GnuTLS

Transport Layer Security Library for the GNU system
for version 3.6.15, 1 June 2020

Nikos Mavrogiannopoulos
Simon Josefsson (bugs@gnutls.org)

This manual is last updated 1 June 2020 for version 3.6.15 of GnuTLS.

Copyright © 2001-2020 Free Software Foundation, Inc.\\ Copyright © 2001-2020 Nikos Mavrogiannopoulos

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Table of Contents

1 Preface 1

2 Introduction to GnuTLS 2

2.1 Downloading and installing 2
2.2 Installing for a software distribution 3
2.3 Overview 4

3 Introduction to TLS and DTLS 5

3.1 TLS Layers 5
3.2 The Transport Layer 6
3.3 The TLS record protocol 6
3.3.1 Encryption algorithms used in the record layer 6
3.3.2 Compression algorithms and the record layer 8
3.3.3 On record padding 8
3.4 The TLS alert protocol 9
3.5 The TLS handshake protocol 10
3.5.1 TLS ciphersuites 10
3.5.2 Authentication 11
3.5.3 Client authentication 11
3.5.4 Resuming sessions 11
3.6 TLS extensions 12
3.6.1 Maximum fragment length negotiation 12
3.6.2 Server name indication 12
3.6.3 Session tickets 12
3.6.4 HeartBeat 12
3.6.5 Safe renegotiation 13
3.6.6 OCSP status request 14
3.6.7 SRTP 15
3.6.8 False Start 16
3.6.9 Application Layer Protocol Negotiation (ALPN) 16
3.6.10 Extensions and Supplemental Data 17
3.7 How to use TLS in application protocols 17
3.7.1 Separate ports 17
3.7.2 Upward negotiation 17
3.8 On SSL 2 and older protocols 19

4 Authentication methods 20

4.1 Certificate authentication 20
4.1.1 X.509 certificates 21
4.1.1.1 X.509 certificate structure 22
4.1.1.2 Importing an X.509 certificate 23
4.1.1.3 X.509 certificate names 24

4.1.1.4 X.509 distinguished names 24
4.1.1.5 X.509 extensions 26
4.1.1.6 Accessing public and private keys 29
4.1.1.7 Verifying X.509 certificate paths 30
4.1.1.8 Verifying a certificate in the context of TLS session ... 36
4.1.1.9 Verifying a certificate using PKCS #11 38
4.1.2 OpenPGP certificates 39
4.1.3 Raw public-keys 40
4.1.3.1 Importing raw public-keys 40
4.1.4 Advanced certificate verification 40
4.1.4.1 Verifying a certificate using trust on first use authentication 40
4.1.4.2 Verifying a certificate using DANE (DNSSEC) 41
4.1.5 Digital signatures 41
4.1.5.1 Trading security for interoperability 42
4.2 More on certificate authentication 43
4.2.1 PKCS #10 certificate requests 43
4.2.2 PKIX certificate revocation lists 46
4.2.3 OCSP certificate status checking 49
4.2.4 OCSP stapling 54
4.2.5 Managing encrypted keys 56
4.2.6 Invoking certtool 61
4.2.7 Invoking ocsptool 82
4.2.8 Invoking danetool 87
4.3 Shared-key and anonymous authentication 92
4.3.1 PSK authentication 92
4.3.1.1 Authentication using PSK 92
4.3.1.2 Invoking psktool 92
4.3.2 SRP authentication 94
4.3.2.1 Authentication using SRP 94
4.3.2.2 Invoking srptool 95
4.3.3 Anonymous authentication 97
4.4 Selecting an appropriate authentication method 97
4.4.1 Two peers with an out-of-band channel 97
4.4.2 Two peers without an out-of-band channel 98
4.4.3 Two peers and a trusted third party 98

5 Abstract key types and Hardware security modules 100

5.1 Abstract key types 100
5.1.1 Public keys 101
5.1.2 Private keys 102
5.1.3 Operations 104
5.2 System and application-specific keys 108
5.2.1 System-specific keys 108
5.2.2 Application-specific keys 108
5.3 Smart cards and HSMs 110
5.3.1 Initialization 110

5.3.2 Manual initialization of user-specific modules 111
5.3.3 Accessing objects that require a PIN 112
5.3.4 Reading objects 113
5.3.5 Writing objects 116
5.3.6 Low Level Access 117
5.3.7 Using a PKCS #11 token with TLS 118
5.3.8 Verifying certificates over PKCS #11 118
5.3.9 Invoking p11tool 119
5.4 Trusted Platform Module (TPM) 130
5.4.1 Keys in TPM 130
5.4.2 Key generation 131
5.4.3 Using keys 131
5.4.4 Invoking tpmtool 133

6 How to use GnuTLS in applications 137
6.1 Introduction 137
6.1.1 General idea 137
6.1.2 Error handling 138
6.1.3 Common types 138
6.1.4 Debugging and auditing 139
6.1.5 Thread safety 141
6.1.6 Running in a sandbox 141
6.1.7 Sessions and fork 142
6.1.8 Callback functions 142
6.2 Preparation 143
6.2.1 Headers 143
6.2.2 Initialization 143
6.2.3 Version check 143
6.2.4 Building the source 143
6.3 Session initialization 144
6.4 Associating the credentials 146
6.4.1 Certificates 147
6.4.2 Raw public-keys 152
6.4.3 SRP 153
6.4.4 PSK 155
6.4.5 Anonymous 156
6.5 Setting up the transport layer 156
6.5.1 Asynchronous operation 160
6.5.1.1 TLS protocol 160
6.5.1.2 Datagram TLS protocol 160
6.5.2 Reducing round-trips 161
6.5.3 Zero-roundtrip mode 162
6.5.4 Anti-replay protection 163
6.5.5 DTLS sessions 165
6.5.6 DTLS and SCTP 165
6.6 TLS handshake 166
6.7 Data transfer and termination 167
6.8 Buffered data transfer 170

6.9 Handling alerts ... 170
6.10 Priority strings ... 172
6.11 Selecting cryptographic key sizes ... 179
6.12 Advanced topics ... 181
6.12.1 Virtual hosts and credentials ... 181
6.12.2 Session resumption ... 183
6.12.3 Certificate verification ... 186
6.12.3.1 Trust on first use ... 186
6.12.3.2 DANE verification ... 188
6.12.4 TLS 1.2 re-authentication ... 190
6.12.4.1 Client side ... 190
6.12.4.2 Server side ... 191
6.12.5 TLS 1.3 re-authentication and re-key ... 191
6.12.6 Parameter generation ... 192
6.12.6.1 Legacy parameter generation ... 192
6.12.7 Deriving keys for other applications/protocols ... 193
6.12.8 Channel bindings ... 194
6.12.9 Interoperability ... 195
6.12.10 Compatibility with the OpenSSL library ... 195

7 GnuTLS application examples ... 196

7.1 Client examples ... 196
7.1.1 Client example with X.509 certificate support ... 196
7.1.2 Datagram TLS client example ... 199
7.1.3 Using a smart card with TLS ... 202
7.1.4 Client with resume capability example ... 206
7.1.5 Client example with SSH-style certificate verification ... 209
7.2 Server examples ... 211
7.2.1 Echo server with X.509 authentication ... 211
7.2.2 DTLS echo server with X.509 authentication ... 216
7.3 More advanced client and servers ... 225
7.3.1 Client example with anonymous authentication ... 225
7.3.2 Using a callback to select the certificate to use ... 227
7.3.3 Obtaining session information ... 232
7.3.4 Advanced certificate verification ... 235
7.3.5 Client example with PSK authentication ... 238
7.3.6 Client example with SRP authentication ... 241
7.3.7 Legacy client example with X.509 certificate support ... 244
7.3.8 Client example using the C++ API ... 248
7.3.9 Echo server with PSK authentication ... 250
7.3.10 Echo server with SRP authentication ... 255
7.3.11 Echo server with anonymous authentication ... 258
7.3.12 Helper functions for TCP connections ... 262
7.3.13 Helper functions for UDP connections ... 263
7.4 OCSP example ... 264
7.5 Miscellaneous examples ... 271
7.5.1 Checking for an alert ... 271
7.5.2 X.509 certificate parsing example ... 272

7.5.3 Listing the ciphersuites in a priority string 275
7.5.4 PKCS #12 structure generation example 276

8 System-wide configuration of the library ... 280

8.1 Application-specific priority strings 280
8.2 Disabling algorithms and protocols 280
8.2.1 Examples 281
8.3 Querying for disabled algorithms and protocols 282
8.4 Overriding the parameter verification profile 282
8.5 Overriding the default priority string 282

9 Using GnuTLS as a cryptographic library .. 283

9.1 Symmetric algorithms 283
9.2 Public key algorithms 286
9.2.1 Key generation 287
9.3 Cryptographic Message Syntax / PKCS7 288
9.4 Hash and MAC functions 290
9.5 Random number generation 294
9.6 Overriding algorithms 294

10 Other included programs 297

10.1 Invoking gnutls-cli 297
10.2 Invoking gnutls-serv 306
10.3 Invoking gnutls-cli-debug 314

11 Internal Architecture of GnuTLS 317

11.1 The TLS Protocol 317
11.2 TLS Handshake Protocol 317
11.3 TLS Authentication Methods 318
11.4 TLS Extension Handling 319
11.5 Cryptographic Backend 325
11.6 Random Number Generators 327
11.7 FIPS140-2 mode 329

Appendix A Upgrading from previous versions .. 332

Appendix B Support 339

B.1 Getting Help 339
B.2 Commercial Support 339
B.3 Bug Reports 339
B.4 Contributing 339
B.5 Certification 340

Appendix C Error Codes and Descriptions ... 341

Appendix D Supported Ciphersuites 350

Appendix E API reference 360

E.1 Core TLS API 360
E.2 Datagram TLS API 490
E.3 X.509 certificate API 493
E.4 PKCS 7 API 620
E.5 OCSP API 629
E.6 PKCS 12 API 641
E.7 Hardware token via PKCS 11 API 649
E.8 TPM API 670
E.9 Abstract key API 671
E.10 Socket specific API 715
E.11 DANE API 716
E.12 Cryptographic API 721
E.13 Compatibility API 737

Appendix F Copying Information 741

Bibliography 749

Function and Data Index 754

Concept Index 765

1 Preface

This document demonstrates and explains the GnuTLS library API. A brief introduction to the protocols and the technology involved is also included so that an application programmer can better understand the GnuTLS purpose and actual offerings. Even if GnuTLS is a typical library software, it operates over several security and cryptographic protocols which require the programmer to make careful and correct usage of them. Otherwise it is likely to only obtain a false sense of security. The term of security is very broad even if restricted to computer software, and cannot be confined to a single cryptographic library. For that reason, do not consider any program secure just because it uses GnuTLS; there are several ways to compromise a program or a communication line and GnuTLS only helps with some of them.

Although this document tries to be self contained, basic network programming and public key infrastructure (PKI) knowledge is assumed in most of it. A good introduction to networking can be found in [[STEVENS], page 752], to public key infrastructure in [[GUTPKI], page 749] and to security engineering in [[ANDERSON], page 753].

Updated versions of the GnuTLS software and this document will be available from `https://www.gnutls.org/`.

2 Introduction to GnuTLS

In brief GnuTLS can be described as a library which offers an API to access secure communication protocols. These protocols provide privacy over insecure lines, and were designed to prevent eavesdropping, tampering, or message forgery.

Technically GnuTLS is a portable ANSI C based library which implements the protocols ranging from SSL 3.0 to TLS 1.3 (see Chapter 3 [Introduction to TLS], page 5, for a detailed description of the protocols), accompanied with the required framework for authentication and public key infrastructure. Important features of the GnuTLS library include:

- Support for TLS 1.3, TLS 1.2, TLS 1.1, TLS 1.0 and optionally SSL 3.0 protocols.
- Support for Datagram TLS 1.0 and 1.2.
- Support for handling and verification of X.509 certificates.
- Support for password authentication using TLS-SRP.
- Support for keyed authentication using TLS-PSK.
- Support for TPM, PKCS #11 tokens and smart-cards.

The GnuTLS library consists of three independent parts, namely the "TLS protocol part", the "Certificate part", and the "Cryptographic back-end" part. The "TLS protocol part" is the actual protocol implementation, and is entirely implemented within the GnuTLS library. The "Certificate part" consists of the certificate parsing, and verification functions and it uses functionality from the libtasn1 library. The "Cryptographic back-end" is provided by the nettle and gmplib libraries.

2.1 Downloading and installing

GnuTLS is available for download at: `https://www.gnutls.org/download.html`

GnuTLS uses a development cycle where even minor version numbers indicate a stable release and a odd minor version number indicate a development release. For example, GnuTLS 1.6.3 denote a stable release since 6 is even, and GnuTLS 1.7.11 denote a development release since 7 is odd.

GnuTLS depends on `nettle` and `gmplib`, and you will need to install it before installing GnuTLS. The `nettle` library is available from `https://www.lysator.liu.se/~nisse/nettle/`, while `gmplib` is available from `https://www.gmplib.org/`. Don't forget to verify the cryptographic signature after downloading source code packages.

The package is then extracted, configured and built like many other packages that use Autoconf. For detailed information on configuring and building it, refer to the `INSTALL` file that is part of the distribution archive. Typically you invoke `./configure` and then `make check install`. There are a number of compile-time parameters, as discussed below.

Several parts of GnuTLS require ASN.1 functionality, which is provided by a library called libtasn1. A copy of libtasn1 is included in GnuTLS. If you want to install it separately (e.g., to make it possibly to use libtasn1 in other programs), you can get it from `https://www.gnu.org/software/libtasn1/`.

The compression library, `libz`, the PKCS #11 helper library `p11-kit`, the TPM library `trousers`, as well as the IDN library `libidn`[1] are optional dependencies. Check the README file in the distribution on how to obtain these libraries.

[1] Needed to use RFC6125 name comparison in internationalized domains.

A few `configure` options may be relevant, summarized below. They disable or enable particular features, to create a smaller library with only the required features. Note however, that although a smaller library is generated, the included programs are not guaranteed to compile if some of these options are given.

```
--disable-srp-authentication
--disable-psk-authentication
--disable-anon-authentication
--disable-dhe
--disable-ecdhe
--disable-openssl-compatibility
--disable-dtls-srtp-support
--disable-alpn-support
--disable-heartbeat-support
--disable-libdane
--without-p11-kit
--without-tpm
--without-zlib
```

For the complete list, refer to the output from `configure --help`.

2.2 Installing for a software distribution

When installing for a software distribution, it is often desirable to preconfigure GnuTLS with the system-wide paths and files. There two important configuration options, one sets the trust store in system, which are the CA certificates to be used by programs by default (if they don't override it), and the other sets to DNSSEC root key file used by unbound for DNSSEC verification.

For the latter the following configuration option is available, and if not specified GnuTLS will try to auto-detect the location of that file.

```
--with-unbound-root-key-file
```

To set the trust store the following options are available.

```
--with-default-trust-store-file
--with-default-trust-store-dir
--with-default-trust-store-pkcs11
```

The first option is used to set a PEM file which contains a list of trusted certificates, while the second will read all certificates in the given path. The recommended option is the last, which allows to use a PKCS #11 trust policy module. That module not only provides the trusted certificates, but allows the categorization of them using purpose, e.g., CAs can be restricted for e-mail usage only, or administrative restrictions of CAs, for examples by restricting a CA to only issue certificates for a given DNS domain using NameConstraints. A publicly available PKCS #11 trust module is p11-kit's trust module[2].

[2] `https://p11-glue.github.io/p11-glue/trust-module.html`

2.3 Overview

In this document we present an overview of the supported security protocols in Chapter 3 [Introduction to TLS], page 5, and continue by providing more information on the certificate authentication in Section 4.1 [Certificate authentication], page 20, and shared-key as well anonymous authentication in Section 4.3 [Shared-key and anonymous authentication], page 92. We elaborate on certificate authentication by demonstrating advanced usage of the API in Section 4.2 [More on certificate authentication], page 43. The core of the TLS library is presented in Chapter 6 [How to use GnuTLS in applications], page 137, and example applications are listed in Chapter 7 [GnuTLS application examples], page 196. In Chapter 10 [Other included programs], page 297, the usage of few included programs that may assist debugging is presented. The last chapter is Chapter 11 [Internal architecture of GnuTLS], page 317, that provides a short introduction to GnuTLS' internal architecture.

3 Introduction to TLS and DTLS

TLS stands for "Transport Layer Security" and is the successor of SSL, the Secure Sockets Layer protocol [[SSL3], page 752] designed by Netscape. TLS is an Internet protocol, defined by IETF[1], described in [[RFC5246], page 750]. The protocol provides confidentiality, and authentication layers over any reliable transport layer. The description, above, refers to TLS 1.0 but applies to all other TLS versions as the differences between the protocols are not major.

The DTLS protocol, or "Datagram TLS" [[RFC4347], page 750] is a protocol with identical goals as TLS, but can operate under unreliable transport layers such as UDP. The discussions below apply to this protocol as well, except when noted otherwise.

3.1 TLS Layers

TLS is a layered protocol, and consists of the record protocol, the handshake protocol and the alert protocol. The record protocol is to serve all other protocols and is above the transport layer. The record protocol offers symmetric encryption, and data authenticity[2]. The alert protocol offers some signaling to the other protocols. It can help informing the peer for the cause of failures and other error conditions. See [The Alert Protocol], page 9, for more information. The alert protocol is above the record protocol.

The handshake protocol is responsible for the security parameters' negotiation, the initial key exchange and authentication. See [The Handshake Protocol], page 10, for more information about the handshake protocol. The protocol layering in TLS is shown in Figure 3.1.

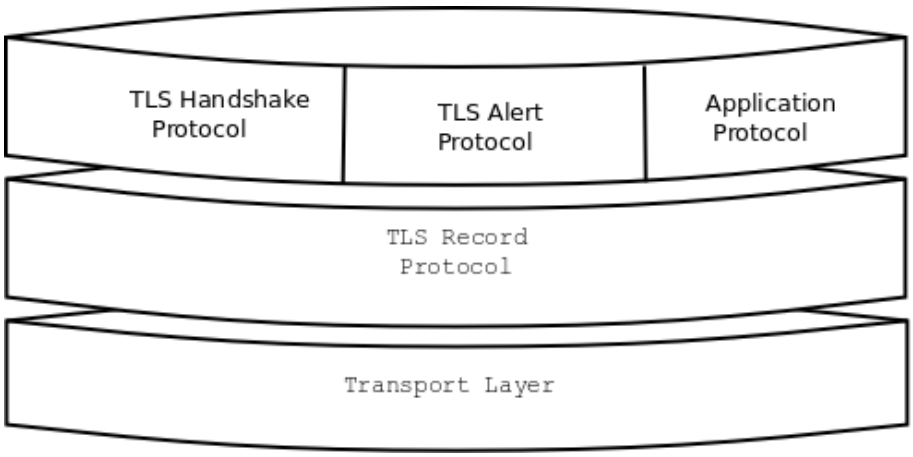

Figure 3.1: The TLS protocol layers.

[1] IETF, or Internet Engineering Task Force, is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.

[2] In early versions of TLS compression was optionally available as well. This is no longer the case in recent versions of the protocol.

3.2 The Transport Layer

TLS is not limited to any transport layer and can be used above any transport layer, as long as it is a reliable one. DTLS can be used over reliable and unreliable transport layers. GnuTLS supports TCP and UDP layers transparently using the Berkeley sockets API. However, any transport layer can be used by providing callbacks for GnuTLS to access the transport layer (for details see Section 6.5 [Setting up the transport layer], page 156).

3.3 The TLS record protocol

The record protocol is the secure communications provider. Its purpose is to encrypt, and authenticate packets. The record layer functions can be called at any time after the handshake process is finished, when there is need to receive or send data. In DTLS however, due to re-transmission timers used in the handshake out-of-order handshake data might be received for some time (maximum 60 seconds) after the handshake process is finished.

The functions to access the record protocol are limited to send and receive functions, which might, given the importance of this protocol in TLS, seem awkward. This is because the record protocol's parameters are all set by the handshake protocol. The record protocol initially starts with NULL parameters, which means no encryption, and no MAC is used. Encryption and authentication begin just after the handshake protocol has finished.

3.3.1 Encryption algorithms used in the record layer

Confidentiality in the record layer is achieved by using symmetric ciphers like `AES` or `CHACHA20`. Ciphers are encryption algorithms that use a single, secret, key to encrypt and decrypt data. Early versions of TLS separated between block and stream ciphers and had message authentication plugged in to them by the protocol, though later versions switched to using authenticated-encryption (AEAD) ciphers. The AEAD ciphers are defined to combine encryption and authentication, and as such they are not only more efficient, as the primitives used are designed to interoperate nicely, but they are also known to interoperate in a secure way.

The supported in GnuTLS ciphers and MAC algorithms are shown in Table 3.1 and Table 3.2.

Algorithm	Type	Applicable Protocols	Description
AES-128-GCM, AES-256-GCM	AEAD	TLS 1.2, TLS 1.3	This is the AES algorithm in the authenticated encryption GCM mode. This mode combines message authentication and encryption and can be extremely fast on CPUs that support hardware acceleration.
AES-128-CCM, AES-256-CCM	AEAD	TLS 1.2, TLS 1.3	This is the AES algorithm in the authenticated encryption CCM mode. This mode combines message authentication and encryption and is often used by systems without AES or GCM acceleration support.
CHACHA20-POLY1305	AEAD	TLS 1.2, TLS 1.3	CHACHA20-POLY1305 is an authenticated encryption algorithm based on CHACHA20 cipher and POLY1305 MAC. CHACHA20 is a refinement of SALSA20 algorithm, an approved cipher by the European ESTREAM project. POLY1305 is Wegman-Carter, one-time authenticator. The combination provides a fast stream cipher suitable for systems where a hardware AES accelerator is not available.
AES-128-CCM-8, AES-256-CCM-8	AEAD	TLS 1.2, TLS 1.3	This is the AES algorithm in the authenticated encryption CCM mode with a truncated to 64-bit authentication tag. This mode is for communication with restricted systems.
CAMELLIA-128-GCM, CAMELLIA-256-GCM	AEAD	TLS 1.2	This is the CAMELLIA algorithm in the authenticated encryption GCM mode.
AES-128-CBC, AES-256-CBC	Legacy (block)	TLS 1.0, TLS 1.1, TLS 1.2	AES or RIJNDAEL is the block cipher algorithm that replaces the old DES algorithm. It has 128 bits block size and is used in CBC mode.
CAMELLIA-128-CBC, CAMELLIA-256-CBC	Legacy (block)	TLS 1.0, TLS 1.1, TLS 1.2	This is an 128-bit block cipher developed by Mitsubishi and NTT. It is one of the approved ciphers of the European NESSIE and Japanese CRYPTREC projects.
3DES-CBC	Legacy (block)	TLS 1.0, TLS 1.1, TLS 1.2	This is the DES block cipher algorithm used with triple encryption (EDE). Has 64 bits block size and is used in CBC mode.
ARCFOUR-128	Legacy (stream)	TLS 1.0, TLS 1.1, TLS 1.2	ARCFOUR-128 is a compatible algorithm with RSA's RC4 algorithm, which is considered to be a trade secret. It is a considered to be broken, and is only used for compatibility purposed. For this reason it is not enabled by default.

Algorithm	Description
MAC-MD5	This is an HMAC based on MD5 a cryptographic hash algorithm designed by Ron Rivest. Outputs 128 bits of data.
MAC-SHA1	An HMAC based on the SHA1 cryptographic hash algorithm designed by NSA. Outputs 160 bits of data.
MAC-SHA256	An HMAC based on SHA2-256. Outputs 256 bits of data.
MAC-SHA384	An HMAC based on SHA2-384. Outputs 384 bits of data.
GOST28147-TC26Z-IMIT	This is a 64-bit block cipher GOST 28147-89 with TC26Z S-Box working in special MAC mode called Imitovstavks. It is one of the approved MAC algorithms in Russia. Outputs 32 bits of data. It is not enabled by default.
MAC-AEAD	This indicates that an authenticated encryption algorithm, such as GCM, is in use.

Table 3.2: Supported MAC algorithms in TLS.

3.3.2 Compression algorithms and the record layer

In early versions of TLS the record layer supported compression. However, that proved to be problematic in many ways, and enabled several attacks based on traffic analysis on the transported data. For that newer versions of the protocol no longer offer compression, and GnuTLS since 3.6.0 no longer implements any support for compression.

3.3.3 On record padding

The TLS 1.3 protocol allows for extra padding of records to prevent statistical analysis based on the length of exchanged messages. GnuTLS takes advantage of this feature, by allowing the user to specify the amount of padding for a particular message. The simplest interface is provided by [gnutls_record_send2], page 452, and is made available when under TLS1.3; alternatively [gnutls_record_can_use_length_hiding], page 448 can be queried.

Note that this interface is not sufficient to completely hide the length of the data. The application code may reveal the data transferred by leaking its data processing time, or by leaking the TLS1.3 record processing time by GnuTLS. That is because under TLS1.3 the padding removal time depends on the padding data for an efficient implementation. To make that processing constant time the [gnutls_init], page 422 function must be called with the flag `GNUTLS_SAFE_PADDING_CHECK`.

`ssize_t gnutls_record_send2` (*gnutls_session_t* `session`, *const void ** `data`, *size_t* `data_size`, *size_t* `pad`, *unsigned* `flags`) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send

data_size: is the length of the data

pad: padding to be added to the record

flags: must be zero

This function is identical to `gnutls_record_send()` except that it takes an extra argument to specify padding to be added the record. To determine the maximum size of padding, use `gnutls_record_get_max_size()` and `gnutls_record_overhead_size()` .

Note that in order for GnuTLS to provide constant time processing of padding and data in TLS1.3, the flag `GNUTLS_SAFE_PADDING_CHECK` must be used in `gnutls_init()` .

Returns: The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

Since: 3.6.3

Older GnuTLS versions provided an API suitable for cases where the sender sends data that are always within a given range. That API is still available, and consists of the following functions.

unsigned [gnutls_record_can_use_length_hiding], page 448, (gnutls_session_t *session*)
ssize_t [gnutls_record_send_range], page 453, (gnutls_session_t *session*, const void * *data*, size_t *data_size*, const gnutls_range_st * *range*)

3.4 The TLS alert protocol

The alert protocol is there to allow signals to be sent between peers. These signals are mostly used to inform the peer about the cause of a protocol failure. Some of these signals are used internally by the protocol and the application protocol does not have to cope with them (e.g. `GNUTLS_A_CLOSE_NOTIFY`), and others refer to the application protocol solely (e.g. `GNUTLS_A_USER_CANCELLED`). An alert signal includes a level indication which may be either fatal or warning (under TLS1.3 all alerts are fatal). Fatal alerts always terminate the current connection, and prevent future re-negotiations using the current session ID. All supported alert messages are summarized in the table below.

The alert messages are protected by the record protocol, thus the information that is included does not leak. You must take extreme care for the alert information not to leak to a possible attacker, via public log files etc.

Alert	ID	Description
GNUTLS_A_CLOSE_NOTIFY	0	Close notify
GNUTLS_A_UNEXPECTED_MESSAGE	10	Unexpected message
GNUTLS_A_BAD_RECORD_MAC	20	Bad record MAC
GNUTLS_A_DECRYPTION_FAILED	21	Decryption failed
GNUTLS_A_RECORD_OVERFLOW	22	Record overflow
GNUTLS_A_DECOMPRESSION_FAILURE	30	Decompression failed
GNUTLS_A_HANDSHAKE_FAILURE	40	Handshake failed

GNUTLS_A_SSL3_NO_CERTIFICATE	41	No certificate (SSL 3.0)
GNUTLS_A_BAD_CERTIFICATE	42	Certificate is bad
GNUTLS_A_UNSUPPORTED_CERTIFICATE	43	Certificate is not supported
GNUTLS_A_CERTIFICATE_REVOKED	44	Certificate was revoked
GNUTLS_A_CERTIFICATE_EXPIRED	45	Certificate is expired
GNUTLS_A_CERTIFICATE_UNKNOWN	46	Unknown certificate
GNUTLS_A_ILLEGAL_PARAMETER	47	Illegal parameter
GNUTLS_A_UNKNOWN_CA	48	CA is unknown
GNUTLS_A_ACCESS_DENIED	49	Access was denied
GNUTLS_A_DECODE_ERROR	50	Decode error
GNUTLS_A_DECRYPT_ERROR	51	Decrypt error
GNUTLS_A_EXPORT_RESTRICTION	60	Export restriction
GNUTLS_A_PROTOCOL_VERSION	70	Error in protocol version
GNUTLS_A_INSUFFICIENT_SECURITY	71	Insufficient security
GNUTLS_A_INTERNAL_ERROR	80	Internal error
GNUTLS_A_INAPPROPRIATE_FALLBACK	86	Inappropriate fallback
GNUTLS_A_USER_CANCELED	90	User canceled
GNUTLS_A_NO_RENEGOTIATION	100	No renegotiation is allowed
GNUTLS_A_MISSING_EXTENSION	109	An extension was expected but was not seen
GNUTLS_A_UNSUPPORTED_EXTENSION	110	An unsupported extension was sent
GNUTLS_A_CERTIFICATE_UNOBTAINABLE	111	Could not retrieve the specified certificate
GNUTLS_A_UNRECOGNIZED_NAME	112	The server name sent was not recognized
GNUTLS_A_UNKNOWN_PSK_IDENTITY	115	The SRP/PSK username is missing or not known
GNUTLS_A_CERTIFICATE_REQUIRED	116	Certificate is required
GNUTLS_A_NO_APPLICATION_PROTOCOL	120	No supported application protocol could be negotiated

3.5 The TLS handshake protocol

The handshake protocol is responsible for the ciphersuite negotiation, the initial key exchange, and the authentication of the two peers. This is fully controlled by the application layer, thus your program has to set up the required parameters. The main handshake function is [gnutls_handshake], page 414. In the next paragraphs we elaborate on the handshake protocol, i.e., the ciphersuite negotiation.

3.5.1 TLS ciphersuites

The TLS cipher suites have slightly different meaning under different protocols. Under TLS 1.3, a cipher suite indicates the symmetric encryption algorithm in use, as well as the pseudo-random function (PRF) used in the TLS session.

Under TLS 1.2 or early the handshake protocol negotiates cipher suites of a special form illustrated by the `TLS_DHE_RSA_WITH_3DES_CBC_SHA` cipher suite name. A typical cipher suite contains these parameters:

- The key exchange algorithm. `DHE_RSA` in the example.
- The Symmetric encryption algorithm and mode `3DES_CBC` in this example.
- The MAC[3] algorithm used for authentication. `MAC_SHA` is used in the above example.

The cipher suite negotiated in the handshake protocol will affect the record protocol, by enabling encryption and data authentication. Note that you should not over rely on TLS to negotiate the strongest available cipher suite. Do not enable ciphers and algorithms that you consider weak.

All the supported ciphersuites are listed in [ciphersuites], page 350.

3.5.2 Authentication

The key exchange algorithms of the TLS protocol offer authentication, which is a prerequisite for a secure connection. The available authentication methods in GnuTLS, under TLS 1.3 or earlier versions, follow.

- Certificate authentication: Authenticated key exchange using public key infrastructure and X.509 certificates.
- PSK authentication: Authenticated key exchange using a pre-shared key.

Under TLS 1.2 or earlier versions, the following authentication methods are also available.

- SRP authentication: Authenticated key exchange using a password.
- Anonymous authentication: Key exchange without peer authentication.

3.5.3 Client authentication

In the case of ciphersuites that use certificate authentication, the authentication of the client is optional in TLS. A server may request a certificate from the client using the [gnutls_certificate_server_set_request], page 373 function. We elaborate in Section 6.4.1 [Certificate credentials], page 147.

3.5.4 Resuming sessions

The TLS handshake process performs expensive calculations and a busy server might easily be put under load. To reduce the load, session resumption may be used. This is a feature of the TLS protocol which allows a client to connect to a server after a successful handshake, without the expensive calculations. This is achieved by re-using the previously established keys, meaning the server needs to store the state of established connections (unless session tickets are used – Section 3.6.3 [Session tickets], page 12).

Session resumption is an integral part of GnuTLS, and Section 6.12.2 [Session resumption], page 183, [ex-resume-client], page 206, illustrate typical uses of it.

[3] MAC stands for Message Authentication Code. It can be described as a keyed hash algorithm. See RFC2104.

3.6 TLS extensions

A number of extensions to the TLS protocol have been proposed mainly in [[TLSEXT], page 752]. The extensions supported in GnuTLS are discussed in the subsections that follow.

3.6.1 Maximum fragment length negotiation

This extension allows a TLS implementation to negotiate a smaller value for record packet maximum length. This extension may be useful to clients with constrained capabilities. The functions shown below can be used to control this extension.

size_t [gnutls_record_get_max_size], page 449, (gnutls_session_t *session*)
ssize_t [gnutls_record_set_max_size], page 454, (gnutls_session_t *session*, size_t *size*)

3.6.2 Server name indication

A common problem in HTTPS servers is the fact that the TLS protocol is not aware of the hostname that a client connects to, when the handshake procedure begins. For that reason the TLS server has no way to know which certificate to send.

This extension solves that problem within the TLS protocol, and allows a client to send the HTTP hostname before the handshake begins within the first handshake packet. The functions [gnutls_server_name_set], page 458 and [gnutls_server_name_get], page 457 can be used to enable this extension, or to retrieve the name sent by a client.

int [gnutls_server_name_set], page 458, (gnutls_session_t *session*, gnutls_server_name_type_t *type*, const void * *name*, size_t *name_length*)
int [gnutls_server_name_get], page 457, (gnutls_session_t *session*, void * *data*, size_t * *data_length*, unsigned int * *type*, unsigned int *indx*)

3.6.3 Session tickets

To resume a TLS session, the server normally stores session parameters. This complicates deployment, and can be avoided by delegating the storage to the client. Because session parameters are sensitive they are encrypted and authenticated with a key only known to the server and then sent to the client. The Session Tickets extension is described in RFC 5077 [[TLSTKT], page 752].

A disadvantage of session tickets is that they eliminate the effects of forward secrecy when a server uses the same key for long time. That is, the secrecy of all sessions on a server using tickets depends on the ticket key being kept secret. For that reason server keys should be rotated and discarded regularly.

Since version 3.1.3 GnuTLS clients transparently support session tickets, unless forward secrecy is explicitly requested (with the PFS priority string).

Under TLS 1.3 session tickets are mandatory for session resumption, and they do not share the forward secrecy concerns as with TLS 1.2 or earlier.

3.6.4 HeartBeat

This is a TLS extension that allows to ping and receive confirmation from the peer, and is described in [[RFC6520], page 751]. The extension is disabled by default and

[gnutls_heartbeat_enable], page 418 can be used to enable it. A policy may be negotiated to only allow sending heartbeat messages or sending and receiving. The current session policy can be checked with [gnutls_heartbeat_allowed], page 418. The requests coming from the peer result to `GNUTLS_E_HEARTBEAT_PING_RECEIVED` being returned from the receive function. Ping requests to peer can be send via [gnutls_heartbeat_ping], page 418.

unsigned [gnutls_heartbeat_allowed], page 418, (gnutls_session_t *session*, unsigned int *type*)
void [gnutls_heartbeat_enable], page 418, (gnutls_session_t *session*, unsigned int *type*)

int [gnutls_heartbeat_ping], page 418, (gnutls_session_t *session*, size_t *data_size*, unsigned int *max_tries*, unsigned int *flags*)
int [gnutls_heartbeat_pong], page 419, (gnutls_session_t *session*, unsigned int *flags*)
void [gnutls_heartbeat_set_timeouts], page 419, (gnutls_session_t *session*, unsigned int *retrans_timeout*, unsigned int *total_timeout*)
unsigned int [gnutls_heartbeat_get_timeout], page 418, (gnutls_session_t *session*)

3.6.5 Safe renegotiation

TLS gives the option to two communicating parties to renegotiate and update their security parameters. One useful example of this feature was for a client to initially connect using anonymous negotiation to a server, and the renegotiate using some authenticated ciphersuite. This occurred to avoid having the client sending its credentials in the clear.

However this renegotiation, as initially designed would not ensure that the party one is renegotiating is the same as the one in the initial negotiation. For example one server could forward all renegotiation traffic to an other server who will see this traffic as an initial negotiation attempt.

This might be seen as a valid design decision, but it seems it was not widely known or understood, thus today some application protocols use the TLS renegotiation feature in a manner that enables a malicious server to insert content of his choice in the beginning of a TLS session.

The most prominent vulnerability was with HTTPS. There servers request a renegotiation to enforce an anonymous user to use a certificate in order to access certain parts of a web site. The attack works by having the attacker simulate a client and connect to a server, with server-only authentication, and send some data intended to cause harm. The server will then require renegotiation from him in order to perform the request. When the proper client attempts to contact the server, the attacker hijacks that connection and forwards traffic to the initial server that requested renegotiation. The attacker will not be able to read the data exchanged between the client and the server. However, the server will (incorrectly) assume that the initial request sent by the attacker was sent by the now authenticated client. The result is a prefix plain-text injection attack.

The above is just one example. Other vulnerabilities exists that do not rely on the TLS renegotiation to change the client's authenticated status (either TLS or application layer).

While fixing these application protocols and implementations would be one natural reaction, an extension to TLS has been designed that cryptographically binds together any renego-

tiated handshakes with the initial negotiation. When the extension is used, the attack is detected and the session can be terminated. The extension is specified in [[RFC5746], page 752].

GnuTLS supports the safe renegotiation extension. The default behavior is as follows. Clients will attempt to negotiate the safe renegotiation extension when talking to servers. Servers will accept the extension when presented by clients. Clients and servers will permit an initial handshake to complete even when the other side does not support the safe renegotiation extension. Clients and servers will refuse renegotiation attempts when the extension has not been negotiated.

Note that permitting clients to connect to servers when the safe renegotiation extension is not enabled, is open up for attacks. Changing this default behavior would prevent interoperability against the majority of deployed servers out there. We will reconsider this default behavior in the future when more servers have been upgraded. Note that it is easy to configure clients to always require the safe renegotiation extension from servers.

To modify the default behavior, we have introduced some new priority strings (see Section 6.10 [Priority Strings], page 172). The `%UNSAFE_RENEGOTIATION` priority string permits (re-)handshakes even when the safe renegotiation extension was not negotiated. The default behavior is `%PARTIAL_RENEGOTIATION` that will prevent renegotiation with clients and servers not supporting the extension. This is secure for servers but leaves clients vulnerable to some attacks, but this is a trade-off between security and compatibility with old servers. The `%SAFE_RENEGOTIATION` priority string makes clients and servers require the extension for every handshake. The latter is the most secure option for clients, at the cost of not being able to connect to legacy servers. Servers will also deny clients that do not support the extension from connecting.

It is possible to disable use of the extension completely, in both clients and servers, by using the `%DISABLE_SAFE_RENEGOTIATION` priority string however we strongly recommend you to only do this for debugging and test purposes.

The default values if the flags above are not specified are:

`Server:` %PARTIAL_RENEGOTIATION

`Client:` %PARTIAL_RENEGOTIATION

For applications we have introduced a new API related to safe renegotiation. The [gnutls_safe_renegotiation_status], page 456 function is used to check if the extension has been negotiated on a session, and can be used both by clients and servers.

3.6.6 OCSP status request

The Online Certificate Status Protocol (OCSP) is a protocol that allows the client to verify the server certificate for revocation without messing with certificate revocation lists. Its drawback is that it requires the client to connect to the server's CA OCSP server and request the status of the certificate. This extension however, enables a TLS server to include its CA OCSP server response in the handshake. That is an HTTPS server may periodically run `ocsptool` (see Section 4.2.7 [ocsptool Invocation], page 82) to obtain its certificate revocation status and serve it to the clients. That way a client avoids an additional connection to the OCSP server.

See Section 4.2.4 [OCSP stapling], page 54, for further information.

Since version 3.1.3 GnuTLS clients transparently support the certificate status request.

3.6.7 SRTP

The TLS protocol was extended in [[RFC5764], page 753] to provide keying material to the Secure RTP (SRTP) protocol. The SRTP protocol provides an encapsulation of encrypted data that is optimized for voice data. With the SRTP TLS extension two peers can negotiate keys using TLS or DTLS and obtain keying material for use with SRTP. The available SRTP profiles are listed below.

`GNUTLS_SRTP_AES128_CM_HMAC_SHA1_80`
: 128 bit AES with a 80 bit HMAC-SHA1

`GNUTLS_SRTP_AES128_CM_HMAC_SHA1_32`
: 128 bit AES with a 32 bit HMAC-SHA1

`GNUTLS_SRTP_NULL_HMAC_SHA1_80`
: NULL cipher with a 80 bit HMAC-SHA1

`GNUTLS_SRTP_NULL_HMAC_SHA1_32`
: NULL cipher with a 32 bit HMAC-SHA1

Figure 3.2: Supported SRTP profiles

To enable use the following functions.

int [gnutls_srtp_set_profile], page 480, (gnutls_session_t *session*, gnutls_srtp_profile_t *profile*)
int [gnutls_srtp_set_profile_direct], page 480, (gnutls_session_t *session*, const char * *profiles*, const char ** *err_pos*)

To obtain the negotiated keys use the function below.

`int gnutls_srtp_get_keys` (*gnutls_session_t* **`session`**, *void *` **`key_material`**, *unsigned int* **`key_material_size`**, *gnutls_datum_t **` **`client_key`**, *gnutls_datum_t **` **`client_salt`**, *gnutls_datum_t **` **`server_key`**, *gnutls_datum_t **` **`server_salt`**) [Function]

session: is a `gnutls_session_t` type.

key_material: Space to hold the generated key material

key_material_size: The maximum size of the key material

client_key: The master client write key, pointing inside the key material

client_salt: The master client write salt, pointing inside the key material

server_key: The master server write key, pointing inside the key material

server_salt: The master server write salt, pointing inside the key material

This is a helper function to generate the keying material for SRTP. It requires the space of the key material to be pre-allocated (should be at least 2x the maximum key size and salt size). The `client_key` , `client_salt` , `server_key` and `server_salt` are convenience datums that point inside the key material. They may be `NULL` .

Returns: On success the size of the key material is returned, otherwise, `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not sufficient, or a negative error code.

Since 3.1.4

Other helper functions are listed below.

int [gnutls_srtp_get_selected_profile], page 479, (gnutls_session_t *session*, gnutls_srtp_profile_t * *profile*)
const char * [gnutls_srtp_get_profile_name], page 479, (gnutls_srtp_profile_t *profile*)
int [gnutls_srtp_get_profile_id], page 479, (const char * *name*, gnutls_srtp_profile_t * *profile*)

3.6.8 False Start

The TLS protocol was extended in [[RFC7918], page 749] to allow the client to send data to server in a single round trip. This change however operates on the borderline of the TLS protocol security guarantees and should be used for the cases where the reduced latency outperforms the risk of an adversary intercepting the transferred data. In GnuTLS applications can use the GNUTLS_ENABLE_FALSE_START as option to [gnutls_init], page 422 to request an early return of the [gnutls_handshake], page 414 function. After that early return the application is expected to transfer any data to be piggybacked on the last handshake message.

After handshake's early termination, the application is expected to transmit data using [gnutls_record_send], page 452, and call [gnutls_record_recv], page 450 on any received data as soon, to ensure that handshake completes timely. That is, especially relevant for applications which set an explicit time limit for the handshake process via [gnutls_handshake_set_timeout], page 417.

Note however, that the API ensures that the early return will not happen if the false start requirements are not satisfied. That is, on ciphersuites which are not whitelisted for false start or on insufficient key sizes, the handshake process will complete properly (i.e., no early return). To verify that false start was used you may use [gnutls_session_get_flags], page 461 and check for the GNUTLS_SFLAGS_FALSE_START flag. For GnuTLS the false start is whitelisted for the following key exchange methods (see [[RFC7918], page 749] for rationale)

- DHE
- ECDHE

but only when the negotiated parameters exceed `GNUTLS_SEC_PARAM_HIGH` –see Table 6.7, and when under (D)TLS 1.2 or later.

3.6.9 Application Layer Protocol Negotiation (ALPN)

The TLS protocol was extended in `RFC7301` to provide the application layer a method of negotiating the application protocol version. This allows for negotiation of the application protocol during the TLS handshake, thus reducing round-trips. The application protocol is described by an opaque string. To enable, use the following functions.

int [gnutls_alpn_set_protocols], page 361, (gnutls_session_t *session*, const gnutls_datum_t * *protocols*, unsigned *protocols_size*, unsigned int *flags*)
int [gnutls_alpn_get_selected_protocol], page 361, (gnutls_session_t *session*, gnutls_datum_t * *protocol*)

Note that these functions are intended to be used with protocols that are registered in the Application Layer Protocol Negotiation IANA registry. While you can use them for other protocols (at the risk of collisions), it is preferable to register them.

3.6.10 Extensions and Supplemental Data

It is possible to transfer supplemental data during the TLS handshake, following [[RFC4680], page 750]. This is for "custom" protocol modifications for applications which may want to transfer additional data (e.g. additional authentication messages). Such an exchange requires a custom extension to be registered. The provided API for this functionality is low-level and described in Section 11.4 [TLS Hello Extension Handling], page 319.

3.7 How to use TLS in application protocols

This chapter is intended to provide some hints on how to use TLS over simple custom made application protocols. The discussion below mainly refers to the TCP/IP transport layer but may be extended to other ones too.

3.7.1 Separate ports

Traditionally SSL was used in application protocols by assigning a new port number for the secure services. By doing this two separate ports were assigned, one for the non-secure sessions, and one for the secure sessions. This method ensures that if a user requests a secure session then the client will attempt to connect to the secure port and fail otherwise. The only possible attack with this method is to perform a denial of service attack. The most famous example of this method is "HTTP over TLS" or HTTPS protocol [[RFC2818], page 751].

Despite its wide use, this method has several issues. This approach starts the TLS Handshake procedure just after the client connects on the —so called— secure port. That way the TLS protocol does not know anything about the client, and popular methods like the host advertising in HTTP do not work[4]. There is no way for the client to say "I connected to YYY server" before the Handshake starts, so the server cannot possibly know which certificate to use.

Other than that it requires two separate ports to run a single service, which is unnecessary complication. Due to the fact that there is a limitation on the available privileged ports, this approach was soon deprecated in favor of upward negotiation.

3.7.2 Upward negotiation

Other application protocols[5] use a different approach to enable the secure layer. They use something often called as the "TLS upgrade" method. This method is quite tricky but it is more flexible. The idea is to extend the application protocol to have a "STARTTLS"

[4] See also the Server Name Indication extension on [serverind], page 12.

[5] See LDAP, IMAP etc.

request, whose purpose it to start the TLS protocols just after the client requests it. This approach does not require any extra port to be reserved. There is even an extension to HTTP protocol to support this method [[RFC2817], page 751].

The tricky part, in this method, is that the "STARTTLS" request is sent in the clear, thus is vulnerable to modifications. A typical attack is to modify the messages in a way that the client is fooled and thinks that the server does not have the "STARTTLS" capability. See a typical conversation of a hypothetical protocol:

(client connects to the server)

CLIENT: HELLO I'M MR. XXX

SERVER: NICE TO MEET YOU XXX

CLIENT: PLEASE START TLS

SERVER: OK

*** TLS STARTS

CLIENT: HERE ARE SOME CONFIDENTIAL DATA

And an example of a conversation where someone is acting in between:

(client connects to the server)

CLIENT: HELLO I'M MR. XXX

SERVER: NICE TO MEET YOU XXX

CLIENT: PLEASE START TLS

(here someone inserts this message)

SERVER: SORRY I DON'T HAVE THIS CAPABILITY

CLIENT: HERE ARE SOME CONFIDENTIAL DATA

As you can see above the client was fooled, and was naïve enough to send the confidential data in the clear, despite the server telling the client that it does not support "STARTTLS".

How do we avoid the above attack? As you may have already noticed this situation is easy to avoid. The client has to ask the user before it connects whether the user requests TLS or not. If the user answered that he certainly wants the secure layer the last conversation should be:

(client connects to the server)

CLIENT: HELLO I'M MR. XXX

SERVER: NICE TO MEET YOU XXX

CLIENT: PLEASE START TLS

(here someone inserts this message)

SERVER: SORRY I DON'T HAVE THIS CAPABILITY

CLIENT: BYE

(the client notifies the user that the secure connection was not possible)

This method, if implemented properly, is far better than the traditional method, and the security properties remain the same, since only denial of service is possible. The benefit is that the server may request additional data before the TLS Handshake protocol starts, in order to send the correct certificate, use the correct password file, or anything else!

3.8 On SSL 2 and older protocols

One of the initial decisions in the GnuTLS development was to implement the known security protocols for the transport layer. Initially TLS 1.0 was implemented since it was the latest at that time, and was considered to be the most advanced in security properties. Later the SSL 3.0 protocol was implemented since it is still the only protocol supported by several servers and there are no serious security vulnerabilities known.

One question that may arise is why we didn't implement SSL 2.0 in the library. There are several reasons, most important being that it has serious security flaws, unacceptable for a modern security library. Other than that, this protocol is barely used by anyone these days since it has been deprecated since 1996. The security problems in SSL 2.0 include:

- Message integrity compromised. The SSLv2 message authentication uses the MD5 function, and is insecure.
- Man-in-the-middle attack. There is no protection of the handshake in SSLv2, which permits a man-in-the-middle attack.
- Truncation attack. SSLv2 relies on TCP FIN to close the session, so the attacker can forge a TCP FIN, and the peer cannot tell if it was a legitimate end of data or not.
- Weak message integrity for export ciphers. The cryptographic keys in SSLv2 are used for both message authentication and encryption, so if weak encryption schemes are negotiated (say 40-bit keys) the message authentication code uses the same weak key, which isn't necessary.

Other protocols such as Microsoft's PCT 1 and PCT 2 were not implemented because they were also abandoned and deprecated by SSL 3.0 and later TLS 1.0.

4 Authentication methods

The initial key exchange of the TLS protocol performs authentication of the peers. In typical scenarios the server is authenticated to the client, and optionally the client to the server.

While many associate TLS with X.509 certificates and public key authentication, the protocol supports various authentication methods, including pre-shared keys, and passwords. In this chapter a description of the existing authentication methods is provided, as well as some guidance on which use-cases each method can be used at.

4.1 Certificate authentication

The most known authentication method of TLS are certificates. The PKIX [[PKIX], page 751] public key infrastructure is daily used by anyone using a browser today. GnuTLS provides a simple API to verify the X.509 certificates as in [[PKIX], page 751].

The key exchange algorithms supported by certificate authentication are shown in Table 4.1.

Key exchange	Description
RSA	The RSA algorithm is used to encrypt a key and send it to the peer. The certificate must allow the key to be used for encryption.
DHE_RSA	The RSA algorithm is used to sign ephemeral Diffie-Hellman parameters which are sent to the peer. The key in the certificate must allow the key to be used for signing. Note that key exchange algorithms which use ephemeral Diffie-Hellman parameters, offer perfect forward secrecy. That means that even if the private key used for signing is compromised, it cannot be used to reveal past session data.
ECDHE_RSA	The RSA algorithm is used to sign ephemeral elliptic curve Diffie-Hellman parameters which are sent to the peer. The key in the certificate must allow the key to be used for signing. It also offers perfect forward secrecy. That means that even if the private key used for signing is compromised, it cannot be used to reveal past session data.
DHE_DSS	The DSA algorithm is used to sign ephemeral Diffie-Hellman parameters which are sent to the peer. The certificate must contain DSA parameters to use this key exchange algorithm. DSA is the algorithm of the Digital Signature Standard (DSS).
ECDHE_ECDSA	The Elliptic curve DSA algorithm is used to sign ephemeral elliptic curve Diffie-Hellman parameters which are sent to the peer. The certificate must contain ECDSA parameters (i.e., EC and marked for signing) to use this key exchange algorithm.

Table 4.1: Supported key exchange algorithms.

4.1.1 X.509 certificates

The X.509 protocols rely on a hierarchical trust model. In this trust model Certification Authorities (CAs) are used to certify entities. Usually more than one certification authorities exist, and certification authorities may certify other authorities to issue certificates as well, following a hierarchical model.

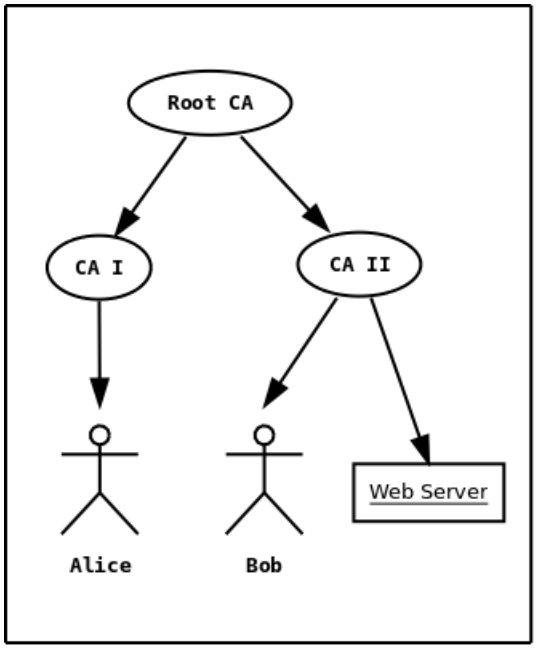

Figure 4.1: An example of the X.509 hierarchical trust model.

One needs to trust one or more CAs for his secure communications. In that case only the certificates issued by the trusted authorities are acceptable. The framework is illustrated on Figure 4.1.

4.1.1.1 X.509 certificate structure

An X.509 certificate usually contains information about the certificate holder, the signer, a unique serial number, expiration dates and some other fields [[PKIX], page 751] as shown in Table 4.2.

Field	Description
version	The field that indicates the version of the certificate.
serialNumber	This field holds a unique serial number per certificate.
signature	The issuing authority's signature.
issuer	Holds the issuer's distinguished name.
validity	The activation and expiration dates.
subject	The subject's distinguished name of the certificate.
extensions	The extensions are fields only present in version 3 certificates.

Table 4.2: X.509 certificate fields.

The certificate's *subject or issuer name* is not just a single string. It is a Distinguished name and in the ASN.1 notation is a sequence of several object identifiers with their corresponding values. Some of available OIDs to be used in an X.509 distinguished name are defined in `gnutls/x509.h`.

The *Version* field in a certificate has values either 1 or 3 for version 3 certificates. Version 1 certificates do not support the extensions field so it is not possible to distinguish a CA from a person, thus their usage should be avoided.

The *validity* dates are there to indicate the date that the specific certificate was activated and the date the certificate's key would be considered invalid.

In GnuTLS the X.509 certificate structures are handled using the `gnutls_x509_crt_t` type and the corresponding private keys with the `gnutls_x509_privkey_t` type. All the available functions for X.509 certificate handling have their prototypes in `gnutls/x509.h`. An example program to demonstrate the X.509 parsing capabilities can be found in [ex-x509-info], page 272.

4.1.1.2 Importing an X.509 certificate

The certificate structure should be initialized using [gnutls_x509_crt_init], page 560, and a certificate structure can be imported using [gnutls_x509_crt_import], page 559.

```
int [gnutls_x509_crt_init], page 560, (gnutls_x509_crt_t * cert)
int [gnutls_x509_crt_import], page 559, (gnutls_x509_crt_t cert, const
gnutls_datum_t * data, gnutls_x509_crt_fmt_t format)
void [gnutls_x509_crt_deinit], page 535, (gnutls_x509_crt_t cert)
```

In several functions an array of certificates is required. To assist in initialization and import the following two functions are provided.

int [gnutls_x509_crt_list_import], page 560, (gnutls_x509_crt_t * *certs*, unsigned int * *cert_max*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, unsigned int *flags*)
int [gnutls_x509_crt_list_import2], page 561, (gnutls_x509_crt_t ** *certs*, unsigned int * *size*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, unsigned int *flags*)

In all cases after use a certificate must be deinitialized using [gnutls_x509_crt_deinit], page 535. Note that although the functions above apply to gnutls_x509_crt_t structure, similar functions exist for the CRL structure gnutls_x509_crl_t.

4.1.1.3 X.509 certificate names

X.509 certificates allow for multiple names and types of names to be specified. CA certificates often rely on X.509 distinguished names (see Section 4.1.1.4 [X.509 distinguished names], page 24) for unique identification, while end-user and server certificates rely on the 'subject alternative names'. The subject alternative names provide a typed name, e.g., a DNS name, or an email address, which identifies the owner of the certificate. The following functions provide access to that names.

int [gnutls_x509_crt_get_subject_alt_name2], page 557, (gnutls_x509_crt_t *cert*, unsigned int *seq*, void * *san*, size_t * *san_size*, unsigned int * *san_type*, unsigned int * *critical*)
int [gnutls_x509_crt_set_subject_alt_name], page 573, (gnutls_x509_crt_t *crt*, gnutls_x509_subject_alt_name_t *type*, const void * *data*, unsigned int *data_size*, unsigned int *flags*)

int [gnutls_subject_alt_names_init], page 495, (gnutls_subject_alt_names_t * *sans*)
int [gnutls_subject_alt_names_get], page 495, (gnutls_subject_alt_names_t *sans*, unsigned int *seq*, unsigned int * *san_type*, gnutls_datum_t * *san*, gnutls_datum_t * *othername_oid*)
int [gnutls_subject_alt_names_set], page 496, (gnutls_subject_alt_names_t *sans*, unsigned int *san_type*, const gnutls_datum_t * *san*, const char * *othername_oid*)

Note however, that server certificates often used the Common Name (CN), part of the certificate DistinguishedName to place a single DNS address. That practice is discouraged (see [[RFC6125], page 749]), because only a single address can be specified, and the CN field is free-form making matching ambiguous.

4.1.1.4 X.509 distinguished names

The "subject" of an X.509 certificate is not described by a single name, but rather with a distinguished name. This in X.509 terminology is a list of strings each associated an object identifier. To make things simple GnuTLS provides [gnutls_x509_crt_get_dn2], page 540 which follows the rules in [[RFC4514], page 750] and returns a single string. Access to each string by individual object identifiers can be accessed using [gnutls_x509_crt_get_dn_by_oid], page 541.

`int gnutls_x509_crt_get_dn2 (`*gnutls_x509_crt_t* `cert`, *gnutls_datum_t ** `dn``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crt_get_dn3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.10

int [gnutls_x509_crt_get_dn], page 539, (gnutls_x509_crt_t *cert*, char * *buf*, size_t * *buf_size*)
int [gnutls_x509_crt_get_dn_by_oid], page 541, (gnutls_x509_crt_t *cert*, const char * *oid*, unsigned *indx*, unsigned int *raw_flag*, void * *buf*, size_t * *buf_size*)
int [gnutls_x509_crt_get_dn_oid], page 541, (gnutls_x509_crt_t *cert*, unsigned *indx*, void * *oid*, size_t * *oid_size*)

Similar functions exist to access the distinguished name of the issuer of the certificate.

int [gnutls_x509_crt_get_issuer_dn], page 547, (gnutls_x509_crt_t *cert*, char * *buf*, size_t * *buf_size*)
int [gnutls_x509_crt_get_issuer_dn2], page 547, (gnutls_x509_crt_t *cert*, gnutls_datum_t * *dn*)
int [gnutls_x509_crt_get_issuer_dn_by_oid], page 548, (gnutls_x509_crt_t *cert*, const char * *oid*, unsigned *indx*, unsigned int *raw_flag*, void * *buf*, size_t * *buf_size*)
int [gnutls_x509_crt_get_issuer_dn_oid], page 549, (gnutls_x509_crt_t *cert*, unsigned *indx*, void * *oid*, size_t * *oid_size*)
int [gnutls_x509_crt_get_issuer], page 545, (gnutls_x509_crt_t *cert*, gnutls_x509_dn_t * *dn*)

The more powerful [gnutls_x509_crt_get_subject], page 556 and [gnutls_x509_dn_get_rdn_ava], page 578 provide efficient but low-level access to the contents of the distinguished name structure.

int [gnutls_x509_crt_get_subject], page 556, (gnutls_x509_crt_t *cert*, gnutls_x509_dn_t * *dn*)
int [gnutls_x509_crt_get_issuer], page 545, (gnutls_x509_crt_t *cert*, gnutls_x509_dn_t * *dn*)

`int gnutls_x509_dn_get_rdn_ava (`*gnutls_x509_dn_t* `dn`, *int* `irdn`, *int* `iava`, *gnutls_x509_ava_st ** `ava``)` [Function]

dn: a pointer to DN

irdn: index of RDN

iava: index of AVA.

ava: Pointer to structure which will hold output information.

Get pointers to data within the DN. The format of the `ava` structure is shown below.

struct gnutls_x509_ava_st { gnutls_datum_t oid; gnutls_datum_t value; unsigned long value_tag; };

The X.509 distinguished name is a sequence of sequences of strings and this is what the `irdn` and `iava` indexes model.

Note that `ava` will contain pointers into the `dn` structure which in turns points to the original certificate. Thus you should not modify any data or deallocate any of those.

This is a low-level function that requires the caller to do the value conversions when necessary (e.g. from UCS-2).

Returns: Returns 0 on success, or an error code.

4.1.1.5 X.509 extensions

X.509 version 3 certificates include a list of extensions that can be used to obtain additional information on the subject or the issuer of the certificate. Those may be e-mail addresses, flags that indicate whether the belongs to a CA etc. All the supported X.509 version 3 extensions are shown in Table 4.3.

The certificate extensions access is split into two parts. The first requires to retrieve the extension, and the second is the parsing part.

To enumerate and retrieve the DER-encoded extension data available in a certificate the following two functions are available.

int [gnutls_x509_crt_get_extension_info], page 543, (gnutls_x509_crt_t *cert*, unsigned *indx*, void * *oid*, size_t * *oid_size*, unsigned int * *critical*)
int [gnutls_x509_crt_get_extension_data2], page 543, (gnutls_x509_crt_t *cert*, unsigned *indx*, gnutls_datum_t * *data*)
int [gnutls_x509_crt_get_extension_by_oid2], page 542, (gnutls_x509_crt_t *cert*, const char * *oid*, unsigned *indx*, gnutls_datum_t * *output*, unsigned int * *critical*)

After a supported DER-encoded extension is retrieved it can be parsed using the APIs in `x509-ext.h`. Complex extensions may require initializing an intermediate structure that holds the parsed extension data. Examples of simple parsing functions are shown below.

int [gnutls_x509_ext_import_basic_constraints], page 585, (const gnutls_datum_t * *ext*, unsigned int * *ca*, int * *pathlen*)
int [gnutls_x509_ext_export_basic_constraints], page 581, (unsigned int *ca*, int *pathlen*, gnutls_datum_t * *ext*)
int [gnutls_x509_ext_import_key_usage], page 586, (const gnutls_datum_t * *ext*, unsigned int * *key_usage*)
int [gnutls_x509_ext_export_key_usage], page 582, (unsigned int *usage*, gnutls_datum_t * *ext*)

More complex extensions, such as Name Constraints, require an intermediate structure, in that case `gnutls_x509_name_constraints_t` to be initialized in order to store the parsed extension data.

int [gnutls_x509_ext_import_name_constraints], page 587, (const gnutls_datum_t * *ext*, gnutls_x509_name_constraints_t *nc*, unsigned int *flags*)
int [gnutls_x509_ext_export_name_constraints], page 582, (gnutls_x509_name_constraints_t *nc*, gnutls_datum_t * *ext*)

After the name constraints are extracted in the structure, the following functions can be used to access them.

int [gnutls_x509_name_constraints_get_permitted], page 592, (gnutls_x509_name_constraints_t *nc*, unsigned *idx*, unsigned * *type*, gnutls_datum_t * *name*)
int [gnutls_x509_name_constraints_get_excluded], page 592, (gnutls_x509_name_constraints_t *nc*, unsigned *idx*, unsigned * *type*, gnutls_datum_t * *name*)
int [gnutls_x509_name_constraints_add_permitted], page 591, (gnutls_x509_name_constraints_t *nc*, gnutls_x509_subject_alt_name_t *type*, const gnutls_datum_t * *name*)
int [gnutls_x509_name_constraints_add_excluded], page 590, (gnutls_x509_name_constraints_t *nc*, gnutls_x509_subject_alt_name_t *type*, const gnutls_datum_t * *name*)

unsigned [gnutls_x509_name_constraints_check], page 591, (gnutls_x509_name_constraints_t *nc*, gnutls_x509_subject_alt_name_t *type*, const gnutls_datum_t * *name*)
unsigned [gnutls_x509_name_constraints_check_crt], page 591, (gnutls_x509_name_constraints_t *nc*, gnutls_x509_subject_alt_name_t *type*, gnutls_x509_crt_t *cert*)

Other utility functions are listed below.

int [gnutls_x509_name_constraints_init], page 593, (gnutls_x509_name_constraints_t * *nc*)
void [gnutls_x509_name_constraints_deinit], page 592, (gnutls_x509_name_constraints_t *nc*)

Similar functions exist for all of the other supported extensions, listed in Table 4.3.

Extension	OID	Description
Subject key id	2.5.29.14	An identifier of the key of the subject.
Key usage	2.5.29.15	Constraints the key's usage of the certificate.
Private key usage period	2.5.29.16	Constraints the validity time of the private key.
Subject alternative name	2.5.29.17	Alternative names to subject's distinguished name.
Issuer alternative name	2.5.29.18	Alternative names to the issuer's distinguished name.
Basic constraints	2.5.29.19	Indicates whether this is a CA certificate or not, and specify the maximum path lengths of certificate chains.
Name constraints	2.5.29.30	A field in CA certificates that restricts the scope of the name of issued certificates.
CRL distribution points	2.5.29.31	This extension is set by the CA, in order to inform about the location of issued Certificate Revocation Lists.
Certificate policy	2.5.29.32	This extension is set to indicate the certificate policy as object identifier and may contain a descriptive string or URL.
Extended key usage	2.5.29.54	Inhibit any policy extension. Constraints the any policy OID (`GNUTLS_X509_OID_POLICY_ANY`) use in the policy extension.
Authority key identifier	2.5.29.35	An identifier of the key of the issuer of the certificate. That is used to distinguish between different keys of the same issuer.
Extended key usage	2.5.29.37	Constraints the purpose of the certificate.
Authority information access	1.3.6.1.5.5.7.1.1	Information on services by the issuer of the certificate.

Note, that there are also direct APIs to access extensions that may be simpler to use for non-complex extensions. They are available in `x509.h` and some examples are listed below.

int [gnutls_x509_crt_get_basic_constraints], page 538, (gnutls_x509_crt_t *cert*, unsigned int * *critical*, unsigned int * *ca*, int * *pathlen*)
int [gnutls_x509_crt_set_basic_constraints], page 563, (gnutls_x509_crt_t *crt*, unsigned int *ca*, int *pathLenConstraint*)
int [gnutls_x509_crt_get_key_usage], page 550, (gnutls_x509_crt_t *cert*, unsigned int * *key_usage*, unsigned int * *critical*)
int [gnutls_x509_crt_set_key_usage], page 570, (gnutls_x509_crt_t *crt*, unsigned int *usage*)

4.1.1.6 Accessing public and private keys

Each X.509 certificate contains a public key that corresponds to a private key. To get a unique identifier of the public key the [gnutls_x509_crt_get_key_id], page 549 function is provided. To export the public key or its parameters you may need to convert the X.509 structure to a `gnutls_pubkey_t`. See Section 5.1.1 [Abstract public keys], page 101, for more information.

`int gnutls_x509_crt_get_key_id` (*gnutls_x509_crt_t* `crt`, *unsigned int* `flags`, *unsigned char * output_data*, *size_t * output_data_size*) [Function]

crt: Holds the certificate

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

Returns: In case of failure a negative error code will be returned, and 0 on success.

The private key parameters may be directly accessed by using one of the following functions.

int [gnutls_x509_privkey_get_pk_algorithm2], page 601, (gnutls_x509_privkey_t *key*, unsigned int * *bits*)
int [gnutls_x509_privkey_export_rsa_raw2], page 599, (gnutls_x509_privkey_t *key*, gnutls_datum_t * *m*, gnutls_datum_t * *e*, gnutls_datum_t * *d*, gnutls_datum_t

* *p*, gnutls_datum_t * *q*, gnutls_datum_t * *u*, gnutls_datum_t * *e1*, gnutls_datum_t * *e2*)
int [gnutls_x509_privkey_export_ecc_raw], page 597, (gnutls_x509_privkey_t *key*, gnutls_ecc_curve_t * *curve*, gnutls_datum_t * *x*, gnutls_datum_t * *y*, gnutls_datum_t * *k*)
int [gnutls_x509_privkey_export_dsa_raw], page 596, (gnutls_x509_privkey_t *key*, gnutls_datum_t * *p*, gnutls_datum_t * *q*, gnutls_datum_t * *g*, gnutls_datum_t * *y*, gnutls_datum_t * *x*)
int [gnutls_x509_privkey_get_key_id], page 601, (gnutls_x509_privkey_t *key*, unsigned int *flags*, unsigned char * *output_data*, size_t * *output_data_size*)

4.1.1.7 Verifying X.509 certificate paths

Verifying certificate paths is important in X.509 authentication. For this purpose the following functions are provided.

`int gnutls_x509_trust_list_add_cas` (*gnutls_x509_trust_list_t* `list`, *const gnutls_x509_crt_t* * `clist`, *unsigned* `clist_size`, *unsigned int* `flags`) [Function]

list: The list

clist: A list of CAs

clist_size: The length of the CA list

flags: flags from `gnutls_trust_list_flags_t`

This function will add the given certificate authorities to the trusted list. The CAs in `clist` must not be deinitialized during the lifetime of `list` .

If the flag `GNUTLS_TL_NO_DUPLICATES` is specified, then this function will ensure that no duplicates will be present in the final trust list.

If the flag `GNUTLS_TL_NO_DUPLICATE_KEY` is specified, then this function will ensure that no certificates with the same key are present in the final trust list.

If either `GNUTLS_TL_NO_DUPLICATE_KEY` or `GNUTLS_TL_NO_DUPLICATES` are given, `gnutls_x509_trust_list_deinit()` must be called with parameter `all` being 1.

Returns: The number of added elements is returned; that includes duplicate entries.

Since: 3.0.0

`int gnutls_x509_trust_list_add_named_crt` (*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* `cert`, *const void* * `name`, *size_t* `name_size`, *unsigned int* `flags`) [Function]

list: The list

cert: A certificate

name: An identifier for the certificate

name_size: The size of the identifier

flags: should be 0.

This function will add the given certificate to the trusted list and associate it with a name. The certificate will not be be used for verification with `gnutls_x509_trust_list_verify_crt()` but with `gnutls_x509_trust_list_verify_named_crt()` or

`gnutls_x509_trust_list_verify_crt2()` - the latter only since GnuTLS 3.4.0 and if a hostname is provided.

In principle this function can be used to set individual "server" certificates that are trusted by the user for that specific server but for no other purposes.

The certificate `cert` must not be deinitialized during the lifetime of the `list` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0.0

`int gnutls_x509_trust_list_add_crls (`*gnutls_x509_trust_list_t* `list`, *const gnutls_x509_crl_t * * `crl_list`, unsigned `crl_size`, unsigned int `flags`, unsigned int `verification_flags`*`)` [Function]

list: The list

crl_list: A list of CRLs

crl_size: The length of the CRL list

flags: flags from `gnutls_trust_list_flags_t`

verification_flags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate revocation lists to the trusted list. The CRLs in `crl_list` must not be deinitialized during the lifetime of `list` .

This function must be called after `gnutls_x509_trust_list_add_cas()` to allow verifying the CRLs for validity. If the flag `GNUTLS_TL_NO_DUPLICATES` is given, then the final CRL list will not contain duplicate entries.

If the flag `GNUTLS_TL_NO_DUPLICATES` is given, `gnutls_x509_trust_list_deinit()` must be called with parameter `all` being 1.

If flag `GNUTLS_TL_VERIFY_CRL` is given the CRLs will be verified before being added, and if verification fails, they will be skipped.

Returns: The number of added elements is returned; that includes duplicate entries.

Since: 3.0

`int gnutls_x509_trust_list_verify_crt (`*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t * `cert_list`, unsigned int `cert_list_size`, unsigned int `flags`, unsigned int * `voutput`, gnutls_verify_output_function `func`*`)` [Function]

list: The list

cert_list: is the certificate list to be verified

cert_list_size: is the certificate list size

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `voutput` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

The details of the verification are the same as in `gnutls_x509_trust_list_verify_crt2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

`int gnutls_x509_trust_list_verify_crt2` [Function]
(*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* * `cert_list`, *unsigned int* `cert_list_size`, *gnutls_typed_vdata_st* * `data`, *unsigned int* `elements`, *unsigned int* `flags`, *unsigned int* * `voutput`, *gnutls_verify_output_function* `func`)

list: The list

cert_list: is the certificate list to be verified

cert_list_size: is the certificate list size

data: an array of typed data

elements: the number of data elements

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will attempt to verify the given certificate chain and return its status. The `voutput` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

When a certificate chain of `cert_list_size` with more than one certificates is provided, the verification status will apply to the first certificate in the chain that failed verification. The verification process starts from the end of the chain (from CA to end certificate). The first certificate in the chain must be the end-certificate while the rest of the members may be sorted or not.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

Additional verification parameters are possible via the `data` types; the acceptable types are `GNUTLS_DT_DNS_HOSTNAME` , `GNUTLS_DT_IP_ADDRESS` and `GNUTLS_DT_KEY_PURPOSE_OID` . The former accepts as data a null-terminated hostname, and the latter a null-terminated object identifier (e.g., `GNUTLS_KP_TLS_WWW_SERVER`). If a DNS hostname is provided then this function will compare the hostname in the end certificate against the given. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set. In addition it will consider certificates provided with `gnutls_x509_trust_list_add_named_crt()` .

If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to match the provided OID or be marked for any purpose, otherwise verification will fail with `GNUTLS_CERT_PURPOSE_MISMATCH` status.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value. Note that verification failure will not result to an error code, only voutput will be updated.

Since: 3.3.8

int gnutls_x509_trust_list_verify_named_crt [Function]
(*gnutls_x509_trust_list_t* **list**, *gnutls_x509_crt_t* **cert**, *const void * **name***, *size_t* **name_size**, *unsigned int* **flags**, *unsigned int * **voutput***, *gnutls_verify_output_function* **func**)

list: The list

cert: is the certificate to be verified

name: is the certificate's name

name_size: is the certificate's name size

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will try to find a certificate that is associated with the provided name – see gnutls_x509_trust_list_add_named_crt() . If a match is found the certificate is considered valid. In addition to that this function will also check CRLs. The voutput parameter will hold an OR'ed sequence of gnutls_certificate_status_t flags.

Additionally a certificate verification profile can be specified from the ones in gnutls_certificate_verification_profiles_t by ORing the result of GNUTLS_PROFILE_TO_VFLAGS() to the verification flags.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.0.0

int gnutls_x509_trust_list_add_trust_file [Function]
(*gnutls_x509_trust_list_t* **list**, *const char * **ca_file***, *const char * **crl_file***, *gnutls_x509_crt_fmt_t* **type**, *unsigned int* **tl_flags**, *unsigned int* **tl_vflags**)

list: The list

ca_file: A file containing a list of CAs (optional)

crl_file: A file containing a list of CRLs (optional)

type: The format of the certificates

tl_flags: flags from gnutls_trust_list_flags_t

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate authorities to the trusted list. PKCS 11 URLs are also accepted, instead of files, by this function. A PKCS 11 URL implies a trust database (a specially marked module in p11-kit); the URL "pkcs11:" implies all trust databases in the system. Only a single URL specifying trust databases can be set; they cannot be stacked with multiple calls.

Returns: The number of added elements is returned.

Since: 3.1

`int gnutls_x509_trust_list_add_trust_mem` [Function]
(*gnutls_x509_trust_list_t* `list`, *const gnutls_datum_t ** `cas`, *const gnutls_datum_t ** `crls`, *gnutls_x509_crt_fmt_t* `type`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The list

cas: A buffer containing a list of CAs (optional)

crls: A buffer containing a list of CRLs (optional)

type: The format of the certificates

tl_flags: flags from `gnutls_trust_list_flags_t`

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate authorities to the trusted list.

If this function is used `gnutls_x509_trust_list_deinit()` must be called with parameter `all` being 1.

Returns: The number of added elements is returned.

Since: 3.1

`int gnutls_x509_trust_list_add_system_trust` [Function]
(*gnutls_x509_trust_list_t* `list`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The structure of the list

tl_flags: GNUTLS_TL_*

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function adds the system's default trusted certificate authorities to the trusted list. Note that on unsupported systems this function returns `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

This function implies the flag `GNUTLS_TL_NO_DUPLICATES` .

Returns: The number of added elements or a negative error code on error.

Since: 3.1

The verification function will verify a given certificate chain against a list of certificate authorities and certificate revocation lists, and output a bit-wise OR of elements of the `gnutls_certificate_status_t` enumeration shown in Figure 4.2. The `GNUTLS_CERT_INVALID` flag is always set on a verification error and more detailed flags will also be set when appropriate.

`GNUTLS_CERT_INVALID`
: The certificate is not signed by one of the known authorities or the signature is invalid (deprecated by the flags `GNUTLS_CERT_SIGNATURE_FAILURE` and `GNUTLS_CERT_SIGNER_NOT_FOUND`).

`GNUTLS_CERT_REVOKED`
: Certificate is revoked by its authority. In X.509 this will be set only if CRLs are checked.

`GNUTLS_CERT_SIGNER_NOT_FOUND`
: The certificate's issuer is not known. This is the case if the issuer is not included in the trusted certificate list.

`GNUTLS_CERT_SIGNER_NOT_CA`
: The certificate's signer was not a CA. This may happen if this was a version 1 certificate, which is common with some CAs, or a version 3 certificate without the basic constrains extension.

`GNUTLS_CERT_INSECURE_ALGORITHM`
: The certificate was signed using an insecure algorithm such as MD2 or MD5. These algorithms have been broken and should not be trusted.

`GNUTLS_CERT_NOT_ACTIVATED`
: The certificate is not yet activated.

`GNUTLS_CERT_EXPIRED`
: The certificate has expired.

`GNUTLS_CERT_SIGNATURE_FAILURE`
: The signature verification failed.

`GNUTLS_CERT_REVOCATION_DATA_SUPERSEDED`
: The revocation data are old and have been superseded.

`GNUTLS_CERT_UNEXPECTED_OWNER`
: The owner is not the expected one.

`GNUTLS_CERT_REVOCATION_DATA_ISSUED_IN_FUTURE`
: The revocation data have a future issue date.

`GNUTLS_CERT_SIGNER_CONSTRAINTS_FAILURE`
: The certificate's signer constraints were violated.

`GNUTLS_CERT_MISMATCH`
: The certificate presented isn't the expected one (TOFU)

`GNUTLS_CERT_PURPOSE_MISMATCH`
: The certificate or an intermediate does not match the intended purpose (extended key usage).

`GNUTLS_CERT_MISSING_OCSP_STATUS`
: The certificate requires the server to send the certifiate status, but no status was received.

`GNUTLS_CERT_INVALID_OCSP_STATUS`
: The received OCSP status response is invalid.

`GNUTLS_CERT_UNKNOWN_CRIT_EXTENSIONS`
: The certificate has extensions marked as critical which are not supported.

Figure 4.2: The `gnutls_certificate_status_t` enumeration.

An example of certificate verification is shown in [ex-verify2], page 235. It is also possible to have a set of certificates that are trusted for a particular server but not to authorize other certificates. This purpose is served by the functions [gnutls_x509_trust_list_add_named_crt], page 613 and [gnutls_x509_trust_list_verify_named_crt], page 620.

4.1.1.8 Verifying a certificate in the context of TLS session

When operating in the context of a TLS session, the trusted certificate authority list may also be set using:

```
int [gnutls_certificate_set_x509_trust_file], page 388,
(gnutls_certificate_credentials_t cred, const char * cafile,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_trust_dir], page 388,
(gnutls_certificate_credentials_t cred, const char * ca_dir,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_crl_file], page 382,
(gnutls_certificate_credentials_t res, const char * crlfile,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_system_trust], page 387,
(gnutls_certificate_credentials_t cred)
```

These functions allow the specification of the trusted certificate authorities, either via a file, a directory or use the system-specified certificate authorities. Unless the authorities are application specific, it is generally recommended to use the system trust storage (see [gnutls_certificate_set_x509_system_trust], page 387).

Unlike the previous section it is not required to setup a trusted list, and there are two approaches to verify the peer's certificate and identity. The recommended in GnuTLS 3.5.0 and later is via the [gnutls_session_set_verify_cert], page 466, but for older GnuTLS versions you may use an explicit callback set via [gnutls_certificate_set_verify_function], page 381 and then utilize [gnutls_certificate_verify_peers3], page 392 for verification. The reported verification status is identical to the verification functions described in the previous section.

Note that in certain cases it is required to check the marked purpose of the end certificate (e.g. `GNUTLS_KP_TLS_WWW_SERVER`); in these cases the more advanced [gnutls_session_set_verify_cert2], page 467 and [gnutls_certificate_verify_peers], page 391 should be used instead.

There is also the possibility to pass some input to the verification functions in the form of flags. For [gnutls_x509_trust_list_verify_crt2], page 619 the flags are passed directly, but for [gnutls_certificate_verify_peers3], page 392, the flags are set using [gnutls_certificate_set_verify_flags], page 380. All the available flags are part of the enumeration `gnutls_certificate_verify_flags` shown in Figure 4.3.

`GNUTLS_VERIFY_DISABLE_CA_SIGN`
: If set a signer does not have to be a certificate authority. This flag should normally be disabled, unless you know what this means.

`GNUTLS_VERIFY_DO_NOT_ALLOW_IP_MATCHES`
: When verifying a hostname prevent textual IP addresses from matching IP addresses in the certificate. Treat the input only as a DNS name.

`GNUTLS_VERIFY_DO_NOT_ALLOW_SAME`
: If a certificate is not signed by anyone trusted but exists in the trusted CA list do not treat it as trusted.

`GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT`
: Allow CA certificates that have version 1 (both root and intermediate). This might be dangerous since those haven't the basicConstraints extension.

`GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2`
: Allow certificates to be signed using the broken MD2 algorithm.

`GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5`
: Allow certificates to be signed using the broken MD5 algorithm.

`GNUTLS_VERIFY_DISABLE_TIME_CHECKS`
: Disable checking of activation and expiration validity periods of certificate chains. Don't set this unless you understand the security implications.

`GNUTLS_VERIFY_DISABLE_TRUSTED_TIME_CHECKS`
: If set a signer in the trusted list is never checked for expiration or activation.

`GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT`
: Do not allow trusted CA certificates that have version 1. This option is to be used to deprecate all certificates of version 1.

`GNUTLS_VERIFY_DISABLE_CRL_CHECKS`
: Disable checking for validity using certificate revocation lists or the available OCSP data.

`GNUTLS_VERIFY_ALLOW_UNSORTED_CHAIN`
: A certificate chain is tolerated if unsorted (the case with many TLS servers out there). This is the default since GnuTLS 3.1.4.

`GNUTLS_VERIFY_DO_NOT_ALLOW_UNSORTED_CHAIN`
: Do not tolerate an unsorted certificate chain.

`GNUTLS_VERIFY_DO_NOT_ALLOW_WILDCARDS`
: When including a hostname check in the verification, do not consider any wildcards.

`GNUTLS_VERIFY_USE_TLS1_RSA`
: This indicates that a (raw) RSA signature is provided as in the TLS 1.0 protocol. Not all functions accept this flag.

`GNUTLS_VERIFY_IGNORE_UNKNOWN_CRIT_EXTENSIONS`
: This signals the verification process, not to fail on unknown critical extensions.

`GNUTLS_VERIFY_ALLOW_SIGN_WITH_SHA1`
: Allow certificates to be signed using the broken SHA1 hash algorithm.

Figure 4.3: The `gnutls_certificate_verify_flags` enumeration.

4.1.1.9 Verifying a certificate using PKCS #11

Some systems provide a system wide trusted certificate storage accessible using the PKCS #11 API. That is, the trusted certificates are queried and accessed using the PKCS #11 API, and trusted certificate properties, such as purpose, are marked using attached extensions. One example is the p11-kit trust module[1].

These special PKCS #11 modules can be used for GnuTLS certificate verification if marked as trust policy modules, i.e., with `trust-policy: yes` in the p11-kit module file. The way to use them is by specifying to the file verification function (e.g., [gnutls_certificate_set_x509_trust_file], page 388), a pkcs11 URL, or simply `pkcs11:` to use all the marked with trust policy modules.

The trust modules of p11-kit assign a purpose to trusted authorities using the extended key usage object identifiers. The common purposes are shown in Table 4.4. Note that typically according to [[RFC5280], page 752] the extended key usage object identifiers apply to end certificates. Their application to CA certificates is an extension used by the trust modules.

[1] see `https://p11-glue.github.io/p11-glue/trust-module.html`.

Purpose	OID	Description
GNUTLS_KP_TLS_WWW_SERVER	1.3.6.1.5.5.7.3.1	The certificate is to be used for TLS WWW authentication. When in a CA certificate, it indicates that the CA is allowed to sign certificates for TLS WWW authentication.
GNUTLS_KP_TLS_WWW_CLIENT	1.3.6.1.5.5.7.3.2	The certificate is to be used for TLS WWW client authentication. When in a CA certificate, it indicates that the CA is allowed to sign certificates for TLS WWW client authentication.
GNUTLS_KP_CODE_SIGNING	1.3.6.1.5.5.7.3.3	The certificate is to be used for code signing. When in a CA certificate, it indicates that the CA is allowed to sign certificates for code signing.
GNUTLS_KP_EMAIL_PROTECTION	1.3.6.1.5.5.7.3.4	The certificate is to be used for email protection. When in a CA certificate, it indicates that the CA is allowed to sign certificates for email users.
GNUTLS_KP_OCSP_SIGNING	1.3.6.1.5.5.7.3.9	The certificate is to be used for signing OCSP responses. When in a CA certificate, it indicates that the CA is allowed to sign certificates which sign OCSP responses.
GNUTLS_KP_ANY	2.5.29.37.0	The certificate is to be used for any purpose. When in a CA certificate, it indicates that the CA is allowed to sign any kind of certificates.

Table 4.4: Key purpose object identifiers.

With such modules, it is recommended to use the verification functions [gnutls_x509_trust_list_verify_crt2], page 619, or [gnutls_certificate_verify_peers], page 391, which allow to explicitly specify the key purpose. The other verification functions which do not allow setting a purpose, would operate as if `GNUTLS_KP_TLS_WWW_SERVER` was requested from the trusted authorities.

4.1.2 OpenPGP certificates

Previous versions of GnuTLS supported limited OpenPGP key authentication. That functionality has been deprecated and is no longer made available. The reason is that, supporting alternative authentication methods, when X.509 and PKIX were new on the Internet and not well established, seemed like a good idea, in today's Internet X.509 is unquestionably the main container for certificates. As such supporting more options with no clear use-cases, is a distraction that consumes considerable resources for improving and testing the library. For that we have decided to drop this functionality completely in 3.6.0.

4.1.3 Raw public-keys

There are situations in which a rather large certificate / certificate chain is undesirable or impractical. An example could be a resource constrained sensor network in which you do want to use authentication of and encryption between your devices but where your devices lack loads of memory or processing power. Furthermore, there are situations in which you don't want to or can't rely on a PKIX. TLS is, next to a PKIX environment, also commonly used with self-signed certificates in smaller deployments where the self-signed certificates are distributed to all involved protocol endpoints out-of-band. This practice does, however, still require the overhead of the certificate generation even though none of the information found in the certificate is actually used.

With raw public-keys, only a subset of the information found in typical certificates is utilized: namely, the SubjectPublicKeyInfo structure (in ASN.1 format) of a PKIX certificate that carries the parameters necessary to describe the public-key. Other parameters found in PKIX certificates are omitted. By omitting various certificate-related structures, the resulting raw public-key is kept fairly small in comparison to the original certificate, and the code to process the keys can be simpler.

It should be noted however, that the authenticity of these raw keys must be verified by an out-of-band mechanism or something like TOFU.

4.1.3.1 Importing raw public-keys

Raw public-keys and their private counterparts can best be handled by using the abstract types `gnutls_pubkey_t` and `gnutls_privkey_t` respectively. To learn how to use these see Section 5.1 [Abstract key types], page 100.

4.1.4 Advanced certificate verification

The verification of X.509 certificates in the HTTPS and other Internet protocols is typically done by loading a trusted list of commercial Certificate Authorities (see [gnutls_certificate_set_x509_system_trust], page 387), and using them as trusted anchors. However, there are several examples (eg. the Diginotar incident) where one of these authorities was compromised. This risk can be mitigated by using in addition to CA certificate verification, other verification methods. In this section we list the available in GnuTLS methods.

4.1.4.1 Verifying a certificate using trust on first use authentication

It is possible to use a trust on first use (TOFU) authentication method in GnuTLS. That is the concept used by the SSH programs, where the public key of the peer is not verified, or verified in an out-of-bound way, but subsequent connections to the same peer require the public key to remain the same. Such a system in combination with the typical CA verification of a certificate, and OCSP revocation checks, can help to provide multiple factor verification, where a single point of failure is not enough to compromise the system. For example a server compromise may be detected using OCSP, and a CA compromise can be detected using the trust on first use method. Such a hybrid system with X.509 and trust on first use authentication is shown in Section 7.1.5 [Client example with SSH-style certificate verification], page 209.

See Section 6.12.3 [Certificate verification], page 186, on how to use the available functionality.

4.1.4.2 Verifying a certificate using DANE (DNSSEC)

The DANE protocol is a protocol that can be used to verify TLS certificates using the DNS (or better DNSSEC) protocols. The DNS security extensions (DNSSEC) provide an alternative public key infrastructure to the commercial CAs that are typically used to sign TLS certificates. The DANE protocol takes advantage of the DNSSEC infrastructure to verify TLS certificates. This can be in addition to the verification by CA infrastructure or may even replace it where DNSSEC is fully deployed. Note however, that DNSSEC deployment is fairly new and it would be better to use it as an additional verification method rather than the only one.

The DANE functionality is provided by the `libgnutls-dane` library that is shipped with GnuTLS and the function prototypes are in `gnutls/dane.h`. See Section 6.12.3 [Certificate verification], page 186, for information on how to use the library.

Note however, that the DANE RFC mandates the verification methods one should use in addition to the validation via DNSSEC TLSA entries. GnuTLS doesn't follow that RFC requirement, and the term DANE verification in this manual refers to the TLSA entry verification. In GnuTLS any other verification methods can be used (e.g., PKIX or TOFU) on top of DANE.

4.1.5 Digital signatures

In this section we will provide some information about digital signatures, how they work, and give the rationale for disabling some of the algorithms used.

Digital signatures work by using somebody's secret key to sign some arbitrary data. Then anybody else could use the public key of that person to verify the signature. Since the data may be arbitrary it is not suitable input to a cryptographic digital signature algorithm. For this reason and also for performance cryptographic hash algorithms are used to preprocess the input to the signature algorithm. This works as long as it is difficult enough to generate two different messages with the same hash algorithm output. In that case the same signature could be used as a proof for both messages. Nobody wants to sign an innocent message of donating 1 euro to Greenpeace and find out that they donated 1.000.000 euros to Bad Inc.

For a hash algorithm to be called cryptographic the following three requirements must hold:

1. Preimage resistance. That means the algorithm must be one way and given the output of the hash function $H(x)$, it is impossible to calculate x.
2. 2nd preimage resistance. That means that given a pair x, y with $y = H(x)$ it is impossible to calculate an x' such that $y = H(x')$.
3. Collision resistance. That means that it is impossible to calculate random x and x' such $H(x') = H(x)$.

The last two requirements in the list are the most important in digital signatures. These protect against somebody who would like to generate two messages with the same hash output. When an algorithm is considered broken usually it means that the Collision resistance of the algorithm is less than brute force. Using the birthday paradox the brute force attack takes $2^{(\text{hash size})/2}$ operations. Today colliding certificates using the MD5 hash algorithm have been generated as shown in [[WEGER], page 753].

There has been cryptographic results for the SHA-1 hash algorithms as well, although they are not yet critical. Before 2004, MD5 had a presumed collision strength of 2^{64}, but it has been showed to have a collision strength well under 2^{50}. As of November 2005, it is believed that SHA-1's collision strength is around 2^{63}. We consider this sufficiently hard so that we still support SHA-1. We anticipate that SHA-256/386/512 will be used in publicly-distributed certificates in the future. When 2^{63} can be considered too weak compared to the computer power available sometime in the future, SHA-1 will be disabled as well. The collision attacks on SHA-1 may also get better, given the new interest in tools for creating them.

4.1.5.1 Trading security for interoperability

If you connect to a server and use GnuTLS' functions to verify the certificate chain, and get a `GNUTLS_CERT_INSECURE_ALGORITHM` validation error (see Section 4.1.1.7 [Verifying X.509 certificate paths], page 30), it means that somewhere in the certificate chain there is a certificate signed using `RSA-MD2` or `RSA-MD5`. These two digital signature algorithms are considered broken, so GnuTLS fails verifying the certificate. In some situations, it may be useful to be able to verify the certificate chain anyway, assuming an attacker did not utilize the fact that these signatures algorithms are broken. This section will give help on how to achieve that.

It is important to know that you do not have to enable any of the flags discussed here to be able to use trusted root CA certificates self-signed using `RSA-MD2` or `RSA-MD5`. The certificates in the trusted list are considered trusted irrespective of the signature.

If you are using [gnutls_certificate_verify_peers3], page 392 to verify the certificate chain, you can call [gnutls_certificate_set_verify_flags], page 380 with the flags:

- `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2`
- `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5`
- `GNUTLS_VERIFY_ALLOW_SIGN_WITH_SHA1`
- `GNUTLS_VERIFY_ALLOW_BROKEN`

as in the following example:

```
gnutls_certificate_set_verify_flags (x509cred,
                                     GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5);
```

This will signal the verifier algorithm to enable `RSA-MD5` when verifying the certificates.

If you are using [gnutls_x509_crt_verify], page 576 or [gnutls_x509_crt_list_verify], page 562, you can pass the `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5` parameter directly in the `flags` parameter.

If you are using these flags, it may also be a good idea to warn the user when verification failure occur for this reason. The simplest is to not use the flags by default, and only fall back to using them after warning the user. If you wish to inspect the certificate chain yourself, you can use [gnutls_certificate_get_peers], page 371 to extract the raw server's certificate chain, [gnutls_x509_crt_list_import], page 560 to parse each of the certificates, and then [gnutls_x509_crt_get_signature_algorithm], page 555 to find out the signing algorithm used for each certificate. If any of the intermediary certificates are using `GNUTLS_SIGN_RSA_MD2` or `GNUTLS_SIGN_RSA_MD5`, you could present a warning.

4.2 More on certificate authentication

Certificates are not the only structures involved in a public key infrastructure. Several other structures that are used for certificate requests, encrypted private keys, revocation lists, GnuTLS abstract key structures, etc., are discussed in this chapter.

4.2.1 PKCS #10 certificate requests

A certificate request is a structure, which contain information about an applicant of a certificate service. It typically contains a public key, a distinguished name and secondary data such as a challenge password. GnuTLS supports the requests defined in PKCS #10 [[RFC2986], page 751]. Other formats of certificate requests are not currently supported by GnuTLS.

A certificate request can be generated by associating it with a private key, setting the subject's information and finally self signing it. The last step ensures that the requester is in possession of the private key.

int [gnutls_x509_crq_set_version], page 531, (gnutls_x509_crq_t *crq*, unsigned int *version*)
int [gnutls_x509_crq_set_dn], page 527, (gnutls_x509_crq_t *crq*, const char * *dn*, const char ** *err*)
int [gnutls_x509_crq_set_dn_by_oid], page 527, (gnutls_x509_crq_t *crq*, const char * *oid*, unsigned int *raw_flag*, const void * *data*, unsigned int *sizeof_data*)
int [gnutls_x509_crq_set_key_usage], page 529, (gnutls_x509_crq_t *crq*, unsigned int *usage*)
int [gnutls_x509_crq_set_key_purpose_oid], page 528, (gnutls_x509_crq_t *crq*, const void * *oid*, unsigned int *critical*)
int [gnutls_x509_crq_set_basic_constraints], page 527, (gnutls_x509_crq_t *crq*, unsigned int *ca*, int *pathLenConstraint*)

The [gnutls_x509_crq_set_key], page 528 and [gnutls_x509_crq_sign2], page 531 functions associate the request with a private key and sign it. If a request is to be signed with a key residing in a PKCS #11 token it is recommended to use the signing functions shown in Section 5.1 [Abstract key types], page 100.

`int gnutls_x509_crq_set_key` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_privkey_t* `key`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

This function will set the public parameters from the given private key to the request.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

`int gnutls_x509_crq_sign2` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_privkey_t* `key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

dig: The message digest to use, i.e., `GNUTLS_DIG_SHA256`

flags: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in `gnutls_x509_crt_set_key()` since a certificate request is self signed.

This must be the last step in a certificate request generation since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed request will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code. `GNUTLS_E_ASN1_VALUE_NOT_FOUND` is returned if you didn't set all information in the certificate request (e.g., the version using `gnutls_x509_crq_set_version()`).

The following example is about generating a certificate request, and a private key. A certificate request can be later be processed by a CA which should return a signed certificate.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/abstract.h>
#include <time.h>

/* This example will generate a private key and a certificate
 * request.
 */

int main(void)
{
        gnutls_x509_crq_t crq;
        gnutls_x509_privkey_t key;
        unsigned char buffer[10 * 1024];
        size_t buffer_size = sizeof(buffer);
        unsigned int bits;

        gnutls_global_init();

        /* Initialize an empty certificate request, and
         * an empty private key.
```

```
 */
gnutls_x509_crq_init(&crq);

gnutls_x509_privkey_init(&key);

/* Generate an RSA key of moderate security.
 */
bits =
    gnutls_sec_param_to_pk_bits(GNUTLS_PK_RSA,
                                GNUTLS_SEC_PARAM_MEDIUM);
gnutls_x509_privkey_generate(key, GNUTLS_PK_RSA, bits, 0);

/* Add stuff to the distinguished name
 */
gnutls_x509_crq_set_dn_by_oid(crq, GNUTLS_OID_X520_COUNTRY_NAME,
                              0, "GR", 2);

gnutls_x509_crq_set_dn_by_oid(crq, GNUTLS_OID_X520_COMMON_NAME,
                              0, "Nikos", strlen("Nikos"));

/* Set the request version.
 */
gnutls_x509_crq_set_version(crq, 1);

/* Set a challenge password.
 */
gnutls_x509_crq_set_challenge_password(crq,
                                       "something to remember here");

/* Associate the request with the private key
 */
gnutls_x509_crq_set_key(crq, key);

/* Self sign the certificate request.
 */
gnutls_x509_crq_sign2(crq, key, GNUTLS_DIG_SHA1, 0);

/* Export the PEM encoded certificate request, and
 * display it.
 */
gnutls_x509_crq_export(crq, GNUTLS_X509_FMT_PEM, buffer,
                       &buffer_size);

printf("Certificate Request: \n%s", buffer);


/* Export the PEM encoded private key, and
```

```
         * display it.
         */
        buffer_size = sizeof(buffer);
        gnutls_x509_privkey_export(key, GNUTLS_X509_FMT_PEM, buffer,
                                   &buffer_size);

        printf("\n\nPrivate key: \n%s", buffer);

        gnutls_x509_crq_deinit(crq);
        gnutls_x509_privkey_deinit(key);

        return 0;

}
```

4.2.2 PKIX certificate revocation lists

A certificate revocation list (CRL) is a structure issued by an authority periodically containing a list of revoked certificates serial numbers. The CRL structure is signed with the issuing authorities' keys. A typical CRL contains the fields as shown in Table 4.5. Certificate revocation lists are used to complement the expiration date of a certificate, in order to account for other reasons of revocation, such as compromised keys, etc.

Each CRL is valid for limited amount of time and is required to provide, except for the current issuing time, also the issuing time of the next update.

Field	**Description**
version	The field that indicates the version of the CRL structure.
signature	A signature by the issuing authority.
issuer	Holds the issuer's distinguished name.
thisUpdate	The issuing time of the revocation list.
nextUpdate	The issuing time of the revocation list that will update that one.
revokedCertificates	List of revoked certificates serial numbers.
extensions	Optional CRL structure extensions.

Table 4.5: Certificate revocation list fields.

The basic CRL structure functions follow.

int [gnutls_x509_crl_init], page 508, (gnutls_x509_crl_t * *crl*)
int [gnutls_x509_crl_import], page 508, (gnutls_x509_crl_t *crl*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*)
int [gnutls_x509_crl_export], page 501, (gnutls_x509_crl_t *crl*, gnutls_x509_crt_fmt_t *format*, void * *output_data*, size_t * *output_data_size*)
int [gnutls_x509_crl_export], page 501, (gnutls_x509_crl_t *crl*, gnutls_x509_crt_fmt_t *format*, void * *output_data*, size_t * *output_data_size*)

Reading a CRL

The most important function that extracts the certificate revocation information from a CRL is [gnutls_x509_crl_get_crt_serial], page 502. Other functions that return other fields of the CRL structure are also provided.

`int gnutls_x509_crl_get_crt_serial (`*gnutls_x509_crl_t* **`crl`**, *unsigned* **`indx`**, *unsigned char* * **`serial`**, *size_t* * **`serial_size`**, *time_t* * **`t`**`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: the index of the certificate to extract (starting from 0)

serial: where the serial number will be copied

serial_size: initially holds the size of serial

t: if non null, will hold the time this certificate was revoked

This function will retrieve the serial number of the specified, by the index, revoked certificate.

Note that this function will have performance issues in large sequences of revoked certificates. In that case use `gnutls_x509_crl_iter_crt_serial()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

int [gnutls_x509_crl_get_version], page 508, (gnutls_x509_crl_t *crl*)
int [gnutls_x509_crl_get_issuer_dn], page 505, (gnutls_x509_crl_t *crl*, char * *buf*, size_t * *sizeof_buf*)
int [gnutls_x509_crl_get_issuer_dn2], page 505, (gnutls_x509_crl_t *crl*, gnutls_datum_t * *dn*)
time_t [gnutls_x509_crl_get_this_update], page 508, (gnutls_x509_crl_t *crl*)
time_t [gnutls_x509_crl_get_next_update], page 506, (gnutls_x509_crl_t *crl*)
int [gnutls_x509_crl_get_crt_count], page 502, (gnutls_x509_crl_t *crl*)

Generation of a CRL

The following functions can be used to generate a CRL.

int [gnutls_x509_crl_set_version], page 512, (gnutls_x509_crl_t *crl*, unsigned int *version*)
int [gnutls_x509_crl_set_crt_serial], page 511, (gnutls_x509_crl_t *crl*, const void * *serial*, size_t *serial_size*, time_t *revocation_time*)
int [gnutls_x509_crl_set_crt], page 511, (gnutls_x509_crl_t *crl*, gnutls_x509_crt_t *crt*, time_t *revocation_time*)
int [gnutls_x509_crl_set_next_update], page 511, (gnutls_x509_crl_t *crl*, time_t *exp_time*)
int [gnutls_x509_crl_set_this_update], page 512, (gnutls_x509_crl_t *crl*, time_t *act_time*)

The [gnutls_x509_crl_sign2], page 513 and [gnutls_x509_crl_privkey_sign], page 713 functions sign the revocation list with a private key. The latter function can be used to sign with a key residing in a PKCS #11 token.

`int gnutls_x509_crl_sign2` (*gnutls_x509_crl_t* ***crl***, *gnutls_x509_crt_t* ***issuer***, *gnutls_x509_privkey_t* ***issuer_key***, *gnutls_digest_algorithm_t* ***dig***, *unsigned int* ***flags***) [Function]

crl: should contain a gnutls_x509_crl_t type

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use. GNUTLS_DIG_SHA256 is the safe choice unless you know what you're doing.

flags: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed CRL will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

`int gnutls_x509_crl_privkey_sign` (*gnutls_x509_crl_t* ***crl***, *gnutls_x509_crt_t* ***issuer***, *gnutls_privkey_t* ***issuer_key***, *gnutls_digest_algorithm_t* ***dig***, *unsigned int* ***flags***) [Function]

crl: should contain a gnutls_x509_crl_t type

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use. GNUTLS_DIG_SHA256 is the safe choice unless you know what you're doing.

flags: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed CRL will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 2.12.0

Few extensions on the CRL structure are supported, including the CRL number extension and the authority key identifier.

int [gnutls_x509_crl_set_number], page 511, (gnutls_x509_crl_t *crl*, const void * *nr*, size_t *nr_size*)
int [gnutls_x509_crl_set_authority_key_id], page 510, (gnutls_x509_crl_t *crl*, const void * *id*, size_t *id_size*)

4.2.3 OCSP certificate status checking

Certificates may be revoked before their expiration time has been reached. There are several reasons for revoking certificates, but a typical situation is when the private key associated with a certificate has been compromised. Traditionally, Certificate Revocation Lists (CRLs) have been used by application to implement revocation checking, however, several problems with CRLs have been identified [[RIVESTCRL], page 753].

The Online Certificate Status Protocol, or OCSP [[RFC2560], page 753], is a widely implemented protocol which performs certificate revocation status checking. An application that wish to verify the identity of a peer will verify the certificate against a set of trusted certificates and then check whether the certificate is listed in a CRL and/or perform an OCSP check for the certificate.

Applications are typically expected to contact the OCSP server in order to request the certificate validity status. The OCSP server replies with an OCSP response. This section describes this online communication (which can be avoided when using OCSP stapled responses, for that, see Section 4.2.4 [OCSP stapling], page 54).

Before performing the OCSP query, the application will need to figure out the address of the OCSP server. The OCSP server address can be provided by the local user in manual configuration or may be stored in the certificate that is being checked. When stored in a certificate the OCSP server is in the extension field called the Authority Information Access (AIA). The following function extracts this information from a certificate.

int [gnutls_x509_crt_get_authority_info_access], page 536,
(gnutls_x509_crt_t *crt*, unsigned int *seq*, int *what*, gnutls_datum_t * *data*,
unsigned int * *critical*)

There are several functions in GnuTLS for creating and manipulating OCSP requests and responses. The general idea is that a client application creates an OCSP request object,

stores some information about the certificate to check in the request, and then exports the request in DER format. The request will then need to be sent to the OCSP responder, which needs to be done by the application (GnuTLS does not send and receive OCSP packets). Normally an OCSP response is received that the application will need to import into an OCSP response object. The digital signature in the OCSP response needs to be verified against a set of trust anchors before the information in the response can be trusted.

The ASN.1 structure of OCSP requests are briefly as follows. It is useful to review the structures to get an understanding of which fields are modified by GnuTLS functions.

```
OCSPRequest     ::=     SEQUENCE {
    tbsRequest                  TBSRequest,
    optionalSignature   [0]     EXPLICIT Signature OPTIONAL }

TBSRequest      ::=     SEQUENCE {
    version             [0]     EXPLICIT Version DEFAULT v1,
    requestorName       [1]     EXPLICIT GeneralName OPTIONAL,
    requestList                 SEQUENCE OF Request,
    requestExtensions   [2]     EXPLICIT Extensions OPTIONAL }

Request         ::=     SEQUENCE {
    reqCert                     CertID,
    singleRequestExtensions     [0] EXPLICIT Extensions OPTIONAL }

CertID          ::=     SEQUENCE {
    hashAlgorithm       AlgorithmIdentifier,
    issuerNameHash      OCTET STRING, -- Hash of Issuer's DN
    issuerKeyHash       OCTET STRING, -- Hash of Issuers public key
    serialNumber        CertificateSerialNumber }
```

The basic functions to initialize, import, export and deallocate OCSP requests are the following.

int [gnutls_ocsp_req_init], page 632, (gnutls_ocsp_req_t * *req*)
void [gnutls_ocsp_req_deinit], page 630, (gnutls_ocsp_req_t *req*)
int [gnutls_ocsp_req_import], page 632, (gnutls_ocsp_req_t *req*, const gnutls_datum_t * *data*)
int [gnutls_ocsp_req_export], page 630, (gnutls_ocsp_req_const_t *req*, gnutls_datum_t * *data*)
int [gnutls_ocsp_req_print], page 632, (gnutls_ocsp_req_const_t *req*, gnutls_ocsp_print_formats_t *format*, gnutls_datum_t * *out*)

To generate an OCSP request the issuer name hash, issuer key hash, and the checked certificate's serial number are required. There are two interfaces available for setting those in an OCSP request. The is a low-level function when you have the issuer name hash, issuer key hash, and certificate serial number in binary form. The second is more useful if you have the certificate (and its issuer) in a `gnutls_x509_crt_t` type. There is also a function to extract this information from existing an OCSP request.

int [gnutls_ocsp_req_add_cert_id], page 630, (gnutls_ocsp_req_t *req*, gnutls_digest_algorithm_t *digest*, const gnutls_datum_t * *issuer_name_hash*, const gnutls_datum_t * *issuer_key_hash*, const gnutls_datum_t * *serial_number*)
int [gnutls_ocsp_req_add_cert], page 629, (gnutls_ocsp_req_t *req*, gnutls_digest_algorithm_t *digest*, gnutls_x509_crt_t *issuer*, gnutls_x509_crt_t *cert*)
int [gnutls_ocsp_req_get_cert_id], page 631, (gnutls_ocsp_req_const_t *req*, unsigned *indx*, gnutls_digest_algorithm_t * *digest*, gnutls_datum_t * *issuer_name_hash*, gnutls_datum_t * *issuer_key_hash*, gnutls_datum_t * *serial_number*)

Each OCSP request may contain a number of extensions. Extensions are identified by an Object Identifier (OID) and an opaque data buffer whose syntax and semantics is implied by the OID. You can extract or set those extensions using the following functions.

int [gnutls_ocsp_req_get_extension], page 631, (gnutls_ocsp_req_const_t *req*, unsigned *indx*, gnutls_datum_t * *oid*, unsigned int * *critical*, gnutls_datum_t * *data*)
int [gnutls_ocsp_req_set_extension], page 633, (gnutls_ocsp_req_t *req*, const char * *oid*, unsigned int *critical*, const gnutls_datum_t * *data*)

A common OCSP Request extension is the nonce extension (OID 1.3.6.1.5.5.7.48.1.2), which is used to avoid replay attacks of earlier recorded OCSP responses. The nonce extension carries a value that is intended to be sufficiently random and unique so that an attacker will not be able to give a stale response for the same nonce.

int [gnutls_ocsp_req_get_nonce], page 632, (gnutls_ocsp_req_const_t *req*, unsigned int * *critical*, gnutls_datum_t * *nonce*)
int [gnutls_ocsp_req_set_nonce], page 633, (gnutls_ocsp_req_t *req*, unsigned int *critical*, const gnutls_datum_t * *nonce*)
int [gnutls_ocsp_req_randomize_nonce], page 633, (gnutls_ocsp_req_t *req*)

The OCSP response structures is a complex structure. A simplified overview of it is in Table 4.6. Note that a response may contain information on multiple certificates.

Field	Description
version	The OCSP response version number (typically 1).
responder ID	An identifier of the responder (DN name or a hash of its key).
issue time	The time the response was generated.
thisUpdate	The issuing time of the revocation information.
nextUpdate	The issuing time of the revocation information that will update that one.
	Revoked certificates
certificate status	The status of the certificate.
certificate serial	The certificate's serial number.
revocationTime	The time the certificate was revoked.
revocationReason	The reason the certificate was revoked.

Table 4.6: The most important OCSP response fields.

We provide basic functions for initialization, importing, exporting and deallocating OCSP responses.

```
int [gnutls_ocsp_resp_init], page 639, (gnutls_ocsp_resp_t * resp)
void [gnutls_ocsp_resp_deinit], page 634, (gnutls_ocsp_resp_t resp)
int [gnutls_ocsp_resp_import], page 639, (gnutls_ocsp_resp_t resp, const
gnutls_datum_t * data)
int [gnutls_ocsp_resp_export], page 634, (gnutls_ocsp_resp_const_t resp,
gnutls_datum_t * data)
int [gnutls_ocsp_resp_print], page 640, (gnutls_ocsp_resp_const_t resp,
gnutls_ocsp_print_formats_t format, gnutls_datum_t * out)
```

The utility function that extracts the revocation as well as other information from a response is shown below.

`int gnutls_ocsp_resp_get_single` (*gnutls_ocsp_resp_const_t* **`resp`**, *unsigned* **`indx`**, *gnutls_digest_algorithm_t ** **`digest`**, *gnutls_datum_t ** **`issuer_name_hash`**, *gnutls_datum_t ** **`issuer_key_hash`**, *gnutls_datum_t ** **`serial_number`**, *unsigned int ** **`cert_status`**, *time_t ** **`this_update`**, *time_t ** **`next_update`**, *time_t ** **`revocation_time`**, *unsigned int ** **`revocation_reason`**) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

indx: Specifies response number to get. Use (0) to get the first one.

digest: output variable with `gnutls_digest_algorithm_t` hash algorithm

issuer_name_hash: output buffer with hash of issuer's DN

issuer_key_hash: output buffer with hash of issuer's public key

serial_number: output buffer with serial number of certificate to check

cert_status: a certificate status, a `gnutls_ocsp_cert_status_t` enum.

this_update: time at which the status is known to be correct.

next_update: when newer information will be available, or (time_t)-1 if unspecified

revocation_time: when `cert_status` is `GNUTLS_OCSP_CERT_REVOKED` , holds time of revocation.

revocation_reason: revocation reason, a `gnutls_x509_crl_reason_t` enum.

This function will return the certificate information of the `indx` 'ed response in the Basic OCSP Response `resp` . The information returned corresponds to the OCSP SingleResponse structure except the final singleExtensions.

Each of the pointers to output variables may be NULL to indicate that the caller is not interested in that value.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

The possible revocation reasons available in an OCSP response are shown below.

`GNUTLS_X509_CRLREASON_UNSPECIFIED`
: Unspecified reason.

`GNUTLS_X509_CRLREASON_KEYCOMPROMISE`
: Private key compromised.

`GNUTLS_X509_CRLREASON_CACOMPROMISE`
: CA compromised.

`GNUTLS_X509_CRLREASON_AFFILIATIONCHANGED`
: Affiliation has changed.

`GNUTLS_X509_CRLREASON_SUPERSEDED`
: Certificate superseded.

`GNUTLS_X509_CRLREASON_CESSATIONOFOPERATION`
: Operation has ceased.

`GNUTLS_X509_CRLREASON_CERTIFICATEHOLD`
: Certificate is on hold.

`GNUTLS_X509_CRLREASON_REMOVEFROMCRL`
: Will be removed from delta CRL.

`GNUTLS_X509_CRLREASON_PRIVILEGEWITHDRAWN`
: Privilege withdrawn.

`GNUTLS_X509_CRLREASON_AACOMPROMISE`
: AA compromised.

Figure 4.4: The revocation reasons

Note, that the OCSP response needs to be verified against some set of trust anchors before it can be relied upon. It is also important to check whether the received OCSP response corresponds to the certificate being checked.

int [gnutls_ocsp_resp_verify], page 640, (gnutls_ocsp_resp_const_t *resp*, gnutls_x509_trust_list_t *trustlist*, unsigned int * *verify*, unsigned int *flags*)
int [gnutls_ocsp_resp_verify_direct], page 641, (gnutls_ocsp_resp_const_t *resp*, gnutls_x509_crt_t *issuer*, unsigned int * *verify*, unsigned int *flags*)
int [gnutls_ocsp_resp_check_crt], page 634, (gnutls_ocsp_resp_const_t *resp*, unsigned int *indx*, gnutls_x509_crt_t *crt*)

4.2.4 OCSP stapling

To avoid applications contacting the OCSP server directly, TLS servers can provide a "stapled" OCSP response in the TLS handshake. That way the client application needs to do nothing more. GnuTLS will automatically consider the stapled OCSP response during the TLS certificate verification (see [gnutls_certificate_verify_peers2], page 392). To disable the automatic OCSP verification the flag `GNUTLS_VERIFY_DISABLE_CRL_CHECKS` should be specified to [gnutls_certificate_set_verify_flags], page 380.

Since GnuTLS 3.5.1 the client certificate verification will consider the [[RFC7633], page 750] OCSP-Must-staple certificate extension, and will consider it while checking for stapled

OCSP responses. If the extension is present and no OCSP staple is found, the certificate verification will fail and the status code `GNUTLS_CERT_MISSING_OCSP_STATUS` will returned from the verification function.

Under TLS 1.2 only one stapled response can be sent by a server, the OCSP response associated with the end-certificate. Under TLS 1.3 a server can send multiple OCSP responses, typically one for each certificate in the certificate chain. The following functions can be used by a client application to retrieve the OCSP responses as sent by the server.

int [gnutls_ocsp_status_request_get], page 425, (gnutls_session_t *session*, gnutls_datum_t * *response*)
int [gnutls_ocsp_status_request_get2], page 425, (gnutls_session_t *session*, unsigned *idx*, gnutls_datum_t * *response*)

GnuTLS servers can provide OCSP responses to their clients using the following functions.

void [gnutls_certificate_set_retrieve_function3], page 673, (gnutls_certificate_credentials_t *cred*, gnutls_certificate_retrieve_function3 * *func*)
int [gnutls_certificate_set_ocsp_status_request_file2], page 375, (gnutls_certificate_credentials_t *sc*, const char * *response_file*, unsigned *idx*, gnutls_x509_crt_fmt_t *fmt*)
unsigned [gnutls_ocsp_status_request_is_checked], page 426, (gnutls_session_t *session*, unsigned int *flags*)

A server is expected to provide the relevant certificate's OCSP responses using [gnutls_certificate_set_ocsp_status_request_file2], page 375, and ensure a periodic reload/renew of the credentials. An estimation of the OCSP responses expiration can be obtained using the [gnutls_certificate_get_ocsp_expiration], page 370 function.

`time_t gnutls_certificate_get_ocsp_expiration` [Function]
(*gnutls_certificate_credentials_t* `sc`, *unsigned* `idx`, *int* `oidx`, *unsigned* `flags`)

sc: is a credentials structure.

idx: is a certificate chain index as returned by `gnutls_certificate_set_key()` and friends

oidx: is an OCSP response index

flags: should be zero

This function returns the validity of the loaded OCSP responses, to provide information on when to reload/refresh them.

Note that the credentials structure should be read-only when in use, thus when reloading, either the credentials structure must not be in use by any sessions, or a new credentials structure should be allocated for new sessions.

When `oidx` is (-1) then the minimum refresh time for all responses is returned. Otherwise the index specifies the response corresponding to the `odix` certificate in the certificate chain.

Returns: On success, the expiration time of the OCSP response. Otherwise (time_t)(-1) on error, or (time_t)-2 on out of bounds.

Since: 3.6.3

Prior to GnuTLS 3.6.4, the functions [gnutls_certificate_set_ocsp_status_request_function2], page 376 [gnutls_certificate_set_ocsp_status_request_file], page 374 were provided to set OCSP responses. These functions are still functional, but cannot be used to set multiple OCSP responses as allowed by TLS1.3.

The responses can be updated periodically using the 'ocsptool' command (see also Section 4.2.7 [ocsptool Invocation], page 82).

```
ocsptool --ask --load-cert server_cert.pem --load-issuer the_issuer.pem
         --load-signer the_issuer.pem --outfile ocsp.resp
```

In order to allow multiple OCSP responses to be concatenated, GnuTLS supports PEM-encoded OCSP responses. These can be generated using 'ocsptool' with the '–no-outder' parameter.

4.2.5 Managing encrypted keys

Transferring or storing private keys in plain may not be a good idea, since any compromise is irreparable. Storing the keys in hardware security modules (see Section 5.3 [Smart cards and HSMs], page 110) could solve the storage problem but it is not always practical or efficient enough. This section describes ways to store and transfer encrypted private keys.

There are methods for key encryption, namely the PKCS #8, PKCS #12 and OpenSSL's custom encrypted private key formats. The PKCS #8 and the OpenSSL's method allow encryption of the private key, while the PKCS #12 method allows, in addition, the bundling of accompanying data into the structure. That is typically the corresponding certificate, as well as a trusted CA certificate.

High level functionality

Generic and higher level private key import functions are available, that import plain or encrypted keys and will auto-detect the encrypted key format.

`int gnutls_privkey_import_x509_raw` (*gnutls_privkey_t* `pkey`, *const gnutls_datum_t ** `data`, *gnutls_x509_crt_fmt_t* `format`, *const char ** `password`, *unsigned int* `flags`) [Function]

pkey: The private key

data: The private key data to be imported

format: The format of the private key

password: A password (optional)

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function will import the given private key to the abstract `gnutls_privkey_t` type.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

`int gnutls_x509_privkey_import2` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t* * `data`, *gnutls_x509_crt_fmt_t* `format`, *const char* * `password`, *unsigned int* `flags`) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded key.

format: One of DER or PEM

password: A password (optional)

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function will import the given DER or PEM encoded key, to the native `gnutls_x509_privkey_t` format, irrespective of the input format. The input format is auto-detected.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

If the provided key is encrypted but no password was given, then `GNUTLS_E_DECRYPTION_FAILED` is returned. Since GnuTLS 3.4.0 this function will utilize the PIN callbacks if any.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Any keys imported using those functions can be imported to a certificate credentials structure using [gnutls_certificate_set_key], page 671, or alternatively they can be directly imported using [gnutls_certificate_set_x509_key_file2], page 384.

PKCS #8 structures

PKCS #8 keys can be imported and exported as normal private keys using the functions below. An addition to the normal import functions, are a password and a flags argument. The flags can be any element of the `gnutls_pkcs_encrypt_flags_t` enumeration. Note however, that GnuTLS only supports the PKCS #5 PBES2 encryption scheme. Keys encrypted with the obsolete PBES1 scheme cannot be decrypted.

int [gnutls_x509_privkey_import_pkcs8], page 605, (gnutls_x509_privkey_t *key*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, const char * *password*, unsigned int *flags*)
int [gnutls_x509_privkey_export_pkcs8], page 598, (gnutls_x509_privkey_t *key*, gnutls_x509_crt_fmt_t *format*, const char * *password*, unsigned int *flags*, void * *output_data*, size_t * *output_data_size*)
int [gnutls_x509_privkey_export2_pkcs8], page 596, (gnutls_x509_privkey_t *key*, gnutls_x509_crt_fmt_t *format*, const char * *password*, unsigned int *flags*, gnutls_datum_t * *out*)

GNUTLS_PKCS_PLAIN
: Unencrypted private key.

GNUTLS_PKCS_PKCS12_3DES
: PKCS-12 3DES.

GNUTLS_PKCS_PKCS12_ARCFOUR
: PKCS-12 ARCFOUR.

GNUTLS_PKCS_PKCS12_RC2_40
: PKCS-12 RC2-40.

GNUTLS_PKCS_PBES2_3DES
: PBES2 3DES.

GNUTLS_PKCS_PBES2_AES_128
: PBES2 AES-128.

GNUTLS_PKCS_PBES2_AES_192
: PBES2 AES-192.

GNUTLS_PKCS_PBES2_AES_256
: PBES2 AES-256.

GNUTLS_PKCS_NULL_PASSWORD
: Some schemas distinguish between an empty and a NULL password.

GNUTLS_PKCS_PBES2_DES
: PBES2 single DES.

GNUTLS_PKCS_PBES1_DES_MD5
: PBES1 with single DES; for compatibility with openssl only.

GNUTLS_PKCS_PBES2_GOST_TC26Z
: PBES2 GOST 28147-89 CFB with TC26-Z S-box.

GNUTLS_PKCS_PBES2_GOST_CPA
: PBES2 GOST 28147-89 CFB with CryptoPro-A S-box.

GNUTLS_PKCS_PBES2_GOST_CPB
: PBES2 GOST 28147-89 CFB with CryptoPro-B S-box.

GNUTLS_PKCS_PBES2_GOST_CPC
: PBES2 GOST 28147-89 CFB with CryptoPro-C S-box.

GNUTLS_PKCS_PBES2_GOST_CPD
: PBES2 GOST 28147-89 CFB with CryptoPro-D S-box.

Figure 4.5: Encryption flags

PKCS #12 structures

A PKCS #12 structure [[PKCS12], page 752] usually contains a user's private keys and certificates. It is commonly used in browsers to export and import the user's identities. A file containing such a key can be directly imported to a certificate credentials structure by using [gnutls_certificate_set_x509_simple_pkcs12_file], page 386.

In GnuTLS the PKCS #12 structures are handled using the `gnutls_pkcs12_t` type. This is an abstract type that may hold several `gnutls_pkcs12_bag_t` types. The bag types are the holders of the actual data, which may be certificates, private keys or encrypted data. A bag of type encrypted should be decrypted in order for its data to be accessed.

To reduce the complexity in parsing the structures the simple helper function [gnutls_pkcs12_simple_parse], page 648 is provided. For more advanced uses, manual parsing of the structure is required using the functions below.

int [gnutls_pkcs12_get_bag], page 647, (gnutls_pkcs12_t *pkcs12*, int *indx*, gnutls_pkcs12_bag_t *bag*)
int [gnutls_pkcs12_verify_mac], page 649, (gnutls_pkcs12_t *pkcs12*, const char * *pass*)
int [gnutls_pkcs12_bag_decrypt], page 641, (gnutls_pkcs12_bag_t *bag*, const char * *pass*)
int [gnutls_pkcs12_bag_get_count], page 643, (gnutls_pkcs12_bag_t *bag*)

`int gnutls_pkcs12_simple_parse` (*gnutls_pkcs12_t* `p12`, *const char* * `password`, *gnutls_x509_privkey_t* * `key`, *gnutls_x509_crt_t* ** `chain`, *unsigned int* * `chain_len`, *gnutls_x509_crt_t* ** `extra_certs`, *unsigned int* * `extra_certs_len`, *gnutls_x509_crl_t* * `crl`, *unsigned int* `flags`) [Function]

p12: A pkcs12 type

password: optional password used to decrypt the structure, bags and keys.

key: a structure to store the parsed private key.

chain: the corresponding to key certificate chain (may be `NULL`)

chain_len: will be updated with the number of additional (may be `NULL`)

extra_certs: optional pointer to receive an array of additional certificates found in the PKCS12 structure (may be `NULL`).

extra_certs_len: will be updated with the number of additional certs (may be `NULL`).

crl: an optional structure to store the parsed CRL (may be `NULL`).

flags: should be zero or one of GNUTLS_PKCS12_SP_*

This function parses a PKCS12 structure in `pkcs12` and extracts the private key, the corresponding certificate chain, any additional certificates and a CRL. The structures in `key` , `chain` `crl` , and `extra_certs` must not be initialized.

The `extra_certs` and `extra_certs_len` parameters are optional and both may be set to `NULL` . If either is non-`NULL` , then both must be set. The value for `extra_certs` is allocated using `gnutls_malloc()` .

Encrypted PKCS12 bags and PKCS8 private keys are supported, but only with password based security and the same password for all operations.

Note that a PKCS12 structure may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. For this reason this function is useful for PKCS12 files that contain only one key/certificate pair and/or one CRL.

If the provided structure has encrypted fields but no password is provided then this function returns `GNUTLS_E_DECRYPTION_FAILED` .

Note that normally the chain constructed does not include self signed certificates, to comply with TLS' requirements. If, however, the flag `GNUTLS_PKCS12_SP_INCLUDE_SELF_SIGNED` is specified then self signed certificates will be included in the chain.

Prior to using this function the PKCS `12` structure integrity must be verified using `gnutls_pkcs12_verify_mac()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

int [gnutls_pkcs12_bag_get_data], page 643, (gnutls_pkcs12_bag_t *bag*, unsigned *indx*, gnutls_datum_t * *data*)
int [gnutls_pkcs12_bag_get_key_id], page 643, (gnutls_pkcs12_bag_t *bag*, unsigned *indx*, gnutls_datum_t * *id*)
int [gnutls_pkcs12_bag_get_friendly_name], page 643, (gnutls_pkcs12_bag_t *bag*, unsigned *indx*, char ** *name*)

The functions below are used to generate a PKCS #12 structure. An example of their usage is shown at Section 7.5.4 [PKCS12 structure generation example], page 276.

int [gnutls_pkcs12_set_bag], page 648, (gnutls_pkcs12_t *pkcs12*, gnutls_pkcs12_bag_t *bag*)
int [gnutls_pkcs12_bag_encrypt], page 642, (gnutls_pkcs12_bag_t *bag*, const char * *pass*, unsigned int *flags*)
int [gnutls_pkcs12_generate_mac], page 646, (gnutls_pkcs12_t *pkcs12*, const char * *pass*)

int [gnutls_pkcs12_bag_set_data], page 644, (gnutls_pkcs12_bag_t *bag*, gnutls_pkcs12_bag_type_t *type*, const gnutls_datum_t * *data*)
int [gnutls_pkcs12_bag_set_crl], page 644, (gnutls_pkcs12_bag_t *bag*, gnutls_x509_crl_t *crl*)
int [gnutls_pkcs12_bag_set_crt], page 644, (gnutls_pkcs12_bag_t *bag*, gnutls_x509_crt_t *crt*)
int [gnutls_pkcs12_bag_set_key_id], page 645, (gnutls_pkcs12_bag_t *bag*, unsigned *indx*, const gnutls_datum_t * *id*)
int [gnutls_pkcs12_bag_set_friendly_name], page 645, (gnutls_pkcs12_bag_t *bag*, unsigned *indx*, const char * *name*)

OpenSSL encrypted keys

Unfortunately the structures discussed in the previous sections are not the only structures that may hold an encrypted private key. For example the OpenSSL library offers a custom key encryption method. Those structures are also supported in GnuTLS with [gnutls_x509_privkey_import_openssl], page 604.

`int gnutls_x509_privkey_import_openssl` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * * `data`, *const char * * `password`) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded key.

password: the password to decrypt the key (if it is encrypted).

This function will convert the given PEM encrypted to the native gnutls_x509_privkey_t format. The output will be stored in `key` .

The `password` should be in ASCII. If the password is not provided or wrong then `GNUTLS_E_DECRYPTION_FAILED` will be returned.

If the Certificate is PEM encoded it should have a header of "PRIVATE KEY" and the "DEK-Info" header.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

4.2.6 Invoking certtool

Tool to parse and generate X.509 certificates, requests and private keys. It can be used interactively or non interactively by specifying the template command line option.

The tool accepts files or supported URIs via the –infile option. In case PIN is required for URI access you can provide it using the environment variables GNUTLS_PIN and GNUTLS_SO_PIN.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `certtool` program. This software is released under the GNU General Public License, version 3 or later.

certtool help/usage (`--help`)

This is the automatically generated usage text for certtool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
certtool - GnuTLS certificate tool
Usage:  certtool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file

Certificate related options:
```

```
   -i, --certificate-info     Print information on the given certificate
       --pubkey-info          Print information on a public key
   -s, --generate-self-signed  Generate a self-signed certificate
   -c, --generate-certificate  Generate a signed certificate
       --generate-proxy       Generates a proxy certificate
   -u, --update-certificate   Update a signed certificate
       --fingerprint          Print the fingerprint of the given certificate
       --key-id               Print the key ID of the given certificate
       --v1                   Generate an X.509 version 1 certificate (with no extensions)
       --sign-params=str      Sign a certificate with a specific signature algorithm

Certificate request related options:

       --crq-info             Print information on the given certificate request
   -q, --generate-request     Generate a PKCS #10 certificate request
                                - prohibits the option 'infile'
       --no-crq-extensions    Do not use extensions in certificate requests

PKCS#12 file related options:

       --p12-info             Print information on a PKCS #12 structure
       --p12-name=str         The PKCS #12 friendly name to use
       --to-p12               Generate a PKCS #12 structure

Private key related options:

   -k, --key-info             Print information on a private key
       --p8-info              Print information on a PKCS #8 structure
       --to-rsa               Convert an RSA-PSS key to raw RSA format
   -p, --generate-privkey     Generate a private key
       --key-type=str         Specify the key type to use on key generation
       --bits=num             Specify the number of bits for key generation
       --curve=str            Specify the curve used for EC key generation
       --sec-param=str        Specify the security level [low, legacy, medium, high, ultra]
       --to-p8                Convert a given key to a PKCS #8 structure
   -8, --pkcs8                Use PKCS #8 format for private keys
       --provable             Generate a private key or parameters from a seed using a prov
       --verify-provable-privkey  Verify a private key generated from a seed using a provab
       --seed=str             When generating a private key use the given hex-encoded seed

CRL related options:

   -l, --crl-info             Print information on the given CRL structure
       --generate-crl         Generate a CRL
       --verify-crl           Verify a Certificate Revocation List using a trusted list
                                - requires the option 'load-ca-certificate'
```

```
Certificate verification related options:

   -e, --verify-chain         Verify a PEM encoded certificate chain
       --verify               Verify a PEM encoded certificate (chain) against a trusted se
       --verify-hostname=str  Specify a hostname to be used for certificate chain verificat
       --verify-email=str     Specify a email to be used for certificate chain verification
                                - prohibits the option 'verify-hostname'
       --verify-purpose=str   Specify a purpose OID to be used for certificate chain verifi
       --verify-allow-broken  Allow broken algorithms, such as MD5 for verification
       --verify-profile=str   Specify a security level profile to be used for verification

PKCS#7 structure options:

       --p7-generate          Generate a PKCS #7 structure
       --p7-sign              Signs using a PKCS #7 structure
       --p7-detached-sign     Signs using a detached PKCS #7 structure
       --p7-include-cert      The signer's certificate will be included in the cert list.
                                - disabled as '--no-p7-include-cert'
                                - enabled by default
       --p7-time              Will include a timestamp in the PKCS #7 structure
                                - disabled as '--no-p7-time'
       --p7-show-data         Will show the embedded data in the PKCS #7 structure
                                - disabled as '--no-p7-show-data'
       --p7-info              Print information on a PKCS #7 structure
       --p7-verify            Verify the provided PKCS #7 structure
       --smime-to-p7          Convert S/MIME to PKCS #7 structure

Other options:

       --get-dh-params        List the included PKCS #3 encoded Diffie-Hellman parameters
       --dh-info              Print information PKCS #3 encoded Diffie-Hellman parameters
       --load-privkey=str     Loads a private key file
       --load-pubkey=str      Loads a public key file
       --load-request=str     Loads a certificate request file
       --load-certificate=str Loads a certificate file
       --load-ca-privkey=str  Loads the certificate authority's private key file
       --load-ca-certificate=str Loads the certificate authority's certificate file
       --load-crl=str         Loads the provided CRL
       --load-data=str        Loads auxiliary data
       --password=str         Password to use
       --null-password        Enforce a NULL password
       --empty-password       Enforce an empty password
       --hex-numbers          Print big number in an easier format to parse
       --cprint               In certain operations it prints the information in C-friendly
       --hash=str             Hash algorithm to use for signing
       --salt-size=num        Specify the RSA-PSS key default salt size
```

```
      --inder                Use DER format for input certificates, private keys, and DH p
                               - disabled as '--no-inder'
      --inraw                an alias for the 'inder' option
      --outder               Use DER format for output certificates, private keys, and DH
                               - disabled as '--no-outder'
      --outraw               an alias for the 'outder' option
      --template=str         Template file to use for non-interactive operation
      --stdout-info          Print information to stdout instead of stderr
      --ask-pass             Enable interaction for entering password when in batch mode.
      --pkcs-cipher=str      Cipher to use for PKCS #8 and #12 operations
      --provider=str         Specify the PKCS #11 provider library
      --text                 Output textual information before PEM-encoded certificates, p
keys, etc
                               - disabled as '--no-text'
                               - enabled by default

Version, usage and configuration options:

   -v, --version[=arg]       output version information and exit
   -h, --help                display extended usage information and exit
   -!, --more-help           extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Tool to parse and generate X.509 certificates, requests and private keys.
It can be used interactively or non interactively by specifying the
template command line option.

The tool accepts files or supported URIs via the --infile option.  In case
PIN is required for URI access you can provide it using the environment
variables GNUTLS_PIN and GNUTLS_SO_PIN.
```

Base options

debug option (-d).

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

cert-options options

Certificate related options.

pubkey-info option.

This is the "print information on a public key" option. The option combined with –load-request, –load-pubkey, –load-privkey and –load-certificate will extract the public key of the object in question.

fingerprint option.

This is the "print the fingerprint of the given certificate" option. This is a simple hash of the DER encoding of the certificate. It can be combined with the –hash parameter. However, it is recommended for identification to use the key-id which depends only on the certificate's key.

key-id option.

This is the "print the key id of the given certificate" option. This is a hash of the public key of the given certificate. It identifies the key uniquely, remains the same on a certificate renewal and depends only on signed fields of the certificate.

certificate-pubkey option.

This is the "print certificate's public key" option. This option is deprecated as a duplicate of –pubkey-info

NOTE: THIS OPTION IS DEPRECATED

sign-params option.

This is the "sign a certificate with a specific signature algorithm" option. This option takes a string argument. This option can be combined with –generate-certificate, to sign the certificate with a specific signature algorithm variant. The only option supported is 'RSA-PSS', and should be specified when the signer does not have a certificate which is marked for RSA-PSS use only.

crq-options options

Certificate request related options.

generate-request option (-q).

This is the "generate a pkcs #10 certificate request" option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: infile.

Will generate a PKCS #10 certificate request. To specify a private key use –load-privkey.

pkcs12-options options

PKCS#12 file related options.

p12-info option.

This is the “print information on a pkcs #12 structure” option. This option will dump the contents and print the metadata of the provided PKCS #12 structure.

p12-name option.

This is the “the pkcs #12 friendly name to use” option. This option takes a string argument. The name to be used for the primary certificate and private key in a PKCS #12 file.

to-p12 option.

This is the “generate a pkcs #12 structure” option. It requires a certificate, a private key and possibly a CA certificate to be specified.

key-options options

Private key related options.

p8-info option.

This is the “print information on a pkcs #8 structure” option. This option will print information about encrypted PKCS #8 structures. That option does not require the decryption of the structure.

to-rsa option.

This is the “convert an rsa-pss key to raw rsa format” option. It requires an RSA-PSS key as input and will output a raw RSA key. This command is necessary for compatibility with applications that cannot read RSA-PSS keys.

generate-privkey option (-p).

This is the “generate a private key” option. When generating RSA-PSS private keys, the –hash option will restrict the allowed hash for the key; in the same keys the –salt-size option is also acceptable.

key-type option.

This is the “specify the key type to use on key generation” option. This option takes a string argument. This option can be combined with –generate-privkey, to specify the key type to be generated. Valid options are, ’rsa’, ’rsa-pss’, ’dsa’, ’ecdsa’, ’ed25519, and ’ed448’.’. When combined with certificate generation it can be used to specify an RSA-PSS certificate when an RSA key is given.

curve option.

This is the “specify the curve used for ec key generation” option. This option takes a string argument. Supported values are secp192r1, secp224r1, secp256r1, secp384r1 and secp521r1.

sec-param option.

This is the "specify the security level [low, legacy, medium, high, ultra]" option. This option takes a string argument `Security parameter`. This is alternative to the bits option.

to-p8 option.

This is the "convert a given key to a pkcs #8 structure" option. This needs to be combined with –load-privkey.

provable option.

This is the "generate a private key or parameters from a seed using a provable method" option. This will use the FIPS PUB186-4 algorithms (i.e., Shawe-Taylor) for provable key generation. When specified the private keys or parameters will be generated from a seed, and can be later validated with –verify-provable-privkey to be correctly generated from the seed. You may specify –seed or allow GnuTLS to generate one (recommended). This option can be combined with –generate-privkey or –generate-dh-params.

That option applies to RSA and DSA keys. On the DSA keys the PQG parameters are generated using the seed, and on RSA the two primes.

verify-provable-privkey option.

This is the "verify a private key generated from a seed using a provable method" option. This will use the FIPS-186-4 algorithms for provable key generation. You may specify –seed or use the seed stored in the private key structure.

seed option.

This is the "when generating a private key use the given hex-encoded seed" option. This option takes a string argument. The seed acts as a security parameter for the private key, and thus a seed size which corresponds to the security level of the private key should be provided (e.g., 256-bits seed).

crl-options options

CRL related options.

generate-crl option.

This is the "generate a crl" option. This option generates a Certificate Revocation List. When combined with –load-crl it would use the loaded CRL as base for the generated (i.e., all revoked certificates in the base will be copied to the new CRL). To add new certificates to the CRL use –load-certificate.

verify-crl option.

This is the "verify a certificate revocation list using a trusted list" option.
This option has some usage constraints. It:

- must appear in combination with the following options: load-ca-certificate.

The trusted certificate list must be loaded with –load-ca-certificate.

cert-verify-options options

Certificate verification related options.

verify-chain option (-e).

This is the "verify a pem encoded certificate chain" option. Verifies the validity of a certificate chain. That is, an ordered set of certificates where each one is the issuer of the previous, and the first is the end-certificate to be validated. In a proper chain the last certificate is a self signed one. It can be combined with –verify-purpose or –verify-hostname.

verify option.

This is the "verify a pem encoded certificate (chain) against a trusted set" option. The trusted certificate list can be loaded with –load-ca-certificate. If no certificate list is provided, then the system's trusted certificate list is used. Note that during verification multiple paths may be explored. On a successful verification the successful path will be the last one. It can be combined with –verify-purpose or –verify-hostname.

verify-hostname option.

This is the "specify a hostname to be used for certificate chain verification" option. This option takes a string argument. This is to be combined with one of the verify certificate options.

verify-email option.

This is the "specify a email to be used for certificate chain verification" option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: verify-hostname.

This is to be combined with one of the verify certificate options.

verify-purpose option.

This is the "specify a purpose oid to be used for certificate chain verification" option. This option takes a string argument. This object identifier restricts the purpose of the certificates to be verified. Example purposes are 1.3.6.1.5.5.7.3.1 (TLS WWW), 1.3.6.1.5.5.7.3.4 (EMAIL) etc. Note that a CA certificate without a purpose set (extended key usage) is valid for any purpose.

verify-allow-broken option.

This is the "allow broken algorithms, such as md5 for verification" option. This can be combined with –p7-verify, –verify or –verify-chain.

verify-profile option.

This is the "specify a security level profile to be used for verification" option. This option takes a string argument. This option can be used to specify a certificate verification profile.

Certificate verification profiles correspond to the security level. This should be one of 'none', 'very weak', 'low', 'legacy', 'medium', 'high', 'ultra', 'future'. Note that by default no profile is applied, unless one is set as minimum in the gnutls configuration file.

pkcs7-options options

PKCS#7 structure options.

p7-generate option.

This is the "generate a pkcs #7 structure" option. This option generates a PKCS #7 certificate container structure. To add certificates in the structure use –load-certificate and –load-crl.

p7-sign option.

This is the "signs using a pkcs #7 structure" option. This option generates a PKCS #7 structure containing a signature for the provided data from infile. The data are stored within the structure. The signer certificate has to be specified using –load-certificate and –load-privkey. The input to –load-certificate can be a list of certificates. In case of a list, the first certificate is used for signing and the other certificates are included in the structure.

p7-detached-sign option.

This is the "signs using a detached pkcs #7 structure" option. This option generates a PKCS #7 structure containing a signature for the provided data from infile. The signer certificate has to be specified using –load-certificate and –load-privkey. The input to –load-certificate can be a list of certificates. In case of a list, the first certificate is used for signing and the other certificates are included in the structure.

p7-include-cert option.

This is the "the signer's certificate will be included in the cert list." option.

This option has some usage constraints. It:

- can be disabled with –no-p7-include-cert.
- It is enabled by default.

This options works with –p7-sign or –p7-detached-sign and will include or exclude the signer's certificate into the generated signature.

p7-time option.

This is the "will include a timestamp in the pkcs #7 structure" option.

This option has some usage constraints. It:

- can be disabled with –no-p7-time.

This option will include a timestamp in the generated signature

p7-show-data option.

This is the “will show the embedded data in the pkcs #7 structure” option.

This option has some usage constraints. It:

- can be disabled with –no-p7-show-data.

This option can be combined with –p7-verify or –p7-info and will display the embedded signed data in the PKCS #7 structure.

p7-verify option.

This is the “verify the provided pkcs #7 structure” option. This option verifies the signed PKCS #7 structure. The certificate list to use for verification can be specified with –load-ca-certificate. When no certificate list is provided, then the system’s certificate list is used. Alternatively a direct signer can be provided using –load-certificate. A key purpose can be enforced with the –verify-purpose option, and the –load-data option will utilize detached data.

other-options options

Other options.

generate-dh-params option.

This is the “generate pkcs #3 encoded diffie-hellman parameters” option. The will generate random parameters to be used with Diffie-Hellman key exchange. The output parameters will be in PKCS #3 format. Note that it is recommended to use the –get-dh-params option instead.

NOTE: THIS OPTION IS DEPRECATED

get-dh-params option.

This is the “list the included pkcs #3 encoded diffie-hellman parameters” option. Returns stored DH parameters in GnuTLS. Those parameters returned are defined in RFC7919, and can be considered standard parameters for a TLS key exchange. This option is provided for old applications which require DH parameters to be specified; modern GnuTLS applications should not require them.

load-privkey option.

This is the “loads a private key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

load-pubkey option.

This is the “loads a public key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

load-request option.

This is the “loads a certificate request file” option. This option takes a string argument. This option can be used with a file

load-certificate option.

This is the “loads a certificate file” option. This option takes a string argument. This option can be used with a file

load-ca-privkey option.

This is the “loads the certificate authority’s private key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

load-ca-certificate option.

This is the “loads the certificate authority’s certificate file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

load-crl option.

This is the “loads the provided crl” option. This option takes a string argument. This option can be used with a file

load-data option.

This is the “loads auxiliary data” option. This option takes a string argument. This option can be used with a file

password option.

This is the “password to use” option. This option takes a string argument. You can use this option to specify the password in the command line instead of reading it from the tty. Note, that the command line arguments are available for view in others in the system. Specifying password as ” is the same as specifying no password.

null-password option.

This is the “enforce a null password” option. This option enforces a NULL password. This is different than the empty or no password in schemas like PKCS #8.

empty-password option.

This is the “enforce an empty password” option. This option enforces an empty password. This is different than the NULL or no password in schemas like PKCS #8.

cprint option.

This is the “in certain operations it prints the information in c-friendly format” option. In certain operations it prints the information in C-friendly format, suitable for including into C programs.

rsa option.

This is the “generate rsa key” option. When combined with –generate-privkey generates an RSA private key.

NOTE: THIS OPTION IS DEPRECATED

dsa option.

This is the “generate dsa key” option. When combined with –generate-privkey generates a DSA private key.

NOTE: THIS OPTION IS DEPRECATED

ecc option.

This is the “generate ecc (ecdsa) key” option. When combined with –generate-privkey generates an elliptic curve private key to be used with ECDSA.

NOTE: THIS OPTION IS DEPRECATED

ecdsa option.

This is an alias for the `ecc` option, see [certtool ecc], page 72.

hash option.

This is the “hash algorithm to use for signing” option. This option takes a string argument. Available hash functions are SHA1, RMD160, SHA256, SHA384, SHA512, SHA3-224, SHA3-256, SHA3-384, SHA3-512.

salt-size option.

This is the “specify the rsa-pss key default salt size” option. This option takes a number argument. Typical keys shouldn’t set or restrict this option.

inder option.

This is the “use der format for input certificates, private keys, and dh parameters ” option. This option has some usage constraints. It:

- can be disabled with –no-inder.

The input files will be assumed to be in DER or RAW format. Unlike options that in PEM input would allow multiple input data (e.g. multiple certificates), when reading in DER format a single data structure is read.

inraw option.

This is an alias for the `inder` option, see [certtool inder], page 72.

outder option.

This is the “use der format for output certificates, private keys, and dh parameters” option. This option has some usage constraints. It:

- can be disabled with –no-outder.

The output will be in DER or RAW format.

outraw option.

This is an alias for the `outder` option, see [certtool outder], page 72.

ask-pass option.

This is the "enable interaction for entering password when in batch mode." option. This option will enable interaction to enter password when in batch mode. That is useful when the template option has been specified.

pkcs-cipher option.

This is the "cipher to use for pkcs #8 and #12 operations" option. This option takes a string argument `Cipher`. Cipher may be one of 3des, 3des-pkcs12, aes-128, aes-192, aes-256, rc2-40, arcfour.

provider option.

This is the "specify the pkcs #11 provider library" option. This option takes a string argument. This will override the default options in /etc/gnutls/pkcs11.conf

text option.

This is the "output textual information before pem-encoded certificates, private keys, etc" option.

This option has some usage constraints. It:

- can be disabled with –no-text.
- It is enabled by default.

Output textual information before PEM-encoded data

certtool exit status

One of the following exit values will be returned:

'`0 (EXIT_SUCCESS)`'
: Successful program execution.

'`1 (EXIT_FAILURE)`'
: The operation failed or the command syntax was not valid.

certtool See Also

p11tool (1), psktool (1), srptool (1)

certtool Examples

Generating private keys

To create an RSA private key, run:

```
$ certtool --generate-privkey --outfile key.pem --rsa
```

To create a DSA or elliptic curves (ECDSA) private key use the above command combined with 'dsa' or 'ecc' options.

Generating certificate requests

To create a certificate request (needed when the certificate is issued by another party), run:

```
certtool --generate-request --load-privkey key.pem \
   --outfile request.pem
```

If the private key is stored in a smart card you can generate a request by specifying the private key object URL.

```
$ ./certtool --generate-request --load-privkey "pkcs11:..." \
  --load-pubkey "pkcs11:..." --outfile request.pem
```

Generating a self-signed certificate

To create a self signed certificate, use the command:

```
$ certtool --generate-privkey --outfile ca-key.pem
$ certtool --generate-self-signed --load-privkey ca-key.pem \
   --outfile ca-cert.pem
```

Note that a self-signed certificate usually belongs to a certificate authority, that signs other certificates.

Generating a certificate

To generate a certificate using the previous request, use the command:

```
$ certtool --generate-certificate --load-request request.pem \
   --outfile cert.pem --load-ca-certificate ca-cert.pem \
   --load-ca-privkey ca-key.pem
```

To generate a certificate using the private key only, use the command:

```
$ certtool --generate-certificate --load-privkey key.pem \
   --outfile cert.pem --load-ca-certificate ca-cert.pem \
   --load-ca-privkey ca-key.pem
```

Certificate information

To view the certificate information, use:

```
$ certtool --certificate-info --infile cert.pem
```

Changing the certificate format

To convert the certificate from PEM to DER format, use:

```
$ certtool --certificate-info --infile cert.pem --outder --outfile cert.der
```

PKCS #12 structure generation

To generate a PKCS #12 structure using the previous key and certificate, use the command:

```
$ certtool --load-certificate cert.pem --load-privkey key.pem \
   --to-p12 --outder --outfile key.p12
```

Some tools (reportedly web browsers) have problems with that file because it does not contain the CA certificate for the certificate. To work around that problem in the tool, you can use the –load-ca-certificate parameter as follows:

```
$ certtool --load-ca-certificate ca.pem \
```

```
--load-certificate cert.pem --load-privkey key.pem \
--to-p12 --outder --outfile key.p12
```

Obtaining Diffie-Hellman parameters

To obtain the RFC7919 parameters for Diffie-Hellman key exchange, use the command:

```
$ certtool --get-dh-params --outfile dh.pem --sec-param medium
```

Verifying a certificate

To verify a certificate in a file against the system's CA trust store use the following command:

```
$ certtool --verify --infile cert.pem
```

It is also possible to simulate hostname verification with the following options:

```
$ certtool --verify --verify-hostname www.example.com --infile cert.pem
```

Proxy certificate generation

Proxy certificate can be used to delegate your credential to a temporary, typically short-lived, certificate. To create one from the previously created certificate, first create a temporary key and then generate a proxy certificate for it, using the commands:

```
$ certtool --generate-privkey > proxy-key.pem
$ certtool --generate-proxy --load-ca-privkey key.pem \
  --load-privkey proxy-key.pem --load-certificate cert.pem \
  --outfile proxy-cert.pem
```

Certificate revocation list generation

To create an empty Certificate Revocation List (CRL) do:

```
$ certtool --generate-crl --load-ca-privkey x509-ca-key.pem \
           --load-ca-certificate x509-ca.pem
```

To create a CRL that contains some revoked certificates, place the certificates in a file and use `--load-certificate` as follows:

```
$ certtool --generate-crl --load-ca-privkey x509-ca-key.pem \
  --load-ca-certificate x509-ca.pem --load-certificate revoked-certs.pem
```

To verify a Certificate Revocation List (CRL) do:

```
$ certtool --verify-crl --load-ca-certificate x509-ca.pem < crl.pem
```

certtool Files

Certtool's template file format

A template file can be used to avoid the interactive questions of certtool. Initially create a file named 'cert.cfg' that contains the information about the certificate. The template can be used as below:

```
$ certtool --generate-certificate --load-privkey key.pem  \
   --template cert.cfg --outfile cert.pem \
   --load-ca-certificate ca-cert.pem --load-ca-privkey ca-key.pem
```

An example certtool template file that can be used to generate a certificate request or a self signed certificate follows.

```
# X.509 Certificate options
#
# DN options

# The organization of the subject.
organization = "Koko inc."

# The organizational unit of the subject.
unit = "sleeping dept."

# The locality of the subject.
# locality =

# The state of the certificate owner.
state = "Attiki"

# The country of the subject. Two letter code.
country = GR

# The common name of the certificate owner.
cn = "Cindy Lauper"

# A user id of the certificate owner.
#uid = "clauper"

# Set domain components
#dc = "name"
#dc = "domain"

# If the supported DN OIDs are not adequate you can set
# any OID here.
# For example set the X.520 Title and the X.520 Pseudonym
# by using OID and string pairs.
#dn_oid = "2.5.4.12 Dr."
#dn_oid = "2.5.4.65 jackal"

# This is deprecated and should not be used in new
# certificates.
# pkcs9_email = "none@none.org"

# An alternative way to set the certificate's distinguished name directly
# is with the "dn" option. The attribute names allowed are:
# C (country), street, O (organization), OU (unit), title, CN (common name),
# L (locality), ST (state), placeOfBirth, gender, countryOfCitizenship,
```

```
# countryOfResidence, serialNumber, telephoneNumber, surName, initials,
# generationQualifier, givenName, pseudonym, dnQualifier, postalCode, name,
# businessCategory, DC, UID, jurisdictionOfIncorporationLocalityName,
# jurisdictionOfIncorporationStateOrProvinceName,
# jurisdictionOfIncorporationCountryName, XmppAddr, and numeric OIDs.

#dn = "cn = Nikos,st = New\, Something,C=GR,surName=Mavrogiannopoulos,2.5.4.9=Arkadi

# The serial number of the certificate
# The value is in decimal (i.e. 1963) or hex (i.e. 0x07ab).
# Comment the field for a random serial number.
serial = 007

# In how many days, counting from today, this certificate will expire.
# Use -1 if there is no expiration date.
expiration_days = 700

# Alternatively you may set concrete dates and time. The GNU date string
# formats are accepted. See:
# https://www.gnu.org/software/tar/manual/html_node/Date-input-formats.html

#activation_date = "2004-02-29 16:21:42"
#expiration_date = "2025-02-29 16:24:41"

# X.509 v3 extensions

# A dnsname in case of a WWW server.
#dns_name = "www.none.org"
#dns_name = "www.morethanone.org"

# An othername defined by an OID and a hex encoded string
#other_name = "1.3.6.1.5.2.2 302ca00d1b0b56414e5245494e2e4f5247a11b3019a006020400000
#other_name_utf8 = "1.2.4.5.6 A UTF8 string"
#other_name_octet = "1.2.4.5.6 A string that will be encoded as ASN.1 octet string"

# Allows writing an XmppAddr Identifier
#xmpp_name = juliet@im.example.com

# Names used in PKINIT
#krb5_principal = user@REALM.COM
#krb5_principal = HTTP/user@REALM.COM

# A subject alternative name URI
#uri = "https://www.example.com"

# An IP address in case of a server.
#ip_address = "192.168.1.1"
```

```
# An email in case of a person
email = "none@none.org"

# TLS feature (rfc7633) extension. That can is used to indicate mandatory TLS
# extension features to be provided by the server. In practice this is used
# to require the Status Request (extid: 5) extension from the server. That is,
# to require the server holding this certificate to provide a stapled OCSP response.
# You can have multiple lines for multiple TLS features.

# To ask for OCSP status request use:
#tls_feature = 5

# Challenge password used in certificate requests
challenge_password = 123456

# Password when encrypting a private key
#password = secret

# An URL that has CRLs (certificate revocation lists)
# available. Needed in CA certificates.
#crl_dist_points = "https://www.getcrl.crl/getcrl/"

# Whether this is a CA certificate or not
#ca

# Subject Unique ID (in hex)
#subject_unique_id = 00153224

# Issuer Unique ID (in hex)
#issuer_unique_id = 00153225

#### Key usage

# The following key usage flags are used by CAs and end certificates

# Whether this certificate will be used to sign data (needed
# in TLS DHE ciphersuites). This is the digitalSignature flag
# in RFC5280 terminology.
signing_key

# Whether this certificate will be used to encrypt data (needed
# in TLS RSA ciphersuites). Note that it is preferred to use different
# keys for encryption and signing. This is the keyEncipherment flag
# in RFC5280 terminology.
encryption_key
```

```
# Whether this key will be used to sign other certificates. The
# keyCertSign flag in RFC5280 terminology.
#cert_signing_key

# Whether this key will be used to sign CRLs. The
# cRLSign flag in RFC5280 terminology.
#crl_signing_key

# The keyAgreement flag of RFC5280. It's purpose is loosely
# defined. Not use it unless required by a protocol.
#key_agreement

# The dataEncipherment flag of RFC5280. It's purpose is loosely
# defined. Not use it unless required by a protocol.
#data_encipherment

# The nonRepudiation flag of RFC5280. It's purpose is loosely
# defined. Not use it unless required by a protocol.
#non_repudiation

#### Extended key usage (key purposes)

# The following extensions are used in an end certificate
# to clarify its purpose. Some CAs also use it to indicate
# the types of certificates they are purposed to sign.


# Whether this certificate will be used for a TLS client;
# this sets the id-kp-clientAuth (1.3.6.1.5.5.7.3.2) of
# extended key usage.
#tls_www_client

# Whether this certificate will be used for a TLS server;
# this sets the id-kp-serverAuth (1.3.6.1.5.5.7.3.1) of
# extended key usage.
#tls_www_server

# Whether this key will be used to sign code. This sets the
# id-kp-codeSigning (1.3.6.1.5.5.7.3.3) of extended key usage
# extension.
#code_signing_key

# Whether this key will be used to sign OCSP data. This sets the
# id-kp-OCSPSigning (1.3.6.1.5.5.7.3.9) of extended key usage extension.
#ocsp_signing_key

# Whether this key will be used for time stamping. This sets the
```

```
# id-kp-timeStamping (1.3.6.1.5.5.7.3.8) of extended key usage extension.
#time_stamping_key

# Whether this key will be used for email protection. This sets the
# id-kp-emailProtection (1.3.6.1.5.5.7.3.4) of extended key usage extension.
#email_protection_key

# Whether this key will be used for IPsec IKE operations (1.3.6.1.5.5.7.3.17).
#ipsec_ike_key

## adding custom key purpose OIDs

# for microsoft smart card logon
# key_purpose_oid = 1.3.6.1.4.1.311.20.2.2

# for email protection
# key_purpose_oid = 1.3.6.1.5.5.7.3.4

# for any purpose (must not be used in intermediate CA certificates)
# key_purpose_oid = 2.5.29.37.0

### end of key purpose OIDs

### Adding arbitrary extensions
# This requires to provide the extension OIDs, as well as the extension data in
# hex format. The following two options are available since GnuTLS 3.5.3.
#add_extension = "1.2.3.4 0x0AAB01ACFE"

# As above but encode the data as an octet string
#add_extension = "1.2.3.4 octet_string(0x0AAB01ACFE)"

# For portability critical extensions shouldn't be set to certificates.
#add_critical_extension = "5.6.7.8 0x1AAB01ACFE"

# When generating a certificate from a certificate
# request, then honor the extensions stored in the request
# and store them in the real certificate.
#honor_crq_extensions

# Alternatively only specific extensions can be copied.
#honor_crq_ext = 2.5.29.17
#honor_crq_ext = 2.5.29.15

# Path length constraint. Sets the maximum number of
# certificates that can be used to certify this certificate.
# (i.e. the certificate chain length)
#path_len = -1
```

```
#path_len = 2

# OCSP URI
# ocsp_uri = https://my.ocsp.server/ocsp

# CA issuers URI
# ca_issuers_uri = https://my.ca.issuer

# Certificate policies
#policy1 = 1.3.6.1.4.1.5484.1.10.99.1.0
#policy1_txt = "This is a long policy to summarize"
#policy1_url = https://www.example.com/a-policy-to-read

#policy2 = 1.3.6.1.4.1.5484.1.10.99.1.1
#policy2_txt = "This is a short policy"
#policy2_url = https://www.example.com/another-policy-to-read

# The number of additional certificates that may appear in a
# path before the anyPolicy is no longer acceptable.
#inhibit_anypolicy_skip_certs 1

# Name constraints

# DNS
#nc_permit_dns = example.com
#nc_exclude_dns = test.example.com

# EMAIL
#nc_permit_email = "nmav@ex.net"

# Exclude subdomains of example.com
#nc_exclude_email = .example.com

# Exclude all e-mail addresses of example.com
#nc_exclude_email = example.com

# IP
#nc_permit_ip = 192.168.0.0/16
#nc_exclude_ip = 192.168.5.0/24
#nc_permit_ip = fc0a:eef2:e7e7:a56e::/64


# Options for proxy certificates
#proxy_policy_language = 1.3.6.1.5.5.7.21.1


# Options for generating a CRL
```

```
# The number of days the next CRL update will be due.
# next CRL update will be in 43 days
#crl_next_update = 43

# this is the 5th CRL by this CA
# The value is in decimal (i.e. 1963) or hex (i.e. 0x07ab).
# Comment the field for a time-based number.
# Time-based CRL numbers generated in GnuTLS 3.6.3 and later
# are significantly larger than those generated in previous
# versions. Since CRL numbers need to be monotonic, you need
# to specify the CRL number here manually if you intend to
# downgrade to an earlier version than 3.6.3 after publishing
# the CRL as it is not possible to specify CRL numbers greater
# than 2**63-2 using hex notation in those versions.
#crl_number = 5

# Specify the update dates more precisely.
#crl_this_update_date = "2004-02-29 16:21:42"
#crl_next_update_date = "2025-02-29 16:24:41"

# The date that the certificates will be made seen as
# being revoked.
#crl_revocation_date = "2025-02-29 16:24:41"
```

4.2.7 Invoking ocsptool

On verification

Responses are typically signed/issued by designated certificates or certificate authorities and thus this tool requires on verification the certificate of the issuer or the full certificate chain in order to determine the appropriate signing authority. The specified certificate of the issuer is assumed trusted.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `ocsptool` program. This software is released under the GNU General Public License, version 3 or later.

ocsptool help/usage (`--help`)

This is the automatically generated usage text for ocsptool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
ocsptool - GnuTLS OCSP tool
Usage:  ocsptool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...
```

```
   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --ask[=arg]            Ask an OCSP/HTTP server on a certificate validity
   -e, --verify-response      Verify response
   -i, --request-info         Print information on a OCSP request
   -j, --response-info        Print information on a OCSP response
   -q, --generate-request     Generates an OCSP request
       --nonce                Use (or not) a nonce to OCSP request
                                - disabled as '--no-nonce'
       --load-chain=file      Reads a set of certificates forming a chain from file
                                - file must pre-exist
       --load-issuer=file     Reads issuer's certificate from file
                                - file must pre-exist
       --load-cert=file       Reads the certificate to check from file
                                - file must pre-exist
       --load-trust=file      Read OCSP trust anchors from file
                                - prohibits the option 'load-signer'
                                - file must pre-exist
       --load-signer=file     Reads the OCSP response signer from file
                                - prohibits the option 'load-trust'
                                - file must pre-exist
       --inder                Use DER format for input certificates and private keys
                                - disabled as '--no-inder'
       --outder               Use DER format for output of responses (this is the default)
       --outpem               Use PEM format for output of responses
   -Q, --load-request=file    Reads the DER encoded OCSP request from file
                                - file must pre-exist
   -S, --load-response=file   Reads the DER encoded OCSP response from file
                                - file must pre-exist
       --ignore-errors        Ignore any verification errors
       --verify-allow-broken  Allow broken algorithms, such as MD5 for verification
   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

ocsptool is a program that can parse and print information about OCSP
requests/responses, generate requests and verify responses.  Unlike other
```

```
GnuTLS applications it outputs DER encoded structures by default unless the
'--outpem' option is specified.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

ask option

This is the "ask an ocsp/http server on a certificate validity" option. This option takes an optional string argument `server name|url`. Connects to the specified HTTP OCSP server and queries on the validity of the loaded certificate. Its argument can be a URL or a plain server name. It can be combined with –load-chain, where it checks all certificates in the provided chain, or with –load-cert and –load-issuer options. The latter checks the provided certificate against its specified issuer certificate.

verify-response option (-e)

This is the "verify response" option. Verifies the provided OCSP response against the system trust anchors (unless –load-trust is provided). It requires the –load-signer or –load-chain options to obtain the signer of the OCSP response.

request-info option (-i)

This is the "print information on a ocsp request" option. Display detailed information on the provided OCSP request.

response-info option (-j)

This is the "print information on a ocsp response" option. Display detailed information on the provided OCSP response.

load-trust option

This is the "read ocsp trust anchors from file" option. This option takes a file argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: load-signer.

When verifying an OCSP response read the trust anchors from the provided file. When this is not provided, the system's trust anchors will be used.

outder option

This is the "use der format for output of responses (this is the default)" option. The output will be in DER encoded format. Unlike other GnuTLS tools, this is the default for this tool

outpem option

This is the "use pem format for output of responses" option. The output will be in PEM format.

verify-allow-broken option

This is the "allow broken algorithms, such as md5 for verification" option. This can be combined with –verify-response.

ocsptool exit status

One of the following exit values will be returned:

'`0 (EXIT_SUCCESS)`'
: Successful program execution.

'`1 (EXIT_FAILURE)`'
: The operation failed or the command syntax was not valid.

ocsptool See Also

certtool (1)

ocsptool Examples

Print information about an OCSP request

To parse an OCSP request and print information about the content, the `-i` or `--request-info` parameter may be used as follows. The `-Q` parameter specify the name of the file containing the OCSP request, and it should contain the OCSP request in binary DER format.

```
$ ocsptool -i -Q ocsp-request.der
```

The input file may also be sent to standard input like this:

```
$ cat ocsp-request.der | ocsptool --request-info
```

Print information about an OCSP response

Similar to parsing OCSP requests, OCSP responses can be parsed using the `-j` or `--response-info` as follows.

```
$ ocsptool -j -Q ocsp-response.der
$ cat ocsp-response.der | ocsptool --response-info
```

Generate an OCSP request

The `-q` or `--generate-request` parameters are used to generate an OCSP request. By default the OCSP request is written to standard output in binary DER format, but can be stored in a file using `--outfile`. To generate an OCSP request the issuer of the certificate to check needs to be specified with `--load-issuer` and the certificate to check with `--load-cert`. By default PEM format is used for these files, although `--inder` can be used to specify that the input files are in DER format.

```
$ ocsptool -q --load-issuer issuer.pem --load-cert client.pem \
```

```
            --outfile ocsp-request.der
```

When generating OCSP requests, the tool will add an OCSP extension containing a nonce. This behaviour can be disabled by specifying `--no-nonce`.

Verify signature in OCSP response

To verify the signature in an OCSP response the `-e` or `--verify-response` parameter is used. The tool will read an OCSP response in DER format from standard input, or from the file specified by `--load-response`. The OCSP response is verified against a set of trust anchors, which are specified using `--load-trust`. The trust anchors are concatenated certificates in PEM format. The certificate that signed the OCSP response needs to be in the set of trust anchors, or the issuer of the signer certificate needs to be in the set of trust anchors and the OCSP Extended Key Usage bit has to be asserted in the signer certificate.

```
$ ocsptool -e --load-trust issuer.pem \
            --load-response ocsp-response.der
```

The tool will print status of verification.

Verify signature in OCSP response against given certificate

It is possible to override the normal trust logic if you know that a certain certificate is supposed to have signed the OCSP response, and you want to use it to check the signature. This is achieved using `--load-signer` instead of `--load-trust`. This will load one certificate and it will be used to verify the signature in the OCSP response. It will not check the Extended Key Usage bit.

```
$ ocsptool -e --load-signer ocsp-signer.pem \
            --load-response ocsp-response.der
```

This approach is normally only relevant in two situations. The first is when the OCSP response does not contain a copy of the signer certificate, so the `--load-trust` code would fail. The second is if you want to avoid the indirect mode where the OCSP response signer certificate is signed by a trust anchor.

Real-world example

Here is an example of how to generate an OCSP request for a certificate and to verify the response. For illustration we'll use the `blog.josefsson.org` host, which (as of writing) uses a certificate from CACert. First we'll use `gnutls-cli` to get a copy of the server certificate chain. The server is not required to send this information, but this particular one is configured to do so.

```
$ echo | gnutls-cli -p 443 blog.josefsson.org --save-cert chain.pem
```

The saved certificates normally contain a pointer to where the OCSP responder is located, in the Authority Information Access Information extension. For example, from `certtool -i < chain.pem` there is this information:

```
Authority Information Access Information (not critical):
Access Method: 1.3.6.1.5.5.7.48.1 (id-ad-ocsp)
Access Location URI: https://ocsp.CAcert.org/
```

This means that ocsptool can discover the servers to contact over HTTP. We can now request information on the chain certificates.

```
$ ocsptool --ask --load-chain chain.pem
```

The request is sent via HTTP to the OCSP server address found in the certificates. It is possible to override the address of the OCSP server as well as ask information on a particular certificate using –load-cert and –load-issuer.

```
$ ocsptool --ask https://ocsp.CAcert.org/ --load-chain chain.pem
```

4.2.8 Invoking danetool

Tool to generate and check DNS resource records for the DANE protocol.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `danetool` program. This software is released under the GNU General Public License, version 3 or later.

danetool help/usage (`--help`)

This is the automatically generated usage text for danetool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
danetool - GnuTLS DANE tool
Usage:  danetool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --load-pubkey=str      Loads a public key file
       --load-certificate=str Loads a certificate file
       --dlv=str              Sets a DLV file
       --hash=str             Hash algorithm to use for signing
       --check=str            Check a host's DANE TLSA entry
       --check-ee             Check only the end-entity's certificate
       --check-ca             Check only the CA's certificate
       --tlsa-rr              Print the DANE RR data on a certificate or public key
                                - requires the option 'host'
       --host=str             Specify the hostname to be used in the DANE RR
       --proto=str            The protocol set for DANE data (tcp, udp etc.)
       --port=str             The port or service to connect to, for DANE data
       --app-proto=str        an alias for the 'starttls-proto' option
       --starttls-proto=str   The application protocol to be used to obtain the server's ce
(https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)
       --ca                   Whether the provided certificate or public key is a Certifica
Authority
```

```
      --x509                Use the hash of the X.509 certificate, rather than the public
      --local               an alias for the 'domain' option
                              - enabled by default
      --domain              The provided certificate or public key is issued by the local
                              - disabled as '--no-domain'
                              - enabled by default
      --local-dns           Use the local DNS server for DNSSEC resolving
                              - disabled as '--no-local-dns'
      --insecure            Do not verify any DNSSEC signature
      --inder               Use DER format for input certificates and private keys
                              - disabled as '--no-inder'
      --inraw               an alias for the 'inder' option
      --print-raw           Print the received DANE data in raw format
                              - disabled as '--no-print-raw'
      --quiet               Suppress several informational messages
  -v, --version[=arg]       output version information and exit
  -h, --help                display extended usage information and exit
  -!, --more-help           extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Tool to generate and check DNS resource records for the DANE protocol.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

load-pubkey option

This is the "loads a public key file" option. This option takes a string argument. This can be either a file or a PKCS #11 URL

load-certificate option

This is the "loads a certificate file" option. This option takes a string argument. This can be either a file or a PKCS #11 URL

dlv option

This is the "sets a dlv file" option. This option takes a string argument. This sets a DLV file to be used for DNSSEC verification.

hash option

This is the "hash algorithm to use for signing" option. This option takes a string argument. Available hash functions are SHA1, RMD160, SHA256, SHA384, SHA512.

check option

This is the “check a host’s dane tlsa entry” option. This option takes a string argument. Obtains the DANE TLSA entry from the given hostname and prints information. Note that the actual certificate of the host can be provided using –load-certificate, otherwise danetool will connect to the server to obtain it. The exit code on verification success will be zero.

check-ee option

This is the “check only the end-entity’s certificate” option. Checks the end-entity’s certificate only. Trust anchors or CAs are not considered.

check-ca option

This is the “check only the ca’s certificate” option. Checks the trust anchor’s and CA’s certificate only. End-entities are not considered.

tlsa-rr option

This is the “print the dane rr data on a certificate or public key” option.
This option has some usage constraints. It:

- must appear in combination with the following options: host.

This command prints the DANE RR data needed to enable DANE on a DNS server.

host option

This is the “specify the hostname to be used in the dane rr” option. This option takes a string argument `Hostname`. This command sets the hostname for the DANE RR.

proto option

This is the “the protocol set for dane data (tcp, udp etc.)” option. This option takes a string argument `Protocol`. This command specifies the protocol for the service set in the DANE data.

app-proto option

This is an alias for the `starttls-proto` option, see [danetool starttls-proto], page 89.

starttls-proto option

This is the “the application protocol to be used to obtain the server’s certificate (https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)” option. This option takes a string argument. When the server’s certificate isn’t provided danetool will connect to the server to obtain the certificate. In that case it is required to know the protocol to talk with the server prior to initiating the TLS handshake.

ca option

This is the “whether the provided certificate or public key is a certificate authority” option. Marks the DANE RR as a CA certificate if specified.

x509 option

This is the “use the hash of the x.509 certificate, rather than the public key” option. This option forces the generated record to contain the hash of the full X.509 certificate. By default only the hash of the public key is used.

local option

This is an alias for the `domain` option, see [danetool domain], page 90.

domain option

This is the “the provided certificate or public key is issued by the local domain” option.

This option has some usage constraints. It:

- can be disabled with –no-domain.
- It is enabled by default.

DANE distinguishes certificates and public keys offered via the DNSSEC to trusted and local entities. This flag indicates that this is a domain-issued certificate, meaning that there could be no CA involved.

local-dns option

This is the “use the local dns server for dnssec resolving” option.

This option has some usage constraints. It:

- can be disabled with –no-local-dns.

This option will use the local DNS server for DNSSEC. This is disabled by default due to many servers not allowing DNSSEC.

insecure option

This is the “do not verify any dnssec signature” option. Ignores any DNSSEC signature verification results.

inder option

This is the “use der format for input certificates and private keys” option.

This option has some usage constraints. It:

- can be disabled with –no-inder.

The input files will be assumed to be in DER or RAW format. Unlike options that in PEM input would allow multiple input data (e.g. multiple certificates), when reading in DER format a single data structure is read.

inraw option

This is an alias for the `inder` option, see [danetool inder], page 90.

print-raw option

This is the "print the received dane data in raw format" option.

This option has some usage constraints. It:

- can be disabled with –no-print-raw.

This option will print the received DANE data.

quiet option

This is the "suppress several informational messages" option. In that case on the exit code can be used as an indication of verification success

danetool exit status

One of the following exit values will be returned:

'0 (EXIT_SUCCESS)'
: Successful program execution.

'1 (EXIT_FAILURE)'
: The operation failed or the command syntax was not valid.

danetool See Also

certtool (1)

danetool Examples

DANE TLSA RR generation

To create a DANE TLSA resource record for a certificate (or public key) that was issued localy and may or may not be signed by a CA use the following command.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem
```

To create a DANE TLSA resource record for a CA signed certificate, which will be marked as such use the following command.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem \
  --no-domain
```

The former is useful to add in your DNS entry even if your certificate is signed by a CA. That way even users who do not trust your CA will be able to verify your certificate using DANE.

In order to create a record for the CA signer of your certificate use the following.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem \
  --ca --no-domain
```

To read a server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443
```

To verify an HTTPS server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443 --load-certificate chain.p
```

To verify an SMTP server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --starttls-proto=smtp --load-certific
```

4.3 Shared-key and anonymous authentication

In addition to certificate authentication, the TLS protocol may be used with password, shared-key and anonymous authentication methods. The rest of this chapter discusses details of these methods.

4.3.1 PSK authentication

4.3.1.1 Authentication using PSK

Authentication using Pre-shared keys is a method to authenticate using usernames and binary keys. This protocol avoids making use of public key infrastructure and expensive calculations, thus it is suitable for constraint clients. It is available under all TLS protocol versions.

The implementation in GnuTLS is based on [[TLSPSK], page 752]. The supported PSK key exchange methods are:

`PSK:` Authentication using the PSK protocol (no forward secrecy).

`DHE-PSK:` Authentication using the PSK protocol and Diffie-Hellman key exchange. This method offers perfect forward secrecy.

`ECDHE-PSK:`
Authentication using the PSK protocol and Elliptic curve Diffie-Hellman key exchange. This method offers perfect forward secrecy.

`RSA-PSK:` Authentication using the PSK protocol for the client and an RSA certificate for the server. This is not available under TLS 1.3.

Helper functions to generate and maintain PSK keys are also included in GnuTLS.

int [gnutls_key_generate], page 422, (gnutls_datum_t * *key*, unsigned int *key_size*)
int [gnutls_hex_encode], page 420, (const gnutls_datum_t * *data*, char * *result*, size_t * *result_size*)
int [gnutls_hex_decode], page 420, (const gnutls_datum_t * *hex_data*, void * *result*, size_t * *result_size*)

4.3.1.2 Invoking psktool

Program that generates random keys for use with TLS-PSK. The keys are stored in hexadecimal format in a key file.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `psktool` program. This software is released under the GNU General Public License, version 3 or later.

psktool help/usage (`--help`)

This is the automatically generated usage text for psktool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function.

The PAGER environment variable is used to select the program, defaulting to more. Both will exit with a status code of 0.

```
psktool - GnuTLS PSK tool
Usage:  psktool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -s, --keysize=num          Specify the key size in bytes (default is 32-bytes or 256-bit
                                - it must be in the range:
                                  0 to 512
   -u, --username=str         Specify the username to use
   -p, --pskfile=str          Specify a pre-shared key file
   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Program that generates random keys for use with TLS-PSK.  The keys are
stored in hexadecimal format in a key file.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

pskfile option (-p)

This is the "specify a pre-shared key file" option. This option takes a string argument. This option will specify the pre-shared key file to store the generated keys.

passwd option

This is an alias for the pskfile option, see [psktool pskfile], page 93.

psktool exit status

One of the following exit values will be returned:

'0 (EXIT_SUCCESS)'
: Successful program execution.

'1 (EXIT_FAILURE)'
: The operation failed or the command syntax was not valid.

psktool See Also

gnutls-cli-debug (1), gnutls-serv (1), srptool (1), certtool (1)

psktool Examples

To add a user 'psk_identity' in `keys.psk` for use with GnuTLS run:

```
$ ./psktool -u psk_identity -p keys.psk
Generating a random key for user 'psk_identity'
Key stored to keys.psk
$ cat keys.psk
psk_identity:88f3824b3e5659f52d00e959bacab954b6540344
$
```

This command will create `keys.psk` if it does not exist and will add user 'psk_identity'.

4.3.2 SRP authentication

4.3.2.1 Authentication using SRP

GnuTLS supports authentication via the Secure Remote Password or SRP protocol (see [[RFC2945], page 751] for a description). The SRP key exchange is an extension to the TLS protocol, and it provides an authenticated with a password key exchange. The peers can be identified using a single password, or there can be combinations where the client is authenticated using SRP and the server using a certificate. It is only available under TLS 1.2 or earlier versions.

The advantage of SRP authentication, over other proposed secure password authentication schemes, is that SRP is not susceptible to off-line dictionary attacks. Moreover, SRP does not require the server to hold the user's password. This kind of protection is similar to the one used traditionally in the UNIX `/etc/passwd` file, where the contents of this file did not cause harm to the system security if they were revealed. The SRP needs instead of the plain password something called a verifier, which is calculated using the user's password, and if stolen cannot be used to impersonate the user.

Typical conventions in SRP are a password file, called `tpasswd` that holds the SRP verifiers (encoded passwords) and another file, `tpasswd.conf`, which holds the allowed SRP parameters. The included in GnuTLS helper follow those conventions. The srptool program, discussed in the next section is a tool to manipulate the SRP parameters.

The implementation in GnuTLS is based on [[TLSSRP], page 752]. The supported key exchange methods are shown below. Enabling any of these key exchange methods in a session disables support for TLS1.3.

`SRP:` Authentication using the SRP protocol.

`SRP_DSS:` Client authentication using the SRP protocol. Server is authenticated using a certificate with DSA parameters.

`SRP_RSA:` Client authentication using the SRP protocol. Server is authenticated using a certificate with RSA parameters.

`int gnutls_srp_verifier` (*const char * **username***, *const char * **password***, *const gnutls_datum_t * **salt***, *const gnutls_datum_t * **generator***, *const gnutls_datum_t * **prime***, *gnutls_datum_t * **res***) [Function]

username: is the user's name

password: is the user's password

salt: should be some randomly generated bytes

generator: is the generator of the group

prime: is the group's prime

res: where the verifier will be stored.

This function will create an SRP verifier, as specified in RFC2945. The `prime` and `generator` should be one of the static parameters defined in gnutls/gnutls.h or may be generated.

The verifier will be allocated with `gnutls_malloc` () and will be stored in `res` using binary format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

int [gnutls_srp_base64_encode2], page 474, (const gnutls_datum_t * *data*, gnutls_datum_t * *result*)
int [gnutls_srp_base64_decode2], page 473, (const gnutls_datum_t * *b64_data*, gnutls_datum_t * *result*)

4.3.2.2 Invoking srptool

Simple program that emulates the programs in the Stanford SRP (Secure Remote Password) libraries using GnuTLS. It is intended for use in places where you don't expect SRP authentication to be the used for system users.

In brief, to use SRP you need to create two files. These are the password file that holds the users and the verifiers associated with them and the configuration file to hold the group parameters (called tpasswd.conf).

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `srptool` program. This software is released under the GNU General Public License, version 3 or later.

srptool help/usage (`--help`)

This is the automatically generated usage text for srptool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
srptool - GnuTLS SRP tool
Usage:  srptool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -i, --index=num            specify the index of the group parameters in tpasswd.conf to
   -u, --username=str         specify a username
   -p, --passwd=str           specify a password file
   -s, --salt=num             specify salt size
       --verify               just verify the password.
```

```
   -v, --passwd-conf=str      specify a password conf file.
       --create-conf=str      Generate a password configuration file.
   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Simple program that emulates the programs in the Stanford SRP (Secure
Remote Password) libraries using GnuTLS.  It is intended for use in places
where you don't expect SRP authentication to be the used for system users.

In brief, to use SRP you need to create two files.  These are the password
file that holds the users and the verifiers associated with them and the
configuration file to hold the group parameters (called tpasswd.conf).
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

verify option

This is the "just verify the password." option. Verifies the password provided against the password file.

passwd-conf option (-v)

This is the "specify a password conf file." option. This option takes a string argument. Specify a filename or a PKCS #11 URL to read the CAs from.

create-conf option

This is the "generate a password configuration file." option. This option takes a string argument. This generates a password configuration file (tpasswd.conf) containing the required for TLS parameters.

srptool exit status

One of the following exit values will be returned:

'0 (EXIT_SUCCESS)'
: Successful program execution.

'1 (EXIT_FAILURE)'
: The operation failed or the command syntax was not valid.

srptool See Also

gnutls-cli-debug (1), gnutls-serv (1), srptool (1), psktool (1), certtool (1)

srptool Examples

To create `tpasswd.conf` which holds the g and n values for SRP protocol (generator and a large prime), run:

```
$ srptool --create-conf /etc/tpasswd.conf
```

This command will create `/etc/tpasswd` and will add user 'test' (you will also be prompted for a password). Verifiers are stored by default in the way libsrp expects.

```
$ srptool --passwd /etc/tpasswd --passwd-conf /etc/tpasswd.conf -u test
```

This command will check against a password. If the password matches the one in `/etc/tpasswd` you will get an ok.

```
$ srptool --passwd /etc/tpasswd --passwd\-conf /etc/tpasswd.conf --verify -u test
```

4.3.3 Anonymous authentication

The anonymous key exchange offers encryption without any indication of the peer's identity. This kind of authentication is vulnerable to a man in the middle attack, but can be used even if there is no prior communication or shared trusted parties with the peer. It is useful to establish a session over which certificate authentication will occur in order to hide the indentities of the participants from passive eavesdroppers. It is only available under TLS 1.2 or earlier versions.

Unless in the above case, it is not recommended to use anonymous authentication. In the cases where there is no prior communication with the peers, an alternative with better properties, such as key continuity, is trust on first use (see Section 4.1.4.1 [Verifying a certificate using trust on first use authentication], page 40).

The available key exchange algorithms for anonymous authentication are shown below, but note that few public servers support them, and they have to be explicitly enabled. These ciphersuites are negotiated only under TLS 1.2.

`ANON_DH:` This algorithm exchanges Diffie-Hellman parameters.

`ANON_ECDH:`
This algorithm exchanges elliptic curve Diffie-Hellman parameters. It is more efficient than ANON_DH on equivalent security levels.

4.4 Selecting an appropriate authentication method

This section provides some guidance on how to use the available authentication methods in GnuTLS in various scenarios.

4.4.1 Two peers with an out-of-band channel

Let's consider two peers who need to communicate over an untrusted channel (the Internet), but have an out-of-band channel available. The latter channel is considered safe from eavesdropping and message modification and thus can be used for an initial bootstrapping of the protocol. The options available are:

- Pre-shared keys (see Section 4.3.1 [PSK authentication], page 92). The server and a client communicate a shared randomly generated key over the trusted channel and use it to negotiate further sessions over the untrusted channel.

- Passwords (see Section 4.3.2 [SRP authentication], page 94). The client communicates to the server its username and password of choice and uses it to negotiate further sessions over the untrusted channel.
- Public keys (see Section 4.1 [Certificate authentication], page 20). The client and the server exchange their public keys (or fingerprints of them) over the trusted channel. On future sessions over the untrusted channel they verify the key being the same (similar to Section 4.1.4.1 [Verifying a certificate using trust on first use authentication], page 40).

Provided that the out-of-band channel is trusted all of the above provide a similar level of protection. An out-of-band channel may be the initial bootstrapping of a user's PC in a corporate environment, in-person communication, communication over an alternative network (e.g. the phone network), etc.

4.4.2 Two peers without an out-of-band channel

When an out-of-band channel is not available a peer cannot be reliably authenticated. What can be done, however, is to allow some form of registration of users connecting for the first time and ensure that their keys remain the same after that initial connection. This is termed key continuity or trust on first use (TOFU).

The available option is to use public key authentication (see Section 4.1 [Certificate authentication], page 20). The client and the server store each other's public keys (or fingerprints of them) and associate them with their identity. On future sessions over the untrusted channel they verify the keys being the same (see Section 4.1.4.1 [Verifying a certificate using trust on first use authentication], page 40).

To mitigate the uncertainty of the information exchanged in the first connection other channels over the Internet may be used, e.g., DNSSEC (see Section 4.1.4.2 [Verifying a certificate using DANE], page 41).

4.4.3 Two peers and a trusted third party

When a trusted third party is available (or a certificate authority) the most suitable option is to use certificate authentication (see Section 4.1 [Certificate authentication], page 20). The client and the server obtain certificates that associate their identity and public keys using a digital signature by the trusted party and use them to on the subsequent communications with each other. Each party verifies the peer's certificate using the trusted third party's signature. The parameters of the third party's signature are present in its certificate which must be available to all communicating parties.

While the above is the typical authentication method for servers in the Internet by using the commercial CAs, the users that act as clients in the protocol rarely possess such certificates. In that case a hybrid method can be used where the server is authenticated by the client using the commercial CAs and the client is authenticated based on some information the client provided over the initial server-authenticated channel. The available options are:

- Passwords (see Section 4.3.2 [SRP authentication], page 94). The client communicates to the server its username and password of choice on the initial server-authenticated connection and uses it to negotiate further sessions. This is possible because the SRP protocol allows for the server to be authenticated using a certificate and the client using the password.

- Public keys (see Section 4.1 [Certificate authentication], page 20). The client sends its public key to the server (or a fingerprint of it) over the initial server-authenticated connection. On future sessions the client verifies the server using the third party certificate and the server verifies that the client's public key remained the same (see Section 4.1.4.1 [Verifying a certificate using trust on first use authentication], page 40).

5 Abstract key types and Hardware security modules

In several cases storing the long term cryptographic keys in a hard disk or even in memory poses a significant risk. Once the system they are stored is compromised the keys must be replaced as the secrecy of future sessions is no longer guaranteed. Moreover, past sessions that were not protected by a perfect forward secrecy offering ciphersuite are also to be assumed compromised.

If such threats need to be addressed, then it may be wise storing the keys in a security module such as a smart card, an HSM or the TPM chip. Those modules ensure the protection of the cryptographic keys by only allowing operations on them and preventing their extraction. The purpose of the abstract key API is to provide an API that will allow the handle of keys in memory and files, as well as keys stored in such modules.

In GnuTLS the approach is to handle all keys transparently by the high level API, e.g., the API that loads a key or certificate from a file. The high-level API will accept URIs in addition to files that specify keys on an HSM or in TPM, and a callback function will be used to obtain any required keys. The URI format is defined in [[PKCS11URI], page 753].

More information on the API is provided in the next sections. Examples of a URI of a certificate stored in an HSM, as well as a key stored in the TPM chip are shown below. To discover the URIs of the objects the `p11tool` (see Section 5.3.9 [p11tool Invocation], page 119).

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315; \
manufacturer=EnterSafe;object=test1;type=cert
```

5.1 Abstract key types

Since there are many forms of a public or private keys supported by GnuTLS such as X.509, PKCS #11 or TPM it is desirable to allow common operations on them. For these reasons the abstract `gnutls_privkey_t` and `gnutls_pubkey_t` were introduced in `gnutls/abstract.h` header. Those types are initialized using a specific type of key and then can be used to perform operations in an abstract way. For example in order to sign an X.509 certificate with a key that resides in a token the following steps can be used.

```
#include <gnutls/abstract.h>

void sign_cert( gnutls_x509_crt_t to_be_signed)
{
gnutls_x509_crt_t ca_cert;
gnutls_privkey_t abs_key;

  /* initialize the abstract key */
  gnutls_privkey_init(&abs_key);

  /* keys stored in tokens are identified by URLs */
  gnutls_privkey_import_url(abs_key, key_url);

  gnutls_x509_crt_init(&ca_cert);
```

```
  gnutls_x509_crt_import_url(&ca_cert, cert_url);

  /* sign the certificate to be signed */
  gnutls_x509_crt_privkey_sign(to_be_signed, ca_cert, abs_key,
                               GNUTLS_DIG_SHA256, 0);
}
```

5.1.1 Public keys

An abstract `gnutls_pubkey_t` can be initialized and freed by using the functions below.

int [gnutls_pubkey_init], page 709, (gnutls_pubkey_t * *key*)
void [gnutls_pubkey_deinit], page 697, (gnutls_pubkey_t *key*)

After initialization its values can be imported from an existing structure like `gnutls_x509_crt_t`, or through an ASN.1 encoding of the X.509 `SubjectPublicKeyInfo` sequence.

int [gnutls_pubkey_import_x509], page 708, (gnutls_pubkey_t *key*, gnutls_x509_crt_t *crt*, unsigned int *flags*)
int [gnutls_pubkey_import_pkcs11], page 706, (gnutls_pubkey_t *key*, gnutls_pkcs11_obj_t *obj*, unsigned int *flags*)
int [gnutls_pubkey_import_url], page 708, (gnutls_pubkey_t *key*, const char * *url*, unsigned int *flags*)
int [gnutls_pubkey_import_privkey], page 706, (gnutls_pubkey_t *key*, gnutls_privkey_t *pkey*, unsigned int *usage*, unsigned int *flags*)
int [gnutls_pubkey_import], page 703, (gnutls_pubkey_t *key*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*)
int [gnutls_pubkey_export], page 698, (gnutls_pubkey_t *key*, gnutls_x509_crt_fmt_t *format*, void * *output_data*, size_t * *output_data_size*)

int gnutls_pubkey_export2 (*gnutls_pubkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t* * `out`) [Function]

key: Holds the certificate

format: the format of output params. One of PEM or DER.

out: will contain a certificate PEM or DER encoded

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

The output buffer will be allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 3.1.3

Other helper functions that allow directly importing from raw X.509 structures are shown below.

int [gnutls_pubkey_import_x509_raw], page 708, (gnutls_pubkey_t *pkey*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, unsigned int *flags*)

An important function is [gnutls_pubkey_import_url], page 708 which will import public keys from URLs that identify objects stored in tokens (see Section 5.3 [Smart cards and

HSMs], page 110, and Section 5.4 [Trusted Platform Module], page 130). A function to check for a supported by GnuTLS URL is [gnutls_url_is_supported], page 488.

`unsigned gnutls_url_is_supported (`*const char * url*`)` [Function]

url: A URI to be tested

Check whether the provided `url` is supported. Depending on the system libraries GnuTLS may support pkcs11, tpmkey or other URLs.

Returns: return non-zero if the given URL is supported, and zero if it is not known.

Since: 3.1.0

Additional functions are available that will return information over a public key, such as a unique key ID, as well as a function that given a public key fingerprint would provide a memorable sketch.

Note that [gnutls_pubkey_get_key_id], page 701 calculates a SHA1 digest of the public key as a DER-formatted, subjectPublicKeyInfo object. Other implementations use different approaches, e.g., some use the "common method" described in section 4.2.1.2 of [[RFC5280], page 752] which calculates a digest on a part of the subjectPublicKeyInfo object.

int [gnutls_pubkey_get_pk_algorithm], page 702, (gnutls_pubkey_t *key*, unsigned int * *bits*)
int [gnutls_pubkey_get_preferred_hash_algorithm], page 703, (gnutls_pubkey_t *key*, gnutls_digest_algorithm_t * *hash*, unsigned int * *mand*)
int [gnutls_pubkey_get_key_id], page 701, (gnutls_pubkey_t *key*, unsigned int *flags*, unsigned char * *output_data*, size_t * *output_data_size*)
int [gnutls_random_art], page 446, (gnutls_random_art_t *type*, const char * *key_type*, unsigned int *key_size*, void * *fpr*, size_t *fpr_size*, gnutls_datum_t * *art*)

To export the key-specific parameters, or obtain a unique key ID the following functions are provided.

int [gnutls_pubkey_export_rsa_raw2], page 701, (gnutls_pubkey_t *key*, gnutls_datum_t * *m*, gnutls_datum_t * *e*, unsigned *flags*)
int [gnutls_pubkey_export_dsa_raw2], page 699, (gnutls_pubkey_t *key*, gnutls_datum_t * *p*, gnutls_datum_t * *q*, gnutls_datum_t * *g*, gnutls_datum_t * *y*, unsigned *flags*)
int [gnutls_pubkey_export_ecc_raw2], page 700, (gnutls_pubkey_t *key*, gnutls_ecc_curve_t * *curve*, gnutls_datum_t * *x*, gnutls_datum_t * *y*, unsigned int *flags*)
int [gnutls_pubkey_export_ecc_x962], page 700, (gnutls_pubkey_t *key*, gnutls_datum_t * *parameters*, gnutls_datum_t * *ecpoint*)

5.1.2 Private keys

An abstract `gnutls_privkey_t` can be initialized and freed by using the functions below.

int [gnutls_privkey_init], page 693, (gnutls_privkey_t * *key*)
void [gnutls_privkey_deinit], page 679, (gnutls_privkey_t *key*)

After initialization its values can be imported from an existing structure like `gnutls_x509_privkey_t`, but unlike public keys it cannot be exported. That is to allow abstraction over keys stored in hardware that makes available only operations.

int [gnutls_privkey_import_x509], page 692, (gnutls_privkey_t *pkey*, gnutls_x509_privkey_t *key*, unsigned int *flags*)
int [gnutls_privkey_import_pkcs11], page 690, (gnutls_privkey_t *pkey*, gnutls_pkcs11_privkey_t *key*, unsigned int *flags*)

Other helper functions that allow directly importing from raw X.509 structures are shown below. Again, as with public keys, private keys can be imported from a hardware module using URLs.

`int gnutls_privkey_import_url` (*gnutls_privkey_t* **key**, *const char * * **url**, *unsigned int* **flags**) [Function]

key: A key of type `gnutls_privkey_t`

url: A PKCS 11 url

flags: should be zero

This function will import a PKCS11 or TPM URL as a private key. The supported URL types can be checked using `gnutls_url_is_supported()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

int [gnutls_privkey_import_x509_raw], page 693, (gnutls_privkey_t *pkey*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, const char * *password*, unsigned int *flags*)
int [gnutls_privkey_get_pk_algorithm], page 684, (gnutls_privkey_t *key*, unsigned int * *bits*)
gnutls_privkey_type_t [gnutls_privkey_get_type], page 685, (gnutls_privkey_t *key*)
int [gnutls_privkey_status], page 696, (gnutls_privkey_t *key*)

In order to support cryptographic operations using an external API, the following function is provided. This allows for a simple extensibility API without resorting to PKCS #11.

`int gnutls_privkey_import_ext4` (*gnutls_privkey_t* **pkey**, *void * **userdata**, *gnutls_privkey_sign_data_func* **sign_data_fn**, *gnutls_privkey_sign_hash_func* **sign_hash_fn**, *gnutls_privkey_decrypt_func* **decrypt_fn**, *gnutls_privkey_deinit_func* **deinit_fn**, *gnutls_privkey_info_func* **info_fn**, *unsigned int* **flags**) [Function]

pkey: The private key

userdata: private data to be provided to the callbacks

sign_data_fn: callback for signature operations (may be `NULL`)

sign_hash_fn: callback for signature operations (may be `NULL`)

decrypt_fn: callback for decryption operations (may be `NULL`)

deinit_fn: a deinitialization function

info_fn: returns info about the public key algorithm (should not be `NULL`)

flags: Flags for the import

This function will associate the given callbacks with the gnutls_privkey_t type. At least one of the callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE .

Note that in contrast with the signing function of gnutls_privkey_import_ext3() , the signing functions provided to this function take explicitly the signature algorithm as parameter and different functions are provided to sign the data and hashes.

The sign_hash_fn is to be called to sign pre-hashed data. The input to the callback is the output of the hash (such as SHA256) corresponding to the signature algorithm. For RSA PKCS1 signatures, the signature algorithm can be set to GNUTLS_SIGN_RSA_RAW , and in that case the data should be handled as if they were an RSA PKCS1 DigestInfo structure.

The sign_data_fn is to be called to sign data. The input data will be he data to be signed (and hashed), with the provided signature algorithm. This function is to be used for signature algorithms like Ed25519 which cannot take pre-hashed data as input.

When both sign_data_fn and sign_hash_fn functions are provided they must be able to operate on all the supported signature algorithms, unless prohibited by the type of the algorithm (e.g., as with Ed25519).

The info_fn must provide information on the signature algorithms supported by this private key, and should support the flags GNUTLS_PRIVKEY_INFO_PK_ALGO , GNUTLS_PRIVKEY_INFO_HAVE_SIGN_ALGO and GNUTLS_PRIVKEY_INFO_PK_ALGO_BITS . It must return -1 on unknown flags.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.6.0

On the private keys where exporting of parameters is possible (i.e., software keys), the following functions are also available.

int [gnutls_privkey_export_rsa_raw2], page 682, (gnutls_privkey_t *key*, gnutls_datum_t * *m*, gnutls_datum_t * *e*, gnutls_datum_t * *d*, gnutls_datum_t * *p*, gnutls_datum_t * *q*, gnutls_datum_t * *u*, gnutls_datum_t * *e1*, gnutls_datum_t * *e2*, unsigned int *flags*)
int [gnutls_privkey_export_dsa_raw2], page 679, (gnutls_privkey_t *key*, gnutls_datum_t * *p*, gnutls_datum_t * *q*, gnutls_datum_t * *g*, gnutls_datum_t * *y*, gnutls_datum_t * *x*, unsigned int *flags*)
int [gnutls_privkey_export_ecc_raw2], page 680, (gnutls_privkey_t *key*, gnutls_ecc_curve_t * *curve*, gnutls_datum_t * *x*, gnutls_datum_t * *y*, gnutls_datum_t * *k*, unsigned int *flags*)

5.1.3 Operations

The abstract key types can be used to access signing and signature verification operations with the underlying keys.

`int gnutls_pubkey_verify_data2` (*gnutls_pubkey_t* `pubkey`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t * * `data`, *const gnutls_datum_t * * `signature`) [Function]

pubkey: Holds the public key

algo: The signature algorithm used

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

data: holds the signed data

signature: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success. For known to be insecure signatures this function will return `GNUTLS_E_INSUFFICIENT_SECURITY` unless the flag `GNUTLS_VERIFY_ALLOW_BROKEN` is specified.

Since: 3.0

`int gnutls_pubkey_verify_hash2` (*gnutls_pubkey_t* `key`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t * * `hash`, *const gnutls_datum_t * * `signature`) [Function]

key: Holds the public key

algo: The signature algorithm used

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

hash: holds the hash digest to be verified

signature: contains the signature

This function will verify the given signed digest, using the parameters from the public key. Note that unlike `gnutls_privkey_sign_hash()` , this function accepts a signature algorithm instead of a digest algorithm. You can use `gnutls_pk_to_sign()` to get the appropriate value.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success. For known to be insecure signatures this function will return `GNUTLS_E_INSUFFICIENT_SECURITY` unless the flag `GNUTLS_VERIFY_ALLOW_BROKEN` is specified.

Since: 3.0

`int gnutls_pubkey_encrypt_data` (*gnutls_pubkey_t* `key`, *unsigned int* `flags`, *const gnutls_datum_t * * `plaintext`, *gnutls_datum_t * * `ciphertext`) [Function]

key: Holds the public key

flags: should be 0 for now

plaintext: The data to be encrypted

ciphertext: contains the encrypted data

This function will encrypt the given data, using the public key. On success the `ciphertext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

`int gnutls_privkey_sign_data (`*gnutls_privkey_t* **`signer`**, *gnutls_digest_algorithm_t* **`hash`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`data`**, *gnutls_datum_t ** **`signature`**`)` [Function]

signer: Holds the key

hash: should be a digest algorithm

flags: Zero or one of `gnutls_privkey_flags_t`

data: holds the data to be signed

signature: will contain the signature allocated with `gnutls_malloc()`

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only the SHA family for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

`int gnutls_privkey_sign_hash (`*gnutls_privkey_t* **`signer`**, *gnutls_digest_algorithm_t* **`hash_algo`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`hash_data`**, *gnutls_datum_t ** **`signature`**`)` [Function]

signer: Holds the signer's key

hash_algo: The hash algorithm used

flags: Zero or one of `gnutls_privkey_flags_t`

hash_data: holds the data to be signed

signature: will contain newly allocated signature

This function will sign the given hashed data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-XXX for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

The flags may be `GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA` or `GNUTLS_PRIVKEY_SIGN_FLAG_RSA_PSS` . In the former case this function will ignore `hash_algo` and perform a raw PKCS1 signature, and in the latter an RSA-PSS signature will be generated.

Note that, not all algorithm support signing already hashed data. When signing with Ed25519, `gnutls_privkey_sign_data()` should be used.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

`int gnutls_privkey_decrypt_data` (*gnutls_privkey_t* `key`, *unsigned int* `flags`, *const gnutls_datum_t ** `ciphertext`, *gnutls_datum_t ** `plaintext`) [Function]

key: Holds the key

flags: zero for now

ciphertext: holds the data to be decrypted

plaintext: will contain the decrypted data, allocated with `gnutls_malloc()`

This function will decrypt the given data using the algorithm supported by the private key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

Signing existing structures, such as certificates, CRLs, or certificate requests, as well as associating public keys with structures is also possible using the key abstractions.

`int gnutls_x509_crq_set_pubkey` (*gnutls_x509_crq_t* `crq`, *gnutls_pubkey_t* `key`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a public key

This function will set the public parameters from the given public key to the request. The `key` can be deallocated after that.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

`int gnutls_x509_crt_set_pubkey` (*gnutls_x509_crt_t* `crt`, *gnutls_pubkey_t* `key`) [Function]

crt: should contain a `gnutls_x509_crt_t` type

key: holds a public key

This function will set the public parameters from the given public key to the certificate. The `key` can be deallocated after that.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

int [gnutls_x509_crt_privkey_sign], page 714, (gnutls_x509_crt_t *crt*, gnutls_x509_crt_t *issuer*, gnutls_privkey_t *issuer_key*, gnutls_digest_algorithm_t *dig*, unsigned int *flags*)
int [gnutls_x509_crl_privkey_sign], page 713, (gnutls_x509_crl_t *crl*, gnutls_x509_crt_t *issuer*, gnutls_privkey_t *issuer_key*, gnutls_digest_algorithm_t *dig*, unsigned int *flags*)
int [gnutls_x509_crq_privkey_sign], page 713, (gnutls_x509_crq_t *crq*, gnutls_privkey_t *key*, gnutls_digest_algorithm_t *dig*, unsigned int *flags*)

5.2 System and application-specific keys

5.2.1 System-specific keys

In several systems there are keystores which allow to read, store and use certificates and private keys. For these systems GnuTLS provides the system-key API in `gnutls/system-keys.h`. That API provides the ability to iterate through all stored keys, add and delete keys as well as use these keys using a URL which starts with `"system:"`. The format of the URLs is system-specific. The `systemkey` tool is also provided to assist in listing keys and debugging.

The systems supported via this API are the following.

- Windows Cryptography API (CNG)

`int gnutls_system_key_iter_get_info (`*gnutls_system_key_iter_t* `* iter`, *unsigned* `cert_type`, *char ** `cert_url`, *char ** `key_url`, *char ** `label`, *gnutls_datum_t * `der`, *unsigned int* `flags`) [Function]

iter: an iterator of the system keys (must be set to `NULL` initially)

cert_type: A value of gnutls_certificate_type_t which indicates the type of certificate to look for

cert_url: The certificate URL of the pair (may be `NULL`)

key_url: The key URL of the pair (may be `NULL`)

label: The friendly name (if any) of the pair (may be `NULL`)

der: if non-NULL the DER data of the certificate

flags: should be zero

This function will return on each call a certificate and key pair URLs, as well as a label associated with them, and the DER-encoded certificate. When the iteration is complete it will return `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` .

Typically `cert_type` should be `GNUTLS_CRT_X509` .

All values set are allocated and must be cleared using `gnutls_free()` ,

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

void [gnutls_system_key_iter_deinit], page 712, (gnutls_system_key_iter_t *iter*)
int [gnutls_system_key_add_x509], page 712, (gnutls_x509_crt_t *crt*, gnutls_x509_privkey_t *privkey*, const char * *label*, char ** *cert_url*, char ** *key_url*)
int [gnutls_system_key_delete], page 712, (const char * *cert_url*, const char * *key_url*)

5.2.2 Application-specific keys

For systems where GnuTLS doesn't provide a system specific store, it may often be desirable to define a custom class of keys that are identified via URLs and available to GnuTLS calls such as [gnutls_certificate_set_x509_key_file2], page 384. Such keys can be

registered using the API in `gnutls/urls.h`. The function which registers such keys is [gnutls_register_custom_url], page 711.

`int gnutls_register_custom_url (`*const gnutls_custom_url_st * st*`)` [Function]

st: A `gnutls_custom_url_st` structure

Register a custom URL. This will affect the following functions: `gnutls_url_is_supported()` , `gnutls_privkey_import_url()` , gnutls_pubkey_import_url, `gnutls_x509_crt_import_url()` and all functions that depend on them, e.g., `gnutls_certificate_set_x509_key_file2()` .

The provided structure and callback functions must be valid throughout the lifetime of the process. The registration of an existing URL type will fail with `GNUTLS_E_INVALID_REQUEST` . Since GnuTLS 3.5.0 this function can be used to override the builtin URLs.

This function is not thread safe.

Returns: returns zero if the given structure was imported or a negative value otherwise.

Since: 3.4.0

The input to this function are three callback functions as well as the prefix of the URL, (e.g., "mypkcs11:") and the length of the prefix. The types of the callbacks are shown below, and are expected to use the exported gnutls functions to import the keys and certificates. E.g., a typical `import_key` callback should use [gnutls_privkey_import_ext4], page 688.

```
typedef int (*gnutls_privkey_import_url_func)(gnutls_privkey_t pkey,
                                              const char *url,
                                              unsigned flags);

typedef int (*gnutls_x509_crt_import_url_func)(gnutls_x509_crt_t pkey,
                                               const char *url,
                                               unsigned flags);

/* The following callbacks are optional */

/* This is to enable gnutls_pubkey_import_url() */
typedef int (*gnutls_pubkey_import_url_func)(gnutls_pubkey_t pkey,
     const char *url, unsigned flags);

/* This is to allow constructing a certificate chain. It will be provided
 * the initial certificate URL and the certificate to find its issuer, and must
 * return zero and the DER encoding of the issuer's certificate. If not available,
 * it should return GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE. */
typedef int (*gnutls_get_raw_issuer_func)(const char *url, gnutls_x509_crt_t crt,
  gnutls_datum_t *issuer_der, unsigned flags);

typedef struct custom_url_st {
        const char *name;
```

```
        unsigned name_size;
        gnutls_privkey_import_url_func import_key;
        gnutls_x509_crt_import_url_func import_crt;
        gnutls_pubkey_import_url_func import_pubkey;
gnutls_get_raw_issuer_func get_issuer;
} gnutls_custom_url_st;
```

5.3 Smart cards and HSMs

In this section we present the smart-card and hardware security module (HSM) support in GnuTLS using PKCS #11 [[PKCS11], page 752]. Hardware security modules and smart cards provide a way to store private keys and perform operations on them without exposing them. This decouples cryptographic keys from the applications that use them and provide an additional security layer against cryptographic key extraction. Since this can also be achieved in software components such as in Gnome keyring, we will use the term security module to describe any cryptographic key separation subsystem.

PKCS #11 is plugin API allowing applications to access cryptographic operations on a security module, as well as to objects residing on it. PKCS #11 modules exist for hardware tokens such as smart cards[1], cryptographic tokens, as well as for software modules like Gnome Keyring. The objects residing on a security module may be certificates, public keys, private keys or secret keys. Of those certificates and public/private key pairs can be used with GnuTLS. PKCS #11's main advantage is that it allows operations on private key objects such as decryption and signing without exposing the key. In GnuTLS the PKCS #11 functionality is available in `gnutls/pkcs11.h`.

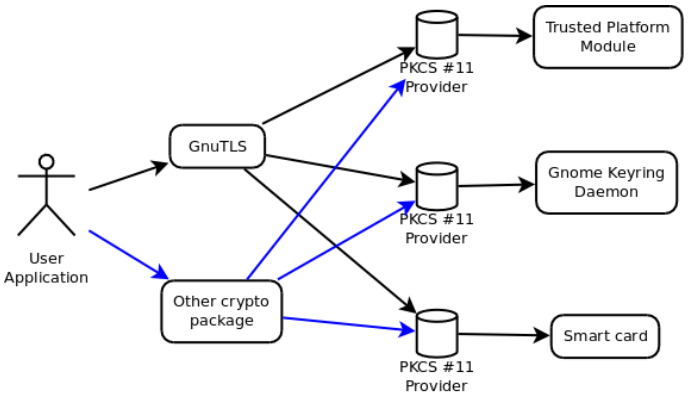

Figure 5.1: PKCS #11 module usage.

5.3.1 Initialization

To allow all GnuTLS applications to transparently access smart cards and tokens, PKCS #11 is automatically initialized during the first call of a PKCS #11 related function, in a thread safe way. The default initialization process, utilizes p11-kit configuration, and loads

[1] For example, OpenSC-supported cards.

any appropriate PKCS #11 modules. The p11-kit configuration files[2] are typically stored in `/etc/pkcs11/modules/`. For example a file that will instruct GnuTLS to load the OpenSC module, could be named `/etc/pkcs11/modules/opensc.module` and contain the following:

```
module: /usr/lib/opensc-pkcs11.so
```

If you use these configuration files, then there is no need for other initialization in GnuTLS, except for the PIN and token callbacks (see next section). In several cases, however, it is desirable to limit badly behaving modules (e.g., modules that add an unacceptable delay on initialization) to single applications. That can be done using the "enable-in:" option followed by the base name of applications that this module should be used.

It is also possible to manually initialize or even disable the PKCS #11 subsystem if the default settings are not desirable or not available (see Section 5.3.2 [PKCS11 Manual Initialization], page 111, for more information).

Note that, PKCS #11 modules behave in a peculiar way after a fork; they require a reinitialization of all the used PKCS #11 resources. While GnuTLS automates that process, there are corner cases where it is not possible to handle it correctly in an automated way[3]. For that, it is recommended not to mix fork() and PKCS #11 module usage. It is recommended to initialize and use any PKCS #11 resources in a single process.

Older versions of GnuTLS required to call [gnutls_pkcs11_reinit], page 665 after a fork() call; since 3.3.0 this is no longer required.

5.3.2 Manual initialization of user-specific modules

In systems where one cannot rely on a globally available p11-kit configuration to be available, it is still possible to utilize PKCS #11 objects. That can be done by loading directly the PKCS #11 shared module in the application using [gnutls_pkcs11_add_provider], page 649, after having called [gnutls_pkcs11_init], page 655 specifying the `GNUTLS_PKCS11_FLAG_MANUAL` flag.

`int gnutls_pkcs11_add_provider (`*const char * `name`, const char * `params`*`)` [Function]

name: The filename of the module

params: should be NULL or a known string (see description)

This function will load and add a PKCS 11 module to the module list used in gnutls. After this function is called the module will be used for PKCS 11 operations.

When loading a module to be used for certificate verification, use the string 'trusted' as `params` .

Note that this function is not thread safe.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

[2] `https://p11-glue.github.io/p11-glue/p11-kit.html`

[3] For example when an open session is to be reinitialized, but the PIN is not available to GnuTLS (e.g., it was entered at a pinpad).

In that case, the application will only have access to the modules explicitly loaded. If the GNUTLS_PKCS11_FLAG_MANUAL flag is specified and no calls to [gnutls_pkcs11_add_provider], page 649 are made, then the PKCS #11 functionality is effectively disabled.

`int gnutls_pkcs11_init (`*unsigned int flags*, *const char ** *deprecated_config_file*`)` [Function]

flags: An ORed sequence of GNUTLS_PKCS11_FLAG_ *

deprecated_config_file: either NULL or the location of a deprecated configuration file

This function will initialize the PKCS 11 subsystem in gnutls. It will read configuration files if GNUTLS_PKCS11_FLAG_AUTO is used or allow you to independently load PKCS 11 modules using gnutls_pkcs11_add_provider() if GNUTLS_PKCS11_FLAG_MANUAL is specified.

You don't need to call this function since GnuTLS 3.3.0 because it is being called during the first request PKCS 11 operation. That call will assume the GNUTLS_PKCS11_FLAG_AUTO flag. If another flags are required then it must be called independently prior to any PKCS 11 operation.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

5.3.3 Accessing objects that require a PIN

Objects stored in token such as a private keys are typically protected from access by a PIN or password. This PIN may be required to either read the object (if allowed) or to perform operations with it. To allow obtaining the PIN when accessing a protected object, as well as probe the user to insert the token the following functions allow to set a callback.

void [gnutls_pkcs11_set_token_function], page 666, (gnutls_pkcs11_token_callback_t *fn*, void * *userdata*)
void [gnutls_pkcs11_set_pin_function], page 665, (gnutls_pin_callback_t *fn*, void * *userdata*)
int [gnutls_pkcs11_add_provider], page 649, (const char * *name*, const char * *params*)
gnutls_pin_callback_t [gnutls_pkcs11_get_pin_function], page 653, (void ** *userdata*)

The callback is of type gnutls_pin_callback_t and will have as input the provided userdata, the PIN attempt number, a URL describing the token, a label describing the object and flags. The PIN must be at most of pin_max size and must be copied to pin variable. The function must return 0 on success or a negative error code otherwise.

```
typedef int (*gnutls_pin_callback_t) (void *userdata, int attempt,
                                      const char *token_url,
                                      const char *token_label,
                                      unsigned int flags,
                                      char *pin, size_t pin_max);
```

The flags are of gnutls_pin_flag_t type and are explained below.

`GNUTLS_PIN_USER`
: The PIN for the user.

`GNUTLS_PIN_SO`
: The PIN for the security officer (admin).

`GNUTLS_PIN_FINAL_TRY`
: This is the final try before blocking.

`GNUTLS_PIN_COUNT_LOW`
: Few tries remain before token blocks.

`GNUTLS_PIN_CONTEXT_SPECIFIC`
: The PIN is for a specific action and key like signing.

`GNUTLS_PIN_WRONG`
: Last given PIN was not correct.

Figure 5.2: The `gnutls_pin_flag_t` enumeration.

Note that due to limitations of PKCS #11 there are issues when multiple libraries are sharing a module. To avoid this problem GnuTLS uses p11-kit that provides a middleware to control access to resources over the multiple users.

To avoid conflicts with multiple registered callbacks for PIN functions, [gnutls_pkcs11_get_pin_function], page 653 may be used to check for any previously set functions. In addition context specific PIN functions are allowed, e.g., by using functions below.

void [gnutls_certificate_set_pin_function], page 377, (gnutls_certificate_credentials_t *cred*, gnutls_pin_callback_t *fn*, void * *userdata*)
void [gnutls_pubkey_set_pin_function], page 710, (gnutls_pubkey_t *key*, gnutls_pin_callback_t *fn*, void * *userdata*)
void [gnutls_privkey_set_pin_function], page 694, (gnutls_privkey_t *key*, gnutls_pin_callback_t *fn*, void * *userdata*)
void [gnutls_pkcs11_obj_set_pin_function], page 661, (gnutls_pkcs11_obj_t *obj*, gnutls_pin_callback_t *fn*, void * *userdata*)
void [gnutls_x509_crt_set_pin_function], page 570, (gnutls_x509_crt_t *crt*, gnutls_pin_callback_t *fn*, void * *userdata*)

5.3.4 Reading objects

All PKCS #11 objects are referenced by GnuTLS functions by URLs as described in [[PKCS11URI], page 753]. This allows for a consistent naming of objects across systems and applications in the same system. For example a public key on a smart card may be referenced as:

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315; \
manufacturer=EnterSafe;object=test1;type=public;\
id=32f153f3e37990b08624141077ca5dec2d15faed
```

while the smart card itself can be referenced as:

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315;manufacturer=EnterSafe
```

Objects stored in a PKCS #11 token can typically be extracted if they are not marked as sensitive. Usually only private keys are marked as sensitive and cannot be extracted, while certificates and other data can be retrieved. The functions that can be used to enumerate and access objects are shown below.

int [gnutls_pkcs11_obj_list_import_url4], page 660, (gnutls_pkcs11_obj_t ** *p_list*, unsigned int * *n_list*, const char * *url*, unsigned int *flags*)
int [gnutls_pkcs11_obj_import_url], page 659, (gnutls_pkcs11_obj_t *obj*, const char * *url*, unsigned int *flags*)
int [gnutls_pkcs11_obj_export_url], page 656, (gnutls_pkcs11_obj_t *obj*, gnutls_pkcs11_url_type_t *detailed*, char ** *url*)

int gnutls_pkcs11_obj_get_info (*gnutls_pkcs11_obj_t* **obj**, *gnutls_pkcs11_obj_info_t* **itype**, *void* * **output**, *size_t* * **output_size**) [Function]

obj: should contain a gnutls_pkcs11_obj_t type

itype: Denotes the type of information requested

output: where output will be stored

output_size: contains the maximum size of the output buffer and will be overwritten with the actual size.

This function will return information about the PKCS11 certificate such as the label, id as well as token information where the key is stored.

When output is text, a null terminated string is written to output and its string length is written to output_size (without null terminator). If the buffer is too small, output_size will contain the expected buffer size (with null terminator for text) and return GNUTLS_E_SHORT_MEMORY_BUFFER .

In versions previously to 3.6.0 this function included the null terminator to output_size . After 3.6.0 the output size doesn't include the terminator character.

Returns: GNUTLS_E_SUCCESS (0) on success or a negative error code on error.

Since: 2.12.0

int [gnutls_x509_crt_import_pkcs11], page 669, (gnutls_x509_crt_t *crt*, gnutls_pkcs11_obj_t *pkcs11_crt*)
int [gnutls_x509_crt_import_url], page 560, (gnutls_x509_crt_t *crt*, const char * *url*, unsigned int *flags*)
int [gnutls_x509_crt_list_import_pkcs11], page 669, (gnutls_x509_crt_t * *certs*, unsigned int *cert_max*, gnutls_pkcs11_obj_t * const *objs*, unsigned int *flags*)

Properties of the physical token can also be accessed and altered with GnuTLS. For example data in a token can be erased (initialized), PIN can be altered, etc.

int [gnutls_pkcs11_token_init], page 668, (const char * *token_url*, const char * *so_pin*, const char * *label*)
int [gnutls_pkcs11_token_get_url], page 668, (unsigned int *seq*, gnutls_pkcs11_url_type_t *detailed*, char ** *url*)
int [gnutls_pkcs11_token_get_info], page 667, (const char * *url*, gnutls_pkcs11_token_info_t *ttype*, void * *output*, size_t * *output_size*)
int [gnutls_pkcs11_token_get_flags], page 666, (const char * *url*, unsigned int * *flags*)
int [gnutls_pkcs11_token_set_pin], page 668, (const char * *token_url*, const char * *oldpin*, const char * *newpin*, unsigned int *flags*)

The following examples demonstrate the usage of the API. The first example will list all available PKCS #11 tokens in a system and the latter will list all certificates in a token that have a corresponding private key.

```
        int i;
        char* url;

        gnutls_global_init();

        for (i=0;;i++)
          {
            ret = gnutls_pkcs11_token_get_url(i, &url);
            if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
              break;

            if (ret < 0)
              exit(1);

            fprintf(stdout, "Token[%d]: URL: %s\n", i, url);
            gnutls_free(url);
          }
        gnutls_global_deinit();
```

```
/* This example code is placed in the public domain. */

#include <config.h>
#include <gnutls/gnutls.h>
#include <gnutls/pkcs11.h>
#include <stdio.h>
#include <stdlib.h>

#define URL "pkcs11:URL"

int main(int argc, char **argv)
{
        gnutls_pkcs11_obj_t *obj_list;
        gnutls_x509_crt_t xcrt;
        unsigned int obj_list_size = 0;
```

```
        gnutls_datum_t cinfo;
        int ret;
        unsigned int i;

        ret = gnutls_pkcs11_obj_list_import_url4(&obj_list, &obj_list_size, URL,
                                                 GNUTLS_PKCS11_OBJ_FLAG_CRT|
                                                 GNUTLS_PKCS11_OBJ_FLAG_WITH_PRIVKEY);
        if (ret < 0)
                return -1;

        /* now all certificates are in obj_list */
        for (i = 0; i < obj_list_size; i++) {

                gnutls_x509_crt_init(&xcrt);

                gnutls_x509_crt_import_pkcs11(xcrt, obj_list[i]);

                gnutls_x509_crt_print(xcrt, GNUTLS_CRT_PRINT_FULL, &cinfo);

                fprintf(stdout, "cert[%d]:\n %s\n\n", i, cinfo.data);

                gnutls_free(cinfo.data);
                gnutls_x509_crt_deinit(xcrt);
        }

        for (i = 0; i < obj_list_size; i++)
                gnutls_pkcs11_obj_deinit(obj_list[i]);
        gnutls_free(obj_list);

        return 0;
}
```

5.3.5 Writing objects

With GnuTLS you can copy existing private keys and certificates to a token. Note that when copying private keys it is recommended to mark them as sensitive using the `GNUTLS_PKCS11_OBJ_FLAG_MARK_SENSITIVE` to prevent its extraction. An object can be marked as private using the flag `GNUTLS_PKCS11_OBJ_FLAG_MARK_PRIVATE`, to require PIN to be entered before accessing the object (for operations or otherwise).

`int gnutls_pkcs11_copy_x509_privkey2` (*const char * `token_url`*, *gnutls_x509_privkey_t `key`*, *const char * `label`*, *const gnutls_datum_t * `cid`*, *unsigned int `key_usage`*, *unsigned int `flags`*) [Function]

token_url: A PKCS 11 URL specifying a token

key: A private key

label: A name to be used for the stored data

cid: The CKA_ID to set for the object -if NULL, the ID will be derived from the public key

key_usage: One of GNUTLS_KEY_*

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will copy a private key into a PKCS 11 token specified by a URL.

Since 3.6.3 the objects are marked as sensitive by default unless GNUTLS_PKCS11_OBJ_FLAG_MARK_NOT_SENSITIVE is specified.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

`int gnutls_pkcs11_copy_x509_crt2` (*const char * `token_url`*, *gnutls_x509_crt_t `crt`*, *const char * `label`*, *const gnutls_datum_t * `cid`*, *unsigned int `flags`*) [Function]

token_url: A PKCS 11 URL specifying a token

crt: The certificate to copy

label: The name to be used for the stored data

cid: The CKA_ID to set for the object -if NULL, the ID will be derived from the public key

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy a certificate into a PKCS 11 token specified by a URL. Valid flags to mark the certificate: GNUTLS_PKCS11_OBJ_FLAG_MARK_TRUSTED , GNUTLS_PKCS11_OBJ_FLAG_MARK_PRIVATE , GNUTLS_PKCS11_OBJ_FLAG_MARK_CA , GNUTLS_PKCS11_OBJ_FLAG_MARK_ALWAYS_AUTH .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

`int gnutls_pkcs11_delete_url` (*const char * `object_url`*, *unsigned int `flags`*) [Function]

object_url: The URL of the object to delete.

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will delete objects matching the given URL. Note that not all tokens support the delete operation.

Returns: On success, the number of objects deleted is returned, otherwise a negative error value.

Since: 2.12.0

5.3.6 Low Level Access

When it is needed to use PKCS#11 functionality which is not wrapped by GnuTLS, it is possible to extract the PKCS#11 session, object or token pointers. That allows an application to still access the low-level functionality, while at the same time take advantage of the URI addressing scheme supported by GnuTLS.

`int gnutls_pkcs11_token_get_ptr` (*const char * `url`, void ** `ptr`, unsigned long * `slot_id`, unsigned int `flags`*) [Function]

url: should contain a PKCS11 URL identifying a token

ptr: will contain the CK_FUNCTION_LIST_PTR pointer

slot_id: will contain the slot_id (may be `NULL`)

flags: should be zero

This function will return the function pointer of the specified token by the URL. The returned pointers are valid until gnutls is deinitialized, c.f. `_global_deinit()` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.6.3

`int gnutls_pkcs11_obj_get_ptr` (*gnutls_pkcs11_obj_t `obj`, void ** `ptr`, void ** `session`, void ** `ohandle`, unsigned long * `slot_id`, unsigned int `flags`*) [Function]

obj: should contain a `gnutls_pkcs11_obj_t` type

ptr: will contain the CK_FUNCTION_LIST_PTR pointer (may be `NULL`)

session: will contain the CK_SESSION_HANDLE of the object

ohandle: will contain the CK_OBJECT_HANDLE of the object

slot_id: the identifier of the slot (may be `NULL`)

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

Obtains the PKCS11 session handles of an object. `session` and `ohandle` must be deinitialized by the caller. The returned pointers are independent of the `obj` lifetime.

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.6.3

5.3.7 Using a PKCS #11 token with TLS

It is possible to use a PKCS #11 token to a TLS session, as shown in [ex-pkcs11-client], page 202. In addition the following functions can be used to load PKCS #11 key and certificates by specifying a PKCS #11 URL instead of a filename.

```
int [gnutls_certificate_set_x509_trust_file], page 388,
(gnutls_certificate_credentials_t cred, const char * cafile,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_key_file2], page 384,
(gnutls_certificate_credentials_t res, const char * certfile, const char *
keyfile, gnutls_x509_crt_fmt_t type, const char * pass, unsigned int flags)
```

5.3.8 Verifying certificates over PKCS #11

The PKCS #11 API can be used to allow all applications in the same operating system to access shared cryptographic keys and certificates in a uniform way, as in Figure 5.1. That way applications could load their trusted certificate list, as well as user certificates from a common PKCS #11 module. Such a provider is the p11-kit trust storage module[4] and

[4] `https://p11-glue.github.io/p11-glue/trust-module.html`

it provides access to the trusted Root CA certificates in a system. That provides a more dynamic list of Root CA certificates, as opposed to a static list in a file or directory.

That store, allows for blacklisting of CAs or certificates, as well as categorization of the Root CAs (Web verification, Code signing, etc.), in addition to restricting their purpose via stapled extensions[5]. GnuTLS will utilize the p11-kit trust module as the default trust store if configured to; i.e., if '–with-default-trust-store-pkcs11=pkcs11:' is given to the configure script.

5.3.9 Invoking p11tool

Program that allows operations on PKCS #11 smart cards and security modules.

To use PKCS #11 tokens with GnuTLS the p11-kit configuration files need to be setup. That is create a .module file in /etc/pkcs11/modules with the contents 'module: /path/to/pkcs11.so'. Alternatively the configuration file /etc/gnutls/pkcs11.conf has to exist and contain a number of lines of the form 'load=/usr/lib/opensc-pkcs11.so'.

You can provide the PIN to be used for the PKCS #11 operations with the environment variables GNUTLS_PIN and GNUTLS_SO_PIN.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `p11tool` program. This software is released under the GNU General Public License, version 3 or later.

p11tool help/usage (`--help`)

This is the automatically generated usage text for p11tool.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
p11tool - GnuTLS PKCS #11 tool
Usage:  p11tool [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [url]


Tokens:

       --list-tokens          List all available tokens
       --list-token-urls      List the URLs available tokens
       --list-mechanisms      List all available mechanisms in a token
       --initialize           Initializes a PKCS #11 token
       --initialize-pin       Initializes/Resets a PKCS #11 token user PIN
       --initialize-so-pin    Initializes/Resets a PKCS #11 token security officer PIN.
       --set-pin=str          Specify the PIN to use on token operations
       --set-so-pin=str       Specify the Security Officer's PIN to use on token initializa

Object listing:
```

[5] See the 'Restricting the scope of CA certificates' post at `https://nmav.gnutls.org/2016/06/restricting-scope-of-ca-certificates.html`

```
       --list-all             List all available objects in a token
       --list-all-certs       List all available certificates in a token
       --list-certs           List all certificates that have an associated private key
       --list-all-privkeys    List all available private keys in a token
       --list-privkeys        an alias for the 'list-all-privkeys' option
       --list-keys            an alias for the 'list-all-privkeys' option
       --list-all-trusted     List all available certificates marked as trusted
       --export               Export the object specified by the URL
                                - prohibits these options:
                                export-stapled
                                export-chain
                                export-pubkey
       --export-stapled       Export the certificate object specified by the URL
                                - prohibits these options:
                                export
                                export-chain
                                export-pubkey
       --export-chain         Export the certificate specified by the URL and its chain of
                                - prohibits these options:
                                export-stapled
                                export
                                export-pubkey
       --export-pubkey        Export the public key for a private key
                                - prohibits these options:
                                export-stapled
                                export
                                export-chain
       --info                 List information on an available object in a token
       --trusted              an alias for the 'mark-trusted' option
       --distrusted           an alias for the 'mark-distrusted' option

Key generation:

       --generate-privkey=str Generate private-public key pair of given type
       --bits=num             Specify the number of bits for the key generate
       --curve=str            Specify the curve used for EC key generation
       --sec-param=str        Specify the security level

Writing objects:

       --set-id=str           Set the CKA_ID (in hex) for the specified by the URL object
                                - prohibits the option 'write'
       --set-label=str        Set the CKA_LABEL for the specified by the URL object
                                - prohibits these options:
                                write
                                set-id
```

```
      --write                Writes the loaded objects to a PKCS #11 token
      --delete               Deletes the objects matching the given PKCS #11 URL
      --label=str            Sets a label for the write operation
      --id=str               Sets an ID for the write operation
      --mark-wrap            Marks the generated key to be a wrapping key
                               - disabled as '--no-mark-wrap'
      --mark-trusted         Marks the object to be written as trusted
                               - prohibits the option 'mark-distrusted'
                               - disabled as '--no-mark-trusted'
      --mark-distrusted      When retrieving objects, it requires the objects to be distru
(blacklisted)
                               - prohibits the option 'mark-trusted'
      --mark-decrypt         Marks the object to be written for decryption
                               - disabled as '--no-mark-decrypt'
      --mark-sign            Marks the object to be written for signature generation
                               - disabled as '--no-mark-sign'
      --mark-ca              Marks the object to be written as a CA
                               - disabled as '--no-mark-ca'
      --mark-private         Marks the object to be written as private
                               - disabled as '--no-mark-private'
      --ca                   an alias for the 'mark-ca' option
      --private              an alias for the 'mark-private' option
      --secret-key=str       Provide a hex encoded secret key
      --load-privkey=file    Private key file to use
                               - file must pre-exist
      --load-pubkey=file     Public key file to use
                               - file must pre-exist
      --load-certificate=file Certificate file to use
                               - file must pre-exist

Other options:

   -d, --debug=num           Enable debugging
                               - it must be in the range:
                                 0 to 9999
      --outfile=str          Output file
      --login                Force (user) login to token
                               - disabled as '--no-login'
      --so-login             Force security officer login to token
                               - disabled as '--no-so-login'
      --admin-login          an alias for the 'so-login' option
      --test-sign            Tests the signature operation of the provided object
      --sign-params=str      Sign with a specific signature algorithm
      --hash=str             Hash algorithm to use for signing
      --generate-random=num  Generate random data
   -8, --pkcs8               Use PKCS #8 format for private keys
      --inder                Use DER/RAW format for input
```

```
                              - disabled as '--no-inder'
      --inraw                 an alias for the 'inder' option
      --outder                Use DER format for output certificates, private keys, and DH
                              - disabled as '--no-outder'
      --outraw                an alias for the 'outder' option
      --provider=file         Specify the PKCS #11 provider library
      --detailed-url          Print detailed URLs
                              - disabled as '--no-detailed-url'
      --only-urls             Print a compact listing using only the URLs
      --batch                 Disable all interaction with the tool

Version, usage and configuration options:

   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.
Operands and options may be intermixed.  They will be reordered.

Program that allows operations on PKCS #11 smart cards and security
modules.

To use PKCS #11 tokens with GnuTLS the p11-kit configuration files need to
be setup.  That is create a .module file in /etc/pkcs11/modules with the
contents 'module: /path/to/pkcs11.so'.  Alternatively the configuration
file /etc/gnutls/pkcs11.conf has to exist and contain a number of lines of
the form 'load=/usr/lib/opensc-pkcs11.so'.

You can provide the PIN to be used for the PKCS #11 operations with the
environment variables GNUTLS_PIN and GNUTLS_SO_PIN.
```

token-related-options options

Tokens.

list-token-urls option.

This is the "list the urls available tokens" option. This is a more compact version of –list-tokens.

initialize-so-pin option.

This is the "initializes/resets a pkcs #11 token security officer pin." option. This initializes the security officer's PIN. When used non-interactively use the GNUTLS_NEW_SO_PIN environment variables to initialize SO's PIN.

set-pin option.

This is the "specify the pin to use on token operations" option. This option takes a string argument. Alternatively the GNUTLS_PIN environment variable may be used.

set-so-pin option.

This is the "specify the security officer's pin to use on token initialization" option. This option takes a string argument. Alternatively the GNUTLS_SO_PIN environment variable may be used.

object-list-related-options options

Object listing.

list-all option.

This is the "list all available objects in a token" option. All objects available in the token will be listed. That includes objects which are potentially unaccessible using this tool.

list-all-certs option.

This is the "list all available certificates in a token" option. That option will also provide more information on the certificates, for example, expand the attached extensions in a trust token (like p11-kit-trust).

list-certs option.

This is the "list all certificates that have an associated private key" option. That option will only display certificates which have a private key associated with them (share the same ID).

list-all-privkeys option.

This is the "list all available private keys in a token" option. Lists all the private keys in a token that match the specified URL.

list-privkeys option.

This is an alias for the `list-all-privkeys` option, see [p11tool list-all-privkeys], page 123.

list-keys option.

This is an alias for the `list-all-privkeys` option, see [p11tool list-all-privkeys], page 123.

export-stapled option.

This is the "export the certificate object specified by the url" option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: export, export-chain, export-pubkey.

Exports the certificate specified by the URL while including any attached extensions to it. Since attached extensions are a p11-kit extension, this option is only available on p11-kit registered trust modules.

export-chain option.

This is the "export the certificate specified by the url and its chain of trust" option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: export-stapled, export, export-pubkey.

Exports the certificate specified by the URL and generates its chain of trust based on the stored certificates in the module.

export-pubkey option.

This is the "export the public key for a private key" option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: export-stapled, export, export-chain.

Exports the public key for the specified private key

trusted option.

This is an alias for the `mark-trusted` option, see [p11tool mark-trusted], page 126.

distrusted option.

This is an alias for the `mark-distrusted` option, see [p11tool mark-distrusted], page 126.

keygen-related-options options

Key generation.

generate-privkey option.

This is the "generate private-public key pair of given type" option. This option takes a string argument. Generates a private-public key pair in the specified token. Acceptable types are RSA, ECDSA, Ed25519, and DSA. Should be combined with –sec-param or –bits.

generate-rsa option.

This is the "generate an rsa private-public key pair" option. Generates an RSA private-public key pair on the specified token. Should be combined with –sec-param or –bits.

NOTE: THIS OPTION IS DEPRECATED

generate-dsa option.

This is the "generate a dsa private-public key pair" option. Generates a DSA private-public key pair on the specified token. Should be combined with –sec-param or –bits.

NOTE: THIS OPTION IS DEPRECATED

generate-ecc option.

This is the "generate an ecdsa private-public key pair" option. Generates an ECDSA private-public key pair on the specified token. Should be combined with –curve, –sec-param or –bits.

NOTE: THIS OPTION IS DEPRECATED

bits option.

This is the "specify the number of bits for the key generate" option. This option takes a number argument. For applications which have no key-size restrictions the –sec-param option is recommended, as the sec-param levels will adapt to the acceptable security levels with the new versions of gnutls.

curve option.

This is the "specify the curve used for ec key generation" option. This option takes a string argument. Supported values are secp192r1, secp224r1, secp256r1, secp384r1 and secp521r1.

sec-param option.

This is the "specify the security level" option. This option takes a string argument `Security parameter`. This is alternative to the bits option. Available options are [low, legacy, medium, high, ultra].

write-object-related-options options

Writing objects.

set-id option.

This is the "set the cka_id (in hex) for the specified by the url object" option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: write.

Modifies or sets the CKA_ID in the specified by the URL object. The ID should be specified in hexadecimal format without a '0x' prefix.

set-label option.

This is the "set the cka_label for the specified by the url object" option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: write, set-id.

Modifies or sets the CKA_LABEL in the specified by the URL object

write option.

This is the “writes the loaded objects to a pkcs #11 token” option. It can be used to write private, public keys, certificates or secret keys to a token. Must be combined with one of –load-privkey, –load-pubkey, –load-certificate option.

id option.

This is the “sets an id for the write operation” option. This option takes a string argument. Sets the CKA_ID to be set by the write operation. The ID should be specified in hexadecimal format without a ’0x’ prefix.

mark-wrap option.

This is the “marks the generated key to be a wrapping key” option.

This option has some usage constraints. It:

- can be disabled with –no-mark-wrap.

Marks the generated key with the CKA_WRAP flag.

mark-trusted option.

This is the “marks the object to be written as trusted” option.

This option has some usage constraints. It:

- can be disabled with –no-mark-trusted.
- must not appear in combination with any of the following options: mark-distrusted.

Marks the object to be generated/written with the CKA_TRUST flag.

mark-distrusted option.

This is the “when retrieving objects, it requires the objects to be distrusted (blacklisted)” option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: mark-trusted.

Ensures that the objects retrieved have the CKA_X_TRUST flag. This is p11-kit trust module extension, thus this flag is only valid with p11-kit registered trust modules.

mark-decrypt option.

This is the “marks the object to be written for decryption” option.

This option has some usage constraints. It:

- can be disabled with –no-mark-decrypt.

Marks the object to be generated/written with the CKA_DECRYPT flag set to true.

mark-sign option.

This is the “marks the object to be written for signature generation” option.
This option has some usage constraints. It:

- can be disabled with –no-mark-sign.

Marks the object to be generated/written with the CKA_SIGN flag set to true.

mark-ca option.

This is the “marks the object to be written as a ca” option.
This option has some usage constraints. It:

- can be disabled with –no-mark-ca.

Marks the object to be generated/written with the CKA_CERTIFICATE_CATEGORY as CA.

mark-private option.

This is the “marks the object to be written as private” option.
This option has some usage constraints. It:

- can be disabled with –no-mark-private.

Marks the object to be generated/written with the CKA_PRIVATE flag. The written object will require a PIN to be used.

ca option.

This is an alias for the `mark-ca` option, see [p11tool mark-ca], page 127.

private option.

This is an alias for the `mark-private` option, see [p11tool mark-private], page 127.

secret-key option.

This is the “provide a hex encoded secret key” option. This option takes a string argument.
This secret key will be written to the module if –write is specified.

other-options options

Other options.

debug option (-d).

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

so-login option.

This is the “force security officer login to token” option.
This option has some usage constraints. It:

- can be disabled with –no-so-login.

Forces login to the token as security officer (admin).

admin-login option.

This is an alias for the `so-login` option, see [p11tool so-login], page 127.

test-sign option.

This is the "tests the signature operation of the provided object" option. It can be used to test the correct operation of the signature operation. If both a private and a public key are available this operation will sign and verify the signed data.

sign-params option.

This is the "sign with a specific signature algorithm" option. This option takes a string argument. This option can be combined with –test-sign, to sign with a specific signature algorithm variant. The only option supported is 'RSA-PSS', and should be specified in order to use RSA-PSS signature on RSA keys.

hash option.

This is the "hash algorithm to use for signing" option. This option takes a string argument. This option can be combined with test-sign. Available hash functions are SHA1, RMD160, SHA256, SHA384, SHA512, SHA3-224, SHA3-256, SHA3-384, SHA3-512.

generate-random option.

This is the "generate random data" option. This option takes a number argument. Asks the token to generate a number of bytes of random bytes.

inder option.

This is the "use der/raw format for input" option.

This option has some usage constraints. It:

- can be disabled with –no-inder.

Use DER/RAW format for input certificates and private keys.

inraw option.

This is an alias for the `inder` option, see [p11tool inder], page 128.

outder option.

This is the "use der format for output certificates, private keys, and dh parameters" option.

This option has some usage constraints. It:

- can be disabled with –no-outder.

The output will be in DER or RAW format.

outraw option.

This is an alias for the `outder` option, see [p11tool outder], page 128.

provider option.

This is the "specify the pkcs #11 provider library" option. This option takes a file argument. This will override the default options in /etc/gnutls/pkcs11.conf

provider-opts option.

This is the "specify parameters for the pkcs #11 provider library" option. This option takes a string argument. This is a PKCS#11 internal option used by few modules. Mainly for testing PKCS#11 modules.

NOTE: THIS OPTION IS DEPRECATED

batch option.

This is the "disable all interaction with the tool" option. In batch mode there will be no prompts, all parameters need to be specified on command line.

p11tool exit status

One of the following exit values will be returned:

'`0 (EXIT_SUCCESS)`'
: Successful program execution.

'`1 (EXIT_FAILURE)`'
: The operation failed or the command syntax was not valid.

p11tool See Also

certtool (1)

p11tool Examples

To view all tokens in your system use:

```
$ p11tool --list-tokens
```

To view all objects in a token use:

```
$ p11tool --login --list-all "pkcs11:TOKEN-URL"
```

To store a private key and a certificate in a token run:

```
$ p11tool --login --write "pkcs11:URL" --load-privkey key.pem \
          --label "Mykey"
$ p11tool --login --write "pkcs11:URL" --load-certificate cert.pem \
          --label "Mykey"
```

Note that some tokens require the same label to be used for the certificate and its corresponding private key.

To generate an RSA private key inside the token use:

```
$ p11tool --login --generate-privkey rsa --bits 1024 --label "MyNewKey" \
          --outfile MyNewKey.pub "pkcs11:TOKEN-URL"
```

The bits parameter in the above example is explicitly set because some tokens only support limited choices in the bit length. The output file is the corresponding public key. This key can be used to general a certificate request with certtool.

```
certtool --generate-request --load-privkey "pkcs11:KEY-URL" \
```

```
--load-pubkey MyNewKey.pub --outfile request.pem
```

5.4 Trusted Platform Module (TPM)

In this section we present the Trusted Platform Module (TPM) support in GnuTLS. Note that we recommend against using TPM with this API because it is restricted to TPM 1.2. We recommend instead to use PKCS#11 wrappers for TPM such as CHAPS[6] or opencryptoki[7]. These will allow using the standard smart card and HSM functionality (see Section 5.3 [Smart cards and HSMs], page 110) for TPM keys.

There was a big hype when the TPM chip was introduced into computers. Briefly it is a co-processor in your PC that allows it to perform calculations independently of the main processor. This has good and bad side-effects. In this section we focus on the good ones; these are the fact that you can use the TPM chip to perform cryptographic operations on keys stored in it, without accessing them. That is very similar to the operation of a PKCS #11 smart card. The chip allows for storage and usage of RSA keys, but has quite some operational differences from PKCS #11 module, and thus require different handling. The basic TPM operations supported and used by GnuTLS, are key generation and signing. That support is currently limited to TPM 1.2.

The next sections assume that the TPM chip in the system is already initialized and in a operational state. If not, ensure that the TPM chip is enabled by your BIOS, that the `tcsd` daemon is running, and that TPM ownership is set (by running `tpm_takeownership`).

In GnuTLS the TPM functionality is available in `gnutls/tpm.h`.

5.4.1 Keys in TPM

The RSA keys in the TPM module may either be stored in a flash memory within TPM or stored in a file in disk. In the former case the key can provide operations as with PKCS #11 and is identified by a URL. The URL is described in [[TPMURI], page 753] and is of the following form.

```
tpmkey:uuid=42309df8-d101-11e1-a89a-97bb33c23ad1;storage=user
```

It consists from a unique identifier of the key as well as the part of the flash memory the key is stored at. The two options for the storage field are 'user' and 'system'. The user keys are typically only available to the generating user and the system keys to all users. The stored in TPM keys are called registered keys.

The keys that are stored in the disk are exported from the TPM but in an encrypted form. To access them two passwords are required. The first is the TPM Storage Root Key (SRK), and the other is a key-specific password. Also those keys are identified by a URL of the form:

```
tpmkey:file=/path/to/file
```

When objects require a PIN to be accessed the same callbacks as with PKCS #11 objects are expected (see Section 5.3.3 [Accessing objects that require a PIN], page 112). Note that the PIN function may be called multiple times to unlock the SRK and the specific key in use. The label in the key function will then be set to 'SRK' when unlocking the SRK key, or to 'TPM' when unlocking any other key.

---

[6] `https://github.com/google/chaps-linux`

[7] `https://sourceforge.net/projects/opencryptoki/`

5.4.2 Key generation

All keys used by the TPM must be generated by the TPM. This can be done using [gnutls_tpm_privkey_generate], page 671.

`int gnutls_tpm_privkey_generate (`*gnutls_pk_algorithm_t* `pk`, *unsigned int* `bits`, *const char ** `srk_password`, *const char ** `key_password`, *gnutls_tpmkey_fmt_t* `format`, *gnutls_x509_crt_fmt_t* `pub_format`, *gnutls_datum_t ** `privkey`, *gnutls_datum_t ** `pubkey`, *unsigned int* `flags`**)** [Function]

pk: the public key algorithm

bits: the security bits

srk_password: a password to protect the exported key (optional)

key_password: the password for the TPM (optional)

format: the format of the private key

pub_format: the format of the public key

privkey: the generated key

pubkey: the corresponding public key (may be null)

flags: should be a list of GNUTLS_TPM_* flags

This function will generate a private key in the TPM chip. The private key will be generated within the chip and will be exported in a wrapped with TPM's master key form. Furthermore the wrapped key can be protected with the provided `password` .

Note that bits in TPM is quantized value. If the input value is not one of the allowed values, then it will be quantized to one of 512, 1024, 2048, 4096, 8192 and 16384.

Allowed flags are:

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

int [gnutls_tpm_get_registered], page 670, (gnutls_tpm_key_list_t * *list*)
void [gnutls_tpm_key_list_deinit], page 670, (gnutls_tpm_key_list_t *list*)
int [gnutls_tpm_key_list_get_url], page 670, (gnutls_tpm_key_list_t *list*, unsigned int *idx*, char ** *url*, unsigned int *flags*)

`int gnutls_tpm_privkey_delete (`*const char ** `url`, *const char ** `srk_password`**)** [Function]

url: the URL describing the key

srk_password: a password for the SRK key

This function will unregister the private key from the TPM chip.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

5.4.3 Using keys

Importing keys

The TPM keys can be used directly by the abstract key types and do not require any special structures. Moreover functions like [gnutls_certificate_set_x509_key_file2], page 384 can access TPM URLs.

int [gnutls_privkey_import_tpm_raw], page 691, (gnutls_privkey_t *pkey*, const gnutls_datum_t * *fdata*, gnutls_tpmkey_fmt_t *format*, const char * *srk_password*, const char * *key_password*, unsigned int *flags*)
int [gnutls_pubkey_import_tpm_raw], page 707, (gnutls_pubkey_t *pkey*, const gnutls_datum_t * *fdata*, gnutls_tpmkey_fmt_t *format*, const char * *srk_password*, unsigned int *flags*)

`int gnutls_privkey_import_tpm_url` (*gnutls_privkey_t* `pkey`, *const char * url*, *const char * srk_password*, *const char * key_password*, *unsigned int flags*) [Function]

pkey: The private key

url: The URL of the TPM key to be imported

srk_password: The password for the SRK key (optional)

key_password: A password for the key (optional)

flags: One of the GNUTLS_PRIVKEY_* flags

This function will import the given private key to the abstract `gnutls_privkey_t` type.

Note that unless `GNUTLS_PRIVKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned and if the key password is wrong or not provided then `GNUTLS_E_TPM_KEY_PASSWORD_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

`int gnutls_pubkey_import_tpm_url` (*gnutls_pubkey_t* `pkey`, *const char * url*, *const char * srk_password*, *unsigned int flags*) [Function]

pkey: The public key

url: The URL of the TPM key to be imported

srk_password: The password for the SRK key (optional)

flags: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` type.

Note that unless `GNUTLS_PUBKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

Listing and deleting keys

The registered keys (that are stored in the TPM) can be listed using one of the following functions. Those keys are unfortunately only identified by their UUID and have no label or other human friendly identifier. Keys can be deleted from permanent storage using [gnutls_tpm_privkey_delete], page 670.

int [gnutls_tpm_get_registered], page 670, (gnutls_tpm_key_list_t * *list*)
void [gnutls_tpm_key_list_deinit], page 670, (gnutls_tpm_key_list_t *list*)
int [gnutls_tpm_key_list_get_url], page 670, (gnutls_tpm_key_list_t *list*, unsigned int *idx*, char ** *url*, unsigned int *flags*)

int gnutls_tpm_privkey_delete (*const char * url*, *const char * srk_password*) [Function]

url: the URL describing the key

srk_password: a password for the SRK key

This function will unregister the private key from the TPM chip.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.1.0

5.4.4 Invoking tpmtool

Program that allows handling cryptographic data from the TPM chip.

This section was generated by **AutoGen**, using the agtexi-cmd template and the option descriptions for the tpmtool program. This software is released under the GNU General Public License, version 3 or later.

tpmtool help/usage (--help)

This is the automatically generated usage text for tpmtool.

The text printed is the same whether selected with the help option (--help) or the more-help option (--more-help). more-help will print the usage text by passing it through a pager program. more-help is disabled on platforms without a working fork(2) function. The PAGER environment variable is used to select the program, defaulting to more. Both will exit with a status code of 0.

```
tpmtool - GnuTLS TPM tool
Usage:  tpmtool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --generate-rsa         Generate an RSA private-public key pair
       --register             Any generated key will be registered in the TPM
                                - requires the option 'generate-rsa'
       --signing              Any generated key will be a signing key
```

```
                              - requires the option 'generate-rsa'
                              -- and prohibits the option 'legacy'
      --legacy                Any generated key will be a legacy key
                              - requires the option 'generate-rsa'
                              -- and prohibits the option 'signing'
      --user                  Any registered key will be a user key
                              - requires the option 'register'
                              -- and prohibits the option 'system'
      --system                Any registered key will be a system key
                              - requires the option 'register'
                              -- and prohibits the option 'user'
      --pubkey=str            Prints the public key of the provided key
      --list                  Lists all stored keys in the TPM
      --delete=str            Delete the key identified by the given URL (UUID).
      --test-sign=str         Tests the signature operation of the provided object
      --sec-param=str         Specify the security level [low, legacy, medium, high, ultra]
      --bits=num              Specify the number of bits for key generate
      --inder                 Use the DER format for keys.
                              - disabled as '--no-inder'
      --outder                Use DER format for output keys
                              - disabled as '--no-outder'
      --srk-well-known        SRK has well known password (20 bytes of zeros)
   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Program that allows handling cryptographic data from the TPM chip.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

generate-rsa option

This is the "generate an rsa private-public key pair" option. Generates an RSA private-public key pair in the TPM chip. The key may be stored in file system and protected by a PIN, or stored (registered) in the TPM chip flash.

user option

This is the "any registered key will be a user key" option.

This option has some usage constraints. It:

- must appear in combination with the following options: register.

- must not appear in combination with any of the following options: system.

The generated key will be stored in a user specific persistent storage.

system option

This is the "any registered key will be a system key" option.

This option has some usage constraints. It:

- must appear in combination with the following options: register.
- must not appear in combination with any of the following options: user.

The generated key will be stored in system persistent storage.

test-sign option

This is the "tests the signature operation of the provided object" option. This option takes a string argument `url`. It can be used to test the correct operation of the signature operation. This operation will sign and verify the signed data.

sec-param option

This is the "specify the security level [low, legacy, medium, high, ultra]." option. This option takes a string argument `Security parameter`. This is alternative to the bits option. Note however that the values allowed by the TPM chip are quantized and given values may be rounded up.

inder option

This is the "use the der format for keys." option.

This option has some usage constraints. It:

- can be disabled with –no-inder.

The input files will be assumed to be in the portable DER format of TPM. The default format is a custom format used by various TPM tools

outder option

This is the "use der format for output keys" option.

This option has some usage constraints. It:

- can be disabled with –no-outder.

The output will be in the TPM portable DER format.

srk-well-known option

This is the "srk has well known password (20 bytes of zeros)" option. This option has no '`doc`' documentation.

tpmtool exit status

One of the following exit values will be returned:

‘`0 (EXIT_SUCCESS)`’
: Successful program execution.

‘`1 (EXIT_FAILURE)`’
: The operation failed or the command syntax was not valid.

tpmtool See Also

p11tool (1), certtool (1)

tpmtool Examples

To generate a key that is to be stored in file system use:

```
$ tpmtool --generate-rsa --bits 2048 --outfile tpmkey.pem
```

To generate a key that is to be stored in TPM's flash use:

```
$ tpmtool --generate-rsa --bits 2048 --register --user
```

To get the public key of a TPM key use:

```
$ tpmtool --pubkey tpmkey:uuid=58ad734b-bde6-45c7-89d8-756a55ad1891;storage=user \
          --outfile pubkey.pem
```

or if the key is stored in the file system:

```
$ tpmtool --pubkey tpmkey:file=tmpkey.pem --outfile pubkey.pem
```

To list all keys stored in TPM use:

```
$ tpmtool --list
```

6 How to use GnuTLS in applications

6.1 Introduction

This chapter tries to explain the basic functionality of the current GnuTLS library. Note that there may be additional functionality not discussed here but included in the library. Checking the header files in `/usr/include/gnutls/` and the manpages is recommended.

6.1.1 General idea

A brief description of how GnuTLS sessions operate is shown at Figure 6.1. This section will become more clear when it is completely read. As shown in the figure, there is a read-only global state that is initialized once by the global initialization function. This global structure, among others, contains the memory allocation functions used, structures needed for the ASN.1 parser and depending on the system's CPU, pointers to hardware accelerated encryption functions. This structure is never modified by any GnuTLS function, except for the deinitialization function which frees all allocated memory and must be called after the program has permanently finished using GnuTLS.

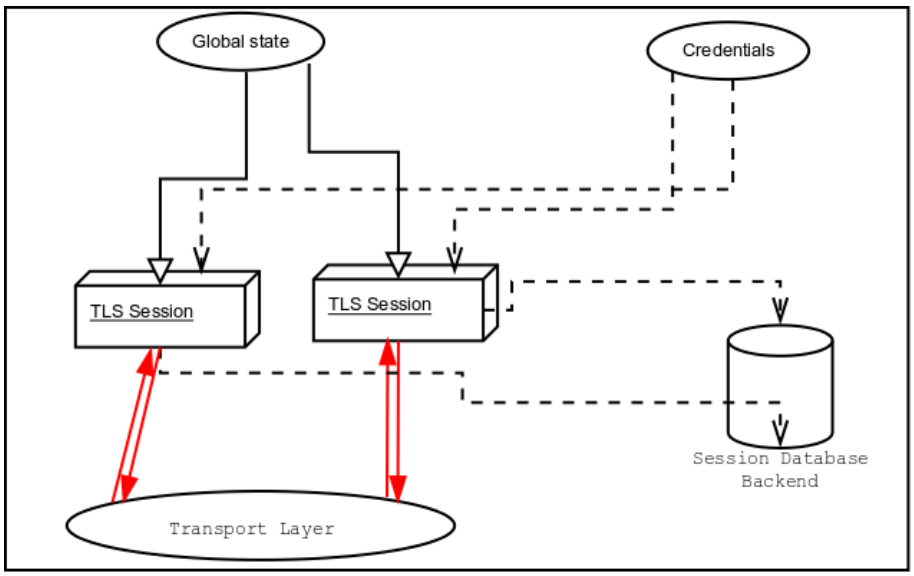

Figure 6.1: High level design of GnuTLS.

The credentials structures are used by the authentication methods, such as certificate authentication. They store certificates, privates keys, and other information that is needed to prove the identity to the peer, and/or verify the identity of the peer. The information stored in the credentials structures is initialized once and then can be shared by many TLS sessions.

A GnuTLS session contains all the required state and information to handle one secure connection. The session communicates with the peers using the provided functions of the transport layer. Every session has a unique session ID shared with the peer.

Since TLS sessions can be resumed, servers need a database back-end to hold the session's parameters. Every GnuTLS session after a successful handshake calls the appropriate back-end function (see [resume], page 11) to store the newly negotiated session. The session database is examined by the server just after having received the client hello[1], and if the session ID sent by the client, matches a stored session, the stored session will be retrieved, and the new session will be a resumed one, and will share the same session ID with the previous one.

6.1.2 Error handling

There two types of GnuTLS functions. The first type returns a boolean value, true (non-zero) or false (zero) value; these functions are defined to return an unsigned integer type. The other type returns a signed integer type with zero (or a positive number) indicating success and a negative value indicating failure. For the latter type it is recommended to check for errors as following.

```
ret = gnutls_function();
if (ret < 0) {
    return -1;
}
```

The above example checks for a failure condition rather than for explicit success (e.g., equality to zero). That has the advantage that future extensions of the API can be extended to provide additional information via positive returned values (see for example [gnutls_certificate_set_x509_key_file], page 383).

For certain operations such as TLS handshake and TLS packet receive there is the notion of fatal and non-fatal error codes. Fatal errors terminate the TLS session immediately and further sends and receives will be disallowed. Such an example is `GNUTLS_E_DECRYPTION_FAILED`. Non-fatal errors may warn about something, i.e., a warning alert was received, or indicate the some action has to be taken. This is the case with the error code `GNUTLS_E_REHANDSHAKE` returned by [gnutls_record_recv], page 450. This error code indicates that the server requests a re-handshake. The client may ignore this request, or may reply with an alert. You can test if an error code is a fatal one by using the [gnutls_error_is_fatal], page 406. All errors can be converted to a descriptive string using [gnutls_strerror], page 481.

If any non fatal errors, that require an action, are to be returned by a function, these error codes will be documented in the function's reference. For example the error codes `GNUTLS_E_WARNING_ALERT_RECEIVED` and `GNUTLS_E_FATAL_ALERT_RECEIVED` that may returned when receiving data, should be handled by notifying the user of the alert (as explained in Section 6.9 [Handling alerts], page 170). See Appendix C [Error codes], page 341, for a description of the available error codes.

6.1.3 Common types

All strings that are to provided as input to GnuTLS functions should be in UTF-8 unless otherwise specified. Output strings are also in UTF-8 format unless otherwise specified. When functions take as input passwords, they will normalize them using [[RFC7613], page 750] rules (since GnuTLS 3.5.7).

[1] The first message in a TLS handshake

When data of a fixed size are provided to GnuTLS functions then the helper structure `gnutls_datum_t` is often used. Its definition is shown below.

```
typedef struct
{
  unsigned char *data;
  unsigned int size;
} gnutls_datum_t;
```

In functions where this structure is a returned type, if the function succeeds, it is expected from the caller to use `gnutls_free()` to deinitialize the data element after use, unless otherwise specified. If the function fails, the contents of the `gnutls_datum_t` should be considered undefined and must not be deinitialized.

Other functions that require data for scattered read use a structure similar to `struct iovec` typically used by `readv`. It is shown below.

```
typedef struct
{
  void *iov_base;             /* Starting address */
  size_t iov_len;             /* Number of bytes to transfer */
} giovec_t;
```

6.1.4 Debugging and auditing

In many cases things may not go as expected and further information, to assist debugging, from GnuTLS is desired. Those are the cases where the [gnutls_global_set_log_level], page 412 and [gnutls_global_set_log_function], page 412 are to be used. Those will print verbose information on the GnuTLS functions internal flow.

void [gnutls_global_set_log_level], page 412, (int *level*)
void [gnutls_global_set_log_function], page 412, (gnutls_log_func *log_func*)

Alternatively the environment variable `GNUTLS_DEBUG_LEVEL` can be set to a logging level and GnuTLS will output debugging output to standard error. Other available environment variables are shown in Table 6.1.

Variable	Purpose
GNUTLS_DEBUG_LEVEL	When set to a numeric value, it sets the default debugging level for GnuTLS applications.
SSLKEYLOGFILE	When set to a filename, GnuTLS will append to it the session keys in the NSS Key Log format. That format can be read by wireshark and will allow decryption of the session for debugging.
GNUTLS_CPUID_OVERRIDE	That environment variable can be used to explicitly enable/disable the use of certain CPU capabilities. Note that CPU detection cannot be overridden, i.e., VIA options cannot be enabled on an Intel CPU. The currently available options are: • 0x1: Disable all run-time detected optimizations • 0x2: Enable AES-NI • 0x4: Enable SSSE3 • 0x8: Enable PCLMUL • 0x10: Enable AVX • 0x20: Enable SHA_NI • 0x100000: Enable VIA padlock • 0x200000: Enable VIA PHE • 0x400000: Enable VIA PHE SHA512
GNUTLS_FORCE_FIPS_MODE	In setups where GnuTLS is compiled with support for FIPS140-2 (see Section 11.7 [FIPS140-2 mode], page 329) if set to one it will force the FIPS mode enablement.

Table 6.1: Environment variables used by the library.

When debugging is not required, important issues, such as detected attacks on the protocol still need to be logged. This is provided by the logging function set by [gnutls_global_set_audit_log_function], page 411. The provided function will receive an message and the corresponding TLS session. The session information might be used to derive IP addresses or other information about the peer involved.

`void gnutls_global_set_audit_log_function` [Function]
(*gnutls_audit_log_func* `log_func`)

log_func: it is the audit log function

This is the function to set the audit logging function. This is a function to report important issues, such as possible attacks in the protocol. This is different from `gnutls_`

`global_set_log_function()` because it will report also session-specific events. The session parameter will be null if there is no corresponding TLS session.

`gnutls_audit_log_func` is of the form, void (*gnutls_audit_log_func)(gnutls_session_t, const char*);

Since: 3.0

6.1.5 Thread safety

The GnuTLS library is thread safe by design, meaning that objects of the library such as TLS sessions, can be safely divided across threads as long as a single thread accesses a single object. This is sufficient to support a server which handles several sessions per thread. Read-only access to objects, for example the credentials holding structures, is also thread-safe.

A `gnutls_session_t` object could also be shared by two threads, one sending, the other receiving. However, care must be taken on the following use cases:

- The re-handshake process in TLS 1.2 or earlier must be handled only in a single thread and no other thread may be performing any operation.
- The flag `GNUTLS_AUTO_REAUTH` cannot be used safely in this mode of operation.
- Any other operation which may send or receive data, like key update (c.f., [gnutls_session_key_update], page 464), must not be performed while threads are receiving or writing.
- The termination of a session should be handled, either by a single thread being active, or by the sender thread using [gnutls_bye], page 367 with `GNUTLS_SHUT_WR` and the receiving thread waiting for a return value of zero (or timeout on certain servers which do not respond).
- The functions [gnutls_transport_set_errno], page 485 and [gnutls_record_get_direction], page 449 should not be relied during parallel operation.

For several aspects of the library (e.g., the random generator, PKCS#11 operations), the library may utilize mutex locks (e.g., pthreads on GNU/Linux and CriticalSection on Windows) which are transparently setup on library initialization. Prior to version 3.3.0 these were setup by explicitly calling [gnutls_global_init], page 411.[2]

Note that, on Glibc systems, unless the application is explicitly linked with the libpthread library, no mutex locks are used and setup by GnuTLS. It will use the Glibc mutex stubs.

6.1.6 Running in a sandbox

Given that TLS protocol handling as well as X.509 certificate parsing are complicated processes involving several thousands lines of code, it is often desirable (and recommended) to run the TLS session handling in a sandbox like seccomp. That has to be allowed by the overall software design, but if available, it adds an additional layer of protection by preventing parsing errors from becoming vessels for further security issues such as code execution.

[2] On special systems you could manually specify the locking system using the function [gnutls_global_set_mutex], page 412 before calling any other GnuTLS function. Setting mutexes manually is not recommended.

GnuTLS requires the following system calls to be available for its proper operation.

- nanosleep
- time
- gettimeofday
- clock_gettime
- getrusage
- getpid
- send
- recv
- sendmsg
- read (to read from /dev/urandom)
- getrandom (this is Linux-kernel specific)
- poll

As well as any calls needed for memory allocation to work. Note however, that GnuTLS depends on libc for the system calls, and there is no guarantee that libc will call the expected system call. For that it is recommended to test your program in all the targeted platforms when filters like seccomp are in place.

An example with a seccomp filter from GnuTLS' test suite is at: `https://gitlab.com/gnutls/gnutls/blob/master/tests/seccomp.c`.

6.1.7 Sessions and fork

A `gnutls_session_t` object can be shared by two processes after a fork, one sending, the other receiving. In that case rehandshakes, cannot and must not be performed. As with threads, the termination of a session should be handled by the sender process using [gnutls_bye], page 367 with `GNUTLS_SHUT_WR` and the receiving process waiting for a return value of zero.

6.1.8 Callback functions

There are several cases where GnuTLS may need out of band input from your program. This is now implemented using some callback functions, which your program is expected to register.

An example of this type of functions are the push and pull callbacks which are used to specify the functions that will retrieve and send data to the transport layer.

void [gnutls_transport_set_push_function], page 488, (gnutls_session_t *session*, gnutls_push_func *push_func*)
void [gnutls_transport_set_pull_function], page 487, (gnutls_session_t *session*, gnutls_pull_func *pull_func*)

Other callback functions may require more complicated input and data to be allocated. Such an example is [gnutls_srp_set_server_credentials_function], page 476. All callbacks should allocate and free memory using `gnutls_malloc` and `gnutls_free`.

6.2 Preparation

To use GnuTLS, you have to perform some changes to your sources and your build system. The necessary changes are explained in the following subsections.

6.2.1 Headers

All the data types and functions of the GnuTLS library are defined in the header file `gnutls/gnutls.h`. This must be included in all programs that make use of the GnuTLS library.

6.2.2 Initialization

The GnuTLS library is initialized on load; prior to 3.3.0 was initialized by calling [gnutls_global_init], page 411[3]. [gnutls_global_init], page 411 in versions after 3.3.0 is thread-safe (see Section 6.1.5 [Thread safety], page 141).

The initialization typically enables CPU-specific acceleration, performs any required precalculations needed, opens any required system devices (e.g., /dev/urandom on Linux) and initializes subsystems that could be used later.

The resources allocated by the initialization process will be released on library deinitialization.

Note that on certain systems file descriptors may be kept open by GnuTLS (e.g. /dev/urandom) on library load. Applications closing all unknown file descriptors must immediately call [gnutls_global_init], page 411, after that, to ensure they don't disrupt GnuTLS' operation.

6.2.3 Version check

It is often desirable to check that the version of 'gnutls' used is indeed one which fits all requirements. Even with binary compatibility new features may have been introduced but due to problem with the dynamic linker an old version is actually used. So you may want to check that the version is okay right after program start-up. See the function [gnutls_check_version], page 393.

On the other hand, it is often desirable to support more than one versions of the library. In that case you could utilize compile-time feature checks using the `GNUTLS_VERSION_NUMBER` macro. For example, to conditionally add code for GnuTLS 3.2.1 or later, you may use:

```
#if GNUTLS_VERSION_NUMBER >= 0x030201
  ...
#endif
```

6.2.4 Building the source

If you want to compile a source file including the `gnutls/gnutls.h` header file, you must make sure that the compiler can find it in the directory hierarchy. This is accomplished by adding the path to the directory in which the header file is located to the compilers include file search path (via the `-I` option).

[3] The original behavior of requiring explicit initialization can obtained by setting the GNUTLS_NO_EXPLICIT_INIT environment variable to 1, or by using the macro GNUTLS_SKIP_GLOBAL_INIT in a global section of your program –the latter works in systems with support for weak symbols only.

However, the path to the include file is determined at the time the source is configured. To solve this problem, the library uses the external package `pkg-config` that knows the path to the include file and other configuration options. The options that need to be added to the compiler invocation at compile time are output by the `--cflags` option to `pkg-config gnutls`. The following example shows how it can be used at the command line:

```
gcc -c foo.c `pkg-config gnutls --cflags`
```

Adding the output of '`pkg-config gnutls --cflags`' to the compilers command line will ensure that the compiler can find the `gnutls/gnutls.h` header file.

A similar problem occurs when linking the program with the library. Again, the compiler has to find the library files. For this to work, the path to the library files has to be added to the library search path (via the `-L` option). For this, the option `--libs` to `pkg-config gnutls` can be used. For convenience, this option also outputs all other options that are required to link the program with the library (for instance, the '`-ltasn1`' option). The example shows how to link `foo.o` with the library to a program `foo`.

```
gcc -o foo foo.o `pkg-config gnutls --libs`
```

Of course you can also combine both examples to a single command by specifying both options to `pkg-config`:

```
gcc -o foo foo.c `pkg-config gnutls --cflags --libs`
```

When a program uses the GNU autoconf system, then the following line or similar can be used to detect the presence of GnuTLS.

```
PKG_CHECK_MODULES([LIBGNUTLS], [gnutls >= 3.3.0])

AC_SUBST([LIBGNUTLS_CFLAGS])
AC_SUBST([LIBGNUTLS_LIBS])
```

6.3 Session initialization

In the previous sections we have discussed the global initialization required for GnuTLS as well as the initialization required for each authentication method's credentials (see Section 3.5.2 [Authentication], page 11). In this section we elaborate on the TLS or DTLS session initiation. Each session is initialized using [gnutls_init], page 422 which among others is used to specify the type of the connection (server or client), and the underlying protocol type, i.e., datagram (UDP) or reliable (TCP).

`int gnutls_init` (*gnutls_session_t * `session`, unsigned int `flags`*) [Function]

session: is a pointer to a `gnutls_session_t` type.

flags: indicate if this session is to be used for server or client.

This function initializes the provided session. Every session must be initialized before use, and must be deinitialized after used by calling `gnutls_deinit()` .

`flags` can be any combination of flags from `gnutls_init_flags_t` .

Note that since version 3.1.2 this function enables some common TLS extensions such as session tickets and OCSP certificate status request in client side by default. To prevent that use the `GNUTLS_NO_EXTENSIONS` flag.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

`GNUTLS_SERVER`
: Connection end is a server.

`GNUTLS_CLIENT`
: Connection end is a client.

`GNUTLS_DATAGRAM`
: Connection is datagram oriented (DTLS). Since 3.0.0.

`GNUTLS_NONBLOCK`
: Connection should not block. Since 3.0.0.

`GNUTLS_NO_EXTENSIONS`
: Do not enable any TLS extensions by default (since 3.1.2). As TLS 1.2 and later require extensions this option is considered obsolete and should not be used.

`GNUTLS_NO_REPLAY_PROTECTION`
: Disable any replay protection in DTLS. This must only be used if replay protection is achieved using other means. Since 3.2.2.

`GNUTLS_NO_SIGNAL`
: In systems where SIGPIPE is delivered on send, it will be disabled. That flag has effect in systems which support the MSG_NOSIGNAL sockets flag (since 3.4.2).

`GNUTLS_ALLOW_ID_CHANGE`
: Allow the peer to replace its certificate, or change its ID during a rehandshake. This change is often used in attacks and thus prohibited by default. Since 3.5.0.

`GNUTLS_ENABLE_FALSE_START`
: Enable the TLS false start on client side if the negotiated ciphersuites allow it. This will enable sending data prior to the handshake being complete, and may introduce a risk of crypto failure when combined with certain key exchanged; for that GnuTLS may not enable that option in ciphersuites that are known to be not safe for false start. Since 3.5.0.

`GNUTLS_FORCE_CLIENT_CERT`
: When in client side and only a single cert is specified, send that certificate irrespective of the issuers expected by the server. Since 3.5.0.

`GNUTLS_NO_TICKETS`
: Flag to indicate that the session should not use resumption with session tickets.

`GNUTLS_KEY_SHARE_TOP`
: Generate key share for the first group which is enabled. For example x25519. This option is the most performant for client (less CPU spent generating keys), but if the server doesn't support the advertized option it may result to more roundtrips needed to discover the server's choice.

`GNUTLS_KEY_SHARE_TOP2`
: Generate key shares for the top-2 different groups which are enabled. For example (ECDH + x25519). This is the default.

`GNUTLS_KEY_SHARE_TOP3`
: Generate key shares for the top-3 different groups which are enabled. That is, as each group is associated with a key type (EC, finite field, x25519), generate three keys using `GNUTLS_PK_DH` , `GNUTLS_PK_EC` , `GNUTLS_PK_ECDH_X25519` if all of them are enabled.

After the session initialization details on the allowed ciphersuites and protocol versions should be set using the priority functions such as [gnutls_priority_set], page 439 and [gnutls_priority_set_direct], page 439. We elaborate on them in Section 6.10 [Priority Strings], page 172. The credentials used for the key exchange method, such as certificates or usernames and passwords should also be associated with the session current session using [gnutls_credentials_set], page 396.

`int gnutls_credentials_set` (*gnutls_session_t* `session`, *gnutls_credentials_type_t* `type`, *void ** `cred`) [Function]

session: is a `gnutls_session_t` type.

type: is the type of the credentials

cred: the credentials to set

Sets the needed credentials for the specified type. E.g. username, password - or public and private keys etc. The `cred` parameter is a structure that depends on the specified type and on the current session (client or server).

In order to minimize memory usage, and share credentials between several threads gnutls keeps a pointer to cred, and not the whole cred structure. Thus you will have to keep the structure allocated until you call `gnutls_deinit()` .

For `GNUTLS_CRD_ANON` , `cred` should be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , `cred` should be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , `cred` should be `gnutls_certificate_credentials_t` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

6.4 Associating the credentials

Each authentication method is associated with a key exchange method, and a credentials type. The contents of the credentials is method-dependent, e.g. certificates for certificate authentication and should be initialized and associated with a session (see [gnutls_credentials_set], page 396). A mapping of the key exchange methods with the credential types is shown in Table 6.2.

Authentication method	Key exchange	Client credentials	Server credentials
Certificate and Raw public-key	KX_RSA, KX_DHE_RSA, KX_DHE_DSS, KX_ECDHE_RSA, KX_ECDHE_ECDSA	CRD_CERTIFICATE	CRD_CERTIFICATE
Password and certificate	KX_SRP_RSA, KX_SRP_DSS	CRD_SRP	CRD_CERTIFICATE, CRD_SRP
Password	KX_SRP	CRD_SRP	CRD_SRP
Anonymous	KX_ANON_DH, KX_ANON_ECDH	CRD_ANON	CRD_ANON
Pre-shared key	KX_PSK, KX_DHE_PSK, KX_ECDHE_PSK	CRD_PSK	CRD_PSK

Table 6.2: Key exchange algorithms and the corresponding credential types.

6.4.1 Certificates

Server certificate authentication

When using certificates the server is required to have at least one certificate and private key pair. Clients may not hold such a pair, but a server could require it. In this section we discuss general issues applying to both client and server certificates. The next section will elaborate on issues arising from client authentication only.

In order to use certificate credentials one must first initialize a credentials structure of type `gnutls_certificate_credentials_t`. After use this structure must be freed. This can be done with the following functions.

int [gnutls_certificate_allocate_credentials], page 367,
(gnutls_certificate_credentials_t * *res*)
void [gnutls_certificate_free_credentials], page 368,
(gnutls_certificate_credentials_t *sc*)

After the credentials structures are initialized, the certificate and key pair must be loaded. This occurs before any TLS session is initialized, and the same structures are reused for multiple sessions. Depending on the certificate type different loading functions are available, as shown below. For X.509 certificates, the functions will accept and use a certificate chain that leads to a trusted authority. The certificate chain must be ordered in such way that every certificate certifies the one before it. The trusted authority's certificate need not to be included since the peer should possess it already.

int [gnutls_certificate_set_x509_key_file2], page 384,
(gnutls_certificate_credentials_t *res*, const char * *certfile*, const char *
keyfile, gnutls_x509_crt_fmt_t *type*, const char * *pass*, unsigned int *flags*)
int [gnutls_certificate_set_x509_key_mem2], page 385,
(gnutls_certificate_credentials_t *res*, const gnutls_datum_t * *cert*, const
gnutls_datum_t * *key*, gnutls_x509_crt_fmt_t *type*, const char * *pass*, unsigned
int *flags*)
int [gnutls_certificate_set_x509_key], page 382,
(gnutls_certificate_credentials_t *res*, gnutls_x509_crt_t * *cert_list*, int
cert_list_size, gnutls_x509_privkey_t *key*)

It is recommended to use the higher level functions such as [gnutls_certificate_set_x509_key_file2], page 384 which accept not only file names but URLs that specify objects stored in token, or system certificates and keys (see Section 5.2 [Application-specific keys], page 108). For these cases, another important function is [gnutls_certificate_set_pin_function], page 377, that allows setting a callback function to retrieve a PIN if the input keys are protected by PIN.

`void gnutls_certificate_set_pin_function` [Function]
(*gnutls_certificate_credentials_t* **cred**, *gnutls_pin_callback_t* **fn**, *void *
userdata)

cred: is a `gnutls_certificate_credentials_t` type.

fn: A PIN callback

userdata: Data to be passed in the callback

This function will set a callback function to be used when required to access a protected object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

Since: 3.1.0

If the imported keys and certificates need to be accessed before any TLS session is established, it is convenient to use [gnutls_certificate_set_key], page 671 in combination with [gnutls_pcert_import_x509_raw], page 677 and [gnutls_privkey_import_x509_raw], page 693.

`int gnutls_certificate_set_key` (*gnutls_certificate_credentials_t* [Function]
res, *const char ** **names**, *int* **names_size**, *gnutls_pcert_st *
pcert_list, *int* **pcert_list_size**, *gnutls_privkey_t* **key**)

res: is a `gnutls_certificate_credentials_t` type.

names: is an array of DNS names belonging to the public-key (NULL if none)

names_size: holds the size of the names list

pcert_list: contains a certificate list (chain) or raw public-key

pcert_list_size: holds the size of the certificate list

key: is a `gnutls_privkey_t` key corresponding to the first public-key in pcert_list

This function sets a public/private key pair in the gnutls_certificate_credentials_t type. The given public key may be encapsulated in a certificate or can be given as a raw key. This function may be called more than once, in case multiple key pairs exist for the server. For clients that want to send more than their own end-

entity certificate (e.g., also an intermediate CA cert), the full certificate chain must be provided in `pcert_list` .

Note that the `key` will become part of the credentials structure and must not be deallocated. It will be automatically deallocated when the `res` structure is deinitialized.

If this function fails, the `res` structure is at an undefined state and it must not be reused to load other keys or certificates.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used for other functions to refer to the added key-pair.

Since GnuTLS 3.6.6 this function also handles raw public keys.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

Since: 3.0

If multiple certificates are used with the functions above each client's request will be served with the certificate that matches the requested name (see Section 3.6.2 [Server name indication], page 12).

As an alternative to loading from files or buffers, a callback may be used for the server or the client to specify the certificate and the key at the handshake time. In that case a certificate should be selected according the peer's signature algorithm preferences. To get those preferences use [gnutls_sign_algorithm_get_requested], page 470. Both functions are shown below.

```
void [gnutls_certificate_set_retrieve_function], page 380,
(gnutls_certificate_credentials_t cred, gnutls_certificate_retrieve_function
* func)
void [gnutls_certificate_set_retrieve_function2], page 672,
(gnutls_certificate_credentials_t cred,
gnutls_certificate_retrieve_function2 * func)
void [gnutls_certificate_set_retrieve_function3], page 673,
(gnutls_certificate_credentials_t cred,
gnutls_certificate_retrieve_function3 * func)
int [gnutls_sign_algorithm_get_requested], page 470, (gnutls_session_t
session, size_t indx, gnutls_sign_algorithm_t * algo)
```

The functions above do not handle the requested server name automatically. A server would need to check the name requested by the client using [gnutls_server_name_get], page 457, and serve the appropriate certificate. Note that some of these functions require the `gnutls_pcert_st` structure to be filled in. Helper functions to fill in the structure are listed below.

```
typedef struct gnutls_pcert_st
{
  gnutls_pubkey_t pubkey;
  gnutls_datum_t cert;
  gnutls_certificate_type_t type;
```

```
} gnutls_pcert_st;
```

int [gnutls_pcert_import_x509], page 676, (gnutls_pcert_st * *pcert*, gnutls_x509_crt_t *crt*, unsigned int *flags*)
int [gnutls_pcert_import_x509_raw], page 677, (gnutls_pcert_st * *pcert*, const gnutls_datum_t * *cert*, gnutls_x509_crt_fmt_t *format*, unsigned int *flags*)
void [gnutls_pcert_deinit], page 674, (gnutls_pcert_st * *pcert*)

In a handshake, the negotiated cipher suite depends on the certificate's parameters, so some key exchange methods might not be available with all certificates. GnuTLS will disable ciphersuites that are not compatible with the key, or the enabled authentication methods. For example keys marked as sign-only, will not be able to access the plain RSA ciphersuites, that require decryption. It is not recommended to use RSA keys for both signing and encryption. If possible use a different key for the `DHE-RSA` which uses signing and `RSA` that requires decryption. All the key exchange methods shown in Table 4.1 are available in certificate authentication.

Client certificate authentication

If a certificate is to be requested from the client during the handshake, the server will send a certificate request message. This behavior is controlled by [gnutls_certificate_server_set_request], page 373. The request contains a list of the by the server accepted certificate signers. This list is constructed using the trusted certificate authorities of the server. In cases where the server supports a large number of certificate authorities it makes sense not to advertise all of the names to save bandwidth. That can be controlled using the function [gnutls_certificate_send_x509_rdn_sequence], page 372. This however will have the side-effect of not restricting the client to certificates signed by server's acceptable signers.

`void gnutls_certificate_server_set_request` (*gnutls_session_t* `session`, *gnutls_certificate_request_t* `req`) [Function]

session: is a `gnutls_session_t` type.

req: is one of GNUTLS_CERT_REQUEST, GNUTLS_CERT_REQUIRE, GNUTLS_CERT_IGNORE

This function specifies if we (in case of a server) are going to send a certificate request message to the client. If `req` is GNUTLS_CERT_REQUIRE then the server will return the `GNUTLS_E_NO_CERTIFICATE_FOUND` error if the peer does not provide a certificate. If you do not call this function then the client will not be asked to send a certificate. Invoking the function with `req` GNUTLS_CERT_IGNORE has the same effect.

`void gnutls_certificate_send_x509_rdn_sequence` (*gnutls_session_t* `session`, *int* `status`) [Function]

session: a `gnutls_session_t` type.

status: is 0 or 1

If status is non zero, this function will order gnutls not to send the rdnSequence in the certificate request message. That is the server will not advertise its trusted CAs to the peer. If status is zero then the default behaviour will take effect, which is to advertise the server's trusted CAs.

This function has no effect in clients, and in authentication methods other than certificate with X.509 certificates.

On the client side, it needs to set its certificates on the credentials structure, similarly to server side from a file, or via a callback. Once the certificates are available in the credentials structure, the client will send them if during the handshake the server requests a certificate signed by the issuer of its CA.

In the case a single certificate is available and the server does not specify a signer's list, then that certificate is always sent. It is, however possible, to send a certificate even when the advertised CA list by the server contains CAs other than its signer. That can be achieved using the `GNUTLS_FORCE_CLIENT_CERT` flag in [gnutls_init], page 422.

int [gnutls_certificate_set_x509_key_file], page 383,
(gnutls_certificate_credentials_t *res*, const char * *certfile*, const char * *keyfile*, gnutls_x509_crt_fmt_t *type*)
int [gnutls_certificate_set_x509_simple_pkcs12_file], page 386,
(gnutls_certificate_credentials_t *res*, const char * *pkcs12file*, gnutls_x509_crt_fmt_t *type*, const char * *password*)
void [gnutls_certificate_set_retrieve_function2], page 672,
(gnutls_certificate_credentials_t *cred*, gnutls_certificate_retrieve_function2 * *func*)

Client or server certificate verification

Certificate verification is possible by loading the trusted authorities into the credentials structure by using the following functions, applicable to X.509 certificates. In modern systems it is recommended to utilize [gnutls_certificate_set_x509_system_trust], page 387 which will load the trusted authorities from the system store.

`int gnutls_certificate_set_x509_system_trust` [Function]
(*gnutls_certificate_credentials_t* *cred*)

cred: is a `gnutls_certificate_credentials_t` type.

This function adds the system's default trusted CAs in order to verify client or server certificates.

In the case the system is currently unsupported `GNUTLS_E_UNIMPLEMENTED_FEATURE` is returned.

Returns: the number of certificates processed or a negative error code on error.

Since: 3.0.20

int [gnutls_certificate_set_x509_trust_file], page 388,
(gnutls_certificate_credentials_t *cred*, const char * *cafile*, gnutls_x509_crt_fmt_t *type*)
int [gnutls_certificate_set_x509_trust_dir], page 388,
(gnutls_certificate_credentials_t *cred*, const char * *ca_dir*, gnutls_x509_crt_fmt_t *type*)

The peer's certificate will be automatically verified if [gnutls_session_set_verify_cert], page 466 is called prior to handshake.

Alternatively, one must set a callback function during the handshake using [gnutls_certificate_set_verify_function], page 381, which will verify the peer's certificate once received. The verification should happen using [gnutls_certificate_verify_peers3], page 392 within the callback. It will verify the certificate's signature and the owner of the certificate. That will provide a brief verification output. If a detailed output is required one should call [gnutls_certificate_get_peers], page 371 to obtain the raw certificate of the peer and verify it using the functions discussed in Section 4.1.1 [X.509 certificates], page 21.

In both the automatic and the manual cases, the verification status returned can be printed using [gnutls_certificate_verification_status_print], page 390.

`void gnutls_session_set_verify_cert` (*gnutls_session_t* `session`, *const char* * `hostname`, *unsigned* `flags`) [Function]

session: is a gnutls session

hostname: is the expected name of the peer; may be `NULL`

flags: flags for certificate verification – `gnutls_certificate_verify_flags`

This function instructs GnuTLS to verify the peer's certificate using the provided hostname. If the verification fails the handshake will also fail with `GNUTLS_E_CERTIFICATE_VERIFICATION_ERROR` . In that case the verification result can be obtained using `gnutls_session_get_verify_cert_status()` .

The `hostname` pointer provided must remain valid for the lifetime of the session. More precisely it should be available during any subsequent handshakes. If no hostname is provided, no hostname verification will be performed. For a more advanced verification function check `gnutls_session_set_verify_cert2()` .

If `flags` is provided which contain a profile, this function should be called after any session priority setting functions.

The `gnutls_session_set_verify_cert()` function is intended to be used by TLS clients to verify the server's certificate.

Since: 3.4.6

int [gnutls_certificate_verify_peers3], page 392, (gnutls_session_t *session*, const char * *hostname*, unsigned int * *status*)
void [gnutls_certificate_set_verify_function], page 381, (gnutls_certificate_credentials_t *cred*, gnutls_certificate_verify_function * *func*)

Note that when using raw public-keys verification will not work because there is no corresponding certificate body belonging to the raw key that can be verified. In that case the [gnutls_certificate_verify_peers], page 391 family of functions will return a GNUTLS_E_INVALID_REQUEST error code. For authenticating raw public-keys one must use an out-of-band mechanism, e.g. by comparing hashes or using trust on first use (see Section 4.1.4.1 [Verifying a certificate using trust on first use authentication], page 40).

6.4.2 Raw public-keys

As of version 3.6.6 GnuTLS supports Section 4.1.3 [Raw public-keys], page 40. With raw public-keys only the public-key part (that is normally embedded in a certificate) is trans-

mitted to the peer. In order to load a raw public-key and its corresponding private key in a credentials structure one can use the following functions.

int [gnutls_certificate_set_key], page 671,
(gnutls_certificate_credentials_t *res*, const char ** *names*, int *names_size*, gnutls_pcert_st * *pcert_list*, int *pcert_list_size*, gnutls_privkey_t *key*)
int [gnutls_certificate_set_rawpk_key_mem], page 379,
(gnutls_certificate_credentials_t *cred*, const gnutls_datum_t* *spki*, const gnutls_datum_t* *pkey*, gnutls_x509_crt_fmt_t *format*, const char* *pass*, unsigned int *key_usage*, const char ** *names*, unsigned int *names_length*, unsigned int *flags*)
int [gnutls_certificate_set_rawpk_key_file], page 377,
(gnutls_certificate_credentials_t *cred*, const char* *rawpkfile*, const char* *privkeyfile*, gnutls_x509_crt_fmt_t *format*, const char * *pass*, unsigned int *key_usage*, const char ** *names*, unsigned int *names_length*, unsigned int *privkey_flags*, unsigned int *pkcs11_flags*)

6.4.3 SRP

The initialization functions in SRP credentials differ between client and server. Clients supporting SRP should set the username and password prior to connection, to the credentials structure. Alternatively [gnutls_srp_set_client_credentials_function], page 475 may be used instead, to specify a callback function that should return the SRP username and password. The callback is called once during the TLS handshake.

int [gnutls_srp_allocate_server_credentials], page 473,
(gnutls_srp_server_credentials_t * *sc*)
int [gnutls_srp_allocate_client_credentials], page 473,
(gnutls_srp_client_credentials_t * *sc*)
void [gnutls_srp_free_server_credentials], page 474,
(gnutls_srp_server_credentials_t *sc*)
void [gnutls_srp_free_client_credentials], page 474,
(gnutls_srp_client_credentials_t *sc*)
int [gnutls_srp_set_client_credentials], page 475,
(gnutls_srp_client_credentials_t *res*, const char * *username*, const char * *password*)

void gnutls_srp_set_client_credentials_function [Function]
(*gnutls_srp_client_credentials_t* **cred**,
gnutls_srp_client_credentials_function * **func**)

cred: is a gnutls_srp_server_credentials_t type.

func: is the callback function

This function can be used to set a callback to retrieve the username and password for client SRP authentication. The callback's function form is:

int (*callback)(gnutls_session_t, char** username, char**password);

The username and password must be allocated using gnutls_malloc() .

The username should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265).

The password can be in ASCII format, or normalized using `gnutls_utf8_password_normalize()` .

The callback function will be called once per handshake before the initial hello message is sent.

The callback should not return a negative error code the second time called, since the handshake procedure will be aborted.

The callback function should return 0 on success. -1 indicates an error.

In server side the default behavior of GnuTLS is to read the usernames and SRP verifiers from password files. These password file format is compatible the with the *Stanford srp libraries* format. If a different password file format is to be used, then [gnutls_srp_set_server_credentials_function], page 476 should be called, to set an appropriate callback.

`int gnutls_srp_set_server_credentials_file` [Function]
(*gnutls_srp_server_credentials_t* ***res***, *const char ** ***password_file***, *const char ** ***password_conf_file***)

res: is a `gnutls_srp_server_credentials_t` type.

password_file: is the SRP password file (tpasswd)

password_conf_file: is the SRP password conf file (tpasswd.conf)

This function sets the password files, in a `gnutls_srp_server_credentials_t` type. Those password files hold usernames and verifiers and will be used for SRP authentication.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

`void gnutls_srp_set_server_credentials_function` [Function]
(*gnutls_srp_server_credentials_t* ***cred***,
*gnutls_srp_server_credentials_function ** ***func***)

cred: is a `gnutls_srp_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the user's SRP credentials. The callback's function form is:

int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t *salt, gnutls_datum_t *verifier, gnutls_datum_t *generator, gnutls_datum_t *prime);

`username` contains the actual username. The `salt` , `verifier` , `generator` and `prime` must be filled in using the `gnutls_malloc()` . For convenience `prime` and `generator` may also be one of the static parameters defined in gnutls.h.

Initially, the data field is NULL in every `gnutls_datum_t` structure that the callback has to fill in. When the callback is done GnuTLS deallocates all of those buffers which are non-NULL, regardless of the return value.

In order to prevent attackers from guessing valid usernames, if a user does not exist, g and n values should be filled in using a random user's parameters. In that case the callback must return the special value (1). See `gnutls_srp_set_server_fake_salt_seed` too. If this is not required for your application, return a negative number from the callback to abort the handshake.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

6.4.4 PSK

The initialization functions in PSK credentials differ between client and server.

int [gnutls_psk_allocate_server_credentials], page 441,
(gnutls_psk_server_credentials_t * *sc*)
int [gnutls_psk_allocate_client_credentials], page 441,
(gnutls_psk_client_credentials_t * *sc*)
void [gnutls_psk_free_server_credentials], page 441,
(gnutls_psk_server_credentials_t *sc*)
void [gnutls_psk_free_client_credentials], page 441,
(gnutls_psk_client_credentials_t *sc*)

Clients supporting PSK should supply the username and key before a TLS session is established. Alternatively [gnutls_psk_set_client_credentials_function], page 443 can be used to specify a callback function. This has the advantage that the callback will be called only if PSK has been negotiated.

int [gnutls_psk_set_client_credentials], page 442,
(gnutls_psk_client_credentials_t *res*, const char * *username*, const gnutls_datum_t * *key*, gnutls_psk_key_flags *flags*)

void gnutls_psk_set_client_credentials_function [Function]
(*gnutls_psk_client_credentials_t* **cred**,
gnutls_psk_client_credentials_function * **func**)

cred: is a gnutls_psk_server_credentials_t type.

func: is the callback function

This function can be used to set a callback to retrieve the username and password for client PSK authentication. The callback's function form is: int (*callback)(gnutls_session_t, char** username, gnutls_datum_t* key);

The username and key ->data must be allocated using gnutls_malloc() . The username should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265).

The callback function will be called once per handshake.

The callback function should return 0 on success. -1 indicates an error.

In server side the default behavior of GnuTLS is to read the usernames and PSK keys from a password file. The password file should contain usernames and keys in hexadecimal format. The name of the password file can be stored to the credentials structure by calling [gnutls_psk_set_server_credentials_file], page 444. If a different password file format is to be used, then a callback should be set instead by [gnutls_psk_set_server_credentials_function], page 444.

The server can help the client chose a suitable username and password, by sending a hint. Note that there is no common profile for the PSK hint and applications are discouraged to use it. A server, may specify the hint by calling [gnutls_psk_set_server_credentials_hint], page 445. The client can retrieve the hint, for example in the callback function, using [gnutls_psk_client_get_hint], page 441.

`int gnutls_psk_set_server_credentials_file` [Function]
(*gnutls_psk_server_credentials_t* **res**, *const char* * **password_file**)

res: is a `gnutls_psk_server_credentials_t` type.

password_file: is the PSK password file (passwd.psk)

This function sets the password file, in a `gnutls_psk_server_credentials_t` type. This password file holds usernames and keys and will be used for PSK authentication.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

void [gnutls_psk_set_server_credentials_function], page 444, (gnutls_psk_server_credentials_t *cred*, gnutls_psk_server_credentials_function * *func*)
int [gnutls_psk_set_server_credentials_hint], page 445, (gnutls_psk_server_credentials_t *res*, const char * *hint*)
const char * [gnutls_psk_client_get_hint], page 441, (gnutls_session_t *session*)

6.4.5 Anonymous

The key exchange methods for anonymous authentication since GnuTLS 3.6.0 will utilize the RFC7919 parameters, unless explicit parameters have been provided and associated with an anonymous credentials structure. Check Section 6.12.6 [Parameter generation], page 192, for more information. The initialization functions for the credentials are shown below.

int [gnutls_anon_allocate_server_credentials], page 362, (gnutls_anon_server_credentials_t * *sc*)
int [gnutls_anon_allocate_client_credentials], page 362, (gnutls_anon_client_credentials_t * *sc*)
void [gnutls_anon_free_server_credentials], page 362, (gnutls_anon_server_credentials_t *sc*)
void [gnutls_anon_free_client_credentials], page 362, (gnutls_anon_client_credentials_t *sc*)

6.5 Setting up the transport layer

The next step is to setup the underlying transport layer details. The Berkeley sockets are implicitly used by GnuTLS, thus a call to [gnutls_transport_set_int], page 486 would be sufficient to specify the socket descriptor.

void [gnutls_transport_set_int], page 486, (gnutls_session_t *session*, int *fd*)
void [gnutls_transport_set_int2], page 486, (gnutls_session_t *session*, int *recv_fd*, int *send_fd*)

If however another transport layer than TCP is selected, then a pointer should be used instead to express the parameter to be passed to custom functions. In that case the following functions should be used instead.

void [gnutls_transport_set_ptr], page 486, (gnutls_session_t *session*, gnutls_transport_ptr_t *ptr*)
void [gnutls_transport_set_ptr2], page 487, (gnutls_session_t *session*, gnutls_transport_ptr_t *recv_ptr*, gnutls_transport_ptr_t *send_ptr*)

Moreover all of the following push and pull callbacks should be set.

`void gnutls_transport_set_push_function (`*gnutls_session_t* ***session***, *gnutls_push_func* ***push_func***`)` [Function]

session: is a `gnutls_session_t` type.

push_func: a callback function similar to `write()`

This is the function where you set a push function for gnutls to use in order to send data. If you are going to use berkeley style sockets, you do not need to use this function since the default send(2) will probably be ok. Otherwise you should specify this function for gnutls to be able to send data. The callback should return a positive number indicating the bytes sent, and -1 on error.

`push_func` is of the form, ssize_t (*gnutls_push_func)(gnutls_transport_ptr_t, const void*, size_t);

`void gnutls_transport_set_vec_push_function (`*gnutls_session_t* ***session***, *gnutls_vec_push_func* ***vec_func***`)` [Function]

session: is a `gnutls_session_t` type.

vec_func: a callback function similar to `writev()`

Using this function you can override the default writev(2) function for gnutls to send data. Setting this callback instead of `gnutls_transport_set_push_function()` is recommended since it introduces less overhead in the TLS handshake process.

`vec_func` is of the form, ssize_t (*gnutls_vec_push_func) (gnutls_transport_ptr_t, const giovec_t * iov, int iovcnt);

Since: 2.12.0

`void gnutls_transport_set_pull_function (`*gnutls_session_t* ***session***, *gnutls_pull_func* ***pull_func***`)` [Function]

session: is a `gnutls_session_t` type.

pull_func: a callback function similar to `read()`

This is the function where you set a function for gnutls to receive data. Normally, if you use berkeley style sockets, do not need to use this function since the default recv(2) will probably be ok. The callback should return 0 on connection termination, a positive number indicating the number of bytes received, and -1 on error.

`gnutls_pull_func` is of the form, ssize_t (*gnutls_pull_func)(gnutls_transport_ptr_t, void*, size_t);

`void gnutls_transport_set_pull_timeout_function (`*gnutls_session_t* ***session***, *gnutls_pull_timeout_func* ***func***`)` [Function]

session: is a `gnutls_session_t` type.

func: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The

callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()` , if the timeout value is zero the callback should return zero if no data are immediately available. The special value `GNUTLS_INDEFINITE_TIMEOUT` indicates that the callback should wait indefinitely for data.

`gnutls_pull_timeout_func` is of the form, int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);

This callback is necessary when `gnutls_handshake_set_timeout()` or `gnutls_record_set_timeout()` are set, under TLS1.3 and for enforcing the DTLS mode timeouts when in blocking mode.

For compatibility with future GnuTLS versions this callback must be set when a custom pull function is registered. The callback will not be used when the session is in TLS mode with non-blocking sockets. That is, when `GNUTLS_NONBLOCK` is specified for a TLS session in `gnutls_init()` .

The helper function `gnutls_system_recv_timeout()` is provided to simplify writing callbacks.

Since: 3.0

The functions above accept a callback function which should return the number of bytes written, or -1 on error and should set `errno` appropriately. In some environments, setting `errno` is unreliable. For example Windows have several errno variables in different CRTs, or in other systems it may be a non thread-local variable. If this is a concern to you, call [gnutls_transport_set_errno], page 485 with the intended errno value instead of setting `errno` directly.

`void gnutls_transport_set_errno (`*gnutls_session_t* `session`*, int* *err*`)` [Function]

session: is a `gnutls_session_t` type.

err: error value to store in session-specific errno variable.

Store `err` in the session-specific errno variable. Useful values for `err` are EINTR, EAGAIN and EMSGSIZE, other values are treated will be treated as real errors in the push/pull function.

This function is useful in replacement push and pull functions set by `gnutls_transport_set_push_function()` and `gnutls_transport_set_pull_function()` under Windows, where the replacements may not have access to the same `errno` variable that is used by GnuTLS (e.g., the application is linked to msvcr71.dll and gnutls is linked to msvcrt.dll).

This function is unreliable if you are using the same `session` in different threads for sending and receiving.

GnuTLS currently only interprets the EINTR, EAGAIN and EMSGSIZE errno values and returns the corresponding GnuTLS error codes:

- `GNUTLS_E_INTERRUPTED`
- `GNUTLS_E_AGAIN`

- GNUTLS_E_LARGE_PACKET

The EINTR and EAGAIN values are returned by interrupted system calls, or when non blocking IO is used. All GnuTLS functions can be resumed (called again), if any of the above error codes is returned. The EMSGSIZE value is returned when attempting to send a large datagram.

In the case of DTLS it is also desirable to override the generic transport functions with functions that emulate the operation of `recvfrom` and `sendto`. In addition DTLS requires timers during the receive of a handshake message, set using the [gnutls_transport_set_pull_timeout_function], page 487 function. To check the retransmission timers the function [gnutls_dtls_get_timeout], page 491 is provided, which returns the time remaining until the next retransmission, or better the time until [gnutls_handshake], page 414 should be called again.

`void gnutls_transport_set_pull_timeout_function` [Function]
(*gnutls_session_t* ***session***, *gnutls_pull_timeout_func* ***func***)

session: is a `gnutls_session_t` type.

func: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()` , if the timeout value is zero the callback should return zero if no data are immediately available. The special value `GNUTLS_INDEFINITE_TIMEOUT` indicates that the callback should wait indefinitely for data.

`gnutls_pull_timeout_func` is of the form, int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);

This callback is necessary when `gnutls_handshake_set_timeout()` or `gnutls_record_set_timeout()` are set, under TLS1.3 and for enforcing the DTLS mode timeouts when in blocking mode.

For compatibility with future GnuTLS versions this callback must be set when a custom pull function is registered. The callback will not be used when the session is in TLS mode with non-blocking sockets. That is, when `GNUTLS_NONBLOCK` is specified for a TLS session in `gnutls_init()` .

The helper function `gnutls_system_recv_timeout()` is provided to simplify writing callbacks.

Since: 3.0

`unsigned int gnutls_dtls_get_timeout (`*gnutls_session_t* [Function]
session)

session: is a `gnutls_session_t` type.

This function will return the milliseconds remaining for a retransmission of the previously sent handshake message. This function is useful when DTLS is used in non-blocking mode, to estimate when to call `gnutls_handshake()` if no packets have been received.

Returns: the remaining time in milliseconds.

Since: 3.0

6.5.1 Asynchronous operation

GnuTLS can be used with asynchronous socket or event-driven programming. The approach is similar to using Berkeley sockets under such an environment. The blocking, due to network interaction, calls such as [gnutls_handshake], page 414, [gnutls_record_recv], page 450, can be set to non-blocking by setting the underlying sockets to non-blocking. If other push and pull functions are setup, then they should behave the same way as `recv` and `send` when used in a non-blocking way, i.e., return -1 and set errno to `EAGAIN`. Since, during a TLS protocol session GnuTLS does not block except for network interaction, the non blocking `EAGAIN` errno will be propagated and GnuTLS functions will return the `GNUTLS_E_AGAIN` error code. Such calls can be resumed the same way as a system call would. The only exception is [gnutls_record_send], page 452, which if interrupted subsequent calls need not to include the data to be sent (can be called with NULL argument).

When using the `poll` or `select` system calls though, one should remember that they only apply to the kernel sockets API. To check for any available buffered data in a GnuTLS session, utilize [gnutls_record_check_pending], page 448, either before the `poll` system call, or after a call to [gnutls_record_recv], page 450. Data queued by [gnutls_record_send], page 452 (when interrupted) can be discarded using [gnutls_record_discard_queued], page 449.

An example of GnuTLS' usage with asynchronous operation can be found in `doc/examples/tlsproxy`.

The following paragraphs describe the detailed requirements for non-blocking operation when using the TLS or DTLS protocols.

6.5.1.1 TLS protocol

There are no special requirements for the TLS protocol operation in non-blocking mode if a non-blocking socket is used.

It is recommended, however, for future compatibility, when in non-blocking mode, to call the [gnutls_init], page 422 function with the `GNUTLS_NONBLOCK` flag set (see Section 6.3 [Session initialization], page 144).

6.5.1.2 Datagram TLS protocol

When in non-blocking mode the function, the [gnutls_init], page 422 function must be called with the `GNUTLS_NONBLOCK` flag set (see Section 6.3 [Session initialization], page 144).

In contrast with the TLS protocol, the pull timeout function is required, but will only be called with a timeout of zero. In that case it should indicate whether there are data to be received or not. When not using the default pull function, then [gnutls_transport_set_pull_timeout_function], page 487 should be called.

Although in the TLS protocol implementation each call to receive or send function implies to restoring the same function that was interrupted, in the DTLS protocol this requirement isn't true. There are cases where a retransmission is required, which are indicated by a received message and thus [gnutls_record_get_direction], page 449 must be called to decide which direction to check prior to restoring a function call.

`int gnutls_record_get_direction` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function is useful to determine whether a GnuTLS function was interrupted while sending or receiving, so that `select()` or `poll()` may be called appropriately.

It provides information about the internals of the record protocol and is only useful if a prior gnutls function call, e.g. `gnutls_handshake()` , was interrupted and returned `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` . After such an interrupt applications may call `select()` or `poll()` before restoring the interrupted GnuTLS function.

This function's output is unreliable if you are using the same `session` in different threads for sending and receiving.

Returns: 0 if interrupted while trying to read data, or 1 while trying to write data.

When calling [gnutls_handshake], page 414 through a multi-plexer, to be able to handle properly the DTLS handshake retransmission timers, the function [gnutls_dtls_get_timeout], page 491 should be used to estimate when to call [gnutls_handshake], page 414 if no data have been received.

6.5.2 Reducing round-trips

The full TLS 1.2 handshake requires 2 round-trips to complete, and when combined with TCP's SYN and SYN-ACK negotiation it extends to 3 full round-trips. While, TLS 1.3 reduces that to two round-trips when under TCP, it still adds considerable latency, making the protocol unsuitable for certain applications.

To optimize the handshake latency, in client side, it is possible to take advantage of the TCP fast open [[RFC7413], page 749] mechanism on operating systems that support it. That can be done either by manually crafting the push and pull callbacks, or by utilizing [gnutls_transport_set_fastopen], page 715. In that case the initial TCP handshake is eliminated, reducing the TLS 1.2 handshake round-trip to 2, and the TLS 1.3 handshake to a single round-trip. Note, that when this function is used, any connection failures will be reported during the [gnutls_handshake], page 414 function call with error code `GNUTLS_E_PUSH_ERROR`.

`void gnutls_transport_set_fastopen` (*gnutls_session_t* `session`, *int* `fd`, *struct sockaddr ** `connect_addr`, *socklen_t* `connect_addrlen`, *unsigned int* `flags`) [Function]

session: is a `gnutls_session_t` type.

fd: is the session's socket descriptor

connect_addr: is the address we want to connect to

connect_addrlen: is the length of `connect_addr`

flags: must be zero

Enables TCP Fast Open (TFO) for the specified TLS client session. That means that TCP connection establishment and the transmission of the first TLS client hello packet are combined. The peer's address must be specified in `connect_addr` and `connect_addrlen` , and the socket specified by `fd` should not be connected.

TFO only works for TCP sockets of type AF_INET and AF_INET6. If the OS doesn't support TCP fast open this function will result to gnutls using `connect()` transparently during the first write.

> **Note:** This function overrides all the transport callback functions. If this is undesirable, TCP Fast Open must be implemented on the user callback functions without calling this function. When using this function, transport callbacks must not be set, and `gnutls_transport_set_ptr()` or `gnutls_transport_set_int()` must not be called.
>
> On GNU/Linux TFO has to be enabled at the system layer, that is in /proc/sys/net/ipv4/tcp_fastopen, bit 0 has to be set.
>
> This function has no effect on server sessions.
>
> **Since:** 3.5.3

When restricted to TLS 1.2, and non-resumed sessions, it is possible to further reduce the round-trips to a single one by taking advantage of the Section 3.6.8 [False Start], page 16, TLS extension. This can be enabled by setting the GNUTLS_ENABLE_FALSE_START flag on [gnutls_init], page 422.

Under TLS 1.3, the server side can start transmitting before the handshake is complete (i.e., while the client Finished message is still in flight), when no client certificate authentication is requested. This, unlike false start, is part of protocol design with no known security implications. It can be enabled by setting the GNUTLS_ENABLE_EARLY_START on [gnutls_init], page 422, and the [gnutls_handshake], page 414 function will return early, allowing the server to send data earlier.

6.5.3 Zero-roundtrip mode

Under TLS 1.3, when the client has already connected to the server and is resuming a session, it can start transmitting application data during handshake. This is called zero round-trip time (0-RTT) mode, and the application data sent in this mode is called early data. The client can send early data with [gnutls_record_send_early_data], page 453. The client should call this function before calling [gnutls_handshake], page 414 and after calling [gnutls_session_set_data], page 465.

Note, however, that early data has weaker security properties than normal application data sent after handshake, such as lack of forward secrecy, no guarantees of non-replay between connections. Thus it is disabled on the server side by default. To enable it, the server needs to:

1. Set GNUTLS_ENABLE_EARLY_DATA on [gnutls_init], page 422. Note that this option only has effect on server.
2. Enable anti-replay measure. See Section 6.5.4 [Anti-replay protection], page 163, for the details.

The server caches the received early data until it is read. To set the maximum amount of data to be stored in the cache, use [gnutls_record_set_max_early_data_size], page 454. After receiving the EndOfEarlyData handshake message, the server can start retrieving the received data with [gnutls_record_recv_early_data], page 451. You can call the function either after the handshake is complete, or through a handshake hook ([gnutls_handshake_set_hook_function], page 415).

When sending early data, the client should respect the maximum amount of early data, which may have been previously advertised by the server. It can be

checked using [gnutls_record_get_max_early_data_size], page 449, right after calling [gnutls_session_set_data], page 465.

After sending early data, to check whether the sent early data was accepted by the server, use [gnutls_session_get_flags], page 461 and compare the result with GNUTLS_SFLAGS_EARLY_DATA. Similarly, on the server side, the same function and flag can be used to check whether it has actually accepted early data.

6.5.4 Anti-replay protection

When 0-RTT mode is used, the server must protect itself from replay attacks, where adversary client reuses duplicate session ticket to send early data, before the server authenticates the client.

GnuTLS provides a simple mechanism against replay attacks, following the method called ClientHello recording. When a session ticket is accepted, the server checks if the ClientHello message has been already seen. If there is a duplicate, the server rejects early data.

The problem of this approach is that the number of recorded messages grows indefinitely. To prevent that, the server can limit the recording to a certain time window, which can be configured with [gnutls_anti_replay_set_window], page 365.

The anti-replay mechanism shall be globally initialized with [gnutls_anti_replay_init], page 364, and then attached to a session using [gnutls_anti_replay_enable], page 364. It can be deinitialized with [gnutls_anti_replay_deinit], page 363.

The server must also set up a database back-end to store ClientHello messages. That can be achieved using [gnutls_anti_replay_set_add_function], page 364 and [gnutls_anti_replay_set_ptr], page 364.

Note that, if the back-end stores arbitrary number of ClientHello, it needs to periodically clean up the stored entries based on the time window set with [gnutls_anti_replay_set_window], page 365. The cleanup can be implemented by iterating through the database entries and calling [gnutls_db_check_entry_expire_time], page 396. This is similar to session database cleanup used by TLS1.2 sessions.

The full set up of the server using early data would be like the following example:

```
#define MAX_EARLY_DATA_SIZE 16384

static int
db_add_func(void *dbf, gnutls_datum_t key, gnutls_datum_t data)
{
    /* Return GNUTLS_E_DB_ENTRY_EXISTS, if KEY is found in the database.
     * Otherwise, store it and return 0.
     */
}

static int
handshake_hook_func(gnutls_session_t session, unsigned int htype,
                    unsigned when, unsigned int incoming, const gnutls_datum_t *msg)
{
    int ret;
    char buf[MAX_EARLY_DATA_SIZE];
```

```
    assert(htype == GNUTLS_HANDSHAKE_END_OF_EARLY_DATA);
    assert(when == GNUTLS_HOOK_POST);

    if (gnutls_session_get_flags(session) & GNUTLS_SFLAGS_EARLY_DATA) {
        ret = gnutls_record_recv_early_data(session, buf, sizeof(buf));
        assert(ret >= 0);
    }

    return ret;
}

int main()
{
  ...
  /* Initialize anti-replay measure, which can be shared
   * among multiple sessions.
   */
  gnutls_anti_replay_init(&anti_replay);

  /* Set the database back-end function for the anti-replay data. */
  gnutls_anti_replay_set_add_function(anti_replay, db_add_func);
  gnutls_anti_replay_set_ptr(anti_replay, NULL);

  ...

  gnutls_init(&server, GNUTLS_SERVER | GNUTLS_ENABLE_EARLY_DATA);
  gnutls_record_set_max_early_data_size(server, MAX_EARLY_DATA_SIZE);

  ...

  /* Set the anti-replay measure to the session.
   */
  gnutls_anti_replay_enable(server, anti_replay);
  ...

  /* Retrieve early data in a handshake hook;
   * you can also do that after handshake.
   */
  gnutls_handshake_set_hook_function(server, GNUTLS_HANDSHAKE_END_OF_EARLY_DATA,
                                     GNUTLS_HOOK_POST, handshake_hook_func);
  ...
}
```

6.5.5 DTLS sessions

Because datagram TLS can operate over connections where the client cannot be reliably verified, functionality in the form of cookies, is available to prevent denial of service attacks to servers. GnuTLS requires a server to generate a secret key that is used to sign a cookie[4]. That cookie is sent to the client using [gnutls_dtls_cookie_send], page 490, and the client must reply using the correct cookie. The server side should verify the initial message sent by client using [gnutls_dtls_cookie_verify], page 490. If successful the session should be initialized and associated with the cookie using [gnutls_dtls_prestate_set], page 491, before proceeding to the handshake.

int [gnutls_key_generate], page 422, (gnutls_datum_t * *key*, unsigned int *key_size*)
int [gnutls_dtls_cookie_send], page 490, (gnutls_datum_t * *key*, void * *client_data*, size_t *client_data_size*, gnutls_dtls_prestate_st * *prestate*, gnutls_transport_ptr_t *ptr*, gnutls_push_func *push_func*)
int [gnutls_dtls_cookie_verify], page 490, (gnutls_datum_t * *key*, void * *client_data*, size_t *client_data_size*, void * *_msg*, size_t *msg_size*, gnutls_dtls_prestate_st * *prestate*)
void [gnutls_dtls_prestate_set], page 491, (gnutls_session_t *session*, gnutls_dtls_prestate_st * *prestate*)

Note that the above apply to server side only and they are not mandatory to be used. Not using them, however, allows denial of service attacks. The client side cookie handling is part of [gnutls_handshake], page 414.

Datagrams are typically restricted by a maximum transfer unit (MTU). For that both client and server side should set the correct maximum transfer unit for the layer underneath GnuTLS. This will allow proper fragmentation of DTLS messages and prevent messages from being silently discarded by the transport layer. The "correct" maximum transfer unit can be obtained through a path MTU discovery mechanism [[RFC4821], page 753].

void [gnutls_dtls_set_mtu], page 492, (gnutls_session_t *session*, unsigned int *mtu*)
unsigned int [gnutls_dtls_get_mtu], page 491, (gnutls_session_t *session*)
unsigned int [gnutls_dtls_get_data_mtu], page 491, (gnutls_session_t *session*)

6.5.6 DTLS and SCTP

Although DTLS can run under any reliable or unreliable layer, there are special requirements for SCTP according to [[RFC6083], page 750]. We summarize the most important below, however for a full treatment we refer to [[RFC6083], page 750].

- The MTU set via [gnutls_dtls_set_mtu], page 492 must be 2^14.
- Replay detection must be disabled; use the flag `GNUTLS_NO_REPLAY_PROTECTION` with [gnutls_init], page 422.
- Retransmission of messages must be disabled; use [gnutls_dtls_set_timeouts], page 492 with a retransmission timeout larger than the total.
- Handshake, Alert and ChangeCipherSpec messages must be sent over stream 0 with unlimited reliability and with the ordered delivery feature.

[4] A key of 128 bits or 16 bytes should be sufficient for this purpose.

- During a rehandshake, the caching of messages with unknown epoch is not handled by GnuTLS; this must be implemented in a special pull function.

6.6 TLS handshake

Once a session has been initialized and a network connection has been set up, TLS and DTLS protocols perform a handshake. The handshake is the actual key exchange.

`int gnutls_handshake` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function performs the handshake of the TLS/SSL protocol, and initializes the TLS session parameters.

The non-fatal errors expected by this function are: `GNUTLS_E_INTERRUPTED` , `GNUTLS_E_AGAIN` , `GNUTLS_E_WARNING_ALERT_RECEIVED` . When this function is called for re-handshake under TLS 1.2 or earlier, the non-fatal error code `GNUTLS_E_GOT_APPLICATION_DATA` may also be returned.

The former two interrupt the handshake procedure due to the transport layer being interrupted, and the latter because of a "warning" alert that was sent by the peer (it is always a good idea to check any received alerts). On these non-fatal errors call this function again, until it returns 0; cf. `gnutls_record_get_direction()` and `gnutls_error_is_fatal()` . In DTLS sessions the non-fatal error `GNUTLS_E_LARGE_PACKET` is also possible, and indicates that the MTU should be adjusted.

When this function is called by a server after a rehandshake request under TLS 1.2 or earlier the `GNUTLS_E_GOT_APPLICATION_DATA` error code indicates that some data were pending prior to peer initiating the handshake. Under TLS 1.3 this function when called after a successful handshake, is a no-op and always succeeds in server side; in client side this function is equivalent to `gnutls_session_key_update()` with `GNUTLS_KU_PEER` flag.

This function handles both full and abbreviated TLS handshakes (resumption). For abbreviated handshakes, in client side, the `gnutls_session_set_data()` should be called prior to this function to set parameters from a previous session. In server side, resumption is handled by either setting a DB back-end, or setting up keys for session tickets.

Returns: `GNUTLS_E_SUCCESS` on a successful handshake, otherwise a negative error code.

`void gnutls_handshake_set_timeout` (*gnutls_session_t* `session`, *unsigned int* `ms`) [Function]

session: is a `gnutls_session_t` type.

ms: is a timeout value in milliseconds

This function sets the timeout for the TLS handshake process to the provided value. Use an `ms` value of zero to disable timeout, or `GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT` for a reasonable default value. For the DTLS protocol, the more detailed `gnutls_dtls_set_timeouts()` is provided.

This function requires to set a pull timeout callback. See `gnutls_transport_set_pull_timeout_function()` .

Since: 3.1.0

In GnuTLS 3.5.0 and later it is recommended to use [gnutls_session_set_verify_cert], page 466 for the handshake process to ensure the verification of the peer's identity. That will verify the peer's certificate, against the trusted CA store while accounting for stapled OCSP responses during the handshake; any error will be returned as a handshake error.

In older GnuTLS versions it is required to verify the peer's certificate during the handshake by setting a callback with [gnutls_certificate_set_verify_function], page 381, and then using [gnutls_certificate_verify_peers3], page 392 from it. See Section 4.1 [Certificate authentication], page 20, for more information.

void [gnutls_session_set_verify_cert], page 466, (gnutls_session_t *session*, const char * *hostname*, unsigned *flags*)
int [gnutls_certificate_verify_peers3], page 392, (gnutls_session_t *session*, const char * *hostname*, unsigned int * *status*)

6.7 Data transfer and termination

Once the handshake is complete and peer's identity has been verified data can be exchanged. The available functions resemble the POSIX `recv` and `send` functions. It is suggested to use [gnutls_error_is_fatal], page 406 to check whether the error codes returned by these functions are fatal for the protocol or can be ignored.

`ssize_t gnutls_record_send` (*gnutls_session_t* ***session***, *const void* ** data*, *size_t* ***data_size***) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send

data_size: is the length of the data

This function has the similar semantics with `send()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. Note that if the send buffer is full, `send()` will block this function. See the `send()` documentation for more information.

You can replace the default push function which is `send()` , by using `gnutls_transport_set_push_function()` .

If the EINTR is returned by the internal push function then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again with the exact same parameters, or provide a `NULL` pointer for `data` and 0 for `data_size` , in order to write the same data as before. If you wish to discard the previous data instead of retrying, you must call `gnutls_record_discard_queued()` before calling this function with different parameters. Note that the latter works only on special transports (e.g., UDP). cf. `gnutls_record_get_direction()` .

Note that in DTLS this function will return the `GNUTLS_E_LARGE_PACKET` error code if the send data exceed the data MTU value - as returned by `gnutls_dtls_get_data_mtu()` . The errno value EMSGSIZE also maps to `GNUTLS_E_LARGE_PACKET` . Note that since 3.2.13 this function can be called under cork in DTLS mode, and will refuse to send data over the MTU size by returning `GNUTLS_E_LARGE_PACKET` .

Returns: The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

`ssize_t gnutls_record_recv` (*gnutls_session_t* ***session***, *void* * ***data***, *size_t* ***data_size***) [Function]

session: is a `gnutls_session_t` type.

data: the buffer that the data will be read into

data_size: the number of requested bytes

This function has the similar semantics with `recv()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. In the special case that the peer requests a renegotiation, the caller will receive an error code of `GNUTLS_E_REHANDSHAKE` . In case of a client, this message may be simply ignored, replied with an alert `GNUTLS_A_NO_RENEGOTIATION` , or replied with a new handshake, depending on the client's will. A server receiving this error code can only initiate a new handshake or terminate the session.

If `EINTR` is returned by the internal pull function (the default is `recv()`) then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again to get the data. See also `gnutls_record_get_direction()` .

Returns: The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error. The number of bytes received might be less than the requested `data_size` .

`int gnutls_error_is_fatal` (*int* ***error***) [Function]

error: is a GnuTLS error code, a negative error code

If a GnuTLS function returns a negative error code you may feed that value to this function to see if the error condition is fatal to a TLS session (i.e., must be terminated).

Note that you may also want to check the error code manually, since some non-fatal errors to the protocol (such as a warning alert or a rehandshake request) may be fatal for your program.

This function is only useful if you are dealing with errors from functions that relate to a TLS session (e.g., record layer or handshake layer handling functions).

Returns: Non-zero value on fatal errors or zero on non-fatal.

Although, in the TLS protocol the receive function can be called at any time, when DTLS is used the GnuTLS receive functions must be called once a message is available for reading, even if no data are expected. This is because in DTLS various (internal) actions may be required due to retransmission timers. Moreover, an extended receive function is shown below, which allows the extraction of the message's sequence number. Due to the unreliable nature of the protocol, this field allows distinguishing out-of-order messages.

`ssize_t gnutls_record_recv_seq` (*gnutls_session_t* ***session***, *void* * ***data***, *size_t* ***data_size***, *unsigned char* * ***seq***) [Function]

session: is a `gnutls_session_t` type.

data: the buffer that the data will be read into

data_size: the number of requested bytes

seq: is the packet's 64-bit sequence number. Should have space for 8 bytes.

This function is the same as `gnutls_record_recv()` , except that it returns in addition to data, the sequence number of the data. This is useful in DTLS where record packets might be received out-of-order. The returned 8-byte sequence number is an integer in big-endian format and should be treated as a unique message identification.

Returns: The number of bytes received and zero on EOF. A negative error code is returned in case of an error. The number of bytes received might be less than `data_size` .

Since: 3.0

The [gnutls_record_check_pending], page 448 helper function is available to allow checking whether data are available to be read in a GnuTLS session buffers. Note that this function complements but does not replace `poll`, i.e., [gnutls_record_check_pending], page 448 reports no data to be read, `poll` should be called to check for data in the network buffers.

`size_t gnutls_record_check_pending` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

This function checks if there are unread data in the gnutls buffers. If the return value is non-zero the next call to `gnutls_record_recv()` is guaranteed not to block.

Returns: Returns the size of the data or zero.

int `[gnutls_record_get_direction], page 449, (gnutls_session_t` *session*`)`

Once a TLS or DTLS session is no longer needed, it is recommended to use [gnutls_bye], page 367 to terminate the session. That way the peer is notified securely about the intention of termination, which allows distinguishing it from a malicious connection termination. A session can be deinitialized with the [gnutls_deinit], page 398 function.

`int gnutls_bye` (*gnutls_session_t* ***session***, *gnutls_close_request_t* ***how***) [Function]

session: is a `gnutls_session_t` type.

how: is an integer

Terminates the current TLS/SSL connection. The connection should have been initiated using `gnutls_handshake()` . `how` should be one of `GNUTLS_SHUT_RDWR` , `GNUTLS_SHUT_WR` .

In case of `GNUTLS_SHUT_RDWR` the TLS session gets terminated and further receives and sends will be disallowed. If the return value is zero you may continue using the underlying transport layer. `GNUTLS_SHUT_RDWR` sends an alert containing a close request and waits for the peer to reply with the same message.

In case of `GNUTLS_SHUT_WR` the TLS session gets terminated and further sends will be disallowed. In order to reuse the connection you should wait for an EOF from the peer. `GNUTLS_SHUT_WR` sends an alert containing a close request.

Note that not all implementations will properly terminate a TLS connection. Some of them, usually for performance reasons, will terminate only the underlying transport layer, and thus not distinguishing between a malicious party prematurely terminating the connection and normal termination.

This function may also return GNUTLS_E_AGAIN or GNUTLS_E_INTERRUPTED ; cf. gnutls_record_get_direction() .

Returns: GNUTLS_E_SUCCESS on success, or an error code, see function documentation for entire semantics.

void gnutls_deinit (*gnutls_session_t session*) [Function]

session: is a gnutls_session_t type.

This function clears all buffers associated with the session . This function will also remove session data from the session database if the session was terminated abnormally.

6.8 Buffered data transfer

Although [gnutls_record_send], page 452 is sufficient to transmit data to the peer, when many small chunks of data are to be transmitted it is inefficient and wastes bandwidth due to the TLS record overhead. In that case it is preferable to combine the small chunks before transmission. The following functions provide that functionality.

void gnutls_record_cork (*gnutls_session_t session*) [Function]

session: is a gnutls_session_t type.

If called, gnutls_record_send() will no longer send any records. Any sent records will be cached until gnutls_record_uncork() is called.

This function is safe to use with DTLS after GnuTLS 3.3.0.

Since: 3.1.9

int gnutls_record_uncork (*gnutls_session_t session*, *unsigned int flags*) [Function]

session: is a gnutls_session_t type.

flags: Could be zero or GNUTLS_RECORD_WAIT

This resets the effect of gnutls_record_cork() , and flushes any pending data. If the GNUTLS_RECORD_WAIT flag is specified then this function will block until the data is sent or a fatal error occurs (i.e., the function will retry on GNUTLS_E_AGAIN and GNUTLS_E_INTERRUPTED).

If the flag GNUTLS_RECORD_WAIT is not specified and the function is interrupted then the GNUTLS_E_AGAIN or GNUTLS_E_INTERRUPTED errors will be returned. To obtain the data left in the corked buffer use gnutls_record_check_corked() .

Returns: On success the number of transmitted data is returned, or otherwise a negative error code.

Since: 3.1.9

6.9 Handling alerts

During a TLS connection alert messages may be exchanged by the two peers. Those messages may be fatal, meaning the connection must be terminated afterwards, or warning when something needs to be reported to the peer, but without interrupting the session. The

error codes GNUTLS_E_WARNING_ALERT_RECEIVED or GNUTLS_E_FATAL_ALERT_RECEIVED signal those alerts when received, and may be returned by all GnuTLS functions that receive data from the peer, being [gnutls_handshake], page 414 and [gnutls_record_recv], page 450.

If those error codes are received the alert and its level should be logged or reported to the peer using the functions below.

`gnutls_alert_description_t gnutls_alert_get (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the last alert number received. This function should be called when GNUTLS_E_WARNING_ALERT_RECEIVED or GNUTLS_E_FATAL_ALERT_RECEIVED errors are returned by a gnutls function. The peer may send alerts if he encounters an error. If no alert has been received the returned value is undefined.

Returns: the last alert received, a `gnutls_alert_description_t` value.

`const char * gnutls_alert_get_name (`*gnutls_alert_description_t* `alert`) [Function]

alert: is an alert number.

This function will return a string that describes the given alert number, or `NULL` . See `gnutls_alert_get()` .

Returns: string corresponding to `gnutls_alert_description_t` value.

The peer may also be warned or notified of a fatal issue by using one of the functions below. All the available alerts are listed in [The Alert Protocol], page 9.

`int gnutls_alert_send (`*gnutls_session_t* `session`, *gnutls_alert_level_t* `level`, *gnutls_alert_description_t* `desc`) [Function]

session: is a `gnutls_session_t` type.

level: is the level of the alert

desc: is the alert description

This function will send an alert to the peer in order to inform him of something important (eg. his Certificate could not be verified). If the alert level is Fatal then the peer is expected to close the connection, otherwise he may ignore the alert and continue.

The error code of the underlying record send function will be returned, so you may also receive GNUTLS_E_INTERRUPTED or GNUTLS_E_AGAIN as well.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise an error code is returned.

`int gnutls_error_to_alert (`*int* `err`, *int ** `level`) [Function]

err: is a negative integer

level: the alert level will be stored there

Get an alert depending on the error code returned by a gnutls function. All alerts sent by this function should be considered fatal. The only exception is when `err` is GNUTLS_E_REHANDSHAKE , where a warning alert should be sent to the peer indicating that no renegotiation will be performed.

If there is no mapping to a valid alert the alert to indicate internal error (`GNUTLS_A_INTERNAL_ERROR`) is returned.

Returns: the alert code to use for a particular error code.

6.10 Priority strings

How to use Priority Strings

The GnuTLS priority strings specify the TLS session's handshake algorithms and options in a compact, easy-to-use format. These strings are intended as a user-specified override of the library defaults.

That is, we recommend applications using the default settings (c.f. [gnutls_set_default_priority], page 469 or [gnutls_set_default_priority_append], page 469), and provide the user with access to priority strings for overriding the default behavior, on configuration files, or other UI. Following such a principle, makes the GnuTLS library as the default settings provider. That is necessary and a good practice, because TLS protocol hardening and phasing out of legacy algorithms, is easier to coordinate when happens in a single library.

int [gnutls_set_default_priority], page 469, (gnutls_session_t *session*)
int [gnutls_set_default_priority_append], page 469, (gnutls_session_t *session*, const char * *add_prio*, const char ** *err_pos*, unsigned *flags*)
int [gnutls_priority_set_direct], page 439, (gnutls_session_t *session*, const char * *priorities*, const char ** *err_pos*)

The priority string translation to the internal GnuTLS form requires processing and the generated internal form also occupies some memory. For that, it is recommended to do that processing once in server side, and share the generated data across sessions. The following functions allow the generation of a "priority cache" and the sharing of it across sessions.

int [gnutls_priority_init2], page 437, (gnutls_priority_t * *priority_cache*, const char * *priorities*, const char ** *err_pos*, unsigned *flags*)
int [gnutls_priority_init], page 436, (gnutls_priority_t * *priority_cache*, const char * *priorities*, const char ** *err_pos*)
int [gnutls_priority_set], page 439, (gnutls_session_t *session*, gnutls_priority_t *priority*)
void [gnutls_priority_deinit], page 435, (gnutls_priority_t *priority_cache*)

Using Priority Strings

A priority string string may contain a single initial keyword such as in Table 6.3 and may be followed by additional algorithm or special keywords. Note that their description is intentionally avoiding specific algorithm details, as the priority strings are not constant between gnutls versions (they are periodically updated to account for cryptographic advances while providing compatibility with old clients and servers).

Keyword	Description
@KEYWORD	Means that a compile-time specified system configuration file (see Chapter 8 [System-wide configuration of the library], page 280) will be used to expand the provided keyword. That is used to impose system-specific policies. It may be followed by additional options that will be appended to the system string (e.g., "@SYSTEM:+SRP"). The system file should have the format 'KEYWORD=VALUE', e.g., 'SYSTEM=NORMAL:+ARCFOUR-128'. Since version 3.5.1 it is allowed to specify fallback keywords such as @KEYWORD1,@KEYWORD2, and the first valid keyword will be used.
PERFORMANCE	All the known to be secure ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance. The message authenticity security level is of 64 bits or more, and the certificate verification profile is set to GNUTLS_PROFILE_LOW (80-bits).
NORMAL	Means all the known to be secure ciphersuites. The ciphers are sorted by security margin, although the 256-bit ciphers are included as a fallback only. The message authenticity security level is of 64 bits or more, and the certificate verification profile is set to GNUTLS_PROFILE_LOW (80-bits). This priority string implicitly enables ECDHE and DHE. The ECDHE ciphersuites are placed first in the priority order, but due to compatibility issues with the DHE ciphersuites they are placed last in the priority order, after the plain RSA ciphersuites.
LEGACY	This sets the NORMAL settings that were used for GnuTLS 3.2.x or earlier. There is no verification profile set, and the allowed DH primes are considered weak today (but are often used by misconfigured servers).
PFS	Means all the known to be secure ciphersuites that support perfect forward secrecy (ECDHE and DHE). The ciphers are sorted by security margin, although the 256-bit ciphers are included as a fallback only. The message authenticity security level is of 80 bits or more, and the certificate verification profile is set to GNUTLS_PROFILE_LOW (80-bits). This option is available since 3.2.4 or later.
SECURE128	Means all known to be secure ciphersuites that offer a security level 128-bit or more. The message authenticity security level is of 80 bits or more, and the certificate verification profile is set to GNUTLS_PROFILE_LOW (80-bits).
SECURE192	Means all the known to be secure ciphersuites that offer a security level 192-bit or more. The message authenticity security level is of 128 bits or more, and the certificate verification

Unless the initial keyword is "NONE" the defaults (in preference order) are for TLS protocols TLS 1.2, TLS1.1, TLS1.0; for certificate types X.509. In key exchange algorithms when in NORMAL or SECURE levels the perfect forward secrecy algorithms take precedence of the other protocols. In all cases all the supported key exchange algorithms are enabled.

Note that the SECURE levels distinguish between overall security level and message authenticity security level. That is because the message authenticity security level requires the adversary to break the algorithms at real-time during the protocol run, whilst the overall security level refers to off-line adversaries (e.g. adversaries breaking the ciphertext years after it was captured).

The NONE keyword, if used, must followed by keywords specifying the algorithms and protocols to be enabled. The other initial keywords do not require, but may be followed by such keywords. All level keywords can be combined, and for example a level of "SECURE256:+SECURE128" is allowed.

The order with which every algorithm or protocol is specified is significant. Algorithms specified before others will take precedence. The supported in the GnuTLS version corresponding to this document algorithms and protocols are shown in Table 6.4; to list the supported algorithms in your currently using version use `gnutls-cli -l`.

To avoid collisions in order to specify a protocol version with "VERS-", signature algorithms with "SIGN-" and certificate types with "CTYPE-". All other algorithms don't need a prefix. Each specified keyword (except for *special keywords*) can be prefixed with any of the following characters.

'!' or '-' appended with an algorithm will remove this algorithm.

"+" appended with an algorithm will add this algorithm.

Type	Keywords
Ciphers	Examples are AES-128-GCM, AES-256-GCM, AES-256-CBC, GOST28147-TC26Z-CNT; see also Table 3.1 for more options. Catch all name is CIPHER-ALL which will add all the algorithms from NORMAL priority. The shortcut for secure GOST algorithms is CIPHER-GOST-ALL.
Key exchange	RSA, DHE-RSA, DHE-DSS, SRP, SRP-RSA, SRP-DSS, PSK, DHE-PSK, ECDHE-PSK, ECDHE-RSA, ECDHE-ECDSA, VKO-GOST-12, ANON-ECDH, ANON-DH. Catch all name is KX-ALL which will add all the algorithms from NORMAL priority. Under TLS1.3, the DHE-PSK and ECDHE-PSK strings are equivalent and instruct for a Diffie-Hellman key exchange using the enabled groups. The shortcut for secure GOST algorithms is KX-GOST-ALL.
MAC	MD5, SHA1, SHA256, SHA384, GOST28147-TC26Z-IMIT, AEAD (used with GCM ciphers only). All algorithms from NORMAL priority can be accessed with MAC-ALL. The shortcut for secure GOST algorithms is MAC-GOST-ALL.
Compression algorithms	COMP-NULL, COMP-DEFLATE. Catch all is COMP-ALL.
TLS versions	VERS-TLS1.0, VERS-TLS1.1, VERS-TLS1.2, VERS-TLS1.3, VERS-DTLS1.0, VERS-DTLS1.2. Catch all are VERS-ALL, and will enable all protocols from NORMAL priority. To distinguish between TLS and DTLS versions you can use VERS-TLS-ALL and VERS-DTLS-ALL.
Signature algorithms	SIGN-RSA-SHA1, SIGN-RSA-SHA224, SIGN-RSA-SHA256, SIGN-RSA-SHA384, SIGN-RSA-SHA512, SIGN-DSA-SHA1, SIGN-DSA-SHA224, SIGN-DSA-SHA256, SIGN-RSA-MD5, SIGN-ECDSA-SHA1, SIGN-ECDSA-SHA224, SIGN-ECDSA-SHA256, SIGN-ECDSA-SHA384, SIGN-ECDSA-SHA512, SIGN-RSA-PSS-SHA256, SIGN-RSA-PSS-SHA384, SIGN-RSA-PSS-SHA512, SIGN-GOSTR341001, SIGN-GOSTR341012-256, SIGN-GOSTR341012-512. Catch all which enables all algorithms from NORMAL priority is SIGN-ALL. Shortcut which enables secure GOST algorithms is SIGN-GOST-ALL. This option is only considered for TLS 1.2 and later.
Groups	GROUP-SECP256R1, GROUP-SECP384R1, GROUP-SECP521R1, GROUP-X25519, GROUP-X448, GROUP-FFDHE2048, GROUP-FFDHE3072, GROUP-FFDHE4096, GROUP-FFDHE6144, and GROUP-FFDHE8192. Groups include both elliptic curve groups, e.g., SECP256R1, as well as finite field groups such as FFDHE2048. Catch all which enables all groups from NORMAL priority is GROUP-ALL. The helper keywords GROUP-DH-ALL, GROUP-GOST-ALL, and GROUP-EC-ALL are also

Note that the finite field groups (indicated by the FFDHE prefix) and DHE key exchange methods are generally slower[5] than their elliptic curves counterpart (ECDHE).

The available special keywords are shown in Table 6.5 and Table 6.6.

[5] It depends on the group in use. Groups with less bits are always faster, but the number of bits ties with the security parameter. See Section 6.11 [Selecting cryptographic key sizes], page 179, for the acceptable security levels.

Keyword	Description
%COMPAT	will enable compatibility mode. It might mean that violations of the protocols are allowed as long as maximum compatibility with problematic clients and servers is achieved. More specifically this string will tolerate packets over the maximum allowed TLS record, and add a padding to TLS Client Hello packet to prevent it being in the 256-512 range which is known to be causing issues with a commonly used firewall (see the %DUMBFW option).
%DUMBFW	will add a private extension with bogus data that make the client hello exceed 512 bytes. This avoids a black hole behavior in some firewalls. This is the [[RFC7685], page 749] client hello padding extension, also enabled with %COMPAT.
%NO_EXTENSIONS	will prevent the sending of any TLS extensions in client side. Note that TLS 1.2 requires extensions to be used, as well as safe renegotiation thus this option must be used with care. When this option is set no versions later than TLS1.2 can be negotiated.
%NO_TICKETS	will prevent the advertizing of the TLS session ticket extension. This is implied by the PFS keyword.
%NO_SESSION_HASH	will prevent the advertizing the TLS extended master secret (session hash) extension.
%SERVER_PRECEDENCE	The ciphersuite will be selected according to server priorities and not the client's.
%SSL3_RECORD_VERSION	will use SSL3.0 record version in client hello. By default GnuTLS will set the minimum supported version as the client hello record version (do not confuse that version with the proposed handshake version at the client hello).

Keyword	Description
%STATELESS_COMPRESSION	ignored; no longer used.
%DISABLE_WILDCARDS	will disable matching wildcards when comparing hostnames in certificates.
%NO_ETM	will disable the encrypt-then-mac TLS extension (RFC7366). This is implied by the %COMPAT keyword.
%FORCE_ETM	negotiate CBC ciphersuites only when both sides of the connection support encrypt-then-mac TLS extension (RFC7366).
%DISABLE_SAFE_RENEGOTIATION	will completely disable safe renegotiation completely. Do not use unless you know what you are doing.
%UNSAFE_RENEGOTIATION	will allow handshakes and re-handshakes without the safe renegotiation extension. Note that for clients this mode is insecure (you may be under attack), and for servers it will allow insecure clients to connect (which could be fooled by an attacker). Do not use unless you know what you are doing and want maximum compatibility.
%PARTIAL_RENEGOTIATION	will allow initial handshakes to proceed, but not re-handshakes. This leaves the client vulnerable to attack, and servers will be compatible with non-upgraded clients for initial handshakes. This is currently the default for clients and servers, for compatibility reasons.
%SAFE_RENEGOTIATION	will enforce safe renegotiation. Clients and servers will refuse to talk to an insecure peer. Currently this causes interoperability problems, but is required for full protection.
%FALLBACK_SCSV	will enable the use of the fallback signaling cipher suite value in the client hello. Note that this should be set only by applications that try to reconnect with a downgraded protocol version.

Finally the ciphersuites enabled by any priority string can be listed using the `gnutls-cli` application (see Section 10.1 [gnutls-cli Invocation], page 297), or by using the priority functions as in Section 7.5.3 [Listing the ciphersuites in a priority string], page 275.

Example priority strings are:

```
The system imposed security level:
    "SYSTEM"

The default priority without the HMAC-MD5:
    "NORMAL:-MD5"

Specifying RSA with AES-128-CBC:
    "NONE:+VERS-TLS-ALL:+MAC-ALL:+RSA:+AES-128-CBC:+SIGN-ALL:+COMP-NULL"

Specifying the defaults plus ARCFOUR-128:
    "NORMAL:+ARCFOUR-128"

Enabling the 128-bit secure ciphers, while disabling TLS 1.0:
    "SECURE128:-VERS-TLS1.0"

Enabling the 128-bit and 192-bit secure ciphers, while disabling all TLS versions
except TLS 1.2:
    "SECURE128:+SECURE192:-VERS-ALL:+VERS-TLS1.2"
```

6.11 Selecting cryptographic key sizes

Because many algorithms are involved in TLS, it is not easy to set a consistent security level. For this reason in Table 6.7 we present some correspondence between key sizes of symmetric algorithms and public key algorithms based on [[ECRYPT], page 753]. Those can be used to generate certificates with appropriate key sizes as well as select parameters for Diffie-Hellman and SRP authentication.

Security bits	RSA, DH and SRP parameter size	ECC key size	Security parameter (profile)	Description
<64	<768	<128	INSECURE	Considered to be insecure
64	768	128	VERY WEAK	Short term protection against individuals
72	1008	160	WEAK	Short term protection against small organizations
80	1024	160	LOW	Very short term protection against agencies (corresponds to ENISA legacy level)
96	1776	192	LEGACY	Legacy standard level
112	2048	224	MEDIUM	Medium-term protection
128	3072	256	HIGH	Long term protection (corresponds to ENISA future level)
192	8192	384	ULTRA	Even longer term protection
256	15424	512	FUTURE	Foreseeable future

Table 6.7: Key sizes and security parameters.

The first column provides a security parameter in a number of bits. This gives an indication of the number of combinations to be tried by an adversary to brute force a key. For example to test all possible keys in a 112 bit security parameter 2^{112} combinations have to be tried. For today's technology this is infeasible. The next two columns correlate the security parameter with actual bit sizes of parameters for DH, RSA, SRP and ECC algorithms. A mapping to `gnutls_sec_param_t` value is given for each security parameter, on the next column, and finally a brief description of the level.

Note, however, that the values suggested here are nothing more than an educated guess that is valid today. There are no guarantees that an algorithm will remain unbreakable or that these values will remain constant in time. There could be scientific breakthroughs that

cannot be predicted or total failure of the current public key systems by quantum computers. On the other hand though the cryptosystems used in TLS are selected in a conservative way and such catastrophic breakthroughs or failures are believed to be unlikely. The NIST publication SP 800-57 [[NISTSP80057], page 749] contains a similar table.

When using GnuTLS and a decision on bit sizes for a public key algorithm is required, use of the following functions is recommended:

`unsigned int gnutls_sec_param_to_pk_bits` [Function]
(*gnutls_pk_algorithm_t* `algo`, *gnutls_sec_param_t* `param`)

algo: is a public key algorithm

param: is a security parameter

When generating private and public key pairs a difficult question is which size of "bits" the modulus will be in RSA and the group size in DSA. The easy answer is 1024, which is also wrong. This function will convert a human understandable security parameter to an appropriate size for the specific algorithm.

Returns: The number of bits, or (0).

Since: 2.12.0

`gnutls_sec_param_t gnutls_pk_bits_to_sec_param` [Function]
(*gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`)

algo: is a public key algorithm

bits: is the number of bits

This is the inverse of `gnutls_sec_param_to_pk_bits()` . Given an algorithm and the number of bits, it will return the security parameter. This is a rough indication.

Returns: The security parameter.

Since: 2.12.0

Those functions will convert a human understandable security parameter of `gnutls_sec_param_t` type, to a number of bits suitable for a public key algorithm.

const char * [gnutls_sec_param_get_name], page 457, (gnutls_sec_param_t *param*)

The following functions will set the minimum acceptable group size for Diffie-Hellman and SRP authentication.

void [gnutls_dh_set_prime_bits], page 403, (gnutls_session_t *session*, unsigned int *bits*)
void [gnutls_srp_set_prime_bits], page 476, (gnutls_session_t *session*, unsigned int *bits*)

6.12 Advanced topics

6.12.1 Virtual hosts and credentials

Often when operating with virtual hosts, one may not want to associate a particular certificate set to the credentials function early, before the virtual host is known. That can be achieved by calling [gnutls_credentials_set], page 396 within a handshake pre-hook for

client hello. That message contains the peer's intended hostname, and if read, and the appropriate credentials are set, gnutls will be able to continue in the handshake process. A brief usage example is shown below.

```
static int ext_hook_func(void *ctx, unsigned tls_id,
                         const unsigned char *data, unsigned size)
{
if (tls_id == 0) { /* server name */
/* figure the advertized name - the following hack
                * relies on the fact that this extension only supports
                * DNS names, and due to a protocol bug cannot be extended
                * to support anything else. */
if (name < 5) return 0;
name = data+5;
name_size = size-5;
}
return 0;
}

static int
handshake_hook_func(gnutls_session_t session, unsigned int htype,
                    unsigned when, unsigned int incoming, const gnutls_datum_t *msg)
{
    int ret;

    assert(htype == GNUTLS_HANDSHAKE_CLIENT_HELLO);
    assert(when == GNUTLS_HOOK_PRE);

    ret = gnutls_ext_raw_parse(NULL, ext_hook_func, msg,
                               GNUTLS_EXT_RAW_FLAG_TLS_CLIENT_HELLO);
    assert(ret >= 0);

    gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, cred);

    return ret;
}

int main()
{
  ...

  gnutls_handshake_set_hook_function(server, GNUTLS_HANDSHAKE_CLIENT_HELLO,
                                     GNUTLS_HOOK_PRE, handshake_hook_func);
  ...
}
```

`void gnutls_handshake_set_hook_function (`*gnutls_session_t* `session`, *unsigned int* `htype`, *int* `when`, *gnutls_handshake_hook_func* `func`) [Function]

session: is a `gnutls_session_t` type

htype: the `gnutls_handshake_description_t` of the message to hook at

when: `GNUTLS_HOOK_` * depending on when the hook function should be called

func: is the function to be called

This function will set a callback to be called after or before the specified handshake message has been received or generated. This is a generalization of `gnutls_handshake_set_post_client_hello_function()` .

To call the hook function prior to the message being generated or processed use `GNUTLS_HOOK_PRE` as `when` parameter, `GNUTLS_HOOK_POST` to call after, and `GNUTLS_HOOK_BOTH` for both cases.

This callback must return 0 on success or a gnutls error code to terminate the handshake.

To hook at all handshake messages use an `htype` of `GNUTLS_HANDSHAKE_ANY` .

Warning: You should not use this function to terminate the handshake based on client input unless you know what you are doing. Before the handshake is finished there is no way to know if there is a man-in-the-middle attack being performed.

6.12.2 Session resumption

To reduce time and network traffic spent in a handshake the client can request session resumption from a server that previously shared a session with the client.

Under TLS 1.2, in order to support resumption a server can either store the session security parameters in a local database or use session tickets (see Section 3.6.3 [Session tickets], page 12) to delegate storage to the client.

Under TLS 1.3, session resumption is only available through session tickets, and multiple tickets could be sent from server to client. That provides the following advantages:

- When tickets are not re-used the subsequent client sessions cannot be associated with each other by an eavesdropper
- On post-handshake authentication the server may send different tickets asynchronously for each identity used by client.

Client side

The client has to retrieve and store the session parameters. Before establishing a new session to the same server the parameters must be re-associated with the GnuTLS session using [gnutls_session_set_data], page 465.

int [gnutls_session_get_data2], page 460, (gnutls_session_t *session*, gnutls_datum_t * *data*)
int [gnutls_session_set_data], page 465, (gnutls_session_t *session*, const void * *session_data*, size_t *session_data_size*)

Keep in mind that sessions will be expired after some time, depending on the server, and a server may choose not to resume a session even when requested to. The expiration is to

prevent temporal session keys from becoming long-term keys. Also note that as a client you must enable, using the priority functions, at least the algorithms used in the last session.

`int gnutls_session_is_resumed` (*gnutls_session_t* ***session***) [Function]
session: is a `gnutls_session_t` type.

Checks whether session is resumed or not. This is functional for both server and client side.

Returns: non zero if this session is resumed, or a zero if this is a new session.

`int gnutls_session_get_id2` (*gnutls_session_t* ***session***, *gnutls_datum_t* * ***session_id***) [Function]
session: is a `gnutls_session_t` type.

session_id: will point to the session ID.

Returns the TLS session identifier. The session ID is selected by the server, and in older versions of TLS was a unique identifier shared between client and server which was persistent across resumption. In the latest version of TLS (1.3) or TLS 1.2 with session tickets, the notion of session identifiers is undefined and cannot be relied for uniquely identifying sessions across client and server.

In client side this function returns the identifier returned by the server, and cannot be assumed to have any relation to session resumption. In server side this function is guaranteed to return a persistent identifier of the session since GnuTLS 3.6.4, which may not necessarily map into the TLS session ID value. Prior to that version the value could only be considered a persistent identifier, under TLS1.2 or earlier and when no session tickets were in use.

The session identifier value returned is always less than `GNUTLS_MAX_SESSION_ID_SIZE` and should be treated as constant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.1.4

Server side

A server enabling both session tickets and a storage for session data would use session tickets when clients support it and the storage otherwise.

A storing server needs to specify callback functions to store, retrieve and delete session data. These can be registered with the functions below. The stored sessions in the database can be checked using [gnutls_db_check_entry], page 396 for expiration.

void [gnutls_db_set_retrieve_function], page 398, (gnutls_session_t *session*, gnutls_db_retr_func *retr_func*)
void [gnutls_db_set_store_function], page 398, (gnutls_session_t *session*, gnutls_db_store_func *store_func*)
void [gnutls_db_set_ptr], page 397, (gnutls_session_t *session*, void * *ptr*)
void [gnutls_db_set_remove_function], page 398, (gnutls_session_t *session*, gnutls_db_remove_func *rem_func*)
int [gnutls_db_check_entry], page 396, (gnutls_session_t *session*, gnutls_datum_t *session_entry*)

A server supporting session tickets must generate ticket encryption and authentication keys using [gnutls_session_ticket_key_generate], page 469. Those keys should be associated with the GnuTLS session using [gnutls_session_ticket_enable_server], page 468.

Those will be the initial keys, but GnuTLS will rotate them regularly. The key rotation interval can be changed with [gnutls_db_set_cache_expiration], page 397 and will be set to three times the ticket expiration time (ie. three times the value given in that function). Every such interval, new keys will be generated from those initial keys. This is a necessary mechanism to prevent the keys from becoming long-term keys and as such preserve forward-secrecy in the issued session tickets. If no explicit key rotation interval is provided, GnuTLS will rotate them every 18 hours by default.

The master key can be shared between processes or between systems. Processes which share the same master key will generate the same rotated subkeys, assuming they share the same time (irrespective of timezone differences).

`int gnutls_session_ticket_enable_server` (*gnutls_session_t session*, *const gnutls_datum_t * key*) [Function]

session: is a `gnutls_session_t` type.

key: key to encrypt session parameters.

Request that the server should attempt session resumption using session tickets, i.e., by delegating storage to the client. `key` must be initialized using `gnutls_session_ticket_key_generate()` . To avoid leaking that key, use `gnutls_memset()` prior to releasing it.

The default ticket expiration time can be overridden using `gnutls_db_set_cache_expiration()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 2.10.0

`int gnutls_session_ticket_key_generate` (*gnutls_datum_t * key*) [Function]

key: is a pointer to a `gnutls_datum_t` which will contain a newly created key.

Generate a random key to encrypt security parameters within SessionTicket.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 2.10.0

`int gnutls_session_resumption_requested` (*gnutls_session_t session*) [Function]

session: is a `gnutls_session_t` type.

Check whether the client has asked for session resumption. This function is valid only on server side.

Returns: non zero if session resumption was asked, or a zero if not.

The expiration time for session resumption, either in tickets or stored data is set using [gnutls_db_set_cache_expiration], page 397. This function also controls the ticket key rotation period. Currently, the session key rotation interval is set to 3 times the expiration time set by this function.

Under TLS 1.3, the server sends by default 2 tickets, and can send additional session tickets at any time using [gnutls_session_ticket_send], page 469.

`int gnutls_session_ticket_send` (*gnutls_session_t* `session`, *unsigned* `nr`, *unsigned* `flags`) [Function]

session: is a `gnutls_session_t` type.

nr: the number of tickets to send

flags: must be zero

Sends a fresh session ticket to the peer. This is relevant only in server side under TLS1.3. This function may also return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` and in that case it must be called again.

Returns: `GNUTLS_E_SUCCESS` on success, or a negative error code.

6.12.3 Certificate verification

In this section the functionality for additional certificate verification methods is listed. These methods are intended to be used in addition to normal PKI verification, in order to reduce the risk of a compromised CA being undetected.

6.12.3.1 Trust on first use

The GnuTLS library includes functionality to use an SSH-like trust on first use authentication. The available functions to store and verify public keys are listed below.

`int gnutls_verify_stored_pubkey` (*const char * db_name*, *gnutls_tdb_t* `tdb`, *const char * host*, *const char * service*, *gnutls_certificate_type_t* `cert_type`, *const gnutls_datum_t * cert*, *unsigned int* `flags`) [Function]

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

cert_type: The type of the certificate

cert: The raw (der) data of the certificate

flags: should be 0.

This function will try to verify a raw public-key or a public-key provided via a raw (DER-encoded) certificate using a list of stored public keys. The `service` field if non-NULL should be a port number.

The `db_name` variable if non-null specifies a custom backend for the retrieval of entries. If it is NULL then the default file backend will be used. In POSIX-like systems the file backend uses the $HOME/.gnutls/known_hosts file.

Note that if the custom storage backend is provided the retrieval function should return `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` if the host/service pair is found but key doesn't match, `GNUTLS_E_NO_CERTIFICATE_FOUND` if no such host/service with the given key is found, and 0 if it was found. The storage function should return 0 on success.

As of GnuTLS 3.6.6 this function also verifies raw public keys.

Returns: If no associated public key is found then `GNUTLS_E_NO_CERTIFICATE_FOUND` will be returned. If a key is found but does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned. On success, `GNUTLS_E_SUCCESS` (0) is returned, or a negative error value on other errors.

Since: 3.0.13

`int gnutls_store_pubkey` (*const char * `db_name`, gnutls_tdb_t `tdb`, const char * `host`, const char * `service`, gnutls_certificate_type_t `cert_type`, const gnutls_datum_t * `cert`, time_t `expiration`, unsigned int `flags`*) [Function]

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

cert_type: The type of the certificate

cert: The data of the certificate

expiration: The expiration time (use 0 to disable expiration)

flags: should be 0.

This function will store a raw public-key or a public-key provided via a raw (DER-encoded) certificate to the list of stored public keys. The key will be considered valid until the provided expiration time.

The `tdb` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Unless an alternative `tdb` is provided, the storage format is a textual format consisting of a line for each host with fields separated by '`|`'. The contents of the fields are a format-identifier which is set to 'g0', the hostname that the rest of the data applies to, the numeric port or host name, the expiration time in seconds since the epoch (0 for no expiration), and a base64 encoding of the raw (DER) public key information (SPKI) of the peer.

As of GnuTLS 3.6.6 this function also accepts raw public keys.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0.13

In addition to the above the [gnutls_store_commitment], page 480 can be used to implement a key-pinning architecture as in [[KEYPIN], page 749]. This provides a way for web server to commit on a public key that is not yet active.

`int gnutls_store_commitment` (*const char * **db_name***, *gnutls_tdb_t* [Function] ***tdb***, *const char * **host***, *const char * **service***, *gnutls_digest_algorithm_t* ***hash_algo***, *const gnutls_datum_t * **hash***, *time_t* ***expiration***, *unsigned int* ***flags***)

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

hash_algo: The hash algorithm type

hash: The raw hash

expiration: The expiration time (use 0 to disable expiration)

flags: should be 0 or `GNUTLS_SCOMMIT_FLAG_ALLOW_BROKEN` .

This function will store the provided hash commitment to the list of stored public keys. The key with the given hash will be considered valid until the provided expiration time.

The `tdb` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Note that this function is not thread safe with the default backend.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

The storage and verification functions may be used with the default text file based back-end, or another back-end may be specified. That should contain storage and retrieval functions and specified as below.

int [gnutls_tdb_init], page 483, (gnutls_tdb_t * *tdb*)
void [gnutls_tdb_deinit], page 483, (gnutls_tdb_t *tdb*)
void [gnutls_tdb_set_verify_func], page 484, (gnutls_tdb_t *tdb*, gnutls_tdb_verify_func *verify*)
void [gnutls_tdb_set_store_func], page 484, (gnutls_tdb_t *tdb*, gnutls_tdb_store_func *store*)
void [gnutls_tdb_set_store_commitment_func], page 484, (gnutls_tdb_t *tdb*, gnutls_tdb_store_commitment_func *cstore*)

6.12.3.2 DANE verification

Since the DANE library is not included in GnuTLS it requires programs to be linked against it. This can be achieved with the following commands.

```
gcc -o foo foo.c ‘pkg-config gnutls-dane --cflags --libs‘
```

When a program uses the GNU autoconf system, then the following line or similar can be used to detect the presence of the library.

```
PKG_CHECK_MODULES([LIBDANE], [gnutls-dane >= 3.0.0])
```

```
AC_SUBST([LIBDANE_CFLAGS])
AC_SUBST([LIBDANE_LIBS])
```

The high level functionality provided by the DANE library is shown below.

`int dane_verify_crt` (*dane_state_t* `s`, *const gnutls_datum_t ** `chain`, *unsigned* `chain_size`, *gnutls_certificate_type_t* `chain_type`, *const char ** `hostname`, *const char ** `proto`, *unsigned int* `port`, *unsigned int* `sflags`, *unsigned int* `vflags`, *unsigned int ** `verify`) [Function]

s: A DANE state structure (may be NULL)

chain: A certificate chain

chain_size: The size of the chain

chain_type: The type of the certificate chain

hostname: The hostname associated with the chain

proto: The protocol of the service connecting (e.g. tcp)

port: The port of the service connecting (e.g. 443)

sflags: Flags for the initialization of `s` (if NULL)

vflags: Verification flags; an OR'ed list of `dane_verify_flags_t` .

verify: An OR'ed list of `dane_verify_status_t` .

This function will verify the given certificate chain against the CA constrains and/or the certificate available via DANE. If no information via DANE can be obtained the flag `DANE_VERIFY_NO_DANE_INFO` is set. If a DNSSEC signature is not available for the DANE record then the verify flag `DANE_VERIFY_NO_DNSSEC_DATA` is set.

Due to the many possible options of DANE, there is no single threat model countered. When notifying the user about DANE verification results it may be better to mention: DANE verification did not reject the certificate, rather than mentioning a successful DANE verication.

Note that this function is designed to be run in addition to PKIX - certificate chain - verification. To be run independently the `DANE_VFLAG_ONLY_CHECK_EE_USAGE` flag should be specified; then the function will check whether the key of the peer matches the key advertized in the DANE entry.

Returns: a negative error code on error and `DANE_E_SUCCESS` (0) when the DANE entries were successfully parsed, irrespective of whether they were verified (see `verify` for that information). If no usable entries were encountered `DANE_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

int [dane_verify_session_crt], page 721, (dane_state_t *s*, gnutls_session_t *session*, const char * *hostname*, const char * *proto*, unsigned int *port*, unsigned int *sflags*, unsigned int *vflags*, unsigned int * *verify*)
*const char ** [dane_strerror], page 719, (int *error*)

Note that the `dane_state_t` structure that is accepted by both verification functions is optional. It is required when many queries are performed to optimize against multiple re-initializations of the resolving back-end and loading of DNSSEC keys.

The following flags are returned by the verify functions to indicate the status of the verification.

`DANE_VERIFY_CA_CONSTRAINTS_VIOLATED`
: The CA constraints were violated.

`DANE_VERIFY_CERT_DIFFERS`
: The certificate obtained via DNS differs.

`DANE_VERIFY_UNKNOWN_DANE_INFO`
: No known DANE data was found in the DNS record.

Figure 6.3: The DANE verification status flags.

In order to generate a DANE TLSA entry to use in a DNS server you may use danetool (see Section 4.2.8 [danetool Invocation], page 87).

6.12.4 TLS 1.2 re-authentication

In TLS 1.2 or earlier there is no distinction between re-key, re-authentication, and re-negotiation. All of these use cases are handled by the TLS' rehandshake process. For that reason in GnuTLS rehandshake is not transparent to the application, and the application must explicitly take control of that process. In addition GnuTLS since version 3.5.0 will not allow the peer to switch identities during a rehandshake. The threat addressed by that behavior depends on the application protocol, but primarily it protects applications from being misled by a rehandshake which switches the peer's identity. Applications can disable this protection by using the `GNUTLS_ALLOW_ID_CHANGE` flag in [gnutls_init], page 422.

The following paragraphs explain how to safely use the rehandshake process.

6.12.4.1 Client side

According to the TLS specification a client may initiate a rehandshake at any time. That can be achieved by calling [gnutls_handshake], page 414 and rely on its return value for the outcome of the handshake (the server may deny a rehandshake). If a server requests a re-handshake, then a call to [gnutls_record_recv], page 450 will return GNUTLS_E_REHANDSHAKE in the client, instructing it to call [gnutls_handshake], page 414. To deny a rehandshake request by the server it is recommended to send a warning alert of type GNUTLS_A_NO_RENEGOTIATION.

Due to limitations of early protocol versions, it is required to check whether safe renegotiation is in place, i.e., using [gnutls_safe_renegotiation_status], page 456, which ensures that the server remains the same as the initial.

To make re-authentication transparent to the application when requested by the server, use the `GNUTLS_AUTO_REAUTH` flag on the [gnutls_init], page 422 call. In that case the re-authentication will happen in the call of [gnutls_record_recv], page 450 that received the reauthentication request.

`unsigned gnutls_safe_renegotiation_status (`*gnutls_session_t session*`)` [Function]

session: is a `gnutls_session_t` type.

Can be used to check whether safe renegotiation is being used in the current session.

Returns: 0 when safe renegotiation is not used and non (0) when safe renegotiation is used.

Since: 2.10.0

6.12.4.2 Server side

A server which wants to instruct the client to re-authenticate, should call [gnutls_rehandshake], page 456 and wait for the client to re-authenticate. It is recommended to only request re-handshake when safe renegotiation is enabled for that session (see [gnutls_safe_renegotiation_status], page 456 and the discussion in Section 3.6.5 [Safe renegotiation], page 13). A server could also encounter the GNUTLS_E_REHANDSHAKE error code while receiving data. That indicates a client-initiated re-handshake request. In that case the server could ignore that request, perform handshake (unsafe when done generally), or even drop the connection.

`int gnutls_rehandshake` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

This function can only be called in server side, and instructs a TLS 1.2 or earlier client to renegotiate parameters (perform a handshake), by sending a hello request message.

If this function succeeds, the calling application should call `gnutls_record_recv()` until `GNUTLS_E_REHANDSHAKE` is returned to clear any pending data. If the `GNUTLS_E_REHANDSHAKE` error code is not seen, then the handshake request was not followed by the peer (the TLS protocol does not require the client to do, and such compliance should be handled by the application protocol).

Once the `GNUTLS_E_REHANDSHAKE` error code is seen, the calling application should proceed to calling `gnutls_handshake()` to negotiate the new parameters.

If the client does not wish to renegotiate parameters he may reply with an alert message, and in that case the return code seen by subsequent `gnutls_record_recv()` will be `GNUTLS_E_WARNING_ALERT_RECEIVED` with the specific alert being `GNUTLS_A_NO_RENEGOTIATION` . A client may also choose to ignore this request.

Under TLS 1.3 this function is equivalent to `gnutls_session_key_update()` with the `GNUTLS_KU_PEER` flag. In that case subsequent calls to `gnutls_record_recv()` will not return `GNUTLS_E_REHANDSHAKE` , and calls to `gnutls_handshake()` in server side are a no-op.

This function always fails with `GNUTLS_E_INVALID_REQUEST` when called in client side.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

6.12.5 TLS 1.3 re-authentication and re-key

The TLS 1.3 protocol distinguishes between re-key and re-authentication. The re-key process ensures that fresh keys are supplied to the already negotiated parameters, and on GnuTLS can be initiated using [gnutls_session_key_update], page 464. The re-key process can be one-way (i.e., the calling party only changes its keys), or two-way where the peer is requested to change keys as well.

The re-authentication process, allows the connected client to switch identity by presenting a new certificate. Unlike TLS 1.2, the server is not allowed to change identities. That client re-authentication, or post-handshake authentication can be initiated only by the server using [gnutls_reauth], page 447, and only if a client has advertized support for it. Both server and client have to explicitly enable support for post handshake authentication using the `GNUTLS_POST_HANDSHAKE_AUTH` flag at [gnutls_init], page 422.

A client receiving a re-authentication request will "see" the error code `GNUTLS_E_REAUTH_REQUEST` at [gnutls_record_recv], page 450. At this point, it should also call [gnutls_reauth], page 447.

To make re-authentication transparent to the application when requested by the server, use the `GNUTLS_AUTO_REAUTH` and `GNUTLS_POST_HANDSHAKE_AUTH` flags on the [gnutls_init], page 422 call. In that case the re-authentication will happen in the call of [gnutls_record_recv], page 450 that received the reauthentication request.

6.12.6 Parameter generation

Prior to GnuTLS 3.6.0 for the ephemeral or anonymous Diffie-Hellman (DH) TLS ciphersuites the application was required to generate or provide DH parameters. That is no longer necessary as GnuTLS utilizes DH parameters and negotiation from [[RFC7919], page 750].

Applications can tune the used parameters by explicitly specifying them in the priority string. In server side applications can set the minimum acceptable level of DH parameters by calling [gnutls_certificate_set_known_dh_params], page 374, [gnutls_anon_set_server_known_dh_params], page 363, or [gnutls_psk_set_server_known_dh_params], page 446, depending on the type of the credentials, to set the lower acceptable parameter limits. Typical applications should rely on the default settings.

int [gnutls_certificate_set_known_dh_params], page 374,
(gnutls_certificate_credentials_t *res*, gnutls_sec_param_t *sec_param*)
int [gnutls_anon_set_server_known_dh_params], page 363,
(gnutls_anon_server_credentials_t *res*, gnutls_sec_param_t *sec_param*)
int [gnutls_psk_set_server_known_dh_params], page 446,
(gnutls_psk_server_credentials_t *res*, gnutls_sec_param_t *sec_param*)

6.12.6.1 Legacy parameter generation

Note that older than 3.5.6 versions of GnuTLS provided functions to generate or import arbitrary DH parameters from a file. This practice is still supported but discouraged in current versions. There is no known advantage from using random parameters, while there have been several occasions where applications were utilizing incorrect, weak or insecure parameters. This is the main reason GnuTLS includes the well-known parameters of [[RFC7919], page 750] and recommends applications utilizing them.

In older applications which require to specify explicit DH parameters, we recommend using `certtool` (of GnuTLS 3.5.6 or later) with the `--get-dh-params` option to obtain the FFDHE parameters discussed above. The output parameters of the tool are in PKCS#3 format and can be imported by most existing applications.

The following functions are still supported but considered obsolete.

int [gnutls_dh_params_generate2], page 401, (gnutls_dh_params_t *dparams*, unsigned int *bits*)
int [gnutls_dh_params_import_pkcs3], page 402, (gnutls_dh_params_t *params*, const gnutls_datum_t * *pkcs3_params*, gnutls_x509_crt_fmt_t *format*)
void [gnutls_certificate_set_dh_params], page 373, (gnutls_certificate_credentials_t *res*, gnutls_dh_params_t *dh_params*)

6.12.7 Deriving keys for other applications/protocols

In several cases, after a TLS connection is established, it is desirable to derive keys to be used in another application or protocol (e.g., in an other TLS session using pre-shared keys). The following describe GnuTLS' implementation of RFC5705 to extract keys based on a session's master secret.

The API to use is [gnutls_prf_rfc5705], page 434. The function needs to be provided with a label, and additional context data to mix in the `context` parameter.

`int gnutls_prf_rfc5705` (*gnutls_session_t* `session`, *size_t* `label_size`, *const char* * `label`, *size_t* `context_size`, *const char* * `context`, *size_t* `outsize`, *char* * `out`) [Function]

session: is a `gnutls_session_t` type.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

context_size: length of the `extra` variable.

context: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-allocated buffer to hold the generated data.

Exports keying material from TLS/DTLS session to an application, as specified in RFC5705.

In the TLS versions prior to 1.3, it applies the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data, seeded with the client and server random fields.

In TLS 1.3, it applies HKDF on the exporter master secret derived from the master secret.

The `label` variable usually contains a string denoting the purpose for the generated data.

The `context` variable can be used to add more data to the seed, after the random variables. It can be used to make sure the generated output is strongly connected to some additional data (e.g., a string used in user authentication).

The output is placed in `out` , which must be pre-allocated.

Note that, to provide the RFC5705 context, the `context` variable must be non-null.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 3.4.4

For example, after establishing a TLS session using [gnutls_handshake], page 414, you can obtain 32-bytes to be used as key, using this call:

```
#define MYLABEL "EXPORTER-My-protocol-name"
#define MYCONTEXT "my-protocol's-1st-session"

char out[32];
rc = gnutls_prf_rfc5705 (session, sizeof(MYLABEL)-1, MYLABEL,
                         sizeof(MYCONTEXT)-1, MYCONTEXT, 32, out);
```

The output key depends on TLS' master secret, and is the same on both client and server.

For legacy applications which need to use a more flexible API, there is [gnutls_prf], page 431, which in addition, allows to switch the mix of the client and server random nonces, using the `server_random_first` parameter. For additional flexibility and low-level access to the TLS1.2 PRF, there is a low-level TLS PRF interface called [gnutls_prf_raw], page 433. That however is not functional under newer protocol versions.

6.12.8 Channel bindings

In user authentication protocols (e.g., EAP or SASL mechanisms) it is useful to have a unique string that identifies the secure channel that is used, to bind together the user authentication with the secure channel. This can protect against man-in-the-middle attacks in some situations. That unique string is called a "channel binding". For background and discussion see [[RFC5056], page 753].

In GnuTLS you can extract a channel binding using the [gnutls_session_channel_binding], page 458 function. Currently only the type `GNUTLS_CB_TLS_UNIQUE` is supported, which corresponds to the `tls-unique` channel binding for TLS defined in [[RFC5929], page 753].

The following example describes how to print the channel binding data. Note that it must be run after a successful TLS handshake.

```
{
  gnutls_datum_t cb;
  int rc;

  rc = gnutls_session_channel_binding (session,
                                       GNUTLS_CB_TLS_UNIQUE,
                                       &cb);
  if (rc)
    fprintf (stderr, "Channel binding error: %s\n",
             gnutls_strerror (rc));
  else
    {
      size_t i;
      printf ("- Channel binding 'tls-unique': ");
      for (i = 0; i < cb.size; i++)
        printf ("%02x", cb.data[i]);
      printf ("\n");
    }
}
```

6.12.9 Interoperability

The TLS protocols support many ciphersuites, extensions and version numbers. As a result, few implementations are not able to properly interoperate once faced with extensions or version protocols they do not support and understand. The TLS protocol allows for a graceful downgrade to the commonly supported options, but practice shows it is not always implemented correctly.

Because there is no way to achieve maximum interoperability with broken peers without sacrificing security, GnuTLS ignores such peers by default. This might not be acceptable in cases where maximum compatibility is required. Thus we allow enabling compatibility with broken peers using priority strings (see Section 6.10 [Priority Strings], page 172). A conservative priority string that would disable certain TLS protocol options that are known to cause compatibility problems, is shown below.

```
NORMAL:%COMPAT
```

For very old broken peers that do not tolerate TLS version numbers over TLS 1.0 another priority string is:

```
NORMAL:-VERS-ALL:+VERS-TLS1.0:+VERS-SSL3.0:%COMPAT
```

This priority string will in addition to above, only enable SSL 3.0 and TLS 1.0 as protocols.

6.12.10 Compatibility with the OpenSSL library

To ease GnuTLS' integration with existing applications, a compatibility layer with the OpenSSL library is included in the `gnutls-openssl` library. This compatibility layer is not complete and it is not intended to completely re-implement the OpenSSL API with GnuTLS. It only provides limited source-level compatibility.

The prototypes for the compatibility functions are in the `gnutls/openssl.h` header file. The limitations imposed by the compatibility layer include:

- Error handling is not thread safe.

7 GnuTLS application examples

In this chapter several examples of real-world use cases are listed. The examples are simplified to promote readability and contain little or no error checking.

7.1 Client examples

This section contains examples of TLS and SSL clients, using GnuTLS. Note that some of the examples require functions implemented by another example.

7.1.1 Client example with X.509 certificate support

Let's assume now that we want to create a TCP client which communicates with servers that use X.509 certificate authentication. The following client is a very simple TLS client, which uses the high level verification functions for certificates, but does not support session resumption.

Note that this client utilizes functionality present in the latest GnuTLS version. For a reasonably portable version see Section 7.3.7 [Legacy client example with X.509 certificate support], page 244.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include "examples.h"

/* A very basic TLS client, with X.509 authentication and server certificate
 * verification. Note that error recovery is minimal for simplicity.
 */

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
```

```
extern void tcp_close(int sd);

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1], *desc;
        gnutls_datum_t out;
        int type;
        unsigned status;
        gnutls_certificate_credentials_t xcred;

        if (gnutls_check_version("3.4.6") == NULL) {
                fprintf(stderr, "GnuTLS 3.4.6 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        /* X509 stuff */
        CHECK(gnutls_certificate_allocate_credentials(&xcred));

        /* sets the system trusted CAs for Internet PKI */
        CHECK(gnutls_certificate_set_x509_system_trust(xcred));

        /* If client holds a certificate it can be set using the following:
         *
         gnutls_certificate_set_x509_key_file (xcred, "cert.pem", "key.pem",
         GNUTLS_X509_FMT_PEM);
         */

        /* Initialize TLS session */
        CHECK(gnutls_init(&session, GNUTLS_CLIENT));

        CHECK(gnutls_server_name_set(session, GNUTLS_NAME_DNS, "www.example.com",
                                     strlen("www.example.com")));

        /* It is recommended to use the default priorities */
        CHECK(gnutls_set_default_priority(session));

        /* put the x509 credentials to the current session
         */
        CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred));
        gnutls_session_set_verify_cert(session, "www.example.com", 0);

        /* connect to the peer
```

```
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);
gnutls_handshake_set_timeout(session,
                             GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

/* Perform the TLS handshake
 */
do {
        ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);
if (ret < 0) {
        if (ret == GNUTLS_E_CERTIFICATE_VERIFICATION_ERROR) {
                /* check certificate verification status */
                type = gnutls_certificate_type_get(session);
                status = gnutls_session_get_verify_cert_status(session);
                CHECK(gnutls_certificate_verification_status_print(status,
                      type, &out, 0));
                printf("cert verify output: %s\n", out.data);
                gnutls_free(out.data);
        }
        fprintf(stderr, "*** Handshake failed: %s\n", gnutls_strerror(ret));
        goto end;
} else {
        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
        gnutls_free(desc);
}

/* send data */
LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n", gnutls_strerror(ret));
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
        goto end;
}

if (ret > 0) {
        printf("- Received %d bytes: ", ret);
```

```
                for (ii = 0; ii < ret; ii++) {
                        fputc(buffer[ii], stdout);
                }
                fputs("\n", stdout);
        }

        CHECK(gnutls_bye(session, GNUTLS_SHUT_RDWR));

      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}
```

7.1.2 Datagram TLS client example

This is a client that uses UDP to connect to a server. This is the DTLS equivalent to the TLS example with X.509 certificates.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <assert.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/dtls.h>

/* A very basic Datagram TLS client, over UDP with X.509 authentication.
 */

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
```

```
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int udp_connect(void);
extern void udp_close(int sd);
extern int verify_certificate_callback(gnutls_session_t session);

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_certificate_credentials_t xcred;

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        /* X509 stuff */
        CHECK(gnutls_certificate_allocate_credentials(&xcred));

        /* sets the system trusted CAs for Internet PKI */
        CHECK(gnutls_certificate_set_x509_system_trust(xcred));

        /* Initialize TLS session */
        CHECK(gnutls_init(&session, GNUTLS_CLIENT | GNUTLS_DATAGRAM));

        /* Use default priorities */
        CHECK(gnutls_set_default_priority(session));

        /* put the x509 credentials to the current session */
        CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred));
        CHECK(gnutls_server_name_set(session, GNUTLS_NAME_DNS, "www.example.com",
                                     strlen("www.example.com")));

        gnutls_session_set_verify_cert(session, "www.example.com", 0);

        /* connect to the peer */
```

```
sd = udp_connect();

gnutls_transport_set_int(session, sd);

/* set the connection MTU */
gnutls_dtls_set_mtu(session, 1000);
/* gnutls_dtls_set_timeouts(session, 1000, 60000); */

/* Perform the TLS handshake */
do {
        ret = gnutls_handshake(session);
}
while (ret == GNUTLS_E_INTERRUPTED || ret == GNUTLS_E_AGAIN);
/* Note that DTLS may also receive GNUTLS_E_LARGE_PACKET */

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
        gnutls_free(desc);
}

LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n", gnutls_strerror(ret));
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
        goto end;
}

if (ret > 0) {
        printf("- Received %d bytes: ", ret);
        for (ii = 0; ii < ret; ii++) {
                fputc(buffer[ii], stdout);
        }
        fputs("\n", stdout);
}
```

```
        /* It is suggested not to use GNUTLS_SHUT_RDWR in DTLS
         * connections because the peer's closure message might
         * be lost */
        CHECK(gnutls_bye(session, GNUTLS_SHUT_WR));

      end:

        udp_close(sd);

        gnutls_deinit(session);

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}
```

7.1.3 Using a smart card with TLS

This example will demonstrate how to load keys and certificates from a smart-card or any other PKCS #11 token, and use it in a TLS connection. The difference between this and the Section 7.1.1 [Client example with X.509 certificate support], page 196, is that the client keys are provided as PKCS #11 URIs instead of files.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/pkcs11.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <getpass.h>            /* for getpass() */
```

```
/* A TLS client that loads the certificate and key.
 */

#define CHECK(x) assert((x)>=0)

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"
#define MIN(x,y) (((x)<(y))?(x):(y))

#define CAFILE "/etc/ssl/certs/ca-certificates.crt"

/* The URLs of the objects can be obtained
 * using p11tool --list-all --login
 */
#define KEY_URL "pkcs11:manufacturer=SomeManufacturer;object=Private%20Key" \
  ";objecttype=private;id=%db%5b%3e%b5%72%33"
#define CERT_URL "pkcs11:manufacturer=SomeManufacturer;object=Certificate;" \
  "objecttype=cert;id=db%5b%3e%b5%72%33"

extern int tcp_connect(void);
extern void tcp_close(int sd);

static int
pin_callback(void *user, int attempt, const char *token_url,
             const char *token_label, unsigned int flags, char *pin,
             size_t pin_max)
{
        const char *password;
        int len;

        printf("PIN required for token '%s' with URL '%s'\n", token_label,
               token_url);
        if (flags & GNUTLS_PIN_FINAL_TRY)
                printf("*** This is the final try before locking!\n");
        if (flags & GNUTLS_PIN_COUNT_LOW)
                printf("*** Only few tries left before locking!\n");
        if (flags & GNUTLS_PIN_WRONG)
                printf("*** Wrong PIN\n");

        password = getpass("Enter pin: ");
        /* FIXME: ensure that we are in UTF-8 locale */
        if (password == NULL || password[0] == 0) {
                fprintf(stderr, "No password given\n");
                exit(1);
        }

        len = MIN(pin_max - 1, strlen(password));
```

```
        memcpy(pin, password, len);
        pin[len] = 0;

        return 0;
}

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_certificate_credentials_t xcred;
        /* Allow connections to servers that have OpenPGP keys as well.
         */

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        /* The PKCS11 private key operations may require PIN.
         * Register a callback. */
        gnutls_pkcs11_set_pin_function(pin_callback, NULL);

        /* X509 stuff */
        CHECK(gnutls_certificate_allocate_credentials(&xcred));

        /* sets the trusted cas file
         */
        CHECK(gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                                     GNUTLS_X509_FMT_PEM));

        CHECK(gnutls_certificate_set_x509_key_file(xcred, CERT_URL, KEY_URL,
                                                   GNUTLS_X509_FMT_DER));

        /* Note that there is no server certificate verification in this example
         */


        /* Initialize TLS session
         */
        CHECK(gnutls_init(&session, GNUTLS_CLIENT));

        /* Use default priorities */
```

```
CHECK(gnutls_set_default_priority(session));

/* put the x509 credentials to the current session
 */
CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred));

/* connect to the peer
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake(session);

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
        gnutls_free(desc);
}

CHECK(gnutls_record_send(session, MSG, strlen(MSG)));

ret = gnutls_record_recv(session, buffer, MAX_BUF);
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
        goto end;
}

printf("- Received %d bytes: ", ret);
for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
}
fputs("\n", stdout);

CHECK(gnutls_bye(session, GNUTLS_SHUT_RDWR));
```

```
      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}
```

7.1.4 Client with resume capability example

This is a modification of the simple client example. Here we demonstrate the use of session resumption. The client tries to connect once using TLS, close the connection and then try to establish a new connection using the previously negotiated data.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <gnutls/gnutls.h>

extern void check_alert(gnutls_session_t session, int ret);
extern int tcp_connect(void);
extern void tcp_close(int sd);

/* A very basic TLS client, with X.509 authentication and server certificate
 * verification as well as session resumption.
 *
 * Note that error recovery is minimal for simplicity.
 */

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)
```

```
#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
        int ret;
        int sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_certificate_credentials_t xcred;

        /* variables used in session resuming
         */
        int t;
        gnutls_datum_t sdata;

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        CHECK(gnutls_certificate_allocate_credentials(&xcred));
        CHECK(gnutls_certificate_set_x509_system_trust(xcred));

        for (t = 0; t < 2; t++) {       /* connect 2 times to the server */

                sd = tcp_connect();

                CHECK(gnutls_init(&session, GNUTLS_CLIENT));

                CHECK(gnutls_server_name_set(session, GNUTLS_NAME_DNS,
                                             "www.example.com",
                                             strlen("www.example.com")));
                gnutls_session_set_verify_cert(session, "www.example.com", 0);

                CHECK(gnutls_set_default_priority(session));

                gnutls_transport_set_int(session, sd);
                gnutls_handshake_set_timeout(session,
                                             GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

                gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                                       xcred);

                if (t > 0) {
                        /* if this is not the first time we connect */
                        CHECK(gnutls_session_set_data(session, sdata.data,
                                                      sdata.size));
                        gnutls_free(sdata.data);
```

```
}

/* Perform the TLS handshake
 */
do {
        ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        printf("- Handshake was completed\n");
}

if (t == 0) {   /* the first time we connect */
        /* get the session data */
        CHECK(gnutls_session_get_data2(session, &sdata));
} else { /* the second time we connect */

        /* check if we actually resumed the previous session */
        if (gnutls_session_is_resumed(session) != 0) {
                printf("- Previous session was resumed\n");
        } else {
                fprintf(stderr,
                        "*** Previous session was NOT resumed\n");
        }
}

LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n",
                gnutls_strerror(ret));
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n",
                gnutls_strerror(ret));
        goto end;
}

if (ret > 0) {
```

```
                        printf("- Received %d bytes: ", ret);
                        for (ii = 0; ii < ret; ii++) {
                                fputc(buffer[ii], stdout);
                        }
                        fputs("\n", stdout);
                }

                gnutls_bye(session, GNUTLS_SHUT_RDWR);

              end:

                tcp_close(sd);

                gnutls_deinit(session);

        }                       /* for() */

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}
```

7.1.5 Client example with SSH-style certificate verification

This is an alternative verification function that will use the X.509 certificate authorities for verification, but also assume an trust on first use (SSH-like) authentication system. That is the user is prompted on unknown public keys and known public keys are considered trusted.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <assert.h>
#include "examples.h"

#define CHECK(x) assert((x)>=0)

/* This function will verify the peer's certificate, check
 * if the hostname matches. In addition it will perform an
```

```
 * SSH-style authentication, where ultimately trusted keys
 * are only the keys that have been seen before.
 */
int _ssh_verify_certificate_callback(gnutls_session_t session)
{
        unsigned int status;
        const gnutls_datum_t *cert_list;
        unsigned int cert_list_size;
        int ret, type;
        gnutls_datum_t out;
        const char *hostname;

        /* read hostname */
        hostname = gnutls_session_get_ptr(session);

        /* This verification function uses the trusted CAs in the credentials
         * structure. So you must have installed one or more CA certificates.
         */
        CHECK(gnutls_certificate_verify_peers3(session, hostname, &status));

        type = gnutls_certificate_type_get(session);

        CHECK(gnutls_certificate_verification_status_print(status,
                                                           type, &out, 0));
        printf("%s", out.data);

        gnutls_free(out.data);

        if (status != 0)        /* Certificate is not trusted */
                return GNUTLS_E_CERTIFICATE_ERROR;

        /* Do SSH verification */
        cert_list = gnutls_certificate_get_peers(session, &cert_list_size);
        if (cert_list == NULL) {
                printf("No certificate was found!\n");
                return GNUTLS_E_CERTIFICATE_ERROR;
        }

        /* service may be obtained alternatively using getservbyport() */
        ret = gnutls_verify_stored_pubkey(NULL, NULL, hostname, "https",
                                          type, &cert_list[0], 0);
        if (ret == GNUTLS_E_NO_CERTIFICATE_FOUND) {
                printf("Host %s is not known.", hostname);
                if (status == 0)
                        printf("Its certificate is valid for %s.\n",
                               hostname);
```

```
                /* the certificate must be printed and user must be asked on
                 * whether it is trustworthy. --see gnutls_x509_crt_print() */

                /* if not trusted */
                return GNUTLS_E_CERTIFICATE_ERROR;
        } else if (ret == GNUTLS_E_CERTIFICATE_KEY_MISMATCH) {
                printf
                    ("Warning: host %s is known but has another key associated.",
                     hostname);
                printf
                    ("It might be that the server has multiple keys, or you are under attac
                if (status == 0)
                        printf("Its certificate is valid for %s.\n",
                               hostname);

                /* the certificate must be printed and user must be asked on
                 * whether it is trustworthy. --see gnutls_x509_crt_print() */

                /* if not trusted */
                return GNUTLS_E_CERTIFICATE_ERROR;
        } else if (ret < 0) {
                printf("gnutls_verify_stored_pubkey: %s\n",
                       gnutls_strerror(ret));
                return ret;
        }

        /* user trusts the key -> store it */
        if (ret != 0) {
                CHECK(gnutls_store_pubkey(NULL, NULL, hostname, "https",
                                          type, &cert_list[0], 0, 0));
        }

        /* notify gnutls to continue handshake normally */
        return 0;
}
```

7.2 Server examples

This section contains examples of TLS and SSL servers, using GnuTLS.

7.2.1 Echo server with X.509 authentication

This example is a very simple echo server which supports X.509 authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <assert.h>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED)

/* The OCSP status file contains up to date information about revocation
 * of the server's certificate. That can be periodically be updated
 * using:
 * $ ocsptool --ask --load-cert your_cert.pem --load-issuer your_issuer.pem
 *            --load-signer your_issuer.pem --outfile ocsp-status.der
 */
#define OCSP_STATUS_FILE "ocsp-status.der"

/* This is a sample TLS 1.0 echo server, using X.509 authentication and
 * OCSP stapling support.
 */

#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */

int main(void)
{
        int listen_sd;
        int sd, ret;
        gnutls_certificate_credentials_t x509_cred;
        gnutls_priority_t priority_cache;
        struct sockaddr_in sa_serv;
        struct sockaddr_in sa_cli;
```

```
        socklen_t client_len;
        char topbuf[512];
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        int optval = 1;

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        CHECK(gnutls_certificate_allocate_credentials(&x509_cred));

        CHECK(gnutls_certificate_set_x509_trust_file(x509_cred, CAFILE,
                                                     GNUTLS_X509_FMT_PEM));

        CHECK(gnutls_certificate_set_x509_crl_file(x509_cred, CRLFILE,
                                                   GNUTLS_X509_FMT_PEM));

        /* The following code sets the certificate key pair as well as,
         * an OCSP response which corresponds to it. It is possible
         * to set multiple key-pairs and multiple OCSP status responses
         * (the latter since 3.5.6). See the manual pages of the individual
         * functions for more information.
         */
        CHECK(gnutls_certificate_set_x509_key_file(x509_cred, CERTFILE,
                                                   KEYFILE,
                                                   GNUTLS_X509_FMT_PEM));

        CHECK(gnutls_certificate_set_ocsp_status_request_file(x509_cred,
                                                              OCSP_STATUS_FILE,
                                                              0));

        CHECK(gnutls_priority_init(&priority_cache, NULL, NULL));

        /* Instead of the default options as shown above one could specify
         * additional options such as server precedence in ciphersuite selection
         * as follows:
         * gnutls_priority_init2(&priority_cache,
         *                       "%SERVER_PRECEDENCE",
         *                       NULL, GNUTLS_PRIORITY_INIT_DEF_APPEND);
         */

#if GNUTLS_VERSION_NUMBER >= 0x030506
        /* only available since GnuTLS 3.5.6, on previous versions see
         * gnutls_certificate_set_dh_params(). */
        gnutls_certificate_set_known_dh_params(x509_cred, GNUTLS_SEC_PARAM_MEDIUM);
#endif
```

```
/* Socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));

listen(listen_sd, 1024);

printf("Server ready. Listening to port '%d'.\n\n", PORT);

client_len = sizeof(sa_cli);
for (;;) {
        CHECK(gnutls_init(&session, GNUTLS_SERVER));
        CHECK(gnutls_priority_set(session, priority_cache));
        CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                                     x509_cred));

        /* We don't request any certificate from the client.
         * If we did we would need to verify it. One way of
         * doing that is shown in the "Verifying a certificate"
         * example.
         */
        gnutls_certificate_server_set_request(session,
                                              GNUTLS_CERT_IGNORE);
        gnutls_handshake_set_timeout(session,
                                     GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

        sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                    &client_len);

        printf("- connection from %s, port %d\n",
               inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                         sizeof(topbuf)), ntohs(sa_cli.sin_port));

        gnutls_transport_set_int(session, sd);

        LOOP_CHECK(ret, gnutls_handshake(session));
        if (ret < 0) {
                close(sd);
```

```
                gnutls_deinit(session);
                fprintf(stderr,
                        "*** Handshake has failed (%s)\n\n",
                        gnutls_strerror(ret));
                continue;
        }
        printf("- Handshake was completed\n");

        /* see the Getting peer's information example */
        /* print_info(session); */

        for (;;) {
                LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));

                if (ret == 0) {
                        printf
                            ("\n- Peer has closed the GnuTLS connection\n");
                        break;
                } else if (ret < 0
                           && gnutls_error_is_fatal(ret) == 0) {
                        fprintf(stderr, "*** Warning: %s\n",
                                gnutls_strerror(ret));
                } else if (ret < 0) {
                        fprintf(stderr, "\n*** Received corrupted "
                                "data(%d). Closing the connection.\n\n",
                                ret);
                        break;
                } else if (ret > 0) {
                        /* echo data back to the client
                         */
                        CHECK(gnutls_record_send(session, buffer, ret));
                }
        }
        printf("\n");
        /* do not wait for the peer to close the connection.
         */
        LOOP_CHECK(ret, gnutls_bye(session, GNUTLS_SHUT_WR));

        close(sd);
        gnutls_deinit(session);

}
close(listen_sd);

gnutls_certificate_free_credentials(x509_cred);
gnutls_priority_deinit(priority_cache);
```

```
        gnutls_global_deinit();

        return 0;

}
```

7.2.2 DTLS echo server with X.509 authentication

This example is a very simple echo server using Datagram TLS and X.509 authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/dtls.h>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"

/* This is a sample DTLS echo server, using X.509 authentication.
 * Note that error checking is minimal to simplify the example.
 */

#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED)

#define MAX_BUFFER 1024
#define PORT 5557

typedef struct {
```

```
        gnutls_session_t session;
        int fd;
        struct sockaddr *cli_addr;
        socklen_t cli_addr_size;
} priv_data_st;

static int pull_timeout_func(gnutls_transport_ptr_t ptr, unsigned int ms);
static ssize_t push_func(gnutls_transport_ptr_t p, const void *data,
                         size_t size);
static ssize_t pull_func(gnutls_transport_ptr_t p, void *data,
                         size_t size);
static const char *human_addr(const struct sockaddr *sa, socklen_t salen,
                              char *buf, size_t buflen);
static int wait_for_connection(int fd);

/* Use global credentials and parameters to simplify
 * the example. */
static gnutls_certificate_credentials_t x509_cred;
static gnutls_priority_t priority_cache;

int main(void)
{
        int listen_sd;
        int sock, ret;
        struct sockaddr_in sa_serv;
        struct sockaddr_in cli_addr;
        socklen_t cli_addr_size;
        gnutls_session_t session;
        char buffer[MAX_BUFFER];
        priv_data_st priv;
        gnutls_datum_t cookie_key;
        gnutls_dtls_prestate_st prestate;
        int mtu = 1400;
        unsigned char sequence[8];

        /* this must be called once in the program
         */
        gnutls_global_init();

        gnutls_certificate_allocate_credentials(&x509_cred);
        gnutls_certificate_set_x509_trust_file(x509_cred, CAFILE,
                                               GNUTLS_X509_FMT_PEM);

        gnutls_certificate_set_x509_crl_file(x509_cred, CRLFILE,
                                             GNUTLS_X509_FMT_PEM);

        ret =
```

```
            gnutls_certificate_set_x509_key_file(x509_cred, CERTFILE,
                                                 KEYFILE,
                                                 GNUTLS_X509_FMT_PEM);
        if (ret < 0) {
                printf("No certificate or key were found\n");
                exit(1);
        }

        gnutls_certificate_set_known_dh_params(x509_cred, GNUTLS_SEC_PARAM_MEDIUM);

        /* pre-3.6.3 equivalent:
         * gnutls_priority_init(&priority_cache,
         *                      "NORMAL:-VERS-TLS-ALL:+VERS-DTLS1.0:%SERVER_PRECEDENCE",
         *                      NULL);
         */
        gnutls_priority_init2(&priority_cache,
                              "%SERVER_PRECEDENCE",
                              NULL, GNUTLS_PRIORITY_INIT_DEF_APPEND);

        gnutls_key_generate(&cookie_key, GNUTLS_COOKIE_KEY_SIZE);

        /* Socket operations
         */
        listen_sd = socket(AF_INET, SOCK_DGRAM, 0);

        memset(&sa_serv, '\0', sizeof(sa_serv));
        sa_serv.sin_family = AF_INET;
        sa_serv.sin_addr.s_addr = INADDR_ANY;
        sa_serv.sin_port = htons(PORT);

        {                       /* DTLS requires the IP don't fragment (DF) bit to be set *
#if defined(IP_DONTFRAG)
                int optval = 1;
                setsockopt(listen_sd, IPPROTO_IP, IP_DONTFRAG,
                           (const void *) &optval, sizeof(optval));
#elif defined(IP_MTU_DISCOVER)
                int optval = IP_PMTUDISC_DO;
                setsockopt(listen_sd, IPPROTO_IP, IP_MTU_DISCOVER,
                           (const void *) &optval, sizeof(optval));
#endif
        }

        bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));

        printf("UDP server ready. Listening to port '%d'.\n\n", PORT);

        for (;;) {
```

```
printf("Waiting for connection...\n");
sock = wait_for_connection(listen_sd);
if (sock < 0)
        continue;

cli_addr_size = sizeof(cli_addr);
ret = recvfrom(sock, buffer, sizeof(buffer), MSG_PEEK,
               (struct sockaddr *) &cli_addr,
               &cli_addr_size);
if (ret > 0) {
        memset(&prestate, 0, sizeof(prestate));
        ret =
            gnutls_dtls_cookie_verify(&cookie_key,
                                      &cli_addr,
                                      sizeof(cli_addr),
                                      buffer, ret,
                                      &prestate);
        if (ret < 0) {  /* cookie not valid */
                priv_data_st s;

                memset(&s, 0, sizeof(s));
                s.fd = sock;
                s.cli_addr = (void *) &cli_addr;
                s.cli_addr_size = sizeof(cli_addr);

                printf
                    ("Sending hello verify request to %s\n",
                     human_addr((struct sockaddr *)
                                &cli_addr,
                                sizeof(cli_addr), buffer,
                                sizeof(buffer)));

                gnutls_dtls_cookie_send(&cookie_key,
                                        &cli_addr,
                                        sizeof(cli_addr),
                                        &prestate,
                                        (gnutls_transport_ptr_t)
                                        & s, push_func);

                /* discard peeked data */
                recvfrom(sock, buffer, sizeof(buffer), 0,
                         (struct sockaddr *) &cli_addr,
                         &cli_addr_size);
                usleep(100);
                continue;
        }
        printf("Accepted connection from %s\n",
```

```
                human_addr((struct sockaddr *)
                           &cli_addr, sizeof(cli_addr),
                           buffer, sizeof(buffer)));
} else
        continue;

gnutls_init(&session, GNUTLS_SERVER | GNUTLS_DATAGRAM);
gnutls_priority_set(session, priority_cache);
gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                       x509_cred);

gnutls_dtls_prestate_set(session, &prestate);
gnutls_dtls_set_mtu(session, mtu);

priv.session = session;
priv.fd = sock;
priv.cli_addr = (struct sockaddr *) &cli_addr;
priv.cli_addr_size = sizeof(cli_addr);

gnutls_transport_set_ptr(session, &priv);
gnutls_transport_set_push_function(session, push_func);
gnutls_transport_set_pull_function(session, pull_func);
gnutls_transport_set_pull_timeout_function(session,
                                           pull_timeout_func);

LOOP_CHECK(ret, gnutls_handshake(session));
/* Note that DTLS may also receive GNUTLS_E_LARGE_PACKET.
 * In that case the MTU should be adjusted.
 */

if (ret < 0) {
        fprintf(stderr, "Error in handshake(): %s\n",
                gnutls_strerror(ret));
        gnutls_deinit(session);
        continue;
}

printf("- Handshake was completed\n");

for (;;) {
        LOOP_CHECK(ret,
                   gnutls_record_recv_seq(session, buffer,
                                          MAX_BUFFER,
                                          sequence));

        if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
                fprintf(stderr, "*** Warning: %s\n",
```

```
                                gnutls_strerror(ret));
                        continue;
                } else if (ret < 0) {
                        fprintf(stderr, "Error in recv(): %s\n",
                                gnutls_strerror(ret));
                        break;
                }

                if (ret == 0) {
                        printf("EOF\n\n");
                        break;
                }

                buffer[ret] = 0;
                printf
                    ("received[%.2x%.2x%.2x%.2x%.2x%.2x%.2x%.2x]: %s\n",
                     sequence[0], sequence[1], sequence[2],
                     sequence[3], sequence[4], sequence[5],
                     sequence[6], sequence[7], buffer);

                /* reply back */
                LOOP_CHECK(ret, gnutls_record_send(session, buffer, ret));
                if (ret < 0) {
                        fprintf(stderr, "Error in send(): %s\n",
                                gnutls_strerror(ret));
                        break;
                }
        }

        LOOP_CHECK(ret, gnutls_bye(session, GNUTLS_SHUT_WR));
        gnutls_deinit(session);

    }
    close(listen_sd);

    gnutls_certificate_free_credentials(x509_cred);
    gnutls_priority_deinit(priority_cache);

    gnutls_global_deinit();

    return 0;

}

static int wait_for_connection(int fd)
{
    fd_set rd, wr;
```

```
        int n;

        FD_ZERO(&rd);
        FD_ZERO(&wr);

        FD_SET(fd, &rd);

        /* waiting part */
        n = select(fd + 1, &rd, &wr, NULL, NULL);
        if (n == -1 && errno == EINTR)
                return -1;
        if (n < 0) {
                perror("select()");
                exit(1);
        }

        return fd;
}

/* Wait for data to be received within a timeout period in milliseconds
 */
static int pull_timeout_func(gnutls_transport_ptr_t ptr, unsigned int ms)
{
        fd_set rfds;
        struct timeval tv;
        priv_data_st *priv = ptr;
        struct sockaddr_in cli_addr;
        socklen_t cli_addr_size;
        int ret;
        char c;

        FD_ZERO(&rfds);
        FD_SET(priv->fd, &rfds);

        tv.tv_sec = ms / 1000;
        tv.tv_usec = (ms % 1000) * 1000;

        ret = select(priv->fd + 1, &rfds, NULL, NULL, &tv);

        if (ret <= 0)
                return ret;

        /* only report ok if the next message is from the peer we expect
         * from
         */
        cli_addr_size = sizeof(cli_addr);
        ret =
```

```
            recvfrom(priv->fd, &c, 1, MSG_PEEK,
                     (struct sockaddr *) &cli_addr, &cli_addr_size);
        if (ret > 0) {
                if (cli_addr_size == priv->cli_addr_size
                    && memcmp(&cli_addr, priv->cli_addr,
                              sizeof(cli_addr)) == 0)
                    return 1;
        }

        return 0;
}

static ssize_t
push_func(gnutls_transport_ptr_t p, const void *data, size_t size)
{
        priv_data_st *priv = p;

        return sendto(priv->fd, data, size, 0, priv->cli_addr,
                      priv->cli_addr_size);
}

static ssize_t pull_func(gnutls_transport_ptr_t p, void *data, size_t size)
{
        priv_data_st *priv = p;
        struct sockaddr_in cli_addr;
        socklen_t cli_addr_size;
        char buffer[64];
        int ret;

        cli_addr_size = sizeof(cli_addr);
        ret =
            recvfrom(priv->fd, data, size, 0,
                     (struct sockaddr *) &cli_addr, &cli_addr_size);
        if (ret == -1)
                return ret;

        if (cli_addr_size == priv->cli_addr_size
            && memcmp(&cli_addr, priv->cli_addr, sizeof(cli_addr)) == 0)
                return ret;

        printf("Denied connection from %s\n",
               human_addr((struct sockaddr *)
                          &cli_addr, sizeof(cli_addr), buffer,
                          sizeof(buffer)));

        gnutls_transport_set_errno(priv->session, EAGAIN);
        return -1;
```

```
}

static const char *human_addr(const struct sockaddr *sa, socklen_t salen,
                              char *buf, size_t buflen)
{
        const char *save_buf = buf;
        size_t l;

        if (!buf || !buflen)
                return NULL;

        *buf = '\0';

        switch (sa->sa_family) {
#if HAVE_IPV6
        case AF_INET6:
                snprintf(buf, buflen, "IPv6 ");
                break;
#endif

        case AF_INET:
                snprintf(buf, buflen, "IPv4 ");
                break;
        }

        l = strlen(buf);
        buf += l;
        buflen -= l;

        if (getnameinfo(sa, salen, buf, buflen, NULL, 0, NI_NUMERICHOST) !=
            0)
                return NULL;

        l = strlen(buf);
        buf += l;
        buflen -= l;

        strncat(buf, " port ", buflen);

        l = strlen(buf);
        buf += l;
        buflen -= l;

        if (getnameinfo(sa, salen, NULL, 0, buf, buflen, NI_NUMERICSERV) !=
            0)
                return NULL;
```

```
        return save_buf;
}
```

7.3 More advanced client and servers

This section has various, more advanced topics in client and servers.

7.3.1 Client example with anonymous authentication

The simplest client using TLS is the one that doesn't do any authentication. This means no external certificates or passwords are needed to set up the connection. As could be expected, the connection is vulnerable to man-in-the-middle (active or redirection) attacks. However, the data are integrity protected and encrypted from passive eavesdroppers.

Note that due to the vulnerable nature of this method very few public servers support it.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <assert.h>
#include <gnutls/gnutls.h>

/* A very basic TLS client, with anonymous authentication.
 */

#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
extern void tcp_close(int sd);

int main(void)
{
```

```
int ret, sd, ii;
gnutls_session_t session;
char buffer[MAX_BUF + 1];
gnutls_anon_client_credentials_t anoncred;
/* Need to enable anonymous KX specifically. */

gnutls_global_init();

gnutls_anon_allocate_client_credentials(&anoncred);

/* Initialize TLS session
 */
gnutls_init(&session, GNUTLS_CLIENT);

/* Use default priorities */
gnutls_priority_set_direct(session,
                           "PERFORMANCE:+ANON-ECDH:+ANON-DH",
                           NULL);

/* put the anonymous credentials to the current session
 */
gnutls_credentials_set(session, GNUTLS_CRD_ANON, anoncred);

/* connect to the peer
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);
gnutls_handshake_set_timeout(session,
                             GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

/* Perform the TLS handshake
 */
do {
        ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
```

```
                gnutls_free(desc);
        }

        LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

        LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
        if (ret == 0) {
                printf("- Peer has closed the TLS connection\n");
                goto end;
        } else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
                fprintf(stderr, "*** Warning: %s\n", gnutls_strerror(ret));
        } else if (ret < 0) {
                fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
                goto end;
        }

        if (ret > 0) {
                printf("- Received %d bytes: ", ret);
                for (ii = 0; ii < ret; ii++) {
                        fputc(buffer[ii], stdout);
                }
                fputs("\n", stdout);
        }

        LOOP_CHECK(ret, gnutls_bye(session, GNUTLS_SHUT_RDWR));

      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_anon_free_client_credentials(anoncred);

        gnutls_global_deinit();

        return 0;
}
```

7.3.2 Using a callback to select the certificate to use

There are cases where a client holds several certificate and key pairs, and may not want to load all of them in the credentials structure. The following example demonstrates the use of the certificate selection callback.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
```

```
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <assert.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/abstract.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* A TLS client that loads the certificate and key.
 */

#define CHECK(x) assert((x)>=0)

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

#define CERT_FILE "cert.pem"
#define KEY_FILE "key.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"

extern int tcp_connect(void);
extern void tcp_close(int sd);

static int
cert_callback(gnutls_session_t session,
              const gnutls_datum_t * req_ca_rdn, int nreqs,
              const gnutls_pk_algorithm_t * sign_algos,
              int sign_algos_length, gnutls_pcert_st ** pcert,
              unsigned int *pcert_length, gnutls_privkey_t * pkey);

gnutls_pcert_st pcrt;
gnutls_privkey_t key;

/* Load the certificate and the private key.
 */
static void load_keys(void)
{
```

```
        gnutls_datum_t data;

        CHECK(gnutls_load_file(CERT_FILE, &data));

        CHECK(gnutls_pcert_import_x509_raw(&pcrt, &data,
                                           GNUTLS_X509_FMT_PEM, 0));

        gnutls_free(data.data);

        CHECK(gnutls_load_file(KEY_FILE, &data));

        CHECK(gnutls_privkey_init(&key));

        CHECK(gnutls_privkey_import_x509_raw(key, &data,
                                             GNUTLS_X509_FMT_PEM,
                                             NULL, 0));
        gnutls_free(data.data);
}

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_certificate_credentials_t xcred;

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        CHECK(gnutls_global_init());

        load_keys();

        /* X509 stuff */
        CHECK(gnutls_certificate_allocate_credentials(&xcred));

        /* sets the trusted cas file
         */
        CHECK(gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                                     GNUTLS_X509_FMT_PEM));

        gnutls_certificate_set_retrieve_function2(xcred, cert_callback);

        /* Initialize TLS session
```

```
 */
CHECK(gnutls_init(&session, GNUTLS_CLIENT));

/* Use default priorities */
CHECK(gnutls_set_default_priority(session));

/* put the x509 credentials to the current session
 */
CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred));

/* connect to the peer
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake(session);

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
        gnutls_free(desc);
}

CHECK(gnutls_record_send(session, MSG, strlen(MSG)));

ret = gnutls_record_recv(session, buffer, MAX_BUF);
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
        goto end;
}

printf("- Received %d bytes: ", ret);
for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
}
```

```
        fputs("\n", stdout);

        CHECK(gnutls_bye(session, GNUTLS_SHUT_RDWR));

      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}



/* This callback should be associated with a session by calling
 * gnutls_certificate_client_set_retrieve_function( session, cert_callback),
 * before a handshake.
 */

static int
cert_callback(gnutls_session_t session,
              const gnutls_datum_t * req_ca_rdn, int nreqs,
              const gnutls_pk_algorithm_t * sign_algos,
              int sign_algos_length, gnutls_pcert_st ** pcert,
              unsigned int *pcert_length, gnutls_privkey_t * pkey)
{
        char issuer_dn[256];
        int i, ret;
        size_t len;
        gnutls_certificate_type_t type;

        /* Print the server's trusted CAs
         */
        if (nreqs > 0)
                printf("- Server's trusted authorities:\n");
        else
                printf
                    ("- Server did not send us any trusted authorities names.\n");

        /* print the names (if any) */
        for (i = 0; i < nreqs; i++) {
                len = sizeof(issuer_dn);
```

```
                ret = gnutls_x509_rdn_get(&req_ca_rdn[i], issuer_dn, &len);
                if (ret >= 0) {
                        printf("   [%d]: ", i);
                        printf("%s\n", issuer_dn);
                }
        }

        /* Select a certificate and return it.
         * The certificate must be of any of the "sign algorithms"
         * supported by the server.
         */
        type = gnutls_certificate_type_get(session);
        if (type == GNUTLS_CRT_X509) {
                *pcert_length = 1;
                *pcert = &pcrt;
                *pkey = key;
        } else {
                return -1;
        }

        return 0;

}
```

7.3.3 Obtaining session information

Most of the times it is desirable to know the security properties of the current established session. This includes the underlying ciphers and the protocols involved. That is the purpose of the following function. Note that this function will print meaningful values only if called after a successful [gnutls_handshake], page 414.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"

/* This function will print some details of the
 * given session.
 */
int print_info(gnutls_session_t session)
```

```
{
        gnutls_credentials_type_t cred;
        gnutls_kx_algorithm_t kx;
        int dhe, ecdh, group;
        char *desc;

        /* get a description of the session connection, protocol,
         * cipher/key exchange */
        desc = gnutls_session_get_desc(session);
        if (desc != NULL) {
                printf("- Session: %s\n", desc);
        }

        dhe = ecdh = 0;

        kx = gnutls_kx_get(session);

        /* Check the authentication type used and switch
         * to the appropriate.
         */
        cred = gnutls_auth_get_type(session);
        switch (cred) {
#ifdef ENABLE_SRP
        case GNUTLS_CRD_SRP:
                printf("- SRP session with username %s\n",
                       gnutls_srp_server_get_username(session));
                break;
#endif

        case GNUTLS_CRD_PSK:
                /* This returns NULL in server side.
                 */
                if (gnutls_psk_client_get_hint(session) != NULL)
                        printf("- PSK authentication. PSK hint '%s'\n",
                               gnutls_psk_client_get_hint(session));
                /* This returns NULL in client side.
                 */
                if (gnutls_psk_server_get_username(session) != NULL)
                        printf("- PSK authentication. Connected as '%s'\n",
                               gnutls_psk_server_get_username(session));

                if (kx == GNUTLS_KX_ECDHE_PSK)
                        ecdh = 1;
                else if (kx == GNUTLS_KX_DHE_PSK)
                        dhe = 1;
                break;
```

```
        case GNUTLS_CRD_ANON:  /* anonymous authentication */

                printf("- Anonymous authentication.\n");
                if (kx == GNUTLS_KX_ANON_ECDH)
                        ecdh = 1;
                else if (kx == GNUTLS_KX_ANON_DH)
                        dhe = 1;
                break;

        case GNUTLS_CRD_CERTIFICATE:   /* certificate authentication */

                /* Check if we have been using ephemeral Diffie-Hellman.
                 */
                if (kx == GNUTLS_KX_DHE_RSA || kx == GNUTLS_KX_DHE_DSS)
                        dhe = 1;
                else if (kx == GNUTLS_KX_ECDHE_RSA
                         || kx == GNUTLS_KX_ECDHE_ECDSA)
                        ecdh = 1;

                /* if the certificate list is available, then
                 * print some information about it.
                 */
                print_x509_certificate_info(session);
                break;
        default:
                break;
        }                       /* switch */

        /* read the negotiated group - if any */
        group = gnutls_group_get(session);
        if (group != 0) {
                printf("- Negotiated group %s\n",
                       gnutls_group_get_name(group));
        } else {
                if (ecdh != 0)
                        printf("- Ephemeral ECDH using curve %s\n",
                               gnutls_ecc_curve_get_name(gnutls_ecc_curve_get
                                                         (session)));
                else if (dhe != 0)
                        printf("- Ephemeral DH using prime of %d bits\n",
                               gnutls_dh_get_prime_bits(session));
        }

        return 0;
}
```

7.3.4 Advanced certificate verification

An example is listed below which uses the high level verification functions to verify a given certificate chain against a set of CAs and CRLs.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"

#define CHECK(x) assert((x)>=0)

/* All the available CRLs
 */
gnutls_x509_crl_t *crl_list;
int crl_list_size;

/* All the available trusted CAs
 */
gnutls_x509_crt_t *ca_list;
int ca_list_size;

static int print_details_func(gnutls_x509_crt_t cert,
                              gnutls_x509_crt_t issuer,
                              gnutls_x509_crl_t crl,
                              unsigned int verification_output);

/* This function will try to verify the peer's certificate chain, and
 * also check if the hostname matches.
 */
void
verify_certificate_chain(const char *hostname,
                         const gnutls_datum_t * cert_chain,
                         int cert_chain_length)
{
        int i;
        gnutls_x509_trust_list_t tlist;
        gnutls_x509_crt_t *cert;
```

```
gnutls_datum_t txt;
unsigned int output;

/* Initialize the trusted certificate list. This should be done
 * once on initialization. gnutls_x509_crt_list_import2() and
 * gnutls_x509_crl_list_import2() can be used to load them.
 */
CHECK(gnutls_x509_trust_list_init(&tlist, 0));

CHECK(gnutls_x509_trust_list_add_cas(tlist, ca_list, ca_list_size, 0));
CHECK(gnutls_x509_trust_list_add_crls(tlist, crl_list, crl_list_size,
                                      GNUTLS_TL_VERIFY_CRL, 0));

cert = malloc(sizeof(*cert) * cert_chain_length);
assert(cert != NULL);

/* Import all the certificates in the chain to
 * native certificate format.
 */
for (i = 0; i < cert_chain_length; i++) {
        CHECK(gnutls_x509_crt_init(&cert[i]));
        CHECK(gnutls_x509_crt_import(cert[i], &cert_chain[i],
                                     GNUTLS_X509_FMT_DER));
}

CHECK(gnutls_x509_trust_list_verify_named_crt(tlist, cert[0],
                                        hostname,
                                        strlen(hostname),
                                        GNUTLS_VERIFY_DISABLE_CRL_CHECKS,
                                        &output,
                                        print_details_func));

/* if this certificate is not explicitly trusted verify against CAs
 */
if (output != 0) {
        CHECK(gnutls_x509_trust_list_verify_crt(tlist, cert,
                                          cert_chain_length, 0,
                                          &output,
                                          print_details_func));
}



if (output & GNUTLS_CERT_INVALID) {
        fprintf(stderr, "Not trusted\n");
        CHECK(gnutls_certificate_verification_status_print(
                                             output,
```

```
                                                GNUTLS_CRT_X509,
                                                &txt, 0));

                fprintf(stderr, "Error: %s\n", txt.data);
                gnutls_free(txt.data);
        } else
                fprintf(stderr, "Trusted\n");

        /* Check if the name in the first certificate matches our destination!
         */
        if (!gnutls_x509_crt_check_hostname(cert[0], hostname)) {
                printf
                    ("The certificate's owner does not match hostname '%s'\n",
                     hostname);
        }

        gnutls_x509_trust_list_deinit(tlist, 1);

        return;
}

static int
print_details_func(gnutls_x509_crt_t cert,
                   gnutls_x509_crt_t issuer, gnutls_x509_crl_t crl,
                   unsigned int verification_output)
{
        char name[512];
        char issuer_name[512];
        size_t name_size;
        size_t issuer_name_size;

        issuer_name_size = sizeof(issuer_name);
        gnutls_x509_crt_get_issuer_dn(cert, issuer_name,
                                      &issuer_name_size);

        name_size = sizeof(name);
        gnutls_x509_crt_get_dn(cert, name, &name_size);

        fprintf(stdout, "\tSubject: %s\n", name);
        fprintf(stdout, "\tIssuer: %s\n", issuer_name);

        if (issuer != NULL) {
                issuer_name_size = sizeof(issuer_name);
                gnutls_x509_crt_get_dn(issuer, issuer_name,
                                       &issuer_name_size);

                fprintf(stdout, "\tVerified against: %s\n", issuer_name);
```

```
        }

        if (crl != NULL) {
                issuer_name_size = sizeof(issuer_name);
                gnutls_x509_crl_get_issuer_dn(crl, issuer_name,
                                              &issuer_name_size);

                fprintf(stdout, "\tVerified against CRL of: %s\n",
                        issuer_name);
        }

        fprintf(stdout, "\tVerification output: %x\n\n",
                verification_output);

        return 0;
}
```

7.3.5 Client example with PSK authentication

The following client is a very simple PSK TLS client which connects to a server and authenticates using a *username* and a *key*.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <assert.h>
#include <gnutls/gnutls.h>

/* A very basic TLS client, with PSK authentication.
 */

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)
```

```
#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
extern void tcp_close(int sd);

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        const char *err;
        gnutls_psk_client_credentials_t pskcred;
        const gnutls_datum_t key = { (void *) "DEADBEEF", 8 };

        if (gnutls_check_version("3.6.3") == NULL) {
                fprintf(stderr, "GnuTLS 3.6.3 or later is required for this example\n");
                exit(1);
        }

        CHECK(gnutls_global_init());

        CHECK(gnutls_psk_allocate_client_credentials(&pskcred));
        CHECK(gnutls_psk_set_client_credentials(pskcred, "test", &key,
                                                GNUTLS_PSK_KEY_HEX));

        /* Initialize TLS session
         */
        CHECK(gnutls_init(&session, GNUTLS_CLIENT));

        ret =
            gnutls_set_default_priority_append(session,
                                               "-KX-ALL:+ECDHE-PSK:+DHE-PSK:+PSK",
                                               &err, 0);

        /* Alternative for pre-3.6.3 versions:
         * gnutls_priority_set_direct(session, "NORMAL:+ECDHE-PSK:+DHE-PSK:+PSK", &err)
         */
        if (ret < 0) {
                if (ret == GNUTLS_E_INVALID_REQUEST) {
                        fprintf(stderr, "Syntax error at: %s\n", err);
                }
                exit(1);
        }

        /* put the x509 credentials to the current session
         */
```

```
CHECK(gnutls_credentials_set(session, GNUTLS_CRD_PSK, pskcred));

/* connect to the peer
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);
gnutls_handshake_set_timeout(session,
                             GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

/* Perform the TLS handshake
 */
do {
        ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
} else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info: %s\n", desc);
        gnutls_free(desc);
}

LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
} else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n", gnutls_strerror(ret));
} else if (ret < 0) {
        fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
        goto end;
}

if (ret > 0) {
        printf("- Received %d bytes: ", ret);
        for (ii = 0; ii < ret; ii++) {
                fputc(buffer[ii], stdout);
        }
```

```
                fputs("\n", stdout);
        }

        CHECK(gnutls_bye(session, GNUTLS_SHUT_RDWR));

      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_psk_free_client_credentials(pskcred);

        gnutls_global_deinit();

        return 0;
}
```

7.3.6 Client example with SRP authentication

The following client is a very simple SRP TLS client which connects to a server and authenticates using a *username* and a *password*. The server may authenticate itself using a certificate, and in that case it has to be verified.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>

/* Those functions are defined in other examples.
 */
extern void check_alert(gnutls_session_t session, int ret);
extern int tcp_connect(void);
extern void tcp_close(int sd);

#define MAX_BUF 1024
#define USERNAME "user"
#define PASSWORD "pass"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main(void)
```

```
{
        int ret;
        int sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_srp_client_credentials_t srp_cred;
        gnutls_certificate_credentials_t cert_cred;

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        gnutls_global_init();

        gnutls_srp_allocate_client_credentials(&srp_cred);
        gnutls_certificate_allocate_credentials(&cert_cred);

        gnutls_certificate_set_x509_trust_file(cert_cred, CAFILE,
                                               GNUTLS_X509_FMT_PEM);
        gnutls_srp_set_client_credentials(srp_cred, USERNAME, PASSWORD);

        /* connects to server
         */
        sd = tcp_connect();

        /* Initialize TLS session
         */
        gnutls_init(&session, GNUTLS_CLIENT);


        /* Set the priorities.
         */
        gnutls_priority_set_direct(session,
                                   "NORMAL:+SRP:+SRP-RSA:+SRP-DSS",
                                   NULL);

        /* put the SRP credentials to the current session
         */
        gnutls_credentials_set(session, GNUTLS_CRD_SRP, srp_cred);
        gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, cert_cred);

        gnutls_transport_set_int(session, sd);
        gnutls_handshake_set_timeout(session,
                                     GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);
```

```
  /* Perform the TLS handshake
   */
  do {
          ret = gnutls_handshake(session);
  }
  while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

  if (ret < 0) {
          fprintf(stderr, "*** Handshake failed\n");
          gnutls_perror(ret);
          goto end;
  } else {
          char *desc;

          desc = gnutls_session_get_desc(session);
          printf("- Session info: %s\n", desc);
          gnutls_free(desc);
  }

  gnutls_record_send(session, MSG, strlen(MSG));

  ret = gnutls_record_recv(session, buffer, MAX_BUF);
  if (gnutls_error_is_fatal(ret) != 0 || ret == 0) {
          if (ret == 0) {
                  printf
                      ("- Peer has closed the GnuTLS connection\n");
                  goto end;
          } else {
                  fprintf(stderr, "*** Error: %s\n",
                          gnutls_strerror(ret));
                  goto end;
          }
  } else
          check_alert(session, ret);

  if (ret > 0) {
          printf("- Received %d bytes: ", ret);
          for (ii = 0; ii < ret; ii++) {
                  fputc(buffer[ii], stdout);
          }
          fputs("\n", stdout);
  }
  gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

  tcp_close(sd);
```

```
        gnutls_deinit(session);

        gnutls_srp_free_client_credentials(srp_cred);
        gnutls_certificate_free_credentials(cert_cred);

        gnutls_global_deinit();

        return 0;
}
```

7.3.7 Legacy client example with X.509 certificate support

For applications that need to maintain compatibility with the GnuTLS 3.1.x library, this client example is identical to Section 7.1.1 [Client example with X.509 certificate support], page 196, but utilizes APIs that were available in GnuTLS 3.1.4.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include "examples.h"

/* A very basic TLS client, with X.509 authentication and server certificate
 * verification utilizing the GnuTLS 3.1.x API.
 * Note that error recovery is minimal for simplicity.
 */

#define CHECK(x) assert((x)>=0)
#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED); \
        assert(rval >= 0)

#define MAX_BUF 1024
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
```

```
extern void tcp_close(int sd);
static int _verify_certificate_callback(gnutls_session_t session);

int main(void)
{
        int ret, sd, ii;
        gnutls_session_t session;
        char buffer[MAX_BUF + 1];
        gnutls_certificate_credentials_t xcred;

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        CHECK(gnutls_global_init());

        /* X509 stuff */
        CHECK(gnutls_certificate_allocate_credentials(&xcred));

        /* sets the trusted cas file
         */
        CHECK(gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                                     GNUTLS_X509_FMT_PEM));
        gnutls_certificate_set_verify_function(xcred,
                                               _verify_certificate_callback);

        /* If client holds a certificate it can be set using the following:
         *
         gnutls_certificate_set_x509_key_file (xcred,
         "cert.pem", "key.pem",
         GNUTLS_X509_FMT_PEM);
         */

        /* Initialize TLS session
         */
        CHECK(gnutls_init(&session, GNUTLS_CLIENT));

        gnutls_session_set_ptr(session, (void *) "www.example.com");

        gnutls_server_name_set(session, GNUTLS_NAME_DNS, "www.example.com",
                               strlen("www.example.com"));

        /* use default priorities */
        CHECK(gnutls_set_default_priority(session));
#if 0
        /* if more fine-graned control is required */
```

```
        ret = gnutls_priority_set_direct(session,
                                         "NORMAL", &err);
        if (ret < 0) {
                if (ret == GNUTLS_E_INVALID_REQUEST) {
                        fprintf(stderr, "Syntax error at: %s\n", err);
                }
                exit(1);
        }
#endif

        /* put the x509 credentials to the current session
         */
        CHECK(gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred));

        /* connect to the peer
         */
        sd = tcp_connect();

        gnutls_transport_set_int(session, sd);
        gnutls_handshake_set_timeout(session,
                                     GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

        /* Perform the TLS handshake
         */
        do {
                ret = gnutls_handshake(session);
        }
        while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

        if (ret < 0) {
                fprintf(stderr, "*** Handshake failed\n");
                gnutls_perror(ret);
                goto end;
        } else {
                char *desc;

                desc = gnutls_session_get_desc(session);
                printf("- Session info: %s\n", desc);
                gnutls_free(desc);
        }

        LOOP_CHECK(ret, gnutls_record_send(session, MSG, strlen(MSG)));

        LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));
        if (ret == 0) {
                printf("- Peer has closed the TLS connection\n");
                goto end;
```

```
        } else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
                fprintf(stderr, "*** Warning: %s\n", gnutls_strerror(ret));
        } else if (ret < 0) {
                fprintf(stderr, "*** Error: %s\n", gnutls_strerror(ret));
                goto end;
        }

        if (ret > 0) {
                printf("- Received %d bytes: ", ret);
                for (ii = 0; ii < ret; ii++) {
                        fputc(buffer[ii], stdout);
                }
                fputs("\n", stdout);
        }

        CHECK(gnutls_bye(session, GNUTLS_SHUT_RDWR));

      end:

        tcp_close(sd);

        gnutls_deinit(session);

        gnutls_certificate_free_credentials(xcred);

        gnutls_global_deinit();

        return 0;
}

/* This function will verify the peer's certificate, and check
 * if the hostname matches, as well as the activation, expiration dates.
 */
static int _verify_certificate_callback(gnutls_session_t session)
{
        unsigned int status;
        int type;
        const char *hostname;
        gnutls_datum_t out;

        /* read hostname */
        hostname = gnutls_session_get_ptr(session);

        /* This verification function uses the trusted CAs in the credentials
         * structure. So you must have installed one or more CA certificates.
         */
```

```
        CHECK(gnutls_certificate_verify_peers3(session, hostname,
                                               &status));

        type = gnutls_certificate_type_get(session);

        CHECK(gnutls_certificate_verification_status_print(status, type,
                                                           &out, 0));

        printf("%s", out.data);

        gnutls_free(out.data);

        if (status != 0)        /* Certificate is not trusted */
                return GNUTLS_E_CERTIFICATE_ERROR;

        /* notify gnutls to continue handshake normally */
        return 0;
}
```

7.3.8 Client example using the C++ API

The following client is a simple example of a client client utilizing the GnuTLS C++ API.

```
#include <config.h>
#include <iostream>
#include <stdexcept>
#include <gnutls/gnutls.h>
#include <gnutls/gnutlsxx.h>
#include <cstring> /* for strlen */

/* A very basic TLS client, with anonymous authentication.
 * written by Eduardo Villanueva Che.
 */

#define MAX_BUF 1024
#define SA struct sockaddr

#define CAFILE "ca.pem"
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern "C"
{
    int tcp_connect(void);
    void tcp_close(int sd);
}


int main(void)
```

```
{
    int sd = -1;
    gnutls_global_init();

    try
    {

        /* Allow connections to servers that have OpenPGP keys as well.
         */
        gnutls::client_session session;

        /* X509 stuff */
        gnutls::certificate_credentials credentials;


        /* sets the trusted cas file
         */
        credentials.set_x509_trust_file(CAFILE, GNUTLS_X509_FMT_PEM);
        /* put the x509 credentials to the current session
         */
        session.set_credentials(credentials);

        /* Use default priorities */
        session.set_priority ("NORMAL", NULL);

        /* connect to the peer
         */
        sd = tcp_connect();
        session.set_transport_ptr((gnutls_transport_ptr_t) (ptrdiff_t)sd);

        /* Perform the TLS handshake
         */
        int ret = session.handshake();
        if (ret < 0)
        {
            throw std::runtime_error("Handshake failed");
        }
        else
        {
            std::cout << "- Handshake was completed" << std::endl;
        }

        session.send(MSG, strlen(MSG));
        char buffer[MAX_BUF + 1];
        ret = session.recv(buffer, MAX_BUF);
        if (ret == 0)
        {
```

```
            throw std::runtime_error("Peer has closed the TLS connection");
        }
        else if (ret < 0)
        {
            throw std::runtime_error(gnutls_strerror(ret));
        }

        std::cout << "- Received " << ret << " bytes:" << std::endl;
        std::cout.write(buffer, ret);
        std::cout << std::endl;

        session.bye(GNUTLS_SHUT_RDWR);
    }
    catch (std::exception &ex)
    {
        std::cerr << "Exception caught: " << ex.what() << std::endl;
    }

    if (sd != -1)
        tcp_close(sd);

    gnutls_global_deinit();

    return 0;
}
```

7.3.9 Echo server with PSK authentication

This is a server which supports PSK authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

#define KEYFILE "key.pem"
```

```
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"

#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED)

/* This is a sample TLS echo server, supporting X.509 and PSK
   authentication.
 */

#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */

static int
pskfunc(gnutls_session_t session, const char *username,
        gnutls_datum_t * key)
{
        printf("psk: username %s\n", username);
        key->data = gnutls_malloc(4);
        key->data[0] = 0xDE;
        key->data[1] = 0xAD;
        key->data[2] = 0xBE;
        key->data[3] = 0xEF;
        key->size = 4;
        return 0;
}

int main(void)
{
        int err, listen_sd;
        int sd, ret;
        struct sockaddr_in sa_serv;
        struct sockaddr_in sa_cli;
        socklen_t client_len;
        char topbuf[512];
        gnutls_session_t session;
        gnutls_certificate_credentials_t x509_cred;
        gnutls_psk_server_credentials_t psk_cred;
        gnutls_priority_t priority_cache;
        char buffer[MAX_BUF + 1];
        int optval = 1;
        int kx;
```

```
if (gnutls_check_version("3.1.4") == NULL) {
        fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
        exit(1);
}

/* for backwards compatibility with gnutls < 3.3.0 */
gnutls_global_init();

gnutls_certificate_allocate_credentials(&x509_cred);
gnutls_certificate_set_x509_trust_file(x509_cred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);

gnutls_certificate_set_x509_crl_file(x509_cred, CRLFILE,
                                     GNUTLS_X509_FMT_PEM);

gnutls_certificate_set_x509_key_file(x509_cred, CERTFILE, KEYFILE,
                                     GNUTLS_X509_FMT_PEM);

gnutls_psk_allocate_server_credentials(&psk_cred);
gnutls_psk_set_server_credentials_function(psk_cred, pskfunc);

/* pre-3.6.3 equivalent:
 * gnutls_priority_init(&priority_cache,
 *                      "NORMAL:+PSK:+ECDHE-PSK:+DHE-PSK",
 *                      NULL);
 */
gnutls_priority_init2(&priority_cache,
                      "+ECDHE-PSK:+DHE-PSK:+PSK",
                      NULL, GNUTLS_PRIORITY_INIT_DEF_APPEND);

gnutls_certificate_set_known_dh_params(x509_cred, GNUTLS_SEC_PARAM_MEDIUM);

/* Socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
SOCKET_ERR(listen_sd, "socket");

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

err =
    bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));
```

```
SOCKET_ERR(err, "bind");
err = listen(listen_sd, 1024);
SOCKET_ERR(err, "listen");

printf("Server ready. Listening to port '%d'.\n\n", PORT);

client_len = sizeof(sa_cli);
for (;;) {
        gnutls_init(&session, GNUTLS_SERVER);
        gnutls_priority_set(session, priority_cache);
        gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                               x509_cred);
        gnutls_credentials_set(session, GNUTLS_CRD_PSK, psk_cred);

        /* request client certificate if any.
         */
        gnutls_certificate_server_set_request(session,
                                              GNUTLS_CERT_REQUEST);

        sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                    &client_len);

        printf("- connection from %s, port %d\n",
               inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                         sizeof(topbuf)), ntohs(sa_cli.sin_port));

        gnutls_transport_set_int(session, sd);
        LOOP_CHECK(ret, gnutls_handshake(session));
        if (ret < 0) {
                close(sd);
                gnutls_deinit(session);
                fprintf(stderr,
                        "*** Handshake has failed (%s)\n\n",
                        gnutls_strerror(ret));
                continue;
        }
        printf("- Handshake was completed\n");

        kx = gnutls_kx_get(session);
        if (kx == GNUTLS_KX_PSK || kx == GNUTLS_KX_DHE_PSK ||
            kx == GNUTLS_KX_ECDHE_PSK) {
                printf("- User %s was connected\n",
                       gnutls_psk_server_get_username(session));
        }

        /* see the Getting peer's information example */
        /* print_info(session); */
```

```
                for (;;) {
                        LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));

                        if (ret == 0) {
                                printf
                                    ("\n- Peer has closed the GnuTLS connection\n");
                                break;
                        } else if (ret < 0
                                   && gnutls_error_is_fatal(ret) == 0) {
                                fprintf(stderr, "*** Warning: %s\n",
                                        gnutls_strerror(ret));
                        } else if (ret < 0) {
                                fprintf(stderr, "\n*** Received corrupted "
                                        "data(%d). Closing the connection.\n\n",
                                        ret);
                                break;
                        } else if (ret > 0) {
                                /* echo data back to the client
                                 */
                                gnutls_record_send(session, buffer, ret);
                        }
                }
                printf("\n");
                /* do not wait for the peer to close the connection.
                 */
                LOOP_CHECK(ret, gnutls_bye(session, GNUTLS_SHUT_WR));

                close(sd);
                gnutls_deinit(session);

        }
        close(listen_sd);

        gnutls_certificate_free_credentials(x509_cred);
        gnutls_psk_free_server_credentials(psk_cred);

        gnutls_priority_deinit(priority_cache);

        gnutls_global_deinit();

        return 0;

}
```

7.3.10 Echo server with SRP authentication

This is a server which supports SRP authentication. It is also possible to combine this functionality with a certificate server. Here it is separate for simplicity.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

#define SRP_PASSWD "tpasswd"
#define SRP_PASSWD_CONF "tpasswd.conf"

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"

#define LOOP_CHECK(rval, cmd) \
        do { \
                rval = cmd; \
        } while(rval == GNUTLS_E_AGAIN || rval == GNUTLS_E_INTERRUPTED)

/* This is a sample TLS-SRP echo server.
 */

#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */

int main(void)
{
        int err, listen_sd;
        int sd, ret;
        struct sockaddr_in sa_serv;
        struct sockaddr_in sa_cli;
        socklen_t client_len;
```

```
char topbuf[512];
gnutls_session_t session;
gnutls_srp_server_credentials_t srp_cred;
gnutls_certificate_credentials_t cert_cred;
char buffer[MAX_BUF + 1];
int optval = 1;
char name[256];

strcpy(name, "Echo Server");

if (gnutls_check_version("3.1.4") == NULL) {
        fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
        exit(1);
}

/* for backwards compatibility with gnutls < 3.3.0 */
gnutls_global_init();

/* SRP_PASSWD a password file (created with the included srptool utility)
 */
gnutls_srp_allocate_server_credentials(&srp_cred);
gnutls_srp_set_server_credentials_file(srp_cred, SRP_PASSWD,
                                       SRP_PASSWD_CONF);

gnutls_certificate_allocate_credentials(&cert_cred);
gnutls_certificate_set_x509_trust_file(cert_cred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);
gnutls_certificate_set_x509_key_file(cert_cred, CERTFILE, KEYFILE,
                                     GNUTLS_X509_FMT_PEM);

/* TCP socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
SOCKET_ERR(listen_sd, "socket");

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

err =
    bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));
SOCKET_ERR(err, "bind");
err = listen(listen_sd, 1024);
```

```
SOCKET_ERR(err, "listen");

printf("%s ready. Listening to port '%d'.\n\n", name, PORT);

client_len = sizeof(sa_cli);
for (;;) {
        gnutls_init(&session, GNUTLS_SERVER);
        gnutls_priority_set_direct(session,
                                   "NORMAL"
                                   ":-KX-ALL:+SRP:+SRP-DSS:+SRP-RSA",
                                   NULL);
        gnutls_credentials_set(session, GNUTLS_CRD_SRP, srp_cred);
        /* for the certificate authenticated ciphersuites.
         */
        gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                               cert_cred);

        /* We don't request any certificate from the client.
         * If we did we would need to verify it. One way of
         * doing that is shown in the "Verifying a certificate"
         * example.
         */
        gnutls_certificate_server_set_request(session,
                                              GNUTLS_CERT_IGNORE);

        sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                    &client_len);

        printf("- connection from %s, port %d\n",
               inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                         sizeof(topbuf)), ntohs(sa_cli.sin_port));

        gnutls_transport_set_int(session, sd);

        LOOP_CHECK(ret, gnutls_handshake(session));
        if (ret < 0) {
                close(sd);
                gnutls_deinit(session);
                fprintf(stderr,
                        "*** Handshake has failed (%s)\n\n",
                        gnutls_strerror(ret));
                continue;
        }
        printf("- Handshake was completed\n");
        printf("- User %s was connected\n",
               gnutls_srp_server_get_username(session));
```

```
                /* print_info(session); */

                for (;;) {
                        LOOP_CHECK(ret, gnutls_record_recv(session, buffer, MAX_BUF));

                        if (ret == 0) {
                                printf
                                    ("\n- Peer has closed the GnuTLS connection\n");
                                break;
                        } else if (ret < 0
                                   && gnutls_error_is_fatal(ret) == 0) {
                                fprintf(stderr, "*** Warning: %s\n",
                                        gnutls_strerror(ret));
                        } else if (ret < 0) {
                                fprintf(stderr, "\n*** Received corrupted "
                                        "data(%d). Closing the connection.\n\n",
                                        ret);
                                break;
                        } else if (ret > 0) {
                                /* echo data back to the client
                                 */
                                gnutls_record_send(session, buffer, ret);
                        }
                }
                printf("\n");
                /* do not wait for the peer to close the connection. */
                LOOP_CHECK(ret, gnutls_bye(session, GNUTLS_SHUT_WR));

                close(sd);
                gnutls_deinit(session);

        }
        close(listen_sd);

        gnutls_srp_free_server_credentials(srp_cred);
        gnutls_certificate_free_credentials(cert_cred);

        gnutls_global_deinit();

        return 0;

}
```

7.3.11 Echo server with anonymous authentication

This example server supports anonymous authentication, and could be used to serve the example client for anonymous authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

/* This is a sample TLS 1.0 echo server, for anonymous authentication only.
 */


#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */

int main(void)
{
        int err, listen_sd;
        int sd, ret;
        struct sockaddr_in sa_serv;
        struct sockaddr_in sa_cli;
        socklen_t client_len;
        char topbuf[512];
        gnutls_session_t session;
        gnutls_anon_server_credentials_t anoncred;
        char buffer[MAX_BUF + 1];
        int optval = 1;

        if (gnutls_check_version("3.1.4") == NULL) {
                fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
                exit(1);
        }

        /* for backwards compatibility with gnutls < 3.3.0 */
        gnutls_global_init();

        gnutls_anon_allocate_server_credentials(&anoncred);
```

```
gnutls_anon_set_server_known_dh_params(anoncred, GNUTLS_SEC_PARAM_MEDIUM);

/* Socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
SOCKET_ERR(listen_sd, "socket");

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

err =
    bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));
SOCKET_ERR(err, "bind");
err = listen(listen_sd, 1024);
SOCKET_ERR(err, "listen");

printf("Server ready. Listening to port '%d'.\n\n", PORT);

client_len = sizeof(sa_cli);
for (;;) {
        gnutls_init(&session, GNUTLS_SERVER);
        gnutls_priority_set_direct(session,
                                   "NORMAL:+ANON-ECDH:+ANON-DH",
                                   NULL);
        gnutls_credentials_set(session, GNUTLS_CRD_ANON, anoncred);

        sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                    &client_len);

        printf("- connection from %s, port %d\n",
               inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                         sizeof(topbuf)), ntohs(sa_cli.sin_port));

        gnutls_transport_set_int(session, sd);

        do {
                ret = gnutls_handshake(session);
        }
        while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

        if (ret < 0) {
```

```
                close(sd);
                gnutls_deinit(session);
                fprintf(stderr,
                        "*** Handshake has failed (%s)\n\n",
                        gnutls_strerror(ret));
                continue;
        }
        printf("- Handshake was completed\n");

        /* see the Getting peer's information example */
        /* print_info(session); */

        for (;;) {
                ret = gnutls_record_recv(session, buffer, MAX_BUF);

                if (ret == 0) {
                        printf
                            ("\n- Peer has closed the GnuTLS connection\n");
                        break;
                } else if (ret < 0
                           && gnutls_error_is_fatal(ret) == 0) {
                        fprintf(stderr, "*** Warning: %s\n",
                                gnutls_strerror(ret));
                } else if (ret < 0) {
                        fprintf(stderr, "\n*** Received corrupted "
                                "data(%d). Closing the connection.\n\n",
                                ret);
                        break;
                } else if (ret > 0) {
                        /* echo data back to the client
                         */
                        gnutls_record_send(session, buffer, ret);
                }
        }
        printf("\n");
        /* do not wait for the peer to close the connection.
         */
        gnutls_bye(session, GNUTLS_SHUT_WR);

        close(sd);
        gnutls_deinit(session);

}
close(listen_sd);

gnutls_anon_free_server_credentials(anoncred);
```

```
        gnutls_global_deinit();

        return 0;

}
```

7.3.12 Helper functions for TCP connections

Those helper function abstract away TCP connection handling from the other examples. It is required to build some examples.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>

/* tcp.c */
int tcp_connect(void);
void tcp_close(int sd);

/* Connects to the peer and returns a socket
 * descriptor.
 */
extern int tcp_connect(void)
{
        const char *PORT = "5556";
        const char *SERVER = "127.0.0.1";
        int err, sd;
        struct sockaddr_in sa;

        /* connects to server
         */
        sd = socket(AF_INET, SOCK_STREAM, 0);

        memset(&sa, '\0', sizeof(sa));
        sa.sin_family = AF_INET;
        sa.sin_port = htons(atoi(PORT));
        inet_pton(AF_INET, SERVER, &sa.sin_addr);
```

```
        err = connect(sd, (struct sockaddr *) &sa, sizeof(sa));
        if (err < 0) {
                fprintf(stderr, "Connect error\n");
                exit(1);
        }

        return sd;
}

/* closes the given socket descriptor.
 */
extern void tcp_close(int sd)
{
        shutdown(sd, SHUT_RDWR);        /* no more receptions */
        close(sd);
}
```

7.3.13 Helper functions for UDP connections

The UDP helper functions abstract away UDP connection handling from the other examples. It is required to build the examples using UDP.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>

/* udp.c */
int udp_connect(void);
void udp_close(int sd);

/* Connects to the peer and returns a socket
 * descriptor.
 */
extern int udp_connect(void)
{
        const char *PORT = "5557";
```

```
        const char *SERVER = "127.0.0.1";
        int err, sd;
#if defined(IP_DONTFRAG) || defined(IP_MTU_DISCOVER)
        int optval;
#endif
        struct sockaddr_in sa;

        /* connects to server
         */
        sd = socket(AF_INET, SOCK_DGRAM, 0);

        memset(&sa, '\0', sizeof(sa));
        sa.sin_family = AF_INET;
        sa.sin_port = htons(atoi(PORT));
        inet_pton(AF_INET, SERVER, &sa.sin_addr);

#if defined(IP_DONTFRAG)
        optval = 1;
        setsockopt(sd, IPPROTO_IP, IP_DONTFRAG,
                   (const void *) &optval, sizeof(optval));
#elif defined(IP_MTU_DISCOVER)
        optval = IP_PMTUDISC_DO;
        setsockopt(sd, IPPROTO_IP, IP_MTU_DISCOVER,
                   (const void *) &optval, sizeof(optval));
#endif

        err = connect(sd, (struct sockaddr *) &sa, sizeof(sa));
        if (err < 0) {
                fprintf(stderr, "Connect error\n");
                exit(1);
        }

        return sd;
}

/* closes the given socket descriptor.
 */
extern void udp_close(int sd)
{
        close(sd);
}
```

7.4 OCSP example

Generate OCSP request

A small tool to generate OCSP requests.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/crypto.h>
#include <gnutls/ocsp.h>
#ifndef NO_LIBCURL
#include <curl/curl.h>
#endif
#include "read-file.h"

size_t get_data(void *buffer, size_t size, size_t nmemb, void *userp);
static gnutls_x509_crt_t load_cert(const char *cert_file);
static void _response_info(const gnutls_datum_t * data);
static void
_generate_request(gnutls_datum_t * rdata, gnutls_x509_crt_t cert,
                  gnutls_x509_crt_t issuer, gnutls_datum_t *nonce);
static int
_verify_response(gnutls_datum_t * data, gnutls_x509_crt_t cert,
                 gnutls_x509_crt_t signer, gnutls_datum_t *nonce);

/* This program queries an OCSP server.
   It expects three files. argv[1] containing the certificate to
   be checked, argv[2] holding the issuer for this certificate,
   and argv[3] holding a trusted certificate to verify OCSP's response.
   argv[4] is optional and should hold the server host name.

   For simplicity the libcurl library is used.
 */

int main(int argc, char *argv[])
{
        gnutls_datum_t ud, tmp;
        int ret;
        gnutls_datum_t req;
        gnutls_x509_crt_t cert, issuer, signer;
#ifndef NO_LIBCURL
        CURL *handle;
        struct curl_slist *headers = NULL;
#endif
        int v, seq;
```

```
const char *cert_file = argv[1];
const char *issuer_file = argv[2];
const char *signer_file = argv[3];
char *hostname = NULL;
unsigned char noncebuf[23];
gnutls_datum_t nonce = { noncebuf, sizeof(noncebuf) };

gnutls_global_init();

if (argc > 4)
        hostname = argv[4];

ret = gnutls_rnd(GNUTLS_RND_NONCE, nonce.data, nonce.size);
if (ret < 0)
        exit(1);

cert = load_cert(cert_file);
issuer = load_cert(issuer_file);
signer = load_cert(signer_file);

if (hostname == NULL) {

        for (seq = 0;; seq++) {
                ret =
                    gnutls_x509_crt_get_authority_info_access(cert,
                                                              seq,
                                                              GNUTLS_IA_OCSP_URI,
                                                              &tmp,
                                                              NULL);
                if (ret == GNUTLS_E_UNKNOWN_ALGORITHM)
                        continue;
                if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE) {
                        fprintf(stderr,
                                "No URI was found in the certificate.\n");
                        exit(1);
                }
                if (ret < 0) {
                        fprintf(stderr, "error: %s\n",
                                gnutls_strerror(ret));
                        exit(1);
                }

                printf("CA issuers URI: %.*s\n", tmp.size,
                       tmp.data);

                hostname = malloc(tmp.size + 1);
                memcpy(hostname, tmp.data, tmp.size);
```

```
                        hostname[tmp.size] = 0;

                        gnutls_free(tmp.data);
                        break;
                }

        }

        /* Note that the OCSP servers hostname might be available
         * using gnutls_x509_crt_get_authority_info_access() in the issuer's
         * certificate */

        memset(&ud, 0, sizeof(ud));
        fprintf(stderr, "Connecting to %s\n", hostname);

        _generate_request(&req, cert, issuer, &nonce);

#ifndef NO_LIBCURL
        curl_global_init(CURL_GLOBAL_ALL);

        handle = curl_easy_init();
        if (handle == NULL)
                exit(1);

        headers =
            curl_slist_append(headers,
                              "Content-Type: application/ocsp-request");

        curl_easy_setopt(handle, CURLOPT_HTTPHEADER, headers);
        curl_easy_setopt(handle, CURLOPT_POSTFIELDS, (void *) req.data);
        curl_easy_setopt(handle, CURLOPT_POSTFIELDSIZE, req.size);
        curl_easy_setopt(handle, CURLOPT_URL, hostname);
        curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, get_data);
        curl_easy_setopt(handle, CURLOPT_WRITEDATA, &ud);

        ret = curl_easy_perform(handle);
        if (ret != 0) {
                fprintf(stderr, "curl[%d] error %d\n", __LINE__, ret);
                exit(1);
        }

        curl_easy_cleanup(handle);
#endif

        _response_info(&ud);

        v = _verify_response(&ud, cert, signer, &nonce);
```

```
        gnutls_x509_crt_deinit(cert);
        gnutls_x509_crt_deinit(issuer);
        gnutls_x509_crt_deinit(signer);
        gnutls_global_deinit();

        return v;
}

static void _response_info(const gnutls_datum_t * data)
{
        gnutls_ocsp_resp_t resp;
        int ret;
        gnutls_datum buf;

        ret = gnutls_ocsp_resp_init(&resp);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_resp_import(resp, data);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_resp_print(resp, GNUTLS_OCSP_PRINT_FULL, &buf);
        if (ret != 0)
                exit(1);

        printf("%.*s", buf.size, buf.data);
        gnutls_free(buf.data);

        gnutls_ocsp_resp_deinit(resp);
}

static gnutls_x509_crt_t load_cert(const char *cert_file)
{
        gnutls_x509_crt_t crt;
        int ret;
        gnutls_datum_t data;
        size_t size;

        ret = gnutls_x509_crt_init(&crt);
        if (ret < 0)
                exit(1);

        data.data = (void *) read_file(cert_file, RF_BINARY, &size);
        data.size = size;
```

```
        if (!data.data) {
                fprintf(stderr, "Cannot open file: %s\n", cert_file);
                exit(1);
        }

        ret = gnutls_x509_crt_import(crt, &data, GNUTLS_X509_FMT_PEM);
        free(data.data);
        if (ret < 0) {
                fprintf(stderr, "Cannot import certificate in %s: %s\n",
                        cert_file, gnutls_strerror(ret));
                exit(1);
        }

        return crt;
}

static void
_generate_request(gnutls_datum_t * rdata, gnutls_x509_crt_t cert,
                  gnutls_x509_crt_t issuer, gnutls_datum_t *nonce)
{
        gnutls_ocsp_req_t req;
        int ret;

        ret = gnutls_ocsp_req_init(&req);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_req_add_cert(req, GNUTLS_DIG_SHA1, issuer, cert);
        if (ret < 0)
                exit(1);


        ret = gnutls_ocsp_req_set_nonce(req, 0, nonce);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_req_export(req, rdata);
        if (ret != 0)
                exit(1);

        gnutls_ocsp_req_deinit(req);

        return;
}

static int
_verify_response(gnutls_datum_t * data, gnutls_x509_crt_t cert,
```

```
                gnutls_x509_crt_t signer, gnutls_datum_t *nonce)
{
        gnutls_ocsp_resp_t resp;
        int ret;
        unsigned verify;
        gnutls_datum_t rnonce;

        ret = gnutls_ocsp_resp_init(&resp);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_resp_import(resp, data);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_resp_check_crt(resp, 0, cert);
        if (ret < 0)
                exit(1);

        ret = gnutls_ocsp_resp_get_nonce(resp, NULL, &rnonce);
        if (ret < 0)
                exit(1);

        if (rnonce.size != nonce->size || memcmp(nonce->data, rnonce.data,
                nonce->size) != 0) {
                exit(1);
        }

        ret = gnutls_ocsp_resp_verify_direct(resp, signer, &verify, 0);
        if (ret < 0)
                exit(1);

        printf("Verifying OCSP Response: ");
        if (verify == 0)
                printf("Verification success!\n");
        else
                printf("Verification error!\n");

        if (verify & GNUTLS_OCSP_VERIFY_SIGNER_NOT_FOUND)
                printf("Signer cert not found\n");

        if (verify & GNUTLS_OCSP_VERIFY_SIGNER_KEYUSAGE_ERROR)
                printf("Signer cert keyusage error\n");

        if (verify & GNUTLS_OCSP_VERIFY_UNTRUSTED_SIGNER)
                printf("Signer cert is not trusted\n");
```

```
        if (verify & GNUTLS_OCSP_VERIFY_INSECURE_ALGORITHM)
                printf("Insecure algorithm\n");

        if (verify & GNUTLS_OCSP_VERIFY_SIGNATURE_FAILURE)
                printf("Signature failure\n");

        if (verify & GNUTLS_OCSP_VERIFY_CERT_NOT_ACTIVATED)
                printf("Signer cert not yet activated\n");

        if (verify & GNUTLS_OCSP_VERIFY_CERT_EXPIRED)
                printf("Signer cert expired\n");

        gnutls_free(rnonce.data);
        gnutls_ocsp_resp_deinit(resp);

        return verify;
}

size_t get_data(void *buffer, size_t size, size_t nmemb, void *userp)
{
        gnutls_datum_t *ud = userp;

        size *= nmemb;

        ud->data = realloc(ud->data, size + ud->size);
        if (ud->data == NULL) {
                fprintf(stderr, "Not enough memory for the request\n");
                exit(1);
        }

        memcpy(&ud->data[ud->size], buffer, size);
        ud->size += size;

        return size;
}
```

7.5 Miscellaneous examples

7.5.1 Checking for an alert

This is a function that checks if an alert has been received in the current session.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>

#include "examples.h"

/* This function will check whether the given return code from
 * a gnutls function (recv/send), is an alert, and will print
 * that alert.
 */
void check_alert(gnutls_session_t session, int ret)
{
        int last_alert;

        if (ret == GNUTLS_E_WARNING_ALERT_RECEIVED
            || ret == GNUTLS_E_FATAL_ALERT_RECEIVED) {
                last_alert = gnutls_alert_get(session);

                /* The check for renegotiation is only useful if we are
                 * a server, and we had requested a rehandshake.
                 */
                if (last_alert == GNUTLS_A_NO_RENEGOTIATION &&
                    ret == GNUTLS_E_WARNING_ALERT_RECEIVED)
                        printf("* Received NO_RENEGOTIATION alert. "
                               "Client Does not support renegotiation.\n");
                else
                        printf("* Received alert '%d': %s.\n", last_alert,
                               gnutls_alert_get_name(last_alert));
        }
}
```

7.5.2 X.509 certificate parsing example

To demonstrate the X.509 parsing capabilities an example program is listed below. That program reads the peer's certificate, and prints information about it.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"
```

```
static const char *bin2hex(const void *bin, size_t bin_size)
{
        static char printable[110];
        const unsigned char *_bin = bin;
        char *print;
        size_t i;

        if (bin_size > 50)
                bin_size = 50;

        print = printable;
        for (i = 0; i < bin_size; i++) {
                sprintf(print, "%.2x ", _bin[i]);
                print += 2;
        }

        return printable;
}

/* This function will print information about this session's peer
 * certificate.
 */
void print_x509_certificate_info(gnutls_session_t session)
{
        char serial[40];
        char dn[256];
        size_t size;
        unsigned int algo, bits;
        time_t expiration_time, activation_time;
        const gnutls_datum_t *cert_list;
        unsigned int cert_list_size = 0;
        gnutls_x509_crt_t cert;
        gnutls_datum_t cinfo;

        /* This function only works for X.509 certificates.
         */
        if (gnutls_certificate_type_get(session) != GNUTLS_CRT_X509)
                return;

        cert_list = gnutls_certificate_get_peers(session, &cert_list_size);

        printf("Peer provided %d certificates.\n", cert_list_size);

        if (cert_list_size > 0) {
                int ret;
```

```
/* we only print information about the first certificate.
 */
gnutls_x509_crt_init(&cert);

gnutls_x509_crt_import(cert, &cert_list[0],
                       GNUTLS_X509_FMT_DER);

printf("Certificate info:\n");

/* This is the preferred way of printing short information about
   a certificate. */

ret =
    gnutls_x509_crt_print(cert, GNUTLS_CRT_PRINT_ONELINE,
                          &cinfo);
if (ret == 0) {
        printf("\t%s\n", cinfo.data);
        gnutls_free(cinfo.data);
}

/* If you want to extract fields manually for some other reason,
   below are popular example calls. */

expiration_time =
    gnutls_x509_crt_get_expiration_time(cert);
activation_time =
    gnutls_x509_crt_get_activation_time(cert);

printf("\tCertificate is valid since: %s",
       ctime(&activation_time));
printf("\tCertificate expires: %s",
       ctime(&expiration_time));

/* Print the serial number of the certificate.
 */
size = sizeof(serial);
gnutls_x509_crt_get_serial(cert, serial, &size);

printf("\tCertificate serial number: %s\n",
       bin2hex(serial, size));

/* Extract some of the public key algorithm's parameters
 */
algo = gnutls_x509_crt_get_pk_algorithm(cert, &bits);

printf("Certificate public key: %s",
       gnutls_pk_algorithm_get_name(algo));
```

```
                /* Print the version of the X.509
                 * certificate.
                 */
                printf("\tCertificate version: #%d\n",
                       gnutls_x509_crt_get_version(cert));

                size = sizeof(dn);
                gnutls_x509_crt_get_dn(cert, dn, &size);
                printf("\tDN: %s\n", dn);

                size = sizeof(dn);
                gnutls_x509_crt_get_issuer_dn(cert, dn, &size);
                printf("\tIssuer's DN: %s\n", dn);

                gnutls_x509_crt_deinit(cert);

        }
}
```

7.5.3 Listing the ciphersuites in a priority string

This is a small program to list the enabled ciphersuites by a priority string.

```
/* This example code is placed in the public domain. */

#include <config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>

static void print_cipher_suite_list(const char *priorities)
{
        size_t i;
        int ret;
        unsigned int idx;
        const char *name;
        const char *err;
        unsigned char id[2];
        gnutls_protocol_t version;
        gnutls_priority_t pcache;

        if (priorities != NULL) {
                printf("Cipher suites for %s\n", priorities);

                ret = gnutls_priority_init(&pcache, priorities, &err);
                if (ret < 0) {
```

```
                        fprintf(stderr, "Syntax error at: %s\n", err);
                        exit(1);
                }

                for (i = 0;; i++) {
                        ret =
                            gnutls_priority_get_cipher_suite_index(pcache,
                                                                   i,
                                                                   &idx);
                        if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
                                break;
                        if (ret == GNUTLS_E_UNKNOWN_CIPHER_SUITE)
                                continue;

                        name =
                            gnutls_cipher_suite_info(idx, id, NULL, NULL,
                                                     NULL, &version);

                        if (name != NULL)
                                printf("%-50s\t0x%02x, 0x%02x\t%s\n",
                                       name, (unsigned char) id[0],
                                       (unsigned char) id[1],
                                       gnutls_protocol_get_name(version));
                }

                return;
        }
}

int main(int argc, char **argv)
{
        if (argc > 1)
                print_cipher_suite_list(argv[1]);
        return 0;
}
```

7.5.4 PKCS #12 structure generation example

This small program demonstrates the usage of the PKCS #12 API, by generating such a structure.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
```

```
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/pkcs12.h>

#include "examples.h"

#define OUTFILE "out.p12"

/* This function will write a pkcs12 structure into a file.
 * cert: is a DER encoded certificate
 * pkcs8_key: is a PKCS #8 encrypted key (note that this must be
 *  encrypted using a PKCS #12 cipher, or some browsers will crash)
 * password: is the password used to encrypt the PKCS #12 packet.
 */
int
write_pkcs12(const gnutls_datum_t * cert,
             const gnutls_datum_t * pkcs8_key, const char *password)
{
        gnutls_pkcs12_t pkcs12;
        int ret, bag_index;
        gnutls_pkcs12_bag_t bag, key_bag;
        char pkcs12_struct[10 * 1024];
        size_t pkcs12_struct_size;
        FILE *fp;

        /* A good idea might be to use gnutls_x509_privkey_get_key_id()
         * to obtain a unique ID.
         */
        gnutls_datum_t key_id = { (void *) "\x00\x00\x07", 3 };

        gnutls_global_init();

        /* Firstly we create two helper bags, which hold the certificate,
         * and the (encrypted) key.
         */

        gnutls_pkcs12_bag_init(&bag);
        gnutls_pkcs12_bag_init(&key_bag);

        ret =
            gnutls_pkcs12_bag_set_data(bag, GNUTLS_BAG_CERTIFICATE, cert);
        if (ret < 0) {
                fprintf(stderr, "ret: %s\n", gnutls_strerror(ret));
                return 1;
        }

        /* ret now holds the bag's index.
```

```
 */
bag_index = ret;

/* Associate a friendly name with the given certificate. Used
 * by browsers.
 */
gnutls_pkcs12_bag_set_friendly_name(bag, bag_index, "My name");

/* Associate the certificate with the key using a unique key
 * ID.
 */
gnutls_pkcs12_bag_set_key_id(bag, bag_index, &key_id);

/* use weak encryption for the certificate.
 */
gnutls_pkcs12_bag_encrypt(bag, password,
                          GNUTLS_PKCS_USE_PKCS12_RC2_40);

/* Now the key.
 */

ret = gnutls_pkcs12_bag_set_data(key_bag,
                                 GNUTLS_BAG_PKCS8_ENCRYPTED_KEY,
                                 pkcs8_key);
if (ret < 0) {
        fprintf(stderr, "ret: %s\n", gnutls_strerror(ret));
        return 1;
}

/* Note that since the PKCS #8 key is already encrypted we don't
 * bother encrypting that bag.
 */
bag_index = ret;

gnutls_pkcs12_bag_set_friendly_name(key_bag, bag_index, "My name");

gnutls_pkcs12_bag_set_key_id(key_bag, bag_index, &key_id);


/* The bags were filled. Now create the PKCS #12 structure.
 */
gnutls_pkcs12_init(&pkcs12);

/* Insert the two bags in the PKCS #12 structure.
 */

gnutls_pkcs12_set_bag(pkcs12, bag);
```

```
        gnutls_pkcs12_set_bag(pkcs12, key_bag);


        /* Generate a message authentication code for the PKCS #12
         * structure.
         */
        gnutls_pkcs12_generate_mac(pkcs12, password);

        pkcs12_struct_size = sizeof(pkcs12_struct);
        ret =
            gnutls_pkcs12_export(pkcs12, GNUTLS_X509_FMT_DER,
                                 pkcs12_struct, &pkcs12_struct_size);
        if (ret < 0) {
                fprintf(stderr, "ret: %s\n", gnutls_strerror(ret));
                return 1;
        }

        fp = fopen(OUTFILE, "w");
        if (fp == NULL) {
                fprintf(stderr, "cannot open file\n");
                return 1;
        }
        fwrite(pkcs12_struct, 1, pkcs12_struct_size, fp);
        fclose(fp);

        gnutls_pkcs12_bag_deinit(bag);
        gnutls_pkcs12_bag_deinit(key_bag);
        gnutls_pkcs12_deinit(pkcs12);

        return 0;
}
```

8 System-wide configuration of the library

GnuTLS 3.6.9 introduced a system-wide configuration of the library which can be used to disable or mark algorithms and protocols as insecure system-wide, overriding the library defaults. The format of this configuration file is of an INI file, with the hash ('#') allowed for commenting. It intentionally does not allow switching algorithms or protocols which were disabled or marked as insecure during compile time to the secure set. This is to prevent the feature from being used to attack the system. Unknown options or sections in the configuration file are skipped unless the environment variable `GNUTLS_SYSTEM_PRIORITY_FAIL_ON_INVALID` is set to 1, where it would cause the library to exit on unknown options.

The location of the default configuration file is `/etc/gnutls/config`, but its actual location may be overridden during compile time or at run-time using the `GNUTLS_SYSTEM_PRIORITY_FILE` environment variable. The file used can be queried using [gnutls_get_system_config_file], page 411.

`const char * gnutls_get_system_config_file ( void)` [Function]

Returns the filename of the system wide configuration file loaded by the library. The returned pointer is valid until the library is unloaded.

Returns: a constant pointer to the config file loaded, or `NULL` if none

Since: 3.6.9

8.1 Application-specific priority strings

It is possible to specify custom cipher priority strings, in addition to the default priority strings (`NORMAL`, `PERFORMANCE`, etc.). These can be used either by individual applications, or even as the default option if the library is compiled with the configuration option `--with-default-priority-string`. In the latter case the defined priority string will be used for applications using [gnutls_set_default_priority], page 469 or [gnutls_set_default_priority_append], page 469.

The priority strings can be specified in the global section of the configuration file, or in the section named `[priorities]`. The format is '`KEYWORD = VALUE`', e.g.,

When used they may be followed by additional options that will be appended to the system string (e.g., '`@EXAMPLE-PRIORITY:+SRP`'). '`EXAMPLE-PRIORITY=NORMAL:+ARCFOUR-128`'. Since version 3.5.1 applications are allowed to specify fallback keywords such as @KEYWORD1,@KEYWORD2, and the first valid keyword will be used.

The following example configuration defines a priority string called `@SYSTEM`. When set, its full settings can be queried using `gnutls-cli --priority @SYSTEM --list`.

```
[priorities]
SYSTEM = NORMAL:-AES-128-CBC:-AES-256-CBC
```

8.2 Disabling algorithms and protocols

The approach above works well to create consistent system-wide settings for co-operative GnuTLS applications. When an application however does not use the [gnutls_set_default_priority], page 469 or [gnutls_set_default_priority_append], page 469 functions, the method is not sufficient to prevent applications from using protocols or

algorithms forbidden by a local policy. The override method described below enables the deprecation of algorithms and protocols system-wide for all applications.

The available options must be set in the `[overrides]` section of the configuration file and can be

- `insecure-sig-for-cert`: to mark the signature algorithm as insecure when used in certificates.
- `insecure-sig`: to mark the signature algorithm as insecure for any use.
- `insecure-hash`: to mark the hash algorithm as insecure for digital signature use (provides a more generic way to disable digital signatures for broken hash algorithms).
- `disabled-version`: to disable the specified TLS versions.
- `tls-disabled-cipher`: to disable the specified ciphers for use in the TLS or DTLS protocols.
- `tls-disabled-mac`: to disable the specified MAC algorithms for use in the TLS or DTLS protocols.
- `tls-disabled-group`: to disable the specified group for use in the TLS or DTLS protocols.
- `tls-disabled-kx`: to disable the specified key exchange algorithms for use in the TLS or DTLS protocols (applies to TLS1.2 or earlier).

Each of the options can be repeated multiple times when multiple values need to be disabled.

The valid values for the options above can be found in the 'Protocols', 'Digests' 'PK-signatures', 'Protocols', 'Ciphrers', and 'MACs' fields of the output of `gnutls-cli --list`.

8.2.1 Examples

The following example marks as insecure all digital signature algorithms which depend on SHA384, as well as the RSA-SHA1 signature algorithm.

```
[overrides]
insecure-hash = sha384
insecure-sig = rsa-sha1
```

The following example marks RSA-SHA256 as insecure for use in certificates and disables the TLS1.0 and TLS1.1 protocols.

```
[overrides]
insecure-sig-for-cert = rsa-sha256
disabled-version = tls1.0
disabled-version = tls1.1
```

The following example disables the `AES-128-CBC` and `AES-256-CBC` ciphers, the `HMAC-SHA1` MAC algorithm and the `GROUP-FFDHE8192` group for TLS and DTLS protocols.

```
[overrides]
tls-disabled-cipher = aes-128-cbc
tls-disabled-cipher = aes-256-cbc
tls-disabled-mac = sha1
tls-disabled-group = group-ffdhe8192
```

8.3 Querying for disabled algorithms and protocols

When necessary applications can query whether a particular algorithm or protocol has been marked as insecure or disabled system-wide. Digital signatures can be queried using the following algorithms.

unsigned [gnutls_sign_is_secure], page 472, (gnutls_sign_algorithm_t *algorithm*)
unsigned [gnutls_sign_is_secure2], page 472, (gnutls_sign_algorithm_t *algorithm*, unsigned int *flags*)

Any disabled protocol versions or elliptic curves will not show up in the lists provided by the following functions.

*const gnutls_protocol_t ** [gnutls_protocol_list], page 440, (*void*)
*const gnutls_group_t ** [gnutls_group_list], page 414, (*void*)
*const gnutls_ecc_curve_t ** [gnutls_ecc_curve_list], page 406, (*void*)

It is not possible to query for insecure hash algorithms directly (only indirectly through the signature API).

8.4 Overriding the parameter verification profile

When verifying a certificate or TLS session parameters, GnuTLS uses a set of profiles associated with the session to determine whether the parameters seen in the session are acceptable. For example, whether the RSA public key size as seen on the wire, or the Diffie-Hellman parameters for the session. These profiles are normally set using the `%PROFILE` priority string (see Section 6.10 [Priority Strings], page 172, and Section 6.11 [Selecting cryptographic key sizes], page 179).

It is possible to set the low bar profile that applications cannot override using the following.

```
[overrides]

# do not allow applications use the LOW or VERY-WEAK profiles.
min-verification-profile = legacy
```

8.5 Overriding the default priority string

GnuTLS uses default priority string which is defined at compiled time. Usually it is set to `NORMAL`. This override allows to set the default priority string to something more appropriate for a given deployment.

Below example sets a more specific default priority string.

```
[overrides]
default-priority-string = SECURE128:-VERS-TLS-ALL:+VERS-TLS1.3
```

9 Using GnuTLS as a cryptographic library

GnuTLS is not a low-level cryptographic library, i.e., it does not provide access to basic cryptographic primitives. However it abstracts the internal cryptographic back-end (see Section 11.5 [Cryptographic Backend], page 325), providing symmetric crypto, hash and HMAC algorithms, as well access to the random number generation. For a low-level crypto API the usage of nettle[1] library is recommended.

9.1 Symmetric algorithms

The available functions to access symmetric crypto algorithms operations are listed in the sections below. The supported algorithms are the algorithms required by the TLS protocol. They are listed in Figure 9.1. Note that there two types of ciphers, the ones providing an authenticated-encryption with associated data (AEAD), and the legacy ciphers which provide raw access to the ciphers. We recommend the use of the AEAD ciphers under the AEAD APIs for new applications as they are designed to minimize the misuse of cryptographic primitives.

[1] See `https://www.lysator.liu.se/~nisse/nettle/`.

`GNUTLS_CIPHER_UNKNOWN`
: Value to identify an unknown/unsupported algorithm.

`GNUTLS_CIPHER_NULL`
: The NULL (identity) encryption algorithm.

`GNUTLS_CIPHER_ARCFOUR_128`
: ARCFOUR stream cipher with 128-bit keys.

`GNUTLS_CIPHER_3DES_CBC`
: 3DES in CBC mode.

`GNUTLS_CIPHER_AES_128_CBC`
: AES in CBC mode with 128-bit keys.

`GNUTLS_CIPHER_AES_256_CBC`
: AES in CBC mode with 256-bit keys.

`GNUTLS_CIPHER_ARCFOUR_40`
: ARCFOUR stream cipher with 40-bit keys.

`GNUTLS_CIPHER_CAMELLIA_128_CBC`
: Camellia in CBC mode with 128-bit keys.

`GNUTLS_CIPHER_CAMELLIA_256_CBC`
: Camellia in CBC mode with 256-bit keys.

`GNUTLS_CIPHER_AES_192_CBC`
: AES in CBC mode with 192-bit keys.

`GNUTLS_CIPHER_AES_128_GCM`
: AES in GCM mode with 128-bit keys (AEAD).

`GNUTLS_CIPHER_AES_256_GCM`
: AES in GCM mode with 256-bit keys (AEAD).

`GNUTLS_CIPHER_CAMELLIA_192_CBC`
: Camellia in CBC mode with 192-bit keys.

`GNUTLS_CIPHER_SALSA20_256`
: Salsa20 with 256-bit keys.

`GNUTLS_CIPHER_ESTREAM_SALSA20_256`
: Estream's Salsa20 variant with 256-bit keys.

`GNUTLS_CIPHER_CAMELLIA_128_GCM`
: CAMELLIA in GCM mode with 128-bit keys (AEAD).

`GNUTLS_CIPHER_CAMELLIA_256_GCM`
: CAMELLIA in GCM mode with 256-bit keys (AEAD).

`GNUTLS_CIPHER_RC2_40_CBC`
: RC2 in CBC mode with 40-bit keys.

`GNUTLS_CIPHER_DES_CBC`
: DES in CBC mode (56-bit keys).

`GNUTLS_CIPHER_AES_128_CCM`
: AES in CCM mode with 128-bit keys (AEAD).

`GNUTLS_CIPHER_AES_256_CCM`
: AES in CCM mode with 256-bit keys (AEAD).

Authenticated-encryption API

The AEAD API provides access to all ciphers supported by GnuTLS which support authenticated encryption with associated data; these ciphers are marked with the AEAD keyword on the table above. The AEAD cipher API is particularly suitable for message or packet-encryption as it provides authentication and encryption on the same API. See RFC5116 for more information on authenticated encryption.

int [gnutls_aead_cipher_init], page 724, (gnutls_aead_cipher_hd_t * *handle*, gnutls_cipher_algorithm_t *cipher*, const gnutls_datum_t * *key*)
int [gnutls_aead_cipher_encrypt], page 723, (gnutls_aead_cipher_hd_t *handle*, const void * *nonce*, size_t *nonce_len*, const void * *auth*, size_t *auth_len*, size_t *tag_size*, const void * *ptext*, size_t *ptext_len*, void * *ctext*, size_t * *ctext_len*)
int [gnutls_aead_cipher_decrypt], page 721, (gnutls_aead_cipher_hd_t *handle*, const void * *nonce*, size_t *nonce_len*, const void * *auth*, size_t *auth_len*, size_t *tag_size*, const void * *ctext*, size_t *ctext_len*, void * *ptext*, size_t * *ptext_len*)
void [gnutls_aead_cipher_deinit], page 722, (gnutls_aead_cipher_hd_t *handle*)

Because the encryption function above may be difficult to use with scattered data, we provide the following API.

`int gnutls_aead_cipher_encryptv` (*gnutls_aead_cipher_hd_t* `handle`, *const void* * `nonce`, *size_t* `nonce_len`, *const giovec_t* * `auth_iov`, *int* `auth_iovcnt`, *size_t* `tag_size`, *const giovec_t* * `iov`, *int* `iovcnt`, *void* * `ctext`, *size_t* * `ctext_len`) [Function]

handle: is a gnutls_aead_cipher_hd_t type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth_iov: additional data to be authenticated

auth_iovcnt: The number of buffers in auth_iov

tag_size: The size of the tag to use (use zero for the default)

iov: the data to be encrypted

iovcnt: The number of buffers in iov

ctext: the encrypted data including authentication tag

ctext_len: the length of encrypted data (initially must hold the maximum available size, including space for tag)

This function will encrypt the provided data buffers using the algorithm specified by the context. The output data will contain the authentication tag.

Returns: Zero or a negative error code on error.

Since: 3.6.3

Legacy API

The legacy API provides low-level access to all legacy ciphers supported by GnuTLS, and some of the AEAD ciphers (e.g., AES-GCM and CHACHA20). The restrictions of the nettle library implementation of the ciphers apply verbatim to this API[2].

int [gnutls_cipher_init], page 727, (gnutls_cipher_hd_t * *handle*, gnutls_cipher_algorithm_t *cipher*, const gnutls_datum_t * *key*, const gnutls_datum_t * *iv*)
int [gnutls_cipher_encrypt2], page 726, (gnutls_cipher_hd_t *handle*, const void * *ptext*, size_t *ptext_len*, void * *ctext*, size_t *ctext_len*)
int [gnutls_cipher_decrypt2], page 725, (gnutls_cipher_hd_t *handle*, const void * *ctext*, size_t *ctext_len*, void * *ptext*, size_t *ptext_len*)
void [gnutls_cipher_set_iv], page 727, (gnutls_cipher_hd_t *handle*, void * *iv*, size_t *ivlen*)
void [gnutls_cipher_deinit], page 725, (gnutls_cipher_hd_t *handle*)

int [gnutls_cipher_add_auth], page 724, (gnutls_cipher_hd_t *handle*, const void * *ptext*, size_t *ptext_size*)
int [gnutls_cipher_tag], page 727, (gnutls_cipher_hd_t *handle*, void * *tag*, size_t *tag_size*)

While the latter two functions allow the same API can be used with authenticated encryption ciphers, it is recommended to use the following functions which are solely for AEAD ciphers. The latter API is designed to be simple to use and also hard to misuse, by handling the tag verification and addition in transparent way.

9.2 Public key algorithms

Public key cryptography algorithms such as RSA, DSA and ECDSA, are accessed using the abstract key API in Section 5.1 [Abstract key types], page 100. This is a high level API with the advantage of transparently handling keys stored in memory and keys present in smart cards.

[2] See the nettle manual `https://www.lysator.liu.se/~nisse/nettle/nettle.html`

int [gnutls_privkey_init], page 693, (gnutls_privkey_t * *key*)
int [gnutls_privkey_import_url], page 692, (gnutls_privkey_t *key*, const char * *url*, unsigned int *flags*)
int [gnutls_privkey_import_x509_raw], page 693, (gnutls_privkey_t *pkey*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*, const char * *password*, unsigned int *flags*)
int [gnutls_privkey_sign_data], page 694, (gnutls_privkey_t *signer*, gnutls_digest_algorithm_t *hash*, unsigned int *flags*, const gnutls_datum_t * *data*, gnutls_datum_t * *signature*)
int [gnutls_privkey_sign_hash], page 695, (gnutls_privkey_t *signer*, gnutls_digest_algorithm_t *hash_algo*, unsigned int *flags*, const gnutls_datum_t * *hash_data*, gnutls_datum_t * *signature*)
void [gnutls_privkey_deinit], page 679, (gnutls_privkey_t *key*)

int [gnutls_pubkey_init], page 709, (gnutls_pubkey_t * *key*)
int [gnutls_pubkey_import_url], page 708, (gnutls_pubkey_t *key*, const char * *url*, unsigned int *flags*)
int [gnutls_pubkey_import_x509], page 708, (gnutls_pubkey_t *key*, gnutls_x509_crt_t *crt*, unsigned int *flags*)
int [gnutls_pubkey_verify_data2], page 710, (gnutls_pubkey_t *pubkey*, gnutls_sign_algorithm_t *algo*, unsigned int *flags*, const gnutls_datum_t * *data*, const gnutls_datum_t * *signature*)
int [gnutls_pubkey_verify_hash2], page 711, (gnutls_pubkey_t *key*, gnutls_sign_algorithm_t *algo*, unsigned int *flags*, const gnutls_datum_t * *hash*, const gnutls_datum_t * *signature*)
void [gnutls_pubkey_deinit], page 697, (gnutls_pubkey_t *key*)

Keys stored in memory can be imported using functions like [gnutls_privkey_import_x509_raw], page 693, while keys on smart cards or HSMs should be imported using their PKCS#11 URL with [gnutls_privkey_import_url], page 692.

If any of the smart card operations require PIN, that should be provided either by setting the global PIN function ([gnutls_pkcs11_set_pin_function], page 665), or better with the targeted to structures functions such as [gnutls_privkey_set_pin_function], page 694.

9.2.1 Key generation

All supported key types (including RSA, DSA, ECDSA, Ed25519, Ed448) can be generated with GnuTLS. They can be generated with the simpler [gnutls_privkey_generate], page 683 or with the more advanced [gnutls_privkey_generate2], page 683.

`int gnutls_privkey_generate2` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`, *const gnutls_keygen_data_st* * `data`, *unsigned* `data_size`) [Function]

pkey: The private key

algo: is one of the algorithms in `gnutls_pk_algorithm_t` .

bits: the size of the modulus

flags: Must be zero or flags from `gnutls_privkey_flags_t` .

data: Allow specifying `gnutls_keygen_data_st` types such as the seed to be used.

data_size: The number of `data` available.

This function will generate a random private key. Note that this function must be called on an initialized private key.

The flag `GNUTLS_PRIVKEY_FLAG_PROVABLE` instructs the key generation process to use algorithms like Shawe-Taylor (from FIPS PUB186-4) which generate provable parameters out of a seed for RSA and DSA keys. On DSA keys the PQG parameters are generated using the seed, while on RSA the two primes. To specify an explicit seed (by default a random seed is used), use the `data` with a `GNUTLS_KEYGEN_SEED` type.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

To export the generated keys in memory or in files it is recommended to use the PKCS8 form as it can handle all key types, and can store additional parameters such as the seed, in case of provable RSA or DSA keys. Generated keys can be exported in memory using `gnutls_privkey_export_x509()` , and then with `gnutls_x509_privkey_export2_pkcs8()` .

If key generation is part of your application, avoid setting the number of bits directly, and instead use `gnutls_sec_param_to_pk_bits()` . That way the generated keys will adapt to the security levels of the underlying GnuTLS library.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

9.3 Cryptographic Message Syntax / PKCS7

The CMS or PKCS #7 format is a commonly used format for digital signatures. PKCS #7 is the name of the original standard when published by RSA, though today the standard is adopted by IETF under the name CMS.

The standards include multiple ways of signing a digital document, e.g., by embedding the data into the signature, or creating detached signatures of the data, including a timestamp, additional certificates etc. In certain cases the same format is also used to transport lists of certificates and CRLs.

It is a relatively popular standard to sign structures, and is being used to sign in PDF files, as well as for signing kernel modules and other structures.

In GnuTLS, the basic functions to initialize, deinitialize, import, export or print information about a PKCS #7 structure are listed below.

int [gnutls_pkcs7_init], page 626, (gnutls_pkcs7_t * *pkcs7*)
void [gnutls_pkcs7_deinit], page 621, (gnutls_pkcs7_t *pkcs7*)
int [gnutls_pkcs7_export2], page 622, (gnutls_pkcs7_t *pkcs7*, gnutls_x509_crt_fmt_t *format*, gnutls_datum_t * *out*)
int [gnutls_pkcs7_import], page 625, (gnutls_pkcs7_t *pkcs7*, const gnutls_datum_t * *data*, gnutls_x509_crt_fmt_t *format*)
int [gnutls_pkcs7_print], page 626, (gnutls_pkcs7_t *pkcs7*, gnutls_certificate_print_formats_t *format*, gnutls_datum_t * *out*)

The following functions allow the verification of a structure using either a trust list, or individual certificates. The [gnutls_pkcs7_sign], page 627 function is the data signing function.

int [gnutls_pkcs7_verify_direct], page 629, (gnutls_pkcs7_t *pkcs7*, gnutls_x509_crt_t *signer*, unsigned *idx*, const gnutls_datum_t * *data*, unsigned *flags*)
int [gnutls_pkcs7_verify], page 628, (gnutls_pkcs7_t *pkcs7*, gnutls_x509_trust_list_t *tl*, gnutls_typed_vdata_st * *vdata*, unsigned int *vdata_size*, unsigned *idx*, const gnutls_datum_t * *data*, unsigned *flags*)

int gnutls_pkcs7_sign (*gnutls_pkcs7_t* **pkcs7**, *gnutls_x509_crt_t* **signer**, *gnutls_privkey_t* **signer_key**, *const gnutls_datum_t* * **data**, *gnutls_pkcs7_attrs_t* **signed_attrs**, *gnutls_pkcs7_attrs_t* **unsigned_attrs**, *gnutls_digest_algorithm_t* **dig**, *unsigned* **flags**) [Function]

pkcs7: should contain a gnutls_pkcs7_t type

signer: the certificate to sign the structure

signer_key: the key to sign the structure

data: The data to be signed or NULL if the data are already embedded

signed_attrs: Any additional attributes to be included in the signed ones (or NULL)

unsigned_attrs: Any additional attributes to be included in the unsigned ones (or NULL)

dig: The digest algorithm to use for signing

flags: Should be zero or one of GNUTLS_PKCS7 flags

This function will add a signature in the provided PKCS 7 structure for the provided data. Multiple signatures can be made with different signers.

The available flags are: GNUTLS_PKCS7_EMBED_DATA , GNUTLS_PKCS7_INCLUDE_TIME , GNUTLS_PKCS7_INCLUDE_CERT , and GNUTLS_PKCS7_WRITE_SPKI . They are explained in the gnutls_pkcs7_sign_flags definition.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.2

`GNUTLS_PKCS7_EMBED_DATA`
: The signed data will be embedded in the structure.

`GNUTLS_PKCS7_INCLUDE_TIME`
: The signing time will be included in the structure.

`GNUTLS_PKCS7_INCLUDE_CERT`
: The signer's certificate will be included in the cert list.

`GNUTLS_PKCS7_WRITE_SPKI`
: Use the signer's key identifier instead of name.

Figure 9.2: Flags applicable to gnutls_pkcs7_sign()

Other helper functions which allow to access the signatures, or certificates attached in the structure are listed below.

int [gnutls_pkcs7_get_signature_count], page 625, (gnutls_pkcs7_t *pkcs7*)
int [gnutls_pkcs7_get_signature_info], page 625, (gnutls_pkcs7_t *pkcs7*, unsigned *idx*, gnutls_pkcs7_signature_info_st * *info*)
int [gnutls_pkcs7_get_crt_count], page 623, (gnutls_pkcs7_t *pkcs7*)
int [gnutls_pkcs7_get_crt_raw2], page 624, (gnutls_pkcs7_t *pkcs7*, unsigned *indx*, gnutls_datum_t * *cert*)
int [gnutls_pkcs7_get_crl_count], page 623, (gnutls_pkcs7_t *pkcs7*)
int [gnutls_pkcs7_get_crl_raw2], page 623, (gnutls_pkcs7_t *pkcs7*, unsigned *indx*, gnutls_datum_t * *crl*)

To append certificates, or CRLs in the structure the following functions are provided.

int [gnutls_pkcs7_set_crt_raw], page 627, (gnutls_pkcs7_t *pkcs7*, const gnutls_datum_t * *crt*)
int [gnutls_pkcs7_set_crt], page 627, (gnutls_pkcs7_t *pkcs7*, gnutls_x509_crt_t *crt*)
int [gnutls_pkcs7_set_crl_raw], page 627, (gnutls_pkcs7_t *pkcs7*, const gnutls_datum_t * *crl*)
int [gnutls_pkcs7_set_crl], page 627, (gnutls_pkcs7_t *pkcs7*, gnutls_x509_crl_t *crl*)

9.4 Hash and MAC functions

The available operations to access hash functions and hash-MAC (HMAC) algorithms are shown below. HMAC algorithms provided keyed hash functionality. The supported MAC and HMAC algorithms are listed in Figure 9.3. Note that, despite the `hmac` part in the name of the MAC functions listed below, they can be used either for HMAC or MAC operations.

`GNUTLS_MAC_UNKNOWN`
: Unknown MAC algorithm.

`GNUTLS_MAC_NULL`
: NULL MAC algorithm (empty output).

`GNUTLS_MAC_MD5`
: HMAC-MD5 algorithm.

`GNUTLS_MAC_SHA1`
: HMAC-SHA-1 algorithm.

`GNUTLS_MAC_RMD160`
: HMAC-RMD160 algorithm.

`GNUTLS_MAC_MD2`
: HMAC-MD2 algorithm.

`GNUTLS_MAC_SHA256`
: HMAC-SHA-256 algorithm.

`GNUTLS_MAC_SHA384`
: HMAC-SHA-384 algorithm.

`GNUTLS_MAC_SHA512`
: HMAC-SHA-512 algorithm.

`GNUTLS_MAC_SHA224`
: HMAC-SHA-224 algorithm.

`GNUTLS_MAC_SHA3_224`
: Reserved; unimplemented.

`GNUTLS_MAC_SHA3_256`
: Reserved; unimplemented.

`GNUTLS_MAC_SHA3_384`
: Reserved; unimplemented.

`GNUTLS_MAC_SHA3_512`
: Reserved; unimplemented.

`GNUTLS_MAC_MD5_SHA1`
: Combined MD5+SHA1 MAC placeholder.

`GNUTLS_MAC_GOSTR_94`
: HMAC GOST R 34.11-94 algorithm.

`GNUTLS_MAC_STREEBOG_256`
: HMAC GOST R 34.11-2001 (Streebog) algorithm, 256 bit.

`GNUTLS_MAC_STREEBOG_512`
: HMAC GOST R 34.11-2001 (Streebog) algorithm, 512 bit.

`GNUTLS_MAC_AEAD`
: MAC implicit through AEAD cipher.

`GNUTLS_MAC_UMAC_96`
: The UMAC-96 MAC algorithm (requires nonce).

`GNUTLS_MAC_UMAC_128`
: The UMAC-128 MAC algorithm (requires nonce).

int [gnutls_hmac_init], page 735, (gnutls_hmac_hd_t * *dig*, gnutls_mac_algorithm_t *algorithm*, const void * *key*, size_t *keylen*)
int [gnutls_hmac], page 734, (gnutls_hmac_hd_t *handle*, const void * *ptext*, size_t *ptext_len*)
void [gnutls_hmac_output], page 736, (gnutls_hmac_hd_t *handle*, void * *digest*)
void [gnutls_hmac_deinit], page 734, (gnutls_hmac_hd_t *handle*, void * *digest*)
unsigned [gnutls_hmac_get_len], page 735, (gnutls_mac_algorithm_t *algorithm*)
int [gnutls_hmac_fast], page 735, (gnutls_mac_algorithm_t *algorithm*, const void * *key*, size_t *keylen*, const void * *ptext*, size_t *ptext_len*, void * *digest*)

The available functions to access hash functions are shown below. The supported hash functions are shown in Figure 9.4.

int [gnutls_hash_init], page 733, (gnutls_hash_hd_t * *dig*, gnutls_digest_algorithm_t *algorithm*)
int [gnutls_hash], page 732, (gnutls_hash_hd_t *handle*, const void * *ptext*, size_t *ptext_len*)
void [gnutls_hash_output], page 733, (gnutls_hash_hd_t *handle*, void * *digest*)
void [gnutls_hash_deinit], page 732, (gnutls_hash_hd_t *handle*, void * *digest*)
unsigned [gnutls_hash_get_len], page 733, (gnutls_digest_algorithm_t *algorithm*)
int [gnutls_hash_fast], page 732, (gnutls_digest_algorithm_t *algorithm*, const void * *ptext*, size_t *ptext_len*, void * *digest*)
int [gnutls_fingerprint], page 410, (gnutls_digest_algorithm_t *algo*, const gnutls_datum_t * *data*, void * *result*, size_t * *result_size*)

GNUTLS_DIG_UNKNOWN
: Unknown hash algorithm.

GNUTLS_DIG_NULL
: NULL hash algorithm (empty output).

GNUTLS_DIG_MD5
: MD5 algorithm.

GNUTLS_DIG_SHA1
: SHA-1 algorithm.

GNUTLS_DIG_RMD160
: RMD160 algorithm.

GNUTLS_DIG_MD2
: MD2 algorithm.

GNUTLS_DIG_SHA256
: SHA-256 algorithm.

GNUTLS_DIG_SHA384
: SHA-384 algorithm.

GNUTLS_DIG_SHA512
: SHA-512 algorithm.

GNUTLS_DIG_SHA224
: SHA-224 algorithm.

GNUTLS_DIG_SHA3_224
: SHA3-224 algorithm.

GNUTLS_DIG_SHA3_256
: SHA3-256 algorithm.

GNUTLS_DIG_SHA3_384
: SHA3-384 algorithm.

GNUTLS_DIG_SHA3_512
: SHA3-512 algorithm.

GNUTLS_DIG_MD5_SHA1
: Combined MD5+SHA1 algorithm.

GNUTLS_DIG_GOSTR_94
: GOST R 34.11-94 algorithm.

9.5 Random number generation

Access to the random number generator is provided using the [gnutls_rnd], page 737 function. It allows obtaining random data of various levels.

`GNUTLS_RND_NONCE`
: Non-predictable random number. Fatal in parts of session if broken, i.e., vulnerable to statistical analysis.

`GNUTLS_RND_RANDOM`
: Pseudo-random cryptographic random number. Fatal in session if broken. Example use: temporal keys.

`GNUTLS_RND_KEY`
: Fatal in many sessions if broken. Example use: Long-term keys.

Figure 9.5: The random number levels.

`int gnutls_rnd` (*gnutls_rnd_level_t* `level`, *void ** `data`, *size_t* `len`) [Function]

level: a security level

data: place to store random bytes

len: The requested size

This function will generate random data and store it to output buffer. The value of `level` should be one of `GNUTLS_RND_NONCE` , `GNUTLS_RND_RANDOM` and `GNUTLS_RND_KEY` . See the manual and `gnutls_rnd_level_t` for detailed information.

This function is thread-safe and also fork-safe.

Returns: Zero on success, or a negative error code on error.

Since: 2.12.0

See Section 11.6 [Random Number Generators-internals], page 327, for more information on the random number generator operation.

9.6 Overriding algorithms

In systems which provide a hardware accelerated cipher implementation that is not directly supported by GnuTLS, it is possible to utilize it. There are functions which allow overriding the default cipher, digest and MAC implementations. Those are described below.

To override public key operations see Section 5.1.2 [Abstract private keys], page 102.

`int gnutls_crypto_register_cipher` (*gnutls_cipher_algorithm_t* `algorithm`, *int* `priority`, *gnutls_cipher_init_func* `init`, *gnutls_cipher_setkey_func* `setkey`, *gnutls_cipher_setiv_func* `setiv`, *gnutls_cipher_encrypt_func* `encrypt`, *gnutls_cipher_decrypt_func* `decrypt`, *gnutls_cipher_deinit_func* `deinit`) [Function]

algorithm: is the gnutls algorithm identifier

priority: is the priority of the algorithm

init: A function which initializes the cipher

setkey: A function which sets the key of the cipher

setiv: A function which sets the nonce/IV of the cipher (non-AEAD)

encrypt: A function which performs encryption (non-AEAD)

decrypt: A function which performs decryption (non-AEAD)

deinit: A function which deinitializes the cipher

This function will register a cipher algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

In the case the registered init or setkey functions return `GNUTLS_E_NEED_FALLBACK` , GnuTLS will attempt to use the next in priority registered cipher.

The functions which are marked as non-AEAD they are not required when registering a cipher to be used with the new AEAD API introduced in GnuTLS 3.4.0. Internally GnuTLS uses the new AEAD API.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

`int gnutls_crypto_register_aead_cipher` [Function]
(*gnutls_cipher_algorithm_t* `algorithm`, *int* `priority`, *gnutls_cipher_init_func* `init`, *gnutls_cipher_setkey_func* `setkey`, *gnutls_cipher_aead_encrypt_func* `aead_encrypt`, *gnutls_cipher_aead_decrypt_func* `aead_decrypt`, *gnutls_cipher_deinit_func* `deinit`)

algorithm: is the gnutls AEAD cipher identifier

priority: is the priority of the algorithm

init: A function which initializes the cipher

setkey: A function which sets the key of the cipher

aead_encrypt: Perform the AEAD encryption

aead_decrypt: Perform the AEAD decryption

deinit: A function which deinitializes the cipher

This function will register a cipher algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

In the case the registered init or setkey functions return `GNUTLS_E_NEED_FALLBACK` , GnuTLS will attempt to use the next in priority registered cipher.

The functions registered will be used with the new AEAD API introduced in GnuTLS 3.4.0. Internally GnuTLS uses the new AEAD API.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

`int gnutls_crypto_register_mac` (*gnutls_mac_algorithm_t* `algorithm`, *int* `priority`, *gnutls_mac_init_func* `init`, *gnutls_mac_setkey_func* `setkey`, *gnutls_mac_setnonce_func* `setnonce`, *gnutls_mac_hash_func* `hash`, *gnutls_mac_output_func* `output`, *gnutls_mac_deinit_func* `deinit`, *gnutls_mac_fast_func* `hash_fast`) [Function]

algorithm: is the gnutls MAC identifier

priority: is the priority of the algorithm

init: A function which initializes the MAC

setkey: A function which sets the key of the MAC

setnonce: A function which sets the nonce for the mac (may be `NULL` for common MAC algorithms)

hash: Perform the hash operation

output: Provide the output of the MAC

deinit: A function which deinitializes the MAC

hash_fast: Perform the MAC operation in one go

This function will register a MAC algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

`int gnutls_crypto_register_digest` (*gnutls_digest_algorithm_t* `algorithm`, *int* `priority`, *gnutls_digest_init_func* `init`, *gnutls_digest_hash_func* `hash`, *gnutls_digest_output_func* `output`, *gnutls_digest_deinit_func* `deinit`, *gnutls_digest_fast_func* `hash_fast`) [Function]

algorithm: is the gnutls digest identifier

priority: is the priority of the algorithm

init: A function which initializes the digest

hash: Perform the hash operation

output: Provide the output of the digest

deinit: A function which deinitializes the digest

hash_fast: Perform the digest operation in one go

This function will register a digest algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

10 Other included programs

Included with GnuTLS are also a few command line tools that let you use the library for common tasks without writing an application. The applications are discussed in this chapter.

10.1 Invoking gnutls-cli

Simple client program to set up a TLS connection to some other computer. It sets up a TLS connection and forwards data from the standard input to the secured socket and vice versa.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `gnutls-cli` program. This software is released under the GNU General Public License, version 3 or later.

gnutls-cli help/usage (`--help`)

This is the automatically generated usage text for gnutls-cli.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
gnutls-cli - GnuTLS client
Usage:  gnutls-cli [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [hostname]

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --tofu                 Enable trust on first use authentication
                                - disabled as '--no-tofu'
       --strict-tofu          Fail to connect if a certificate is unknown or a known certif
changed
                                - disabled as '--no-strict-tofu'
       --dane                 Enable DANE certificate verification (DNSSEC)
                                - disabled as '--no-dane'
       --local-dns            Use the local DNS server for DNSSEC resolving
                                - disabled as '--no-local-dns'
       --ca-verification      Enable CA certificate verification
                                - disabled as '--no-ca-verification'
                                - enabled by default
       --ocsp                 Enable OCSP certificate verification
                                - disabled as '--no-ocsp'
   -r, --resume               Establish a session and resume
       --earlydata=str        Send early data on resumption from the specified file
```

```
   -e, --rehandshake          Establish a session and rehandshake
       --sni-hostname=str     Server's hostname for server name indication extension
       --verify-hostname=str  Server's hostname to use for validation
   -s, --starttls             Connect, establish a plain session and start TLS
       --app-proto=str        an alias for the 'starttls-proto' option
       --starttls-proto=str   The application protocol to be used to obtain the server's ce
(https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)
                                - prohibits the option 'starttls'
   -u, --udp                  Use DTLS (datagram TLS) over UDP
       --mtu=num              Set MTU for datagram TLS
                                - it must be in the range:
                                  0 to 17000
       --crlf                 Send CR LF instead of LF
       --fastopen             Enable TCP Fast Open
       --x509fmtder           Use DER format for certificates to read from
       --print-cert           Print peer's certificate in PEM format
       --save-cert=str        Save the peer's certificate chain in the specified file in PE
       --save-ocsp=str        Save the peer's OCSP status response in the provided file
                                - prohibits the option 'save-ocsp-multi'
       --save-ocsp-multi=str  Save all OCSP responses provided by the peer in this file
                                - prohibits the option 'save-ocsp'
       --save-server-trace=str Save the server-side TLS message trace in the provided file
       --save-client-trace=str Save the client-side TLS message trace in the provided file
       --dh-bits=num          The minimum number of bits allowed for DH
       --priority=str         Priorities string
       --x509cafile=str       Certificate file or PKCS #11 URL to use
       --x509crlfile=file     CRL file to use
                                - file must pre-exist
       --x509keyfile=str      X.509 key file or PKCS #11 URL to use
       --x509certfile=str     X.509 Certificate file or PKCS #11 URL to use
                                - requires the option 'x509keyfile'
       --rawpkkeyfile=str     Private key file (PKCS #8 or PKCS #12) or PKCS #11 URL to use
       --rawpkfile=str        Raw public-key file to use
                                - requires the option 'rawpkkeyfile'
       --srpusername=str      SRP username to use
       --srppasswd=str        SRP password to use
       --pskusername=str      PSK username to use
       --pskkey=str           PSK key (in hex) to use
   -p, --port=str             The port or service to connect to
       --insecure             Don't abort program if server certificate can't be validated
       --verify-allow-broken  Allow broken algorithms, such as MD5 for certificate verifica
       --benchmark-ciphers    Benchmark individual ciphers
       --benchmark-tls-kx     Benchmark TLS key exchange methods
       --benchmark-tls-ciphers  Benchmark TLS ciphers
   -l, --list                 Print a list of the supported algorithms and modes
                                - prohibits the option 'port'
       --priority-list        Print a list of the supported priority strings
```

```
      --noticket              Don't allow session tickets
      --srtp-profiles=str     Offer SRTP profiles
      --alpn=str              Application layer protocol
                                - may appear multiple times
  -b, --heartbeat             Activate heartbeat support
      --recordsize=num        The maximum record size to advertise
                                - it must be in the range:
                                  0 to 4096
      --disable-sni           Do not send a Server Name Indication (SNI)
      --single-key-share      Send a single key share under TLS1.3
      --post-handshake-auth   Enable post-handshake authentication under TLS1.3
      --inline-commands       Inline commands of the form ^<cmd>^
      --inline-commands-prefix=str Change the default delimiter for inline commands.
      --provider=file         Specify the PKCS #11 provider library
                                - file must pre-exist
      --fips140-mode          Reports the status of the FIPS140-2 mode in gnutls library
      --logfile=str           Redirect informational messages to a specific file.
      --keymatexport=str      Label used for exporting keying material
      --keymatexportsize=num Size of the exported keying material
      --waitresumption        Block waiting for the resumption data under TLS1.3
  -v, --version[=arg]         output version information and exit
  -h, --help                  display extended usage information and exit
  -!, --more-help             extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.
Operands and options may be intermixed.  They will be reordered.

Simple client program to set up a TLS connection to some other computer.  It
sets up a TLS connection and forwards data from the standard input to the
secured socket and vice versa.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

tofu option

This is the "enable trust on first use authentication" option.

This option has some usage constraints. It:

- can be disabled with –no-tofu.

This option will, in addition to certificate authentication, perform authentication based on previously seen public keys, a model similar to SSH authentication. Note that when tofu is specified (PKI) and DANE authentication will become advisory to assist the public key acceptance process.

strict-tofu option

This is the “fail to connect if a certificate is unknown or a known certificate has changed” option.

This option has some usage constraints. It:

- can be disabled with –no-strict-tofu.

This option will perform authentication as with option –tofu; however, no questions shall be asked whatsoever, neither to accept an unknown certificate nor a changed one.

dane option

This is the “enable dane certificate verification (dnssec)” option.

This option has some usage constraints. It:

- can be disabled with –no-dane.

This option will, in addition to certificate authentication using the trusted CAs, verify the server certificates using on the DANE information available via DNSSEC.

local-dns option

This is the “use the local dns server for dnssec resolving” option.

This option has some usage constraints. It:

- can be disabled with –no-local-dns.

This option will use the local DNS server for DNSSEC. This is disabled by default due to many servers not allowing DNSSEC.

ca-verification option

This is the “enable ca certificate verification” option.

This option has some usage constraints. It:

- can be disabled with –no-ca-verification.
- It is enabled by default.

This option can be used to enable or disable CA certificate verification. It is to be used with the –dane or –tofu options.

ocsp option

This is the “enable ocsp certificate verification” option.

This option has some usage constraints. It:

- can be disabled with –no-ocsp.

This option will enable verification of the peer’s certificate using ocsp

resume option (-r)

This is the “establish a session and resume” option. Connect, establish a session, reconnect and resume.

rehandshake option (-e)

This is the "establish a session and rehandshake" option. Connect, establish a session and rehandshake immediately.

sni-hostname option

This is the "server's hostname for server name indication extension" option. This option takes a string argument. Set explicitly the server name used in the TLS server name indication extension. That is useful when testing with servers setup on different DNS name than the intended. If not specified, the provided hostname is used. Even with this option server certificate verification still uses the hostname passed on the main commandline. Use –verify-hostname to change this.

verify-hostname option

This is the "server's hostname to use for validation" option. This option takes a string argument. Set explicitly the server name to be used when validating the server's certificate.

starttls option (-s)

This is the "connect, establish a plain session and start tls" option. The TLS session will be initiated when EOF or a SIGALRM is received.

app-proto option

This is an alias for the `starttls-proto` option, see [gnutls-cli starttls-proto], page 301.

starttls-proto option

This is the "the application protocol to be used to obtain the server's certificate (https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)" option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: starttls.

Specify the application layer protocol for STARTTLS. If the protocol is supported, gnutls-cli will proceed to the TLS negotiation.

save-ocsp-multi option

This is the "save all ocsp responses provided by the peer in this file" option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: save-ocsp.

The file will contain a list of PEM encoded OCSP status responses if any were provided by the peer, starting with the one for the peer's server certificate.

dh-bits option

This is the "the minimum number of bits allowed for dh" option. This option takes a number argument. This option sets the minimum number of bits allowed for a Diffie-Hellman key exchange. You may want to lower the default value if the peer sends a weak prime and you get an connection error with unacceptable prime.

priority option

This is the "priorities string" option. This option takes a string argument. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as PERFORMANCE, NORMAL, PFS, SECURE128, SECURE256. The default is NORMAL.

Check the GnuTLS manual on section "Priority strings" for more information on the allowed keywords

rawpkkeyfile option

This is the "private key file (pkcs #8 or pkcs #12) or pkcs #11 url to use" option. This option takes a string argument. In order to instruct the application to negotiate raw public keys one must enable the respective certificate types via the priority strings (i.e. CTYPE-CLI-* and CTYPE-SRV-* flags).

Check the GnuTLS manual on section "Priority strings" for more information on how to set certificate types.

rawpkfile option

This is the "raw public-key file to use" option. This option takes a string argument.

This option has some usage constraints. It:

- must appear in combination with the following options: rawpkkeyfile.

In order to instruct the application to negotiate raw public keys one must enable the respective certificate types via the priority strings (i.e. CTYPE-CLI-* and CTYPE-SRV-* flags).

Check the GnuTLS manual on section "Priority strings" for more information on how to set certificate types.

ranges option

This is the "use length-hiding padding to prevent traffic analysis" option. When possible (e.g., when using CBC ciphersuites), use length-hiding padding to prevent traffic analysis.

NOTE: THIS OPTION IS DEPRECATED

benchmark-ciphers option

This is the "benchmark individual ciphers" option. By default the benchmarked ciphers will utilize any capabilities of the local CPU to improve performance. To test against the raw software implementation set the environment variable GNUTLS_CPUID_OVERRIDE to 0x1.

benchmark-tls-ciphers option

This is the “benchmark tls ciphers” option. By default the benchmarked ciphers will utilize any capabilities of the local CPU to improve performance. To test against the raw software implementation set the environment variable GNUTLS_CPUID_OVERRIDE to 0x1.

list option (-l)

This is the “print a list of the supported algorithms and modes” option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: port.

Print a list of the supported algorithms and modes. If a priority string is given then only the enabled ciphersuites are shown.

priority-list option

This is the “print a list of the supported priority strings” option. Print a list of the supported priority strings. The ciphersuites corresponding to each priority string can be examined using -l -p.

noticket option

This is the “don’t allow session tickets” option. Disable the request of receiving of session tickets under TLS1.2 or earlier

alpn option

This is the “application layer protocol” option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

This option will set and enable the Application Layer Protocol Negotiation (ALPN) in the TLS protocol.

disable-extensions option

This is the “disable all the tls extensions” option. This option disables all TLS extensions. Deprecated option. Use the priority string.

NOTE: THIS OPTION IS DEPRECATED

single-key-share option

This is the “send a single key share under tls1.3” option. This option switches the default mode of sending multiple key shares, to send a single one (the top one).

post-handshake-auth option

This is the “enable post-handshake authentication under tls1.3” option. This option enables post-handshake authentication when under TLS1.3.

inline-commands option

This is the “inline commands of the form ^<cmd>^” option. Enable inline commands of the form ^<cmd>^. The inline commands are expected to be in a line by themselves. The available commands are: resume, rekey1 (local rekey), rekey (rekey on both peers) and renegotiate.

inline-commands-prefix option

This is the “change the default delimiter for inline commands.” option. This option takes a string argument. Change the default delimiter (^) used for inline commands. The delimiter is expected to be a single US-ASCII character (octets 0 - 127). This option is only relevant if inline commands are enabled via the inline-commands option

provider option

This is the “specify the pkcs #11 provider library” option. This option takes a file argument. This will override the default options in /etc/gnutls/pkcs11.conf

logfile option

This is the “redirect informational messages to a specific file.” option. This option takes a string argument. Redirect informational messages to a specific file. The file may be /dev/null also to make the gnutls client quiet to use it in piped server connections where only the server communication may appear on stdout.

waitresumption option

This is the “block waiting for the resumption data under tls1.3” option. This option makes the client to block waiting for the resumption data under TLS1.3. The option has effect only when –resume is provided.

gnutls-cli exit status

One of the following exit values will be returned:

‘0 (EXIT_SUCCESS)’
: Successful program execution.

‘1 (EXIT_FAILURE)’
: The operation failed or the command syntax was not valid.

gnutls-cli See Also

gnutls-cli-debug(1), gnutls-serv(1)

gnutls-cli Examples

Connecting using PSK authentication

To connect to a server using PSK authentication, you need to enable the choice of PSK by using a cipher priority parameter such as in the example below.

```
$ ./gnutls-cli -p 5556 localhost --pskusername psk_identity \
```

```
    --pskkey 88f3824b3e5659f52d00e959bacab954b6540344 \
    --priority NORMAL:-KX-ALL:+ECDHE-PSK:+DHE-PSK:+PSK
Resolving 'localhost'...
Connecting to '127.0.0.1:5556'...
- PSK authentication.
- Version: TLS1.1
- Key Exchange: PSK
- Cipher: AES-128-CBC
- MAC: SHA1
- Compression: NULL
- Handshake was completed

- Simple Client Mode:
```

By keeping the –pskusername parameter and removing the –pskkey parameter, it will query only for the password during the handshake.

Connecting using raw public-key authentication

To connect to a server using raw public-key authentication, you need to enable the option to negotiate raw public-keys via the priority strings such as in the example below.

```
$ ./gnutls-cli -p 5556 localhost --priority NORMAL:-CTYPE-CLI-ALL:+CTYPE-CLI-RAWPK \
    --rawpkkeyfile cli.key.pem \
    --rawpkfile cli.rawpk.pem
Processed 1 client raw public key pair...
Resolving 'localhost'...
Connecting to '127.0.0.1:5556'...
- Successfully sent 1 certificate(s) to server.
- Server has requested a certificate.
- Certificate type: X.509
- Got a certificate list of 1 certificates.
- Certificate[0] info:
 - skipped
- Description: (TLS1.3-Raw Public Key-X.509)-(ECDHE-SECP256R1)-(RSA-PSS-RSAE-SHA256)
- Options:
- Handshake was completed

- Simple Client Mode:
```

Connecting to STARTTLS services

You could also use the client to connect to services with starttls capability.

```
$ gnutls-cli --starttls-proto smtp --port 25 localhost
```

Listing ciphersuites in a priority string

To list the ciphersuites in a priority string:

```
$ ./gnutls-cli --priority SECURE192 -l
Cipher suites for SECURE192
TLS_ECDHE_ECDSA_AES_256_CBC_SHA384         0xc0, 0x24 TLS1.2
```

```
TLS_ECDHE_ECDSA_AES_256_GCM_SHA384         0xc0, 0x2e TLS1.2
TLS_ECDHE_RSA_AES_256_GCM_SHA384           0xc0, 0x30 TLS1.2
TLS_DHE_RSA_AES_256_CBC_SHA256             0x00, 0x6b TLS1.2
TLS_DHE_DSS_AES_256_CBC_SHA256             0x00, 0x6a TLS1.2
TLS_RSA_AES_256_CBC_SHA256                 0x00, 0x3d TLS1.2

Certificate types: CTYPE-X.509
Protocols: VERS-TLS1.2, VERS-TLS1.1, VERS-TLS1.0, VERS-SSL3.0, VERS-DTLS1.0
Compression: COMP-NULL
Elliptic curves: CURVE-SECP384R1, CURVE-SECP521R1
PK-signatures: SIGN-RSA-SHA384, SIGN-ECDSA-SHA384, SIGN-RSA-SHA512, SIGN-ECDSA-SHA51
```

Connecting using a PKCS #11 token

To connect to a server using a certificate and a private key present in a PKCS #11 token you need to substitute the PKCS 11 URLs in the x509certfile and x509keyfile parameters.

Those can be found using "p11tool –list-tokens" and then listing all the objects in the needed token, and using the appropriate.

```
$ p11tool --list-tokens

Token 0:
URL: pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test
Label: Test
Manufacturer: EnterSafe
Model: PKCS15
Serial: 1234

$ p11tool --login --list-certs "pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;t

Object 0:
URL: pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=client;typ
Type: X.509 Certificate
Label: client
ID: 2a:97:0d:58:d1:51:3c:23:07:ae:4e:0d:72:26:03:7d:99:06:02:6a

$ MYCERT="pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=clien
$ MYKEY="pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=client
$ export MYCERT MYKEY

$ gnutls-cli www.example.com --x509keyfile $MYKEY --x509certfile $MYCERT
```

Notice that the private key only differs from the certificate in the type.

10.2 Invoking gnutls-serv

Server program that listens to incoming TLS connections.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `gnutls-serv` program. This software is released under the GNU General Public License, version 3 or later.

gnutls-serv help/usage (`--help`)

This is the automatically generated usage text for gnutls-serv.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
gnutls-serv - GnuTLS server
Usage:  gnutls-serv [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
       --sni-hostname=str     Server's hostname for server name extension
       --sni-hostname-fatal   Send fatal alert on sni-hostname mismatch
       --alpn=str             Specify ALPN protocol to be enabled by the server
                                - may appear multiple times
       --alpn-fatal           Send fatal alert on non-matching ALPN name
       --noticket             Don't accept session tickets
       --earlydata            Accept early data
       --maxearlydata=num     The maximum early data size to accept
                                - it must be in the range:
                                  greater than or equal to 1
       --nocookie             Don't require cookie on DTLS sessions
   -g, --generate             Generate Diffie-Hellman parameters
   -q, --quiet                Suppress some messages
       --nodb                 Do not use a resumption database
       --http                 Act as an HTTP server
       --echo                 Act as an Echo server
   -u, --udp                  Use DTLS (datagram TLS) over UDP
       --mtu=num              Set MTU for datagram TLS
                                - it must be in the range:
                                  0 to 17000
       --srtp-profiles=str    Offer SRTP profiles
   -a, --disable-client-cert  Do not request a client certificate
                                - prohibits the option 'require-client-cert'
   -r, --require-client-cert  Require a client certificate
       --verify-client-cert   If a client certificate is sent then verify it.
   -b, --heartbeat            Activate heartbeat support
       --x509fmtder           Use DER format for certificates to read from
       --priority=str         Priorities string
       --dhparams=file        DH params file to use
```

```
                              - file must pre-exist
      --x509cafile=str        Certificate file or PKCS #11 URL to use
      --x509crlfile=file      CRL file to use
                              - file must pre-exist
      --x509keyfile=str       X.509 key file or PKCS #11 URL to use
                              - may appear multiple times
      --x509certfile=str      X.509 Certificate file or PKCS #11 URL to use
                              - may appear multiple times
      --rawpkkeyfile=str      Private key file (PKCS #8 or PKCS #12) or PKCS #11 URL to use
                              - may appear multiple times
      --rawpkfile=str         Raw public-key file to use
                              - requires the option 'rawpkkeyfile'
                              - may appear multiple times
      --srppasswd=file        SRP password file to use
                              - file must pre-exist
      --srppasswdconf=file    SRP password configuration file to use
                              - file must pre-exist
      --pskpasswd=file        PSK password file to use
                              - file must pre-exist
      --pskhint=str           PSK identity hint to use
      --ocsp-response=str     The OCSP response to send to client
                              - may appear multiple times
      --ignore-ocsp-response-errors  Ignore any errors when setting the OCSP response
  -p, --port=num              The port to connect to
  -l, --list                  Print a list of the supported algorithms and modes
      --provider=file         Specify the PKCS #11 provider library
                              - file must pre-exist
      --keymatexport=str      Label used for exporting keying material
      --keymatexportsize=num Size of the exported keying material
      --recordsize=num        The maximum record size to advertise
                              - it must be in the range:
                                0 to 16384
      --httpdata=file         The data used as HTTP response
                              - file must pre-exist
  -v, --version[=arg]         output version information and exit
  -h, --help                  display extended usage information and exit
  -!, --more-help             extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.

Server program that listens to incoming TLS connections.
```

debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

sni-hostname option

This is the “server’s hostname for server name extension” option. This option takes a string argument. Server name of type host_name that the server will recognise as its own. If the server receives client hello with different name, it will send a warning-level unrecognized_name alert.

alpn option

This is the “specify alpn protocol to be enabled by the server” option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Specify the (textual) ALPN protocol for the server to use.

require-client-cert option (-r)

This is the “require a client certificate” option. This option before 3.6.0 used to imply –verify-client-cert. Since 3.6.0 it will no longer verify the certificate by default.

verify-client-cert option

This is the “if a client certificate is sent then verify it.” option. Do not require, but if a client certificate is sent then verify it and close the connection if invalid.

heartbeat option (-b)

This is the “activate heartbeat support” option. Regularly ping client via heartbeat extension messages

priority option

This is the “priorities string” option. This option takes a string argument. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as PERFORMANCE, NORMAL, SECURE128, SECURE256. The default is NORMAL.

Check the GnuTLS manual on section “Priority strings” for more information on allowed keywords

x509keyfile option

This is the “x.509 key file or pkcs #11 url to use” option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Specify the private key file or URI to use; it must correspond to the certificate specified in –x509certfile. Multiple keys and certificates can be specified with this option and in that case each occurrence of keyfile must be followed by the corresponding x509certfile or vice-versa.

x509certfile option

This is the "x.509 certificate file or pkcs #11 url to use" option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Specify the certificate file or URI to use; it must correspond to the key specified in –x509keyfile. Multiple keys and certificates can be specified with this option and in that case each occurrence of keyfile must be followed by the corresponding x509certfile or vice-versa.

x509dsakeyfile option

This is an alias for the `x509keyfile` option, see [gnutls-serv x509keyfile], page 309.

x509dsacertfile option

This is an alias for the `x509certfile` option, see [gnutls-serv x509certfile], page 310.

x509ecckeyfile option

This is an alias for the `x509keyfile` option, see [gnutls-serv x509keyfile], page 309.

x509ecccertfile option

This is an alias for the `x509certfile` option, see [gnutls-serv x509certfile], page 310.

rawpkkeyfile option

This is the "private key file (pkcs #8 or pkcs #12) or pkcs #11 url to use" option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

Specify the private key file or URI to use; it must correspond to the raw public-key specified in –rawpkfile. Multiple key pairs can be specified with this option and in that case each occurrence of keyfile must be followed by the corresponding rawpkfile or vice-versa.

In order to instruct the application to negotiate raw public keys one must enable the respective certificate types via the priority strings (i.e. CTYPE-CLI-* and CTYPE-SRV-* flags).

Check the GnuTLS manual on section "Priority strings" for more information on how to set certificate types.

rawpkfile option

This is the "raw public-key file to use" option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.
- must appear in combination with the following options: rawpkkeyfile.

Specify the raw public-key file to use; it must correspond to the private key specified in –rawpkkeyfile. Multiple key pairs can be specified with this option and in that case each occurrence of keyfile must be followed by the corresponding rawpkfile or vice-versa.

In order to instruct the application to negotiate raw public keys one must enable the respective certificate types via the priority strings (i.e. CTYPE-CLI-* and CTYPE-SRV-* flags).

Check the GnuTLS manual on section "Priority strings" for more information on how to set certificate types.

ocsp-response option

This is the "the ocsp response to send to client" option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

If the client requested an OCSP response, return data from this file to the client.

ignore-ocsp-response-errors option

This is the "ignore any errors when setting the ocsp response" option. That option instructs gnutls to not attempt to match the provided OCSP responses with the certificates.

list option (-l)

This is the "print a list of the supported algorithms and modes" option. Print a list of the supported algorithms and modes. If a priority string is given then only the enabled ciphersuites are shown.

provider option

This is the "specify the pkcs #11 provider library" option. This option takes a file argument. This will override the default options in /etc/gnutls/pkcs11.conf

gnutls-serv exit status

One of the following exit values will be returned:

'`0 (EXIT_SUCCESS)`'
: Successful program execution.

'`1 (EXIT_FAILURE)`'
: The operation failed or the command syntax was not valid.

gnutls-serv See Also

gnutls-cli-debug(1), gnutls-cli(1)

gnutls-serv Examples

Running your own TLS server based on GnuTLS can be useful when debugging clients and/or GnuTLS itself. This section describes how to use `gnutls-serv` as a simple HTTPS server.

The most basic server can be started as:

```
gnutls-serv --http --priority "NORMAL:+ANON-ECDH:+ANON-DH"
```

It will only support anonymous ciphersuites, which many TLS clients refuse to use.

The next step is to add support for X.509. First we generate a CA:

```
$ certtool --generate-privkey > x509-ca-key.pem
$ echo 'cn = GnuTLS test CA' > ca.tmpl
$ echo 'ca' >> ca.tmpl
$ echo 'cert_signing_key' >> ca.tmpl
$ certtool --generate-self-signed --load-privkey x509-ca-key.pem \
  --template ca.tmpl --outfile x509-ca.pem
```

Then generate a server certificate. Remember to change the dns_name value to the name of your server host, or skip that command to avoid the field.

```
$ certtool --generate-privkey > x509-server-key.pem
$ echo 'organization = GnuTLS test server' > server.tmpl
$ echo 'cn = test.gnutls.org' >> server.tmpl
$ echo 'tls_www_server' >> server.tmpl
$ echo 'encryption_key' >> server.tmpl
$ echo 'signing_key' >> server.tmpl
$ echo 'dns_name = test.gnutls.org' >> server.tmpl
$ certtool --generate-certificate --load-privkey x509-server-key.pem \
  --load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
  --template server.tmpl --outfile x509-server.pem
```

For use in the client, you may want to generate a client certificate as well.

```
$ certtool --generate-privkey > x509-client-key.pem
$ echo 'cn = GnuTLS test client' > client.tmpl
$ echo 'tls_www_client' >> client.tmpl
$ echo 'encryption_key' >> client.tmpl
$ echo 'signing_key' >> client.tmpl
$ certtool --generate-certificate --load-privkey x509-client-key.pem \
  --load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
  --template client.tmpl --outfile x509-client.pem
```

To be able to import the client key/certificate into some applications, you will need to convert them into a PKCS#12 structure. This also encrypts the security sensitive key with a password.

```
$ certtool --to-p12 --load-ca-certificate x509-ca.pem \
  --load-privkey x509-client-key.pem --load-certificate x509-client.pem \
  --outder --outfile x509-client.p12
```

For icing, we'll create a proxy certificate for the client too.

```
$ certtool --generate-privkey > x509-proxy-key.pem
$ echo 'cn = GnuTLS test client proxy' > proxy.tmpl
```

```
$ certtool --generate-proxy --load-privkey x509-proxy-key.pem \
  --load-ca-certificate x509-client.pem --load-ca-privkey x509-client-key.pem \
  --load-certificate x509-client.pem --template proxy.tmpl \
  --outfile x509-proxy.pem
```

Then start the server again:

```
$ gnutls-serv --http \
            --x509cafile x509-ca.pem \
            --x509keyfile x509-server-key.pem \
            --x509certfile x509-server.pem
```

Try connecting to the server using your web browser. Note that the server listens to port 5556 by default.

While you are at it, to allow connections using ECDSA, you can also create a ECDSA key and certificate for the server. These credentials will be used in the final example below.

```
$ certtool --generate-privkey --ecdsa > x509-server-key-ecc.pem
$ certtool --generate-certificate --load-privkey x509-server-key-ecc.pem \
  --load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
  --template server.tmpl --outfile x509-server-ecc.pem
```

The next step is to add support for SRP authentication. This requires an SRP password file created with `srptool`. To start the server with SRP support:

```
gnutls-serv --http --priority NORMAL:+SRP-RSA:+SRP \
            --srppasswdconf srp-tpasswd.conf \
            --srppasswd srp-passwd.txt
```

Let's also start a server with support for PSK. This would require a password file created with `psktool`.

```
gnutls-serv --http --priority NORMAL:+ECDHE-PSK:+PSK \
            --pskpasswd psk-passwd.txt
```

If you want a server with support for raw public-keys we can also add these credentials. Note however that there is no identity information linked to these keys as is the case with regular x509 certificates. Authentication must be done via different means. Also we need to explicitly enable raw public-key certificates via the priority strings.

```
gnutls-serv --http --priority NORMAL:+CTYPE-CLI-RAWPK:+CTYPE-SRV-RAWPK \
            --rawpkfile srv.rawpk.pem \
            --rawpkkeyfile srv.key.pem
```

Finally, we start the server with all the earlier parameters and you get this command:

```
gnutls-serv --http --priority NORMAL:+PSK:+SRP:+CTYPE-CLI-RAWPK:+CTYPE-SRV-RAWPK \
            --x509cafile x509-ca.pem \
            --x509keyfile x509-server-key.pem \
            --x509certfile x509-server.pem \
            --x509keyfile x509-server-key-ecc.pem \
            --x509certfile x509-server-ecc.pem \
            --srppasswdconf srp-tpasswd.conf \
            --srppasswd srp-passwd.txt \
            --pskpasswd psk-passwd.txt \
            --rawpkfile srv.rawpk.pem \
            --rawpkkeyfile srv.key.pem
```

10.3 Invoking gnutls-cli-debug

TLS debug client. It sets up multiple TLS connections to a server and queries its capabilities. It was created to assist in debugging GnuTLS, but it might be useful to extract a TLS server's capabilities. It connects to a TLS server, performs tests and print the server's capabilities. If called with the '-V' parameter more checks will be performed. Can be used to check for servers with special needs or bugs.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `gnutls-cli-debug` program. This software is released under the GNU General Public License, version 3 or later.

gnutls-cli-debug help/usage (`--help`)

This is the automatically generated usage text for gnutls-cli-debug.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
gnutls-cli-debug - GnuTLS debug client
Usage:  gnutls-cli-debug [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
   -p, --port=num             The port to connect to
                                - it must be in the range:
                                  0 to 65536
       --app-proto=str        an alias for the 'starttls-proto' option
       --starttls-proto=str   The application protocol to be used to obtain the server's ce
(https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)
   -v, --version[=arg]        output version information and exit
   -h, --help                 display extended usage information and exit
   -!, --more-help            extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.
Operands and options may be intermixed.  They will be reordered.

TLS debug client.  It sets up multiple TLS connections to a server and
queries its capabilities.  It was created to assist in debugging GnuTLS,
but it might be useful to extract a TLS server's capabilities.  It connects
to a TLS server, performs tests and print the server's capabilities.  If
called with the '-V' parameter more checks will be performed.  Can be used
to check for servers with special needs or bugs.
```

debug option (-d)

This is the "enable debugging" option. This option takes a number argument. Specifies the debug level.

app-proto option

This is an alias for the `starttls-proto` option, see [gnutls-cli-debug starttls-proto], page 315.

starttls-proto option

This is the "the application protocol to be used to obtain the server's certificate (https, ftp, smtp, imap, ldap, xmpp, lmtp, pop3, nntp, sieve, postgres)" option. This option takes a string argument. Specify the application layer protocol for STARTTLS. If the protocol is supported, gnutls-cli will proceed to the TLS negotiation.

gnutls-cli-debug exit status

One of the following exit values will be returned:

'`0 (EXIT_SUCCESS)`'
: Successful program execution.

'`1 (EXIT_FAILURE)`'
: The operation failed or the command syntax was not valid.

gnutls-cli-debug See Also

gnutls-cli(1), gnutls-serv(1)

gnutls-cli-debug Examples

```
$ gnutls-cli-debug localhost
GnuTLS debug client 3.5.0
Checking localhost:443
                              for SSL 3.0 (RFC6101) support... yes
                         whether we need to disable TLS 1.2... no
                         whether we need to disable TLS 1.1... no
                         whether we need to disable TLS 1.0... no
                         whether %NO_EXTENSIONS is required... no
                           whether %COMPAT is required... no
                         for TLS 1.0 (RFC2246) support... yes
                         for TLS 1.1 (RFC4346) support... yes
                         for TLS 1.2 (RFC5246) support... yes
                              fallback from TLS 1.6 to... TLS1.2
                         for RFC7507 inappropriate fallback... yes
                                    for HTTPS server name... Local
                                for certificate chain order... sorted
                    for safe renegotiation (RFC5746) support... yes
                       for Safe renegotiation support (SCSV)... no
                      for encrypt-then-MAC (RFC7366) support... no
```

```
                    for ext master secret (RFC7627) support... no
                            for heartbeat (RFC6520) support... no
                        for version rollback bug in RSA PMS... dunno
                     for version rollback bug in Client Hello... no
               whether the server ignores the RSA PMS version... yes
whether small records (512 bytes) are tolerated on handshake... yes
    whether cipher suites not in SSL 3.0 spec are accepted... yes
whether a bogus TLS record version in the client hello is accepted... yes
         whether the server understands TLS closure alerts... partially
            whether the server supports session resumption... yes
                      for anonymous authentication support... no
                      for ephemeral Diffie-Hellman support... no
                   for ephemeral EC Diffie-Hellman support... yes
                     ephemeral EC Diffie-Hellman group info... SECP256R1
                    for AES-128-GCM cipher (RFC5288) support... yes
                    for AES-128-CCM cipher (RFC6655) support... no
                 for AES-128-CCM-8 cipher (RFC6655) support... no
                    for AES-128-CBC cipher (RFC3268) support... yes
                for CAMELLIA-128-GCM cipher (RFC6367) support... no
                for CAMELLIA-128-CBC cipher (RFC5932) support... no
                        for 3DES-CBC cipher (RFC2246) support... yes
                    for ARCFOUR 128 cipher (RFC2246) support... yes
                                       for MD5 MAC support... yes
                                      for SHA1 MAC support... yes
                                   for SHA256 MAC support... yes
                               for ZLIB compression support... no
                          for max record size (RFC6066) support... no
                     for OCSP status response (RFC6066) support... no
                   for OpenPGP authentication (RFC6091) support... no
```

You could also use the client to debug services with starttls capability.

```
$ gnutls-cli-debug --starttls-proto smtp --port 25 localhost
```

11 Internal Architecture of GnuTLS

This chapter is to give a brief description of the way GnuTLS works. The focus is to give an idea to potential developers and those who want to know what happens inside the black box.

11.1 The TLS Protocol

The main use case for the TLS protocol is shown in Figure 11.1. A user of a library implementing the protocol expects no less than this functionality, i.e., to be able to set parameters such as the accepted security level, perform a negotiation with the peer and be able to exchange data.

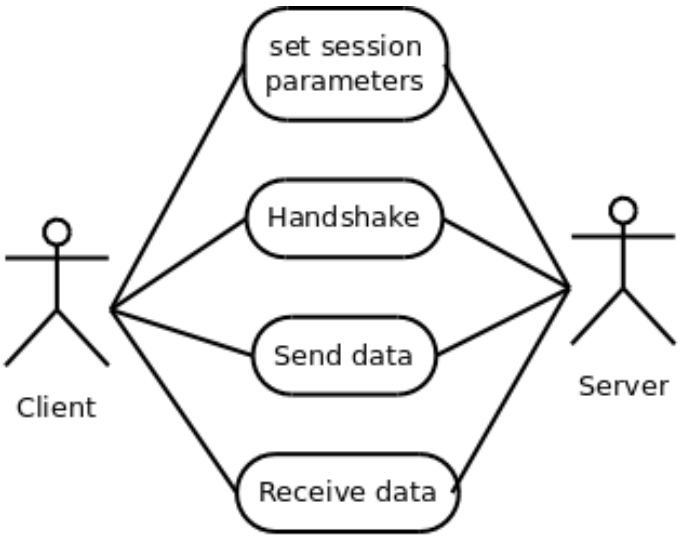

Figure 11.1: TLS protocol use case.

11.2 TLS Handshake Protocol

The GnuTLS handshake protocol is implemented as a state machine that waits for input or returns immediately when the non-blocking transport layer functions are used. The main idea is shown in Figure 11.2.

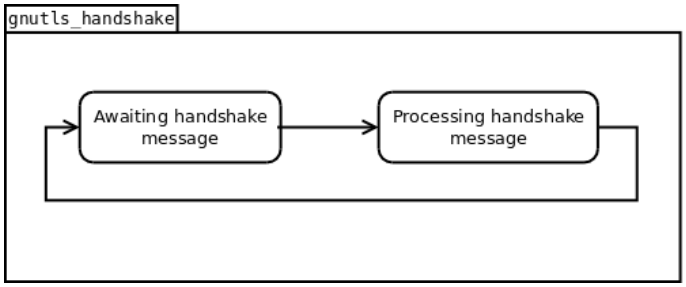

Figure 11.2: GnuTLS handshake state machine.

Also the way the input is processed varies per ciphersuite. Several implementations of the internal handlers are available and [gnutls_handshake], page 414 only multiplexes the input to the appropriate handler. For example a PSK ciphersuite has a different implementation of the `process_client_key_exchange` than a certificate ciphersuite. We illustrate the idea in Figure 11.3.

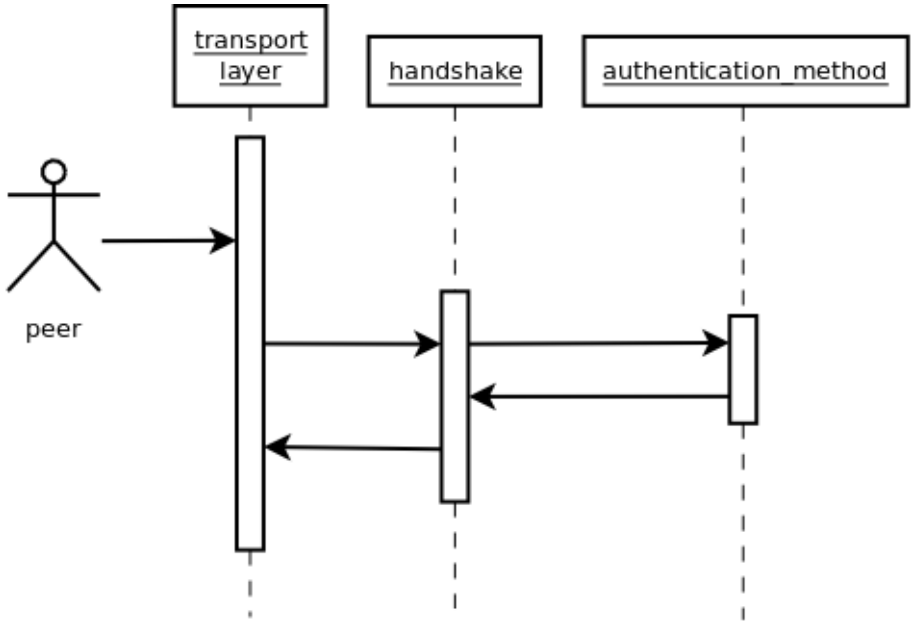

Figure 11.3: GnuTLS handshake process sequence.

11.3 TLS Authentication Methods

In GnuTLS authentication methods can be implemented quite easily. Since the required changes to add a new authentication method affect only the handshake protocol, a simple interface is used. An authentication method needs to implement the functions shown below.

```
typedef struct
{
```

```
  const char *name;
  int (*gnutls_generate_server_certificate) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_certificate) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_server_kx) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_kx) (gnutls_session_t, gnutls_buffer_st*);
  int (*gnutls_generate_client_cert_vrfy) (gnutls_session_t, gnutls_buffer_st *);
  int (*gnutls_generate_server_certificate_request) (gnutls_session_t,
                                                     gnutls_buffer_st *);

  int (*gnutls_process_server_certificate) (gnutls_session_t, opaque *,
                                            size_t);
  int (*gnutls_process_client_certificate) (gnutls_session_t, opaque *,
                                            size_t);
  int (*gnutls_process_server_kx) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_client_kx) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_client_cert_vrfy) (gnutls_session_t, opaque *, size_t);
  int (*gnutls_process_server_certificate_request) (gnutls_session_t,
                                                    opaque *, size_t);
} mod_auth_st;
```

Those functions are responsible for the interpretation of the handshake protocol messages. It is common for such functions to read data from one or more `credentials_t` structures[1] and write data, such as certificates, usernames etc. to `auth_info_t` structures.

Simple examples of existing authentication methods can be seen in `auth/psk.c` for PSK ciphersuites and `auth/srp.c` for SRP ciphersuites. After implementing these functions the structure holding its pointers has to be registered in `gnutls_algorithms.c` in the `_gnutls_kx_algorithms` structure.

11.4 TLS Extension Handling

As with authentication methods, adding TLS hello extensions can be done quite easily by implementing the interface shown below.

```
typedef int (*gnutls_ext_recv_func) (gnutls_session_t session,
                                     const unsigned char *data, size_t len);
typedef int (*gnutls_ext_send_func) (gnutls_session_t session,
                                     gnutls_buffer_st *extdata);
```

Here there are two main functions, one for parsing the received extension data and one for formatting the extension data that must be send. These functions have to check internally whether they operate within a client or a server session.

A simple example of an extension handler can be seen in `lib/ext/srp.c` in GnuTLS' source code. After implementing these functions, the extension has to be registered. Registering an extension can be done in two ways. You can create a GnuTLS internal extension and register it in `hello_ext.c` or write an external extension (not inside GnuTLS but inside an application using GnuTLS) and register it via the exported functions [gnutls_session_ext_register], page 459 or [gnutls_ext_register], page 409.

[1] such as the `gnutls_certificate_credentials_t` structures

Adding a new TLS hello extension

Adding support for a new TLS hello extension is done from time to time, and the process to do so is not difficult. Here are the steps you need to follow if you wish to do this yourself. For the sake of discussion, let's consider adding support for the hypothetical TLS extension `foobar`. The following section is about adding an hello extension to GnuTLS itself. For custom application extensions you should check the exported functions [gnutls_session_ext_register], page 459 or [gnutls_ext_register], page 409.

Add `configure` option like `--enable-foobar` or `--disable-foobar`.

This step is useful when the extension code is large and it might be desirable under some circumstances to be able to leave out the extension during compilation of GnuTLS. If you don't need this kind of feature this step can be safely skipped.

Whether to choose enable or disable depends on whether you intend to make the extension be enabled by default. Look at existing checks (i.e., SRP, authz) for how to model the code. For example:

```
AC_MSG_CHECKING([whether to disable foobar support])
AC_ARG_ENABLE(foobar,
AS_HELP_STRING([--disable-foobar],
[disable foobar support]),
ac_enable_foobar=no)
if test x$ac_enable_foobar != xno; then
 AC_MSG_RESULT(no)
 AC_DEFINE(ENABLE_FOOBAR, 1, [enable foobar])
else
 ac_full=0
 AC_MSG_RESULT(yes)
fi
AM_CONDITIONAL(ENABLE_FOOBAR, test "$ac_enable_foobar" != "no")
```

These lines should go in `lib/m4/hooks.m4`.

Add an extension identifier to `extensions_t` in `gnutls_int.h`.

A good name for the identifier would be GNUTLS_EXTENSION_FOOBAR. If the extension that you are implementing is an extension that is officially registered by IANA then it is recommended to use its official name such that the extension can be correctly identified by other developers. Check with `https://www.iana.org/assignments/tls-extensiontype-values` for registered extensions.

Register the extension in `lib/hello_ext.c`.

In order for the extension to be executed you need to register it in the `static hello_ext_entry_st const *extfunc[]` list in `lib/hello_ext.c`.

A typical entry would be:

```
#ifdef ENABLE_FOOBAR
[GNUTLS_EXTENSION_FOOBAR] = &ext_mod_foobar,
#endif
```

Also for every extension you need to create an `hello_ext_entry_st` that describes the extension. This structure is placed in the designated c file for your extension and its name is used in the registration entry as depicted above.

The structure of `hello_ext_entry_st` is as follows:

```
const hello_ext_entry_st ext_mod_foobar = {
  .name = "FOOBAR",
  .tls_id = 255,
  .gid = GNUTLS_EXTENSION_FOOBAR,
  .parse_type = GNUTLS_EXT_TLS,
  .validity = GNUTLS_EXT_FLAG_CLIENT_HELLO |
GNUTLS_EXT_FLAG_TLS12_SERVER_HELLO |
GNUTLS_EXT_FLAG_TLS13_SERVER_HELLO |
GNUTLS_EXT_FLAG_TLS,
    .recv_func = _gnutls_foobar_recv_params,
    .send_func = _gnutls_foobar_send_params,
    .pack_func = _gnutls_foobar_pack,
    .unpack_func = _gnutls_foobar_unpack,
    .deinit_func = _gnutls_foobar_deinit,
    .cannot_be_overriden = 1
};
```

The GNUTLS_EXTENSION_FOOBAR is the identifier that you've added to `gnutls_int.h` earlier. The `.tls_id` should contain the number that IANA has assigned to this extension, or an unassigned number of your choice if this is an unregistered extension. In the rest of this structure you specify the functions to handle the extension data. The `receive` function will be called upon reception of the data and will be used to parse or interpret the extension data. The `send` function will be called prior to sending the extension data on the wire and will be used to format the data such that it can be send over the wire. The `pack` and `unpack` functions will be used to prepare the data for storage in case of session resumption (and vice versa). The `deinit` function will be called to deinitialize the extension's private parameters, if any.

Look at `gnutls_ext_parse_type_t` and `gnutls_ext_flags_t` for a complete list of available flags.

Note that the conditional `ENABLE_FOOBAR` definition should only be used if step 1 with the `configure` options has taken place.

Add new files that implement the hello extension.

To keep things structured every extension should have its own files. The functions that you should (at least) add are those referenced in the struct from the previous step. Use descriptive file names such as `lib/ext/foobar.c` and for the corresponding header `lib/ext/foobar.h`. As a starter, you could add this:

```
int
_gnutls_foobar_recv_params (gnutls_session_t session, const uint8_t * data,
                    size_t data_size)
{
  return 0;
```

```
}

int
_gnutls_foobar_send_params (gnutls_session_t session, gnutls_buffer_st* data)
{
  return 0;
}

int
_gnutls_foobar_pack (extension_priv_data_t epriv, gnutls_buffer_st * ps)
{
   /* Append the extension's internal state to buffer */
   return 0;
}

int
_gnutls_foobar_unpack (gnutls_buffer_st * ps, extension_priv_data_t * epriv)
{
   /* Read the internal state from buffer */
   return 0;
}
```

The `_gnutls_foobar_recv_params` function is responsible for parsing incoming extension data (both in the client and server).

The `_gnutls_foobar_send_params` function is responsible for formatting extension data such that it can be send over the wire (both in the client and server). It should append data to provided buffer and return a positive (or zero) number on success or a negative error code. Previous to 3.6.0 versions of GnuTLS required that function to return the number of bytes that were written. If zero is returned and no bytes are appended the extension will not be sent. If a zero byte extension is to be sent this function must return `GNUTLS_E_INT_RET_0`.

If you receive length fields that don't match, return `GNUTLS_E_UNEXPECTED_PACKET_LENGTH`. If you receive invalid data, return `GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER`. You can use other error codes from the list in Appendix C [Error codes], page 341. Return 0 on success.

An extension typically stores private information in the `session` data for later usage. That can be done using the functions `_gnutls_hello_ext_set_datum` and `_gnutls_hello_ext_get_datum`. You can check simple examples at `lib/ext/max_record.c` and `lib/ext/server_name.c` extensions. That private information can be saved and restored across session resumption if the following functions are set:

The `_gnutls_foobar_pack` function is responsible for packing internal extension data to save them in the session resumption storage.

The `_gnutls_foobar_unpack` function is responsible for restoring session data from the session resumption storage.

When the internal data is stored using the `_gnutls_hello_ext_set_datum`, then you can rely on the default pack and unpack functions: `_gnutls_hello_ext_default_pack` and `_gnutls_hello_ext_default_unpack`.

Recall that both for the client and server, the send and receive functions most likely will need to do different things depending on which mode they are in. It may be useful to make this distinction explicit in the code. Thus, for example, a better template than above would be:

```
int
_gnutls_foobar_recv_params (gnutls_session_t session,
                            const uint8_t * data,
                            size_t data_size)
{
  if (session->security_parameters.entity == GNUTLS_CLIENT)
    return foobar_recv_client (session, data, data_size);
  else
    return foobar_recv_server (session, data, data_size);
}

int
_gnutls_foobar_send_params (gnutls_session_t session,
                            gnutls_buffer_st * data)
{
  if (session->security_parameters.entity == GNUTLS_CLIENT)
    return foobar_send_client (session, data);
  else
    return foobar_send_server (session, data);
}
```

The functions used would be declared as `static` functions, of the appropriate prototype, in the same file.

When adding the new extension files, you'll need to add them to `lib/ext/Makefile.am` as well, for example:

```
if ENABLE_FOOBAR
libgnutls_ext_la_SOURCES += ext/foobar.c ext/foobar.h
endif
```

Add API functions to use the extension.

It might be desirable to allow users of the extension to request the use of the extension, or set extension specific data. This can be implemented by adding extension specific function calls that can be added to `includes/gnutls/gnutls.h`, as long as the LGPLv2.1+ applies. The implementation of these functions should lie in the `lib/ext/foobar.c` file.

To make the API available in the shared library you need to add the added symbols in `lib/libgnutls.map`, so that the symbols are exported properly.

When writing GTK-DOC style documentation for your new APIs, don't forget to add `Since:` tags to indicate the GnuTLS version the API was introduced in.

Adding a new Supplemental Data Handshake Message

TLS handshake extensions allow to send so called supplemental data handshake messages [[RFC4680], page 750]. This short section explains how to implement a supplemental data handshake message for a given TLS extension.

First of all, modify your extension `foobar` in the way, to instruct the handshake process to send and receive supplemental data, as shown below.

```
int
_gnutls_foobar_recv_params (gnutls_session_t session, const opaque * data,
                                    size_t _data_size)
{
   ...
   gnutls_supplemental_recv(session, 1);
   ...
}

int
_gnutls_foobar_send_params (gnutls_session_t session, gnutls_buffer_st *extdata)
{
   ...
   gnutls_supplemental_send(session, 1);
   ...
}
```

Furthermore you'll need two new functions `_foobar_supp_recv_params` and `_foobar_supp_send_params`, which must conform to the following prototypes.

```
typedef int (*gnutls_supp_recv_func)(gnutls_session_t session,
                                     const unsigned char *data,
                                     size_t data_size);
typedef int (*gnutls_supp_send_func)(gnutls_session_t session,
                                     gnutls_buffer_t buf);
```

The following example code shows how to send a "Hello World" string in the supplemental data handshake message.

```
int
_foobar_supp_recv_params(gnutls_session_t session, const opaque *data, size_t _data_
{
   uint8_t len = _data_size;
   unsigned char *msg;

   msg = gnutls_malloc(len);
   if (msg == NULL) return GNUTLS_E_MEMORY_ERROR;

   memcpy(msg, data, len);
   msg[len]='\0';

   /* do something with msg */
   gnutls_free(msg);

   return len;
}

int
```

```
_foobar_supp_send_params(gnutls_session_t session, gnutls_buffer_t buf)
{
   unsigned char *msg = "hello world";
   int len = strlen(msg);

   if (gnutls_buffer_append_data(buf, msg, len) < 0)
       abort();

   return len;
}
```

Afterwards, register the new supplemental data using [gnutls_session_supplemental_register], page 467, or [gnutls_supplemental_register], page 482 at some point in your program.

11.5 Cryptographic Backend

Today most new processors, either for embedded or desktop systems include either instructions intended to speed up cryptographic operations, or a co-processor with cryptographic capabilities. Taking advantage of those is a challenging task for every cryptographic application or library. GnuTLS handles the cryptographic provider in a modular way, following a layered approach to access cryptographic operations as in Figure 11.4.

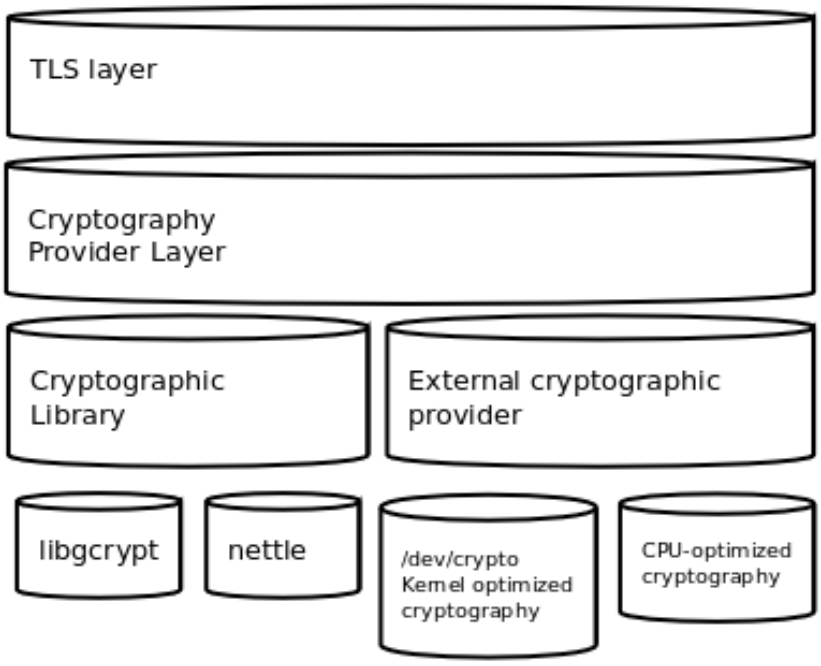

Figure 11.4: GnuTLS cryptographic back-end design.

The TLS layer uses a cryptographic provider layer, that will in turn either use the default crypto provider – a software crypto library, or use an external crypto provider, if available in the local system. The reason of handling the external cryptographic provider in GnuTLS and not delegating it to the cryptographic libraries, is that none of the supported cryptographic libraries support `/dev/crypto` or CPU-optimized cryptography in an efficient way.

Cryptographic library layer

The Cryptographic library layer, currently supports only libnettle. Older versions of GnuTLS used to support libgcrypt, but it was switched with nettle mainly for performance reasons[2] and secondary because it is a simpler library to use. In the future other cryptographic libraries might be supported as well.

External cryptography provider

Systems that include a cryptographic co-processor, typically come with kernel drivers to utilize the operations from software. For this reason GnuTLS provides a layer where each individual algorithm used can be replaced by another implementation, i.e., the one provided by the driver. The FreeBSD, OpenBSD and Linux kernels[3] include already a number of hardware assisted implementations, and also provide an interface to access them, called `/dev/crypto`. GnuTLS will take advantage of this interface if compiled with special options. That is because in most systems where hardware-assisted cryptographic operations are not available, using this interface might actually harm performance.

In systems that include cryptographic instructions with the CPU's instructions set, using the kernel interface will introduce an unneeded layer. For this reason GnuTLS includes such optimizations found in popular processors such as the AES-NI or VIA PADLOCK instruction sets. This is achieved using a mechanism that detects CPU capabilities and overrides parts of crypto back-end at runtime. The next section discusses the registration of a detected algorithm optimization. For more information please consult the GnuTLS source code in `lib/accelerated/`.

Overriding specific algorithms

When an optimized implementation of a single algorithm is available, say a hardware assisted version of AES-CBC then the following functions, from `crypto.h`, can be used to register those algorithms.

- [gnutls_crypto_register_cipher], page 728: To register a cipher algorithm.
- [gnutls_crypto_register_aead_cipher], page 727: To register an AEAD cipher algorithm.
- [gnutls_crypto_register_mac], page 729: To register a MAC algorithm.
- [gnutls_crypto_register_digest], page 729: To register a hash algorithm.

Those registration functions will only replace the specified algorithm and leave the rest of subsystem intact.

---

[2] See `https://lists.gnu.org/archive/html/gnutls-devel/2011-02/msg00079.html`.

[3] Check `https://home.gna.org/cryptodev-linux/` for the Linux kernel implementation of `/dev/crypto`.

Protecting keys through isolation

For asymmetric or public keys, GnuTLS supports PKCS #11 which allows operation without access to long term keys, in addition to CPU offloading. For more information see Chapter 5 [Hardware security modules and abstract key types], page 100.

11.6 Random Number Generators

About the generators

GnuTLS provides two random generators. The default, and the AES-DRBG random generator which is only used when the library is compiled with support for FIPS140-2 and the system is in FIPS140-2 mode.

The default generator - inner workings

The random number generator levels in `gnutls_rnd_level_t` map to two CHACHA-based random generators which are initially seeded using the OS random device, e.g., `/dev/urandom` or `getrandom()`. These random generators are unique per thread, and are automatically re-seeded when a fork is detected.

The reason the CHACHA cipher was selected for the GnuTLS' PRNG is the fact that CHACHA is considered a secure and fast stream cipher, and is already defined for use in TLS protocol. As such, the utilization of it would not stress the CPU caches, and would allow for better performance on busy servers, irrespective of their architecture (e.g., even if AES is not available with an optimized instruction set).

The generators are unique per thread to allow lock-free operation. That induces a cost of around 140-bytes for the state of the generators per thread, on threads that would utilize [gnutls_rnd], page 737. At the same time it allows fast and lock-free access to the generators. The lock-free access benefits servers which utilize more than 4 threads, while imposes no cost on single threaded processes.

On the first call to [gnutls_rnd], page 737 the generators are seeded with two independent keys obtained from the OS random device. Their seed is used to output a fixed amount of bytes before re-seeding; the number of bytes output varies per generator.

One generator is dedicated for the `GNUTLS_RND_NONCE` level, and the second is shared for the `GNUTLS_RND_KEY` and `GNUTLS_RND_RANDOM` levels. For the rest of this section we refer to the first as the nonce generator and the second as the key generator.

The nonce generator will reseed after outputting a fixed amount of bytes (typically few megabytes), or after few hours of operation without reaching the limit has passed. It is being re-seed using the key generator to obtain a new key for the CHACHA cipher, which is mixed with its old one.

Similarly, the key generator, will also re-seed after a fixed amount of bytes is generated (typically less than the nonce), and will also re-seed based on time, i.e., after few hours of operation without reaching the limit for a re-seed. For its re-seed it mixes mixes data obtained from the OS random device with the previous key.

Although the key generator used to provide data for the `GNUTLS_RND_RANDOM` and `GNUTLS_RND_KEY` levels is identical, when used with the `GNUTLS_RND_KEY` level a re-key of the PRNG using its own output, is additionally performed. That ensures that the recovery of the PRNG state will not be sufficient to recover previously generated values.

The AES-DRBG generator - inner workings

Similar with the default generator, the random number generator levels in `gnutls_rnd_level_t` map to two AES-DRBG random generators which are initially seeded using the OS random device, e.g., `/dev/urandom` or `getrandom()`. These random generators are unique per thread, and are automatically re-seeded when a fork is detected.

The AES-DRBG generator is based on the AES cipher in counter mode and is re-seeded after a fixed amount of bytes are generated.

Defense against PRNG attacks

This section describes the counter-measures available in the Pseudo-random number generator (PRNG) of GnuTLS for known attacks as described in [[PRNGATTACKS], page 749]. Note that, the attacks on a PRNG such as state-compromise, assume a quite powerful adversary which has in practice access to the PRNG state.

Cryptanalytic

To defend against cryptanalytic attacks GnuTLS' PRNG is a stream cipher designed to defend against the same attacks. As such, GnuTLS' PRNG strength with regards to this attack relies on the underlying crypto block, which at the time of writing is CHACHA. That is easily replaceable in the future if attacks are found to be possible in that cipher.

Input-based attacks

These attacks assume that the attacker can influence the input that is used to form the state of the PRNG. To counter these attacks GnuTLS does not gather input from the system environment but rather relies on the OS provided random generator. That is the `/dev/urandom` or `getentropy`/`getrandom` system calls. As such, GnuTLS' PRNG is as strong as the system random generator can assure with regards to input-based attacks.

State-compromise: Backtracking

A backtracking attack, assumes that an adversary obtains at some point of time access to the generator state, and wants to recover past bytes. As the GnuTLS generator is fine-tuned to provide multiple levels, such an attack mainly concerns levels `GNUTLS_RND_RANDOM` and `GNUTLS_RND_KEY`, since `GNUTLS_RND_NONCE` is intended to output non-secret data. The `GNUTLS_RND_RANDOM` generator at the time of writing can output 2MB prior to being re-seeded thus this is its upper bound for previously generated data recovered using this attack. That assumes that the state of the operating system random generator is unknown to the attacker, and we carry that assumption on the next paragraphs. The usage of `GNUTLS_RND_KEY` level ensures that no backtracking is possible for all output data, by re-keying the PRNG using its own output.

Such an attack reflects the real world scenario where application's memory is temporarily compromised, while the kernel's memory is inaccessible.

State-compromise: Permanent Compromise Attack

A permanent compromise attack implies that once an attacker compromises the state of GnuTLS' random generator at a specific time, future and past outputs from the generator are compromised. For past outputs the previous paragraph applies. For future outputs,

both the GNUTLS_RND_RANDOM and the GNUTLS_RND_KEY will recover after 2MB of data have been generated or few hours have passed (two at the time of writing). Similarly the GNUTLS_RND_NONCE level generator will recover after several megabytes of output is generated, or its re-key time is reached.

State-compromise: Iterative guessing

This attack assumes that after an attacker obtained the PRNG state at some point, is able to recover the state at a later time by observing outputs of the PRNG. That is countered by switching the key to generators using a combination of a fresh key and the old one (using XOR), at re-seed time. All levels are immune to such attack after a re-seed.

State-compromise: Meet-in-the-Middle

This attack assumes that the attacker obtained the PRNG state at two distinct times, and being able to recover the state at the third time after observing the output of the PRNG. Given the approach described on the above paragraph, all levels are immune to such attack.

11.7 FIPS140-2 mode

GnuTLS can operate in a special mode for FIPS140-2. That mode of operation is for the conformance to NIST's FIPS140-2 publication, which consists of policies for cryptographic modules (such as software libraries). Its implementation in GnuTLS is designed for Red Hat Enterprise Linux, and can only be enabled when the library is explicitly compiled with the '–enable-fips140-mode' configure option.

There are two distinct library states with regard to FIPS140-2: the FIPS140-2 mode is *installed* if /etc/system-fips is present, and the FIPS140-2 mode is *enabled* if /proc/sys/crypto/fips_enabled contains '1', which is typically set with the "fips=1" kernel command line option.

When the FIPS140-2 mode is installed, the operation of the library is modified as follows.

- The random generator used switches to DRBG-AES
- The integrity of the GnuTLS and dependent libraries is checked on startup
- Algorithm self-tests are run on library load

When the FIPS140-2 mode is enabled, The operation of the library is in addition modified as follows.

- Only approved by FIPS140-2 algorithms are enabled
- Only approved by FIPS140-2 key lengths are allowed for key generation
- Any cryptographic operation will be refused if any of the self-tests failed

There are also few environment variables which modify that operation. The environment variable GNUTLS_SKIP_FIPS_INTEGRITY_CHECKS will disable the library integrity tests on startup, and the variable GNUTLS_FORCE_FIPS_MODE can be set to force a value from Figure 11.5, i.e., '1' will enable the FIPS140-2 mode, while '0' will disable it.

The integrity checks for the dependent libraries and GnuTLS are performed using '.hmac' files which are present at the same path as the library. The key for the operations can be provided on compile-time with the configure option '–with-fips140-key'. The MAC algorithm used is HMAC-SHA256.

On runtime an application can verify whether the library is in FIPS140-2 mode using the [gnutls_fips140_mode_enabled], page 410 function.

Relaxing FIPS140-2 requirements

The library by default operates in a strict enforcing mode, ensuring that all constraints imposed by the FIPS140-2 specification are enforced. However the application can relax these requirements via [gnutls_fips140_set_mode], page 410 which can switch to alternative modes as in Figure 11.5.

`GNUTLS_FIPS140_DISABLED`
: The FIPS140-2 mode is disabled.

`GNUTLS_FIPS140_STRICT`
: The default mode; all forbidden operations will cause an operation failure via error code.

`GNUTLS_FIPS140_SELFTESTS`
: A transient state during library initialization. That state cannot be set or seen by applications.

`GNUTLS_FIPS140_LAX`
: The library still uses the FIPS140-2 relevant algorithms but all forbidden by FIPS140-2 operations are allowed; this is useful when the application is aware of the followed security policy, and needs to utilize disallowed operations for other reasons (e.g., compatibility).

`GNUTLS_FIPS140_LOG`
: Similarly to `GNUTLS_FIPS140_LAX` , it allows forbidden operations; any use of them results to a message to the audit callback functions.

Figure 11.5: The `gnutls_fips_mode_t` enumeration.

The intention of this API is to be used by applications which may run in FIPS140-2 mode, while they utilize few algorithms not in the allowed set, e.g., for non-security related purposes. In these cases applications should wrap the non-compliant code within blocks like the following.

```
GNUTLS_FIPS140_SET_LAX_MODE();

_gnutls_hash_fast(GNUTLS_DIG_MD5, buffer, sizeof(buffer), output);

GNUTLS_FIPS140_SET_STRICT_MODE();
```

The `GNUTLS_FIPS140_SET_LAX_MODE` and `GNUTLS_FIPS140_SET_STRICT_MODE` are macros to simplify the following sequence of calls.

```
if (gnutls_fips140_mode_enabled())
  gnutls_fips140_set_mode(GNUTLS_FIPS140_LAX, GNUTLS_FIPS140_SET_MODE_THREAD);

_gnutls_hash_fast(GNUTLS_DIG_MD5, buffer, sizeof(buffer), output);

if (gnutls_fips140_mode_enabled())
```

```
gnutls_fips140_set_mode(GNUTLS_FIPS140_STRICT, GNUTLS_FIPS140_SET_MODE_THREAD);
```

The reason of the `GNUTLS_FIPS140_SET_MODE_THREAD` flag in the previous calls is to localize the change in the mode. Note also, that such a block has no effect when the library is not operating under FIPS140-2 mode, and thus it can be considered a no-op.

Applications could also switch FIPS140-2 mode explicitly off, by calling

```
gnutls_fips140_set_mode(GNUTLS_FIPS140_LAX, 0);
```

Appendix A Upgrading from previous versions

The GnuTLS library typically maintains binary and source code compatibility across versions. The releases that have the major version increased break binary compatibility but source compatibility is provided. This section lists exceptional cases where changes to existing code are required due to library changes.

Upgrading to 2.12.x from previous versions

GnuTLS 2.12.x is binary compatible with previous versions but changes the semantics of `gnutls_transport_set_lowat`, which might cause breakage in applications that relied on its default value be 1. Two fixes are proposed:

- Quick fix. Explicitly call `gnutls_transport_set_lowat (session, 1);` after [gnutls_init], page 422.
- Long term fix. Because later versions of gnutls abolish the functionality of using the system call `select` to check for gnutls pending data, the function [gnutls_record_check_pending], page 448 has to be used to achieve the same functionality as described in Section 6.5.1 [Asynchronous operation], page 160.

Upgrading to 3.0.x from 2.12.x

GnuTLS 3.0.x is source compatible with previous versions except for the functions listed below.

Old function	**Replacement**
`gnutls_transport_set_lowat`	To replace its functionality the function [gnutls_record_check_pending], page 448 has to be used, as described in Section 6.5.1 [Asynchronous operation], page 160,
`gnutls_session_get_server_random`, `gnutls_session_get_client_random`	They are replaced by the safer function [gnutls_session_get_random], page 463
`gnutls_session_get_master_secret`	Replaced by the keying material exporters discussed in Section 6.12.7 [Deriving keys for other applications/protocols], page 193,
`gnutls_transport_set_global_errno`	Replaced by using the system's errno facility or [gnutls_transport_set_errno], page 485.
`gnutls_x509_privkey_verify_data`	Replaced by [gnutls_pubkey_verify_data2], page 710.
`gnutls_certificate_verify_peers`	Replaced by [gnutls_certificate_verify_peers2], page 392.

`gnutls_psk_netconf_derive_key`	Removed. The key derivation function was never standardized.
`gnutls_session_set_finished_function`	Removed.
`gnutls_ext_register`	Removed. Extension registration API is now internal to allow easier changes in the API.
`gnutls_certificate_get_x509_crls`, `gnutls_certificate_get_x509_cas`	Removed to allow updating the internal structures. Replaced by [gnutls_certificate_get_issuer], page 369.
`gnutls_certificate_get_openpgp_keyring`	Removed.
`gnutls_ia_`	Removed. The inner application extensions were completely removed (they failed to be standardized).

Upgrading to 3.1.x from 3.0.x

GnuTLS 3.1.x is source and binary compatible with GnuTLS 3.0.x releases. Few functions have been deprecated and are listed below.

Old function	**Replacement**
`gnutls_pubkey_verify_hash`	The function [gnutls_pubkey_verify_hash2], page 711 is provided and is functionally equivalent and safer to use.
`gnutls_pubkey_verify_data`	The function [gnutls_pubkey_verify_data2], page 710 is provided and is functionally equivalent and safer to use.

Upgrading to 3.2.x from 3.1.x

GnuTLS 3.2.x is source and binary compatible with GnuTLS 3.1.x releases. Few functions have been deprecated and are listed below.

Old function	**Replacement**
`gnutls_privkey_sign_raw_data`	The function [gnutls_privkey_sign_hash], page 695 is equivalent when the flag `GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA` is specified.

Upgrading to 3.3.x from 3.2.x

GnuTLS 3.3.x is source and binary compatible with GnuTLS 3.2.x releases; however there few changes in semantics which are listed below.

Old function	Replacement
`gnutls_global_init`	No longer required. The library is initialized using a constructor.
`gnutls_global_deinit`	No longer required. The library is deinitialized using a destructor.

Upgrading to 3.4.x from 3.3.x

GnuTLS 3.4.x is source compatible with GnuTLS 3.3.x releases; however, several deprecated functions were removed, and are listed below.

Old function	Replacement
Priority string "NORMAL" has been modified	The following string emulates the 3.3.x behavior "NORMAL:+VERS-SSL3.0:+ARCFOUR-128:+DHE-DSS:+SIGN-DSA-SHA512:+SIGN-DSA-SHA256:+SIGN-DSA-SHA1"
`gnutls_certificate_client_set_retrieve_function`, `gnutls_certificate_server_set_retrieve_function`	[gnutls_certificate_set_retrieve_function], page 380
`gnutls_certificate_set_rsa_export_params`, `gnutls_rsa_export_get_modulus_bits`, `gnutls_rsa_export_get_pubkey`, `gnutls_rsa_params_cpy`, `gnutls_rsa_params_deinit`, `gnutls_rsa_params_export_pkcs1`, `gnutls_rsa_params_export_raw`, `gnutls_rsa_params_generate2`, `gnutls_rsa_params_import_pkcs1`, `gnutls_rsa_params_import_raw`, `gnutls_rsa_params_init`	No replacement; the library does not support the RSA-EXPORT ciphersuites.

`gnutls_pubkey_verify_hash`,	[gnutls_pubkey_verify_hash2], page 711.
`gnutls_pubkey_verify_data`,	[gnutls_pubkey_verify_data2], page 710.
`gnutls_x509_crt_get_verify_algorithm`,	No replacement; a similar function is [gnutls_x509_crt_get_signature_algorithm], page 555.
`gnutls_pubkey_get_verify_algorithm`,	No replacement; a similar function is [gnutls_pubkey_get_preferred_hash_algorithm], page 703.
`gnutls_certificate_type_set_priority`, `gnutls_cipher_set_priority`, `gnutls_compression_set_priority`, `gnutls_kx_set_priority`, `gnutls_mac_set_priority`, `gnutls_protocol_set_priority`	[gnutls_priority_set_direct], page 439.
`gnutls_sign_callback_get`, `gnutls_sign_callback_set`	[gnutls_privkey_import_ext3], page 687
`gnutls_x509_crt_verify_hash`	[gnutls_pubkey_verify_hash2], page 711
`gnutls_x509_crt_verify_data`	[gnutls_pubkey_verify_data2], page 710
`gnutls_privkey_sign_raw_data`	[gnutls_privkey_sign_hash], page 695 with the flag GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA

Upgrading to 3.6.x from 3.5.x

GnuTLS 3.6.x is source and binary compatible with GnuTLS 3.5.x releases; however, there are minor differences, listed below.

Old functionality	**Replacement**
The priority strings "+COMP" are a no-op	TLS compression is no longer available.
The SSL 3.0 protocol is a no-op	SSL 3.0 is no longer compiled in by default. It is a legacy protocol which is completely eliminated from public internet. As such it was removed to reduce the attack vector for applications using the library.

The hash function SHA2-224 is a no-op for TLS1.2	TLS 1.3 no longer uses SHA2-224, and it was never a widespread hash algorithm. As such it was removed for simplicity.
The SRP key exchange accepted parameters outside the [[TLSSRP], page 752] spec	The SRP key exchange is restricted to [[TLSSRP], page 752] spec parameters to protect clients from MitM attacks.
The compression-related functions are deprecated	No longer use `gnutls_compression_get`, `gnutls_compression_get_name`, `gnutls_compression_list`, and `gnutls_compression_get_id`.
[gnutls_x509_crt_sign], page 575, [gnutls_x509_crl_sign], page 512, [gnutls_x509_crq_sign], page 531	These signing functions will no longer sign using SHA1, but with a secure hash algorithm.
[gnutls_certificate_set_ocsp_status_request_file], page 374	This function will return an error if the loaded response doesn't match any of the present certificates. To revert to previous semantics set the `GNUTLS_CERTIFICATE_SKIP_OCSP_RESPONSE_CHECK` flag using [gnutls_certificate_set_flags], page 373.
The callback [gnutls_privkey_import_ext3], page 687 is not flexible enough for new signature algorithms such as RSA-PSS	It is replaced with [gnutls_privkey_import_ext4], page 688
Re-handshake functionality is not applicable under TLS 1.3.	It is replaced by separate key update and re-authentication functionality which can be accessed directly via [gnutls_session_key_update], page 464 and [gnutls_reauth], page 447.
TLS session identifiers are not shared with the server under TLS 1.3.	The TLS session identifiers are persistent across resumption only on server side and can be obtained as before via [gnutls_session_get_id2], page 462.
[gnutls_pkcs11_privkey_generate3], page 663, [gnutls_pkcs11_copy_secret_key], page 651, [gnutls_pkcs11_copy_x509_privkey2], page 652	These functions no longer create an exportable key by default; they require the flag `GNUTLS_PKCS11_OBJ_FLAG_MARK_NOT_SENSITIVE` to do so.

[gnutls_db_set_retrieve_function], page 398, [gnutls_db_set_store_function], page 398, [gnutls_db_set_remove_function], page 398	These functions are no longer relevant under TLS 1.3; resumption under TLS 1.3 is done via session tickets, c.f. [gnutls_session_ticket_enable_server], page 468.
[gnutls_session_get_data2], page 460, [gnutls_session_get_data], page 460	These functions may introduce a slight delay under TLS 1.3 for few milliseconds. Check output of [gnutls_session_get_flags], page 461 for GNUTLS_SFLAGS_SESSION_TICKET before calling this function to avoid delays. To work efficiently under TLS 1.3 this function requires the application setting [gnutls_transport_set_pull_timeout_function], page 487.
SRP and RSA-PSK key exchanges are not supported under TLS 1.3	SRP and RSA-PSK key exchanges are not supported in TLS 1.3, so when these key exchanges are present in a priority string, TLS 1.3 is disabled.
Anonymous key exchange is not supported under TLS 1.3	There is no anonymous key exchange supported under TLS 1.3, so if an anonymous key exchange method is set in a priority string, and no certificate credentials are set in the client or server, TLS 1.3 will not be negotiated.
ECDHE-PSK and DHE-PSK keywords have the same meaning under TLS 1.3	In the priority strings, both `ECDHEPSK` and `DHEPSK` indicate the intent to support an ephemeral key exchange with the pre-shared key. The parameters of the key exchange are negotiated with the supported groups specified in the priority string.
Authentication-only ciphersuites are not supported under TLS 1.3	Ciphersuites with the `NULL` cipher (i.e., authentication-only) are not supported in TLS 1.3, so when they are specified in a priority string, TLS 1.3 is disabled.
Supplemental data is not supported under TLS 1.3	The TLS supplemental data handshake message (RFC 4680) is not supported under TLS 1.3, so if the application calls [gnutls_supplemental_register], page 482 or [gnutls_session_supplemental_register], page 467, TLS 1.3 is disabled.

The GNUTLS_X509_NO_WELL_DEFINED_EXPIRATION macro is a no-op	The macro was non-functional and because of the nature of the extension of the no-well-defined date for certificates (a real date), it will not be fixed or re-introduced.

Appendix B Support

B.1 Getting Help

A mailing list where users may help each other exists, and you can reach it by sending e-mail to `gnutls-help@gnutls.org`. Archives of the mailing list discussions, and an interface to manage subscriptions, is available through the World Wide Web at `https://lists.gnutls.org/pipermail/gnutls-help/`.

A mailing list for developers are also available, see `https://www.gnutls.org/lists.html`. Bug reports should be sent to `bugs@gnutls.org`, see Section B.3 [Bug Reports], page 339.

B.2 Commercial Support

Commercial support is available for users of GnuTLS. See `https://www.gnutls.org/commercial.html` for more information.

B.3 Bug Reports

If you think you have found a bug in GnuTLS, please investigate it and report it.

- Please make sure that the bug is really in GnuTLS, and preferably also check that it hasn't already been fixed in the latest version.
- You have to send us a test case that makes it possible for us to reproduce the bug.
- You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way. Make sure that the bug report includes all information you would need to fix this kind of bug for someone else.

Please make an effort to produce a self-contained report, with something definite that can be tested or debugged. Vague queries or piecemeal messages are difficult to act on and don't help the development effort.

If your bug report is good, we will do our best to help you to get a corrected version of the software; if the bug report is poor, we won't do anything about it (apart from asking you to send better bug reports).

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please also send a note.

Send your bug report to:

'`bugs@gnutls.org`'

B.4 Contributing

If you want to submit a patch for inclusion – from solving a typo you discovered, up to adding support for a new feature – you should submit it as a bug report, using the process in Section B.3 [Bug Reports], page 339. There are some things that you can do to increase the chances for it to be included in the official package.

Unless your patch is very small (say, under 10 lines) we require that you assign the copyright of your work to the Free Software Foundation. This is to protect the freedom of the project. If you have not already signed papers, we will send you the necessary information when you submit your contribution.

For contributions that doesn't consist of actual programming code, the only guidelines are common sense. For code contributions, a number of style guides will help you:

- Coding Style. Follow the GNU Standards document.

  If you normally code using another coding standard, there is no problem, but you should use '`indent`' to reformat the code before submitting your work.
- Use the unified diff format '`diff -u`'.
- Return errors. No reason whatsoever should abort the execution of the library. Even memory allocation errors, e.g. when malloc return NULL, should work although result in an error code.
- Design with thread safety in mind. Don't use global variables. Don't even write to per-handle global variables unless the documented behaviour of the function you write is to write to the per-handle global variable.
- Avoid using the C math library. It causes problems for embedded implementations, and in most situations it is very easy to avoid using it.
- Document your functions. Use comments before each function headers, that, if properly formatted, are extracted into Texinfo manuals and GTK-DOC web pages.
- Supply a ChangeLog and NEWS entries, where appropriate.

B.5 Certification

There are certifications from national or international bodies which "prove" to an auditor that the crypto component follows some best practices, such as unit testing and reliance on well known crypto primitives.

GnuTLS has support for the FIPS 140-2 certification under Red Hat Enterprise Linux. See Section 11.7 [FIPS140-2 mode], page 329, for more information.

Appendix C Error Codes and Descriptions

The error codes used throughout the library are described below. The return code `GNUTLS_E_SUCCESS` indicates a successful operation, and is guaranteed to have the value 0, so you can use it in logical expressions.

0	GNUTLS_E_SUCCESS	Success.
-3	GNUTLS_E_UNKNOWN_COMPRESSION_ALGORITHM	Could not negotiate a supported compression method.
-6	GNUTLS_E_UNKNOWN_CIPHER_TYPE	The cipher type is unsupported.
-7	GNUTLS_E_LARGE_PACKET	The transmitted packet is too large (EMSGSIZE).
-8	GNUTLS_E_UNSUPPORTED_VERSION_PACKET	A packet with illegal or unsupported version was received.
-9	GNUTLS_E_UNEXPECTED_PACKET_LENGTH	Error decoding the received TLS packet.
-10	GNUTLS_E_INVALID_SESSION	The specified session has been invalidated for some reason.
-12	GNUTLS_E_FATAL_ALERT_RECEIVED	A TLS fatal alert has been received.
-15	GNUTLS_E_UNEXPECTED_PACKET	An unexpected TLS packet was received.
-16	GNUTLS_E_WARNING_ALERT_RECEIVED	A TLS warning alert has been received.
-18	GNUTLS_E_ERROR_IN_FINISHED_PACKET	An error was encountered at the TLS Finished packet calculation.
-19	GNUTLS_E_UNEXPECTED_HANDSHAKE_PACKET	An unexpected TLS handshake packet was received.
-21	GNUTLS_E_UNKNOWN_CIPHER_SUITE	Could not negotiate a supported cipher suite.
-22	GNUTLS_E_UNWANTED_ALGORITHM	An algorithm that is not enabled was negotiated.
-23	GNUTLS_E_MPI_SCAN_FAILED	The scanning of a large integer has failed.
-24	GNUTLS_E_DECRYPTION_FAILED	Decryption has failed.
-25	GNUTLS_E_MEMORY_ERROR	Internal error in memory allocation.
-26	GNUTLS_E_DECOMPRESSION_FAILED	Decompression of the TLS record packet has failed.
-27	GNUTLS_E_COMPRESSION_FAILED	Compression of the TLS record packet has failed.
-28	GNUTLS_E_AGAIN	Resource temporarily unavailable, try again.

-29	GNUTLS_E_EXPIRED	The session or certificate has expired.
-30	GNUTLS_E_DB_ERROR	Error in Database backend.
-31	GNUTLS_E_SRP_PWD_ERROR	Error in password/key file.
-32	GNUTLS_E_INSUFFICIENT_CREDENTIALS	Insufficient credentials for that request.
-33	GNUTLS_E_HASH_FAILED	Hashing has failed.
-34	GNUTLS_E_BASE64_DECODING_ERROR	Base64 decoding error.
-35	GNUTLS_E_MPI_PRINT_FAILED	Could not export a large integer.
-37	GNUTLS_E_REHANDSHAKE	Rehandshake was requested by the peer.
-38	GNUTLS_E_GOT_APPLICATION_DATA	TLS Application data were received, while expecting handshake data.
-39	GNUTLS_E_RECORD_LIMIT_REACHED	The upper limit of record packet sequence numbers has been reached. Wow!
-40	GNUTLS_E_ENCRYPTION_FAILED	Encryption has failed.
-43	GNUTLS_E_CERTIFICATE_ERROR	Error in the certificate.
-44	GNUTLS_E_PK_ENCRYPTION_FAILED	Public key encryption has failed.
-45	GNUTLS_E_PK_DECRYPTION_FAILED	Public key decryption has failed.
-46	GNUTLS_E_PK_SIGN_FAILED	Public key signing has failed.
-47	GNUTLS_E_X509_UNSUPPORTED_CRITICAL_EXTENSION	Unsupported critical extension in X.509 certificate.
-48	GNUTLS_E_KEY_USAGE_VIOLATION	Key usage violation in certificate has been detected.
-49	GNUTLS_E_NO_CERTIFICATE_FOUND	No certificate was found.
-50	GNUTLS_E_INVALID_REQUEST	The request is invalid.
-51	GNUTLS_E_SHORT_MEMORY_BUFFER	The given memory buffer is too short to hold parameters.
-52	GNUTLS_E_INTERRUPTED	Function was interrupted.
-53	GNUTLS_E_PUSH_ERROR	Error in the push function.
-54	GNUTLS_E_PULL_ERROR	Error in the pull function.
-55	GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER	An illegal parameter has been received.

-56	GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE	The requested data were not available.
-57	GNUTLS_E_PKCS1_WRONG_PAD	Wrong padding in PKCS1 packet.
-58	GNUTLS_E_RECEIVED_ILLEGAL_EXTENSION	An illegal TLS extension was received.
-59	GNUTLS_E_INTERNAL_ERROR	GnuTLS internal error.
-60	GNUTLS_E_CERTIFICATE_KEY_MISMATCH	The certificate and the given key do not match.
-61	GNUTLS_E_UNSUPPORTED_CERTIFICATE_TYPE	The certificate type is not supported.
-62	GNUTLS_E_X509_UNKNOWN_SAN	Unknown Subject Alternative name in X.509 certificate.
-63	GNUTLS_E_DH_PRIME_UNACCEPTABLE	The Diffie-Hellman prime sent by the server is not acceptable (not long enough).
-64	GNUTLS_E_FILE_ERROR	Error while reading file.
-67	GNUTLS_E_ASN1_ELEMENT_NOT_FOUND	ASN1 parser: Element was not found.
-68	GNUTLS_E_ASN1_IDENTIFIER_NOT_FOUND	ASN1 parser: Identifier was not found
-69	GNUTLS_E_ASN1_DER_ERROR	ASN1 parser: Error in DER parsing.
-70	GNUTLS_E_ASN1_VALUE_NOT_FOUND	ASN1 parser: Value was not found.
-71	GNUTLS_E_ASN1_GENERIC_ERROR	ASN1 parser: Generic parsing error.
-72	GNUTLS_E_ASN1_VALUE_NOT_VALID	ASN1 parser: Value is not valid.
-73	GNUTLS_E_ASN1_TAG_ERROR	ASN1 parser: Error in TAG.
-74	GNUTLS_E_ASN1_TAG_IMPLICIT	ASN1 parser: error in implicit tag
-75	GNUTLS_E_ASN1_TYPE_ANY_ERROR	ASN1 parser: Error in type 'ANY'.
-76	GNUTLS_E_ASN1_SYNTAX_ERROR	ASN1 parser: Syntax error.
-77	GNUTLS_E_ASN1_DER_OVERFLOW	ASN1 parser: Overflow in DER parsing.
-78	GNUTLS_E_TOO_MANY_EMPTY_PACKETS	Too many empty record packets have been received.
-79	GNUTLS_E_OPENPGP_UID_REVOKED	The OpenPGP User ID is revoked.

-80	GNUTLS_E_UNKNOWN_PK_ALGORITHM	An unknown public key algorithm was encountered.
-81	GNUTLS_E_TOO_MANY_HANDSHAKE_PACKETS	Too many handshake packets have been received.
-82	GNUTLS_E_RECEIVED_DISALLOWED_NAME	A disallowed SNI server name has been received.
-84	GNUTLS_E_NO_TEMPORARY_RSA_PARAMS	No temporary RSA parameters were found.
-86	GNUTLS_E_NO_COMPRESSION_ALGORITHMS	No supported compression algorithms have been found.
-87	GNUTLS_E_NO_CIPHER_SUITES	No supported cipher suites have been found.
-88	GNUTLS_E_OPENPGP_GETKEY_FAILED	Could not get OpenPGP key.
-89	GNUTLS_E_PK_SIG_VERIFY_FAILED	Public key signature verification has failed.
-90	GNUTLS_E_ILLEGAL_SRP_USERNAME	The SRP username supplied is illegal.
-91	GNUTLS_E_SRP_PWD_PARSING_ERROR	Parsing error in password/key file.
-93	GNUTLS_E_NO_TEMPORARY_DH_PARAMS	No temporary DH parameters were found.
-94	GNUTLS_E_OPENPGP_FINGERPRINT_UNSUPPORTED	The OpenPGP fingerprint is not supported.
-95	GNUTLS_E_X509_UNSUPPORTED_ATTRIBUTE	The certificate has unsupported attributes.
-96	GNUTLS_E_UNKNOWN_HASH_ALGORITHM	The hash algorithm is unknown.
-97	GNUTLS_E_UNKNOWN_PKCS_CONTENT_TYPE	The PKCS structure's content type is unknown.
-98	GNUTLS_E_UNKNOWN_PKCS_BAG_TYPE	The PKCS structure's bag type is unknown.
-99	GNUTLS_E_INVALID_PASSWORD	The given password contains invalid characters.
-100	GNUTLS_E_MAC_VERIFY_FAILED	The Message Authentication Code verification failed.
-101	GNUTLS_E_CONSTRAINT_ERROR	Some constraint limits were reached.
-104	GNUTLS_E_IA_VERIFY_FAILED	Verifying TLS/IA phase checksum failed
-105	GNUTLS_E_UNKNOWN_ALGORITHM	The specified algorithm or protocol is unknown.

-106	GNUTLS_E_UNSUPPORTED_-SIGNATURE_ALGORITHM	The signature algorithm is not supported.
-107	GNUTLS_E_SAFE_-RENEGOTIATION_FAILED	Safe renegotiation failed.
-108	GNUTLS_E_UNSAFE_-RENEGOTIATION_DENIED	Unsafe renegotiation denied.
-109	GNUTLS_E_UNKNOWN_SRP_-USERNAME	The username supplied is unknown.
-110	GNUTLS_E_PREMATURE_-TERMINATION	The TLS connection was non-properly terminated.
-111	GNUTLS_E_MALFORMED_-CIDR	CIDR name constraint is malformed in size or structure.
-112	GNUTLS_E_CERTIFICATE_-REQUIRED	Certificate is required.
-201	GNUTLS_E_BASE64_-ENCODING_ERROR	Base64 encoding error.
-202	GNUTLS_E_INCOMPATIBLE_-GCRYPT_LIBRARY	The crypto library version is too old.
-203	GNUTLS_E_INCOMPATIBLE_-LIBTASN1_LIBRARY	The tasn1 library version is too old.
-204	GNUTLS_E_OPENPGP_-KEYRING_ERROR	Error loading the keyring.
-205	GNUTLS_E_X509_-UNSUPPORTED_OID	The OID is not supported.
-206	GNUTLS_E_RANDOM_FAILED	Failed to acquire random data.
-207	GNUTLS_E_BASE64_-UNEXPECTED_HEADER_-ERROR	Base64 unexpected header error.
-208	GNUTLS_E_OPENPGP_-SUBKEY_ERROR	Could not find OpenPGP subkey.
-209	GNUTLS_E_CRYPTO_-ALREADY_REGISTERED	There is already a crypto algorithm with lower priority.
-210	GNUTLS_E_HANDSHAKE_-TOO_LARGE	The handshake data size is too large.
-211	GNUTLS_E_CRYPTODEV_-IOCTL_ERROR	Error interfacing with /dev/crypto
-212	GNUTLS_E_CRYPTODEV_-DEVICE_ERROR	Error opening /dev/crypto
-213	GNUTLS_E_CHANNEL_-BINDING_NOT_AVAILABLE	Channel binding data not available
-214	GNUTLS_E_BAD_COOKIE	The cookie was bad.
-215	GNUTLS_E_OPENPGP_-PREFERRED_KEY_ERROR	The OpenPGP key has not a preferred key set.

-216	GNUTLS_E_INCOMPAT_DSA_KEY_WITH_TLS_PROTOCOL	The given DSA key is incompatible with the selected TLS protocol.
-217	GNUTLS_E_INSUFFICIENT_SECURITY	One of the involved algorithms has insufficient security level.
-292	GNUTLS_E_HEARTBEAT_PONG_RECEIVED	A heartbeat pong message was received.
-293	GNUTLS_E_HEARTBEAT_PING_RECEIVED	A heartbeat ping message was received.
-294	GNUTLS_E_UNRECOGNIZED_NAME	The SNI host name not recognised.
-300	GNUTLS_E_PKCS11_ERROR	PKCS #11 error.
-301	GNUTLS_E_PKCS11_LOAD_ERROR	PKCS #11 initialization error.
-302	GNUTLS_E_PARSING_ERROR	Error in parsing.
-303	GNUTLS_E_PKCS11_PIN_ERROR	Error in provided PIN.
-305	GNUTLS_E_PKCS11_SLOT_ERROR	PKCS #11 error in slot
-306	GNUTLS_E_LOCKING_ERROR	Thread locking error
-307	GNUTLS_E_PKCS11_ATTRIBUTE_ERROR	PKCS #11 error in attribute
-308	GNUTLS_E_PKCS11_DEVICE_ERROR	PKCS #11 error in device
-309	GNUTLS_E_PKCS11_DATA_ERROR	PKCS #11 error in data
-310	GNUTLS_E_PKCS11_UNSUPPORTED_FEATURE_ERROR	PKCS #11 unsupported feature
-311	GNUTLS_E_PKCS11_KEY_ERROR	PKCS #11 error in key
-312	GNUTLS_E_PKCS11_PIN_EXPIRED	PKCS #11 PIN expired
-313	GNUTLS_E_PKCS11_PIN_LOCKED	PKCS #11 PIN locked
-314	GNUTLS_E_PKCS11_SESSION_ERROR	PKCS #11 error in session
-315	GNUTLS_E_PKCS11_SIGNATURE_ERROR	PKCS #11 error in signature
-316	GNUTLS_E_PKCS11_TOKEN_ERROR	PKCS #11 error in token
-317	GNUTLS_E_PKCS11_USER_ERROR	PKCS #11 user error
-318	GNUTLS_E_CRYPTO_INIT_FAILED	The initialization of crypto backend has failed.

-319	GNUTLS_E_TIMEDOUT	The operation timed out
-320	GNUTLS_E_USER_ERROR	The operation was cancelled due to user error
-321	GNUTLS_E_ECC_NO_SUPPORTED_CURVES	No supported ECC curves were found
-322	GNUTLS_E_ECC_UNSUPPORTED_CURVE	The curve is unsupported
-323	GNUTLS_E_PKCS11_REQUESTED_OBJECT_NOT_AVAILBLE	The requested PKCS #11 object is not available
-324	GNUTLS_E_CERTIFICATE_LIST_UNSORTED	The provided X.509 certificate list is not sorted (in subject to issuer order)
-325	GNUTLS_E_ILLEGAL_PARAMETER	An illegal parameter was found.
-326	GNUTLS_E_NO_PRIORITIES_WERE_SET	No or insufficient priorities were set.
-327	GNUTLS_E_X509_UNSUPPORTED_EXTENSION	Unsupported extension in X.509 certificate.
-328	GNUTLS_E_SESSION_EOF	Peer has terminated the connection
-329	GNUTLS_E_TPM_ERROR	TPM error.
-330	GNUTLS_E_TPM_KEY_PASSWORD_ERROR	Error in provided password for key to be loaded in TPM.
-331	GNUTLS_E_TPM_SRK_PASSWORD_ERROR	Error in provided SRK password for TPM.
-332	GNUTLS_E_TPM_SESSION_ERROR	Cannot initialize a session with the TPM.
-333	GNUTLS_E_TPM_KEY_NOT_FOUND	TPM key was not found in persistent storage.
-334	GNUTLS_E_TPM_UNINITIALIZED	TPM is not initialized.
-335	GNUTLS_E_TPM_NO_LIB	The TPM library (trousers) cannot be found.
-340	GNUTLS_E_NO_CERTIFICATE_STATUS	There is no certificate status (OCSP).
-341	GNUTLS_E_OCSP_RESPONSE_ERROR	The OCSP response is invalid
-342	GNUTLS_E_RANDOM_DEVICE_ERROR	Error in the system's randomness device.
-343	GNUTLS_E_AUTH_ERROR	Could not authenticate peer.
-344	GNUTLS_E_NO_APPLICATION_PROTOCOL	No common application protocol could be negotiated.
-345	GNUTLS_E_SOCKETS_INIT_ERROR	Error in sockets initialization.

-346	GNUTLS_E_KEY_IMPORT_-FAILED	Failed to import the key into store.
-347	GNUTLS_E_-INAPPROPRIATE_FALLBACK	A connection with inappropriate fallback was attempted.
-348	GNUTLS_E_CERTIFICATE_-VERIFICATION_ERROR	Error in the certificate verification.
-349	GNUTLS_E_PRIVKEY_-VERIFICATION_ERROR	Error in the private key verification; seed doesn't match.
-350	GNUTLS_E_UNEXPECTED_-EXTENSIONS_LENGTH	Invalid TLS extensions length field.
-351	GNUTLS_E_ASN1_-EMBEDDED_NULL_IN_-STRING	The provided string has an embedded null.
-400	GNUTLS_E_SELF_TEST_-ERROR	Error while performing self checks.
-401	GNUTLS_E_NO_SELF_TEST	There is no self test for this algorithm.
-402	GNUTLS_E_LIB_IN_ERROR_-STATE	An error has been detected in the library and cannot continue operations.
-403	GNUTLS_E_PK_-GENERATION_ERROR	Error in public key generation.
-404	GNUTLS_E_IDNA_ERROR	There was an issue converting to or from UTF8.
-406	GNUTLS_E_SESSION_USER_-ID_CHANGED	Peer's certificate or username has changed during a rehandshake.
-407	GNUTLS_E_HANDSHAKE_-DURING_FALSE_START	Attempted handshake during false start.
-408	GNUTLS_E_UNAVAILABLE_-DURING_HANDSHAKE	Cannot perform this action while handshake is in progress.
-409	GNUTLS_E_PK_INVALID_-PUBKEY	The public key is invalid.
-410	GNUTLS_E_PK_INVALID_-PRIVKEY	The private key is invalid.
-411	GNUTLS_E_NOT_YET_-ACTIVATED	The certificate is not yet activated.
-412	GNUTLS_E_INVALID_UTF8_-STRING	The given string contains invalid UTF-8 characters.
-413	GNUTLS_E_NO_EMBEDDED_-DATA	There are no embedded data in the structure.
-414	GNUTLS_E_INVALID_UTF8_-EMAIL	The given email string contains non-ASCII characters before '@'.
-415	GNUTLS_E_INVALID_-PASSWORD_STRING	The given password contains invalid characters.

-416	GNUTLS_E_CERTIFICATE_-TIME_ERROR	Error in the time fields of certificate.
-417	GNUTLS_E_RECORD_-OVERFLOW	A TLS record packet with invalid length was received.
-418	GNUTLS_E_ASN1_TIME_-ERROR	The DER time encoding is invalid.
-419	GNUTLS_E_INCOMPATIBLE_-SIG_WITH_KEY	The signature is incompatible with the public key.
-420	GNUTLS_E_PK_INVALID_-PUBKEY_PARAMS	The public key parameters are invalid.
-421	GNUTLS_E_PK_NO_-VALIDATION_PARAMS	There are no validation parameters present.
-422	GNUTLS_E_OCSP_-MISMATCH_WITH_CERTS	The OCSP response provided doesn't match the available certificates
-423	GNUTLS_E_NO_COMMON_-KEY_SHARE	No common key share with peer.
-424	GNUTLS_E_REAUTH_-REQUEST	Re-authentication was requested by the peer.
-425	GNUTLS_E_TOO_MANY_-MATCHES	More than a single object matches the criteria.
-426	GNUTLS_E_CRL_-VERIFICATION_ERROR	Error in the CRL verification.
-427	GNUTLS_E_MISSING_-EXTENSION	An required TLS extension was received.
-428	GNUTLS_E_DB_ENTRY_-EXISTS	The Database entry already exists.
-429	GNUTLS_E_EARLY_DATA_-REJECTED	The early data were rejected.
-430	GNUTLS_E_X509_-DUPLICATE_EXTENSION	Duplicate extension in X.509 certificate.

Appendix D Supported Ciphersuites

Ciphersuites

Ciphersuite name	TLS ID	Since
TLS_AES_128_GCM_SHA256	0x13 0x01	TLS1.3
TLS_AES_256_GCM_SHA384	0x13 0x02	TLS1.3
TLS_CHACHA20_POLY1305_SHA256	0x13 0x03	TLS1.3
TLS_AES_128_CCM_SHA256	0x13 0x04	TLS1.3
TLS_AES_128_CCM_8_SHA256	0x13 0x05	TLS1.3
TLS_RSA_NULL_MD5	0x00 0x01	TLS1.0
TLS_RSA_NULL_SHA1	0x00 0x02	TLS1.0
TLS_RSA_NULL_SHA256	0x00 0x3B	TLS1.2
TLS_RSA_ARCFOUR_128_SHA1	0x00 0x05	TLS1.0
TLS_RSA_ARCFOUR_128_MD5	0x00 0x04	TLS1.0
TLS_RSA_3DES_EDE_CBC_SHA1	0x00 0x0A	TLS1.0
TLS_RSA_AES_128_CBC_SHA1	0x00 0x2F	TLS1.0
TLS_RSA_AES_256_CBC_SHA1	0x00 0x35	TLS1.0
TLS_RSA_CAMELLIA_128_CBC_SHA256	0x00 0xBA	TLS1.2
TLS_RSA_CAMELLIA_256_CBC_SHA256	0x00 0xC0	TLS1.2
TLS_RSA_CAMELLIA_128_CBC_SHA1	0x00 0x41	TLS1.0
TLS_RSA_CAMELLIA_256_CBC_SHA1	0x00 0x84	TLS1.0
TLS_RSA_AES_128_CBC_SHA256	0x00 0x3C	TLS1.2
TLS_RSA_AES_256_CBC_SHA256	0x00 0x3D	TLS1.2
TLS_RSA_AES_128_GCM_SHA256	0x00 0x9C	TLS1.2
TLS_RSA_AES_256_GCM_SHA384	0x00 0x9D	TLS1.2
TLS_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x7A	TLS1.2
TLS_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x7B	TLS1.2
TLS_RSA_AES_128_CCM	0xC0 0x9C	TLS1.2
TLS_RSA_AES_256_CCM	0xC0 0x9D	TLS1.2
TLS_RSA_AES_128_CCM_8	0xC0 0xA0	TLS1.2
TLS_RSA_AES_256_CCM_8	0xC0 0xA1	TLS1.2
TLS_DHE_DSS_ARCFOUR_128_SHA1	0x00 0x66	TLS1.0
TLS_DHE_DSS_3DES_EDE_CBC_SHA1	0x00 0x13	TLS1.0
TLS_DHE_DSS_AES_128_CBC_SHA1	0x00 0x32	TLS1.0
TLS_DHE_DSS_AES_256_CBC_SHA1	0x00 0x38	TLS1.0
TLS_DHE_DSS_CAMELLIA_128_CBC_SHA256	0x00 0xBD	TLS1.2
TLS_DHE_DSS_CAMELLIA_256_CBC_SHA256	0x00 0xC3	TLS1.2
TLS_DHE_DSS_CAMELLIA_128_CBC_SHA1	0x00 0x44	TLS1.0
TLS_DHE_DSS_CAMELLIA_256_CBC_SHA1	0x00 0x87	TLS1.0
TLS_DHE_DSS_AES_128_CBC_SHA256	0x00 0x40	TLS1.2
TLS_DHE_DSS_AES_256_CBC_SHA256	0x00 0x6A	TLS1.2
TLS_DHE_DSS_AES_128_GCM_SHA256	0x00 0xA2	TLS1.2
TLS_DHE_DSS_AES_256_GCM_SHA384	0x00 0xA3	TLS1.2
TLS_DHE_DSS_CAMELLIA_128_GCM_SHA256	0xC0 0x80	TLS1.2

TLS_DHE_DSS_CAMELLIA_256_GCM_SHA384	0xC0 0x81	TLS1.2
TLS_DHE_RSA_3DES_EDE_CBC_SHA1	0x00 0x16	TLS1.0
TLS_DHE_RSA_AES_128_CBC_SHA1	0x00 0x33	TLS1.0
TLS_DHE_RSA_AES_256_CBC_SHA1	0x00 0x39	TLS1.0
TLS_DHE_RSA_CAMELLIA_128_CBC_SHA256	0x00 0xBE	TLS1.2
TLS_DHE_RSA_CAMELLIA_256_CBC_SHA256	0x00 0xC4	TLS1.2
TLS_DHE_RSA_CAMELLIA_128_CBC_SHA1	0x00 0x45	TLS1.0
TLS_DHE_RSA_CAMELLIA_256_CBC_SHA1	0x00 0x88	TLS1.0
TLS_DHE_RSA_AES_128_CBC_SHA256	0x00 0x67	TLS1.2
TLS_DHE_RSA_AES_256_CBC_SHA256	0x00 0x6B	TLS1.2
TLS_DHE_RSA_AES_128_GCM_SHA256	0x00 0x9E	TLS1.2
TLS_DHE_RSA_AES_256_GCM_SHA384	0x00 0x9F	TLS1.2
TLS_DHE_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x7C	TLS1.2
TLS_DHE_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x7D	TLS1.2
TLS_DHE_RSA_CHACHA20_POLY1305	0xCC 0xAA	TLS1.2
TLS_DHE_RSA_AES_128_CCM	0xC0 0x9E	TLS1.2
TLS_DHE_RSA_AES_256_CCM	0xC0 0x9F	TLS1.2
TLS_DHE_RSA_AES_128_CCM_8	0xC0 0xA2	TLS1.2
TLS_DHE_RSA_AES_256_CCM_8	0xC0 0xA3	TLS1.2
TLS_ECDHE_RSA_NULL_SHA1	0xC0 0x10	TLS1.0
TLS_ECDHE_RSA_3DES_EDE_CBC_SHA1	0xC0 0x12	TLS1.0
TLS_ECDHE_RSA_AES_128_CBC_SHA1	0xC0 0x13	TLS1.0
TLS_ECDHE_RSA_AES_256_CBC_SHA1	0xC0 0x14	TLS1.0
TLS_ECDHE_RSA_AES_256_CBC_SHA384	0xC0 0x28	TLS1.2
TLS_ECDHE_RSA_ARCFOUR_128_SHA1	0xC0 0x11	TLS1.0
TLS_ECDHE_RSA_CAMELLIA_128_CBC_SHA256	0xC0 0x76	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_256_CBC_SHA384	0xC0 0x77	TLS1.2
TLS_ECDHE_ECDSA_NULL_SHA1	0xC0 0x06	TLS1.0
TLS_ECDHE_ECDSA_3DES_EDE_CBC_SHA1	0xC0 0x08	TLS1.0
TLS_ECDHE_ECDSA_AES_128_CBC_SHA1	0xC0 0x09	TLS1.0
TLS_ECDHE_ECDSA_AES_256_CBC_SHA1	0xC0 0x0A	TLS1.0
TLS_ECDHE_ECDSA_ARCFOUR_128_SHA1	0xC0 0x07	TLS1.0
TLS_ECDHE_ECDSA_CAMELLIA_128_CBC_-SHA256	0xC0 0x72	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_256_CBC_-SHA384	0xC0 0x73	TLS1.2
TLS_ECDHE_ECDSA_AES_128_CBC_SHA256	0xC0 0x23	TLS1.2
TLS_ECDHE_RSA_AES_128_CBC_SHA256	0xC0 0x27	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_128_GCM_-SHA256	0xC0 0x86	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_256_GCM_-SHA384	0xC0 0x87	TLS1.2
TLS_ECDHE_ECDSA_AES_128_GCM_SHA256	0xC0 0x2B	TLS1.2
TLS_ECDHE_ECDSA_AES_256_GCM_SHA384	0xC0 0x2C	TLS1.2
TLS_ECDHE_RSA_AES_128_GCM_SHA256	0xC0 0x2F	TLS1.2
TLS_ECDHE_RSA_AES_256_GCM_SHA384	0xC0 0x30	TLS1.2

TLS_ECDHE_ECDSA_AES_256_CBC_SHA384	0xC0 0x24	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x8A	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x8B	TLS1.2
TLS_ECDHE_RSA_CHACHA20_POLY1305	0xCC 0xA8	TLS1.2
TLS_ECDHE_ECDSA_CHACHA20_POLY1305	0xCC 0xA9	TLS1.2
TLS_ECDHE_ECDSA_AES_128_CCM	0xC0 0xAC	TLS1.2
TLS_ECDHE_ECDSA_AES_256_CCM	0xC0 0xAD	TLS1.2
TLS_ECDHE_ECDSA_AES_128_CCM_8	0xC0 0xAE	TLS1.2
TLS_ECDHE_ECDSA_AES_256_CCM_8	0xC0 0xAF	TLS1.2
TLS_ECDHE_PSK_3DES_EDE_CBC_SHA1	0xC0 0x34	TLS1.0
TLS_ECDHE_PSK_AES_128_CBC_SHA1	0xC0 0x35	TLS1.0
TLS_ECDHE_PSK_AES_256_CBC_SHA1	0xC0 0x36	TLS1.0
TLS_ECDHE_PSK_AES_128_CBC_SHA256	0xC0 0x37	TLS1.2
TLS_ECDHE_PSK_AES_256_CBC_SHA384	0xC0 0x38	TLS1.2
TLS_ECDHE_PSK_ARCFOUR_128_SHA1	0xC0 0x33	TLS1.0
TLS_ECDHE_PSK_NULL_SHA1	0xC0 0x39	TLS1.0
TLS_ECDHE_PSK_NULL_SHA256	0xC0 0x3A	TLS1.2
TLS_ECDHE_PSK_NULL_SHA384	0xC0 0x3B	TLS1.0
TLS_ECDHE_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x9A	TLS1.2
TLS_ECDHE_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x9B	TLS1.2
TLS_PSK_ARCFOUR_128_SHA1	0x00 0x8A	TLS1.0
TLS_PSK_3DES_EDE_CBC_SHA1	0x00 0x8B	TLS1.0
TLS_PSK_AES_128_CBC_SHA1	0x00 0x8C	TLS1.0
TLS_PSK_AES_256_CBC_SHA1	0x00 0x8D	TLS1.0
TLS_PSK_AES_128_CBC_SHA256	0x00 0xAE	TLS1.2
TLS_PSK_AES_256_GCM_SHA384	0x00 0xA9	TLS1.2
TLS_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x8E	TLS1.2
TLS_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x8F	TLS1.2
TLS_PSK_AES_128_GCM_SHA256	0x00 0xA8	TLS1.2
TLS_PSK_NULL_SHA1	0x00 0x2C	TLS1.0
TLS_PSK_NULL_SHA256	0x00 0xB0	TLS1.2
TLS_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x94	TLS1.2
TLS_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x95	TLS1.2
TLS_PSK_AES_256_CBC_SHA384	0x00 0xAF	TLS1.2
TLS_PSK_NULL_SHA384	0x00 0xB1	TLS1.2
TLS_RSA_PSK_ARCFOUR_128_SHA1	0x00 0x92	TLS1.0
TLS_RSA_PSK_3DES_EDE_CBC_SHA1	0x00 0x93	TLS1.0
TLS_RSA_PSK_AES_128_CBC_SHA1	0x00 0x94	TLS1.0
TLS_RSA_PSK_AES_256_CBC_SHA1	0x00 0x95	TLS1.0
TLS_RSA_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x92	TLS1.2
TLS_RSA_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x93	TLS1.2
TLS_RSA_PSK_AES_128_GCM_SHA256	0x00 0xAC	TLS1.2
TLS_RSA_PSK_AES_128_CBC_SHA256	0x00 0xB6	TLS1.2
TLS_RSA_PSK_NULL_SHA1	0x00 0x2E	TLS1.0
TLS_RSA_PSK_NULL_SHA256	0x00 0xB8	TLS1.2
TLS_RSA_PSK_AES_256_GCM_SHA384	0x00 0xAD	TLS1.2
TLS_RSA_PSK_AES_256_CBC_SHA384	0x00 0xB7	TLS1.2

TLS_RSA_PSK_NULL_SHA384	0x00 0xB9	TLS1.2
TLS_RSA_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x98	TLS1.2
TLS_RSA_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x99	TLS1.2
TLS_DHE_PSK_ARCFOUR_128_SHA1	0x00 0x8E	TLS1.0
TLS_DHE_PSK_3DES_EDE_CBC_SHA1	0x00 0x8F	TLS1.0
TLS_DHE_PSK_AES_128_CBC_SHA1	0x00 0x90	TLS1.0
TLS_DHE_PSK_AES_256_CBC_SHA1	0x00 0x91	TLS1.0
TLS_DHE_PSK_AES_128_CBC_SHA256	0x00 0xB2	TLS1.2
TLS_DHE_PSK_AES_128_GCM_SHA256	0x00 0xAA	TLS1.2
TLS_DHE_PSK_NULL_SHA1	0x00 0x2D	TLS1.0
TLS_DHE_PSK_NULL_SHA256	0x00 0xB4	TLS1.2
TLS_DHE_PSK_NULL_SHA384	0x00 0xB5	TLS1.2
TLS_DHE_PSK_AES_256_CBC_SHA384	0x00 0xB3	TLS1.2
TLS_DHE_PSK_AES_256_GCM_SHA384	0x00 0xAB	TLS1.2
TLS_DHE_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x96	TLS1.2
TLS_DHE_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x97	TLS1.2
TLS_DHE_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x90	TLS1.2
TLS_DHE_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x91	TLS1.2
TLS_PSK_AES_128_CCM	0xC0 0xA4	TLS1.2
TLS_PSK_AES_256_CCM	0xC0 0xA5	TLS1.2
TLS_DHE_PSK_AES_128_CCM	0xC0 0xA6	TLS1.2
TLS_DHE_PSK_AES_256_CCM	0xC0 0xA7	TLS1.2
TLS_PSK_AES_128_CCM_8	0xC0 0xA8	TLS1.2
TLS_PSK_AES_256_CCM_8	0xC0 0xA9	TLS1.2
TLS_DHE_PSK_AES_128_CCM_8	0xC0 0xAA	TLS1.2
TLS_DHE_PSK_AES_256_CCM_8	0xC0 0xAB	TLS1.2
TLS_DHE_PSK_CHACHA20_POLY1305	0xCC 0xAD	TLS1.2
TLS_ECDHE_PSK_CHACHA20_POLY1305	0xCC 0xAC	TLS1.2
TLS_RSA_PSK_CHACHA20_POLY1305	0xCC 0xAE	TLS1.2
TLS_PSK_CHACHA20_POLY1305	0xCC 0xAB	TLS1.2
TLS_DH_ANON_ARCFOUR_128_MD5	0x00 0x18	TLS1.0
TLS_DH_ANON_3DES_EDE_CBC_SHA1	0x00 0x1B	TLS1.0
TLS_DH_ANON_AES_128_CBC_SHA1	0x00 0x34	TLS1.0
TLS_DH_ANON_AES_256_CBC_SHA1	0x00 0x3A	TLS1.0
TLS_DH_ANON_CAMELLIA_128_CBC_SHA256	0x00 0xBF	TLS1.2
TLS_DH_ANON_CAMELLIA_256_CBC_SHA256	0x00 0xC5	TLS1.2
TLS_DH_ANON_CAMELLIA_128_CBC_SHA1	0x00 0x46	TLS1.0
TLS_DH_ANON_CAMELLIA_256_CBC_SHA1	0x00 0x89	TLS1.0
TLS_DH_ANON_AES_128_CBC_SHA256	0x00 0x6C	TLS1.2
TLS_DH_ANON_AES_256_CBC_SHA256	0x00 0x6D	TLS1.2
TLS_DH_ANON_AES_128_GCM_SHA256	0x00 0xA6	TLS1.2
TLS_DH_ANON_AES_256_GCM_SHA384	0x00 0xA7	TLS1.2
TLS_DH_ANON_CAMELLIA_128_GCM_SHA256	0xC0 0x84	TLS1.2
TLS_DH_ANON_CAMELLIA_256_GCM_SHA384	0xC0 0x85	TLS1.2
TLS_ECDH_ANON_NULL_SHA1	0xC0 0x15	TLS1.0
TLS_ECDH_ANON_3DES_EDE_CBC_SHA1	0xC0 0x17	TLS1.0
TLS_ECDH_ANON_AES_128_CBC_SHA1	0xC0 0x18	TLS1.0

TLS_ECDH_ANON_AES_256_CBC_SHA1	0xC0 0x19	TLS1.0
TLS_ECDH_ANON_ARCFOUR_128_SHA1	0xC0 0x16	TLS1.0
TLS_SRP_SHA_3DES_EDE_CBC_SHA1	0xC0 0x1A	TLS1.0
TLS_SRP_SHA_AES_128_CBC_SHA1	0xC0 0x1D	TLS1.0
TLS_SRP_SHA_AES_256_CBC_SHA1	0xC0 0x20	TLS1.0
TLS_SRP_SHA_DSS_3DES_EDE_CBC_SHA1	0xC0 0x1C	TLS1.0
TLS_SRP_SHA_RSA_3DES_EDE_CBC_SHA1	0xC0 0x1B	TLS1.0
TLS_SRP_SHA_DSS_AES_128_CBC_SHA1	0xC0 0x1F	TLS1.0
TLS_SRP_SHA_RSA_AES_128_CBC_SHA1	0xC0 0x1E	TLS1.0
TLS_SRP_SHA_DSS_AES_256_CBC_SHA1	0xC0 0x22	TLS1.0
TLS_SRP_SHA_RSA_AES_256_CBC_SHA1	0xC0 0x21	TLS1.0
TLS_GOSTR341112_256_28147_CNT_IMIT	0xC1 0x02	TLS1.2

Certificate types

`X.509`

`Raw Public Key`

Protocols

`TLS1.0`

`TLS1.1`

`TLS1.2`

`TLS1.3`

`DTLS0.9`

`DTLS1.0`

`DTLS1.2`

Ciphers

AES-256-CBC
AES-192-CBC
AES-128-CBC
AES-128-GCM
AES-192-GCM
AES-256-GCM
AES-128-CCM
AES-256-CCM
AES-128-CCM-8
AES-256-CCM-8
ARCFOUR-128
ESTREAM-SALSA20-256
SALSA20-256
CHACHA20-32
CHACHA20-64
CAMELLIA-256-CBC
CAMELLIA-192-CBC
CAMELLIA-128-CBC
CHACHA20-POLY1305
CAMELLIA-128-GCM
CAMELLIA-256-GCM
GOST28147-TC26Z-CFB
GOST28147-CPA-CFB
GOST28147-CPB-CFB
GOST28147-CPC-CFB
GOST28147-CPD-CFB
AES-128-CFB8
AES-192-CFB8
AES-256-CFB8
AES-128-XTS
AES-256-XTS
AES-128-SIV
AES-256-SIV
GOST28147-TC26Z-CNT
3DES-CBC

DES-CBC

RC2-40

NULL

MAC algorithms

SHA1

SHA256

SHA384

SHA512

SHA224

UMAC-96

UMAC-128

AEAD

MD5

GOSTR341194
STREEBOG-256
STREEBOG-512
AES-CMAC-128
AES-CMAC-256
AES-GMAC-128
AES-GMAC-192
AES-GMAC-256
GOST28147-TC26Z-IMIT

Key exchange methods

ECDHE-RSA
ECDHE-ECDSA
RSA

DHE-RSA

DHE-DSS

PSK

RSA-PSK

DHE-PSK

ECDHE-PSK
SRP-DSS

SRP-RSA

SRP

ANON-DH

ANON-ECDH
VKO-GOST-12
RSA-EXPORT

Public key algorithms

RSA

RSA-PSS

RSA

DSA

GOST R 34.10-2012-512
GOST R 34.10-2012-256
GOST R 34.10-2001
EC/ECDSA

EdDSA (Ed25519)
EdDSA (Ed448)
DH

ECDH (X25519)
ECDH (X448)

Public key signature algorithms

RSA-SHA256
RSA-SHA384
RSA-SHA512
RSA-PSS-SHA256
RSA-PSS-RSAE-SHA256
RSA-PSS-SHA384
RSA-PSS-RSAE-SHA384
RSA-PSS-SHA512
RSA-PSS-RSAE-SHA512
EdDSA-Ed25519
EdDSA-Ed448
ECDSA-SHA256
ECDSA-SHA384
ECDSA-SHA512
ECDSA-SECP256R1-SHA256
ECDSA-SECP384R1-SHA384
ECDSA-SECP521R1-SHA512
ECDSA-SHA3-224
ECDSA-SHA3-256
ECDSA-SHA3-384
ECDSA-SHA3-512
RSA-SHA3-224
RSA-SHA3-256
RSA-SHA3-384
RSA-SHA3-512
DSA-SHA3-224
DSA-SHA3-256
DSA-SHA3-384
DSA-SHA3-512
RSA-RAW

RSA-SHA1

RSA-SHA1

RSA-SHA224
RSA-RMD160
DSA-SHA1

DSA-SHA1

DSA-SHA224
DSA-SHA256
RSA-MD5

RSA-MD5

RSA-MD2

ECDSA-SHA1
ECDSA-SHA224
GOSTR341012-512
GOSTR341012-256
GOSTR341001
DSA-SHA384
DSA-SHA512

Groups

SECP256R1
SECP384R1
SECP521R1
X25519

GC256B

GC512A

X448

FFDHE2048
FFDHE3072
FFDHE4096
FFDHE6144
FFDHE8192

Appendix E API reference

E.1 Core TLS API

The prototypes for the following functions lie in `gnutls/gnutls.h`.

gnutls_alert_get

`gnutls_alert_description_t gnutls_alert_get (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the last alert number received. This function should be called when `GNUTLS_E_WARNING_ALERT_RECEIVED` or `GNUTLS_E_FATAL_ALERT_RECEIVED` errors are returned by a gnutls function. The peer may send alerts if he encounters an error. If no alert has been received the returned value is undefined.

Returns: the last alert received, a `gnutls_alert_description_t` value.

gnutls_alert_get_name

`const char * gnutls_alert_get_name (`*gnutls_alert_description_t* `alert`) [Function]

alert: is an alert number.

This function will return a string that describes the given alert number, or `NULL` . See `gnutls_alert_get()` .

Returns: string corresponding to `gnutls_alert_description_t` value.

gnutls_alert_get_strname

`const char * gnutls_alert_get_strname` [Function]
(*gnutls_alert_description_t* `alert`)

alert: is an alert number.

This function will return a string of the name of the alert.

Returns: string corresponding to `gnutls_alert_description_t` value.

Since: 3.0

gnutls_alert_send

`int gnutls_alert_send (`*gnutls_session_t* `session`, [Function]
gnutls_alert_level_t `level`, *gnutls_alert_description_t* `desc`)

session: is a `gnutls_session_t` type.

level: is the level of the alert

desc: is the alert description

This function will send an alert to the peer in order to inform him of something important (eg. his Certificate could not be verified). If the alert level is Fatal then the peer is expected to close the connection, otherwise he may ignore the alert and continue.

The error code of the underlying record send function will be returned, so you may also receive GNUTLS_E_INTERRUPTED or GNUTLS_E_AGAIN as well.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise an error code is returned.

gnutls_alert_send_appropriate

`int gnutls_alert_send_appropriate (`*gnutls_session_t* ***session***, *int* ***err***`)` [Function]

session: is a gnutls_session_t type.

err: is an error code returned by another GnuTLS function

Sends an alert to the peer depending on the error code returned by a gnutls function. This function will call gnutls_error_to_alert() to determine the appropriate alert to send.

This function may also return GNUTLS_E_AGAIN , or GNUTLS_E_INTERRUPTED .

This function historically was always sending an alert to the peer, even if err was inappropriate to respond with an alert (e.g., GNUTLS_E_SUCCESS). Since 3.6.6 this function returns success without transmitting any data on error codes that should not result to an alert.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise an error code is returned.

gnutls_alpn_get_selected_protocol

`int gnutls_alpn_get_selected_protocol (`*gnutls_session_t* ***session***, *gnutls_datum_t ** ***protocol***`)` [Function]

session: is a gnutls_session_t type.

protocol: will hold the protocol name

This function allows you to get the negotiated protocol name. The returned protocol should be treated as opaque, constant value and only valid during the session life.

The selected protocol is the first supported by the list sent by the client.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code is returned.

Since 3.2.0

gnutls_alpn_set_protocols

`int gnutls_alpn_set_protocols (`*gnutls_session_t* ***session***, *const gnutls_datum_t ** ***protocols***, *unsigned* ***protocols_size***, *unsigned int* ***flags***`)` [Function]

session: is a gnutls_session_t type.

protocols: is the protocol names to add.

protocols_size: the number of protocols to add.

flags: zero or a sequence of gnutls_alpn_flags_t

This function is to be used by both clients and servers, to declare the supported ALPN protocols, which are used during negotiation with peer.

See `gnutls_alpn_flags_t` description for the documentation of available flags.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.2.0

gnutls_anon_allocate_client_credentials

`int gnutls_anon_allocate_client_credentials` [Function]
(*gnutls_anon_client_credentials_t * sc*)

sc: is a pointer to a `gnutls_anon_client_credentials_t` type.

Allocate a gnutls_anon_client_credentials_t structure.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_anon_allocate_server_credentials

`int gnutls_anon_allocate_server_credentials` [Function]
(*gnutls_anon_server_credentials_t * sc*)

sc: is a pointer to a `gnutls_anon_server_credentials_t` type.

Allocate a gnutls_anon_server_credentials_t structure.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_anon_free_client_credentials

`void gnutls_anon_free_client_credentials` [Function]
(*gnutls_anon_client_credentials_t sc*)

sc: is a `gnutls_anon_client_credentials_t` type.

Free a gnutls_anon_client_credentials_t structure.

gnutls_anon_free_server_credentials

`void gnutls_anon_free_server_credentials` [Function]
(*gnutls_anon_server_credentials_t sc*)

sc: is a `gnutls_anon_server_credentials_t` type.

Free a gnutls_anon_server_credentials_t structure.

gnutls_anon_set_params_function

`void gnutls_anon_set_params_function` [Function]
(*gnutls_anon_server_credentials_t res, gnutls_params_function * func*)

res: is a gnutls_anon_server_credentials_t type

func: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for anonymous authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_anon_set_server_dh_params

`void gnutls_anon_set_server_dh_params` [Function]
(*gnutls_anon_server_credentials_t* `res`, *gnutls_dh_params_t* `dh_params`)

res: is a gnutls_anon_server_credentials_t type

dh_params: The Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for an anonymous server to use. These parameters will be used in Anonymous Diffie-Hellman cipher suites.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_anon_set_server_known_dh_params

`int gnutls_anon_set_server_known_dh_params` [Function]
(*gnutls_anon_server_credentials_t* `res`, *gnutls_sec_param_t* `sec_param`)

res: is a gnutls_anon_server_credentials_t type

sec_param: is an option of the `gnutls_sec_param_t` enumeration

This function will set the Diffie-Hellman parameters for an anonymous server to use. These parameters will be used in Anonymous Diffie-Hellman cipher suites and will be selected from the FFDHE set of RFC7919 according to the security level provided.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.6

gnutls_anon_set_server_params_function

`void gnutls_anon_set_server_params_function` [Function]
(*gnutls_anon_server_credentials_t* `res`, *gnutls_params_function* * `func`)

res: is a gnutls_certificate_credentials_t type

func: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman parameters for anonymous authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_anti_replay_deinit

`void gnutls_anti_replay_deinit` (*gnutls_anti_replay_t* `anti_replay`) [Function]

anti_replay: is a `gnutls_anti_replay` type

This function will deinitialize all resources occupied by the given anti-replay context.

Since: 3.6.5

gnutls_anti_replay_enable

`void gnutls_anti_replay_enable (`*gnutls_session_t* ***session***, *gnutls_anti_replay_t* ***anti_replay***`)` [Function]

session: is a `gnutls_session_t` type.

anti_replay: is a `gnutls_anti_replay_t` type.

Request that the server should use anti-replay mechanism.

Since: 3.6.5

gnutls_anti_replay_init

`int gnutls_anti_replay_init (`*gnutls_anti_replay_t ** ***anti_replay***`)` [Function]

anti_replay: is a pointer to `gnutls_anti_replay_t` type

This function will allocate and initialize the `anti_replay` context to be usable for detect replay attacks. The context can then be attached to a `gnutls_session_t` with `gnutls_anti_replay_enable()` .

Returns: Zero or a negative error code on error.

Since: 3.6.5

gnutls_anti_replay_set_add_function

`void gnutls_anti_replay_set_add_function (`*gnutls_anti_replay_t* ***anti_replay***, *gnutls_db_add_func* ***add_func***`)` [Function]

anti_replay: is a `gnutls_anti_replay_t` type.

add_func: is the function.

Sets the function that will be used to store an entry if it is not already present in the resumed sessions database. This function returns 0 if the entry is successfully stored, and a negative error code otherwise. In particular, if the entry is found in the database, it returns `GNUTLS_E_DB_ENTRY_EXISTS` .

The arguments to the `add_func` are: - `ptr` : the pointer set with `gnutls_anti_replay_set_ptr()` - `exp_time` : the expiration time of the entry - `key` : a pointer to the key - `data` : a pointer to data to store

The data set by this function can be examined using `gnutls_db_check_entry_expire_time()` and `gnutls_db_check_entry_time()` .

Since: 3.6.5

gnutls_anti_replay_set_ptr

`void gnutls_anti_replay_set_ptr (`*gnutls_anti_replay_t* ***anti_replay***, *void ** ***ptr***`)` [Function]

anti_replay: is a `gnutls_anti_replay_t` type.

ptr: is the pointer

Sets the pointer that will be provided to db add function as the first argument.

gnutls_anti_replay_set_window

`void gnutls_anti_replay_set_window (`*gnutls_anti_replay_t* `anti_replay`, *unsigned int* `window``)` [Function]

anti_replay: is a `gnutls_anti_replay_t` type.

window: is the time window recording ClientHello, in milliseconds

Sets the time window used for ClientHello recording. In order to protect against replay attacks, the server records ClientHello messages within this time period from the last update, and considers it a replay when a ClientHello outside of the period; if a ClientHello arrives within this period, the server checks the database and detects duplicates.

For the details of the algorithm, see RFC 8446, section 8.2.

Since: 3.6.5

gnutls_auth_client_get_type

`gnutls_credentials_type_t gnutls_auth_client_get_type (`*gnutls_session_t* `session``)` [Function]

session: is a `gnutls_session_t` type.

Returns the type of credentials that were used for client authentication. The returned information is to be used to distinguish the function used to access authentication data.

Note that on resumed sessions, this function returns the schema used in the original session authentication.

Returns: The type of credentials for the client authentication schema, a `gnutls_credentials_type_t` type.

gnutls_auth_get_type

`gnutls_credentials_type_t gnutls_auth_get_type (`*gnutls_session_t* `session``)` [Function]

session: is a `gnutls_session_t` type.

Returns type of credentials for the current authentication schema. The returned information is to be used to distinguish the function used to access authentication data.

Eg. for CERTIFICATE ciphersuites (key exchange algorithms: `GNUTLS_KX_RSA` , `GNUTLS_KX_DHE_RSA`), the same function are to be used to access the authentication data.

Note that on resumed sessions, this function returns the schema used in the original session authentication.

Returns: The type of credentials for the current authentication schema, a `gnutls_credentials_type_t` type.

gnutls_auth_server_get_type

`gnutls_credentials_type_t gnutls_auth_server_get_type` [Function]
(*gnutls_session_t* `session`)

session: is a `gnutls_session_t` type.

Returns the type of credentials that were used for server authentication. The returned information is to be used to distinguish the function used to access authentication data.

Note that on resumed sessions, this function returns the schema used in the original session authentication.

Returns: The type of credentials for the server authentication schema, a `gnutls_credentials_type_t` type.

gnutls_base64_decode2

`int gnutls_base64_decode2` (*const gnutls_datum_t ** `base64`, [Function]
*gnutls_datum_t ** `result`)

base64: contains the encoded data

result: the location of decoded data

This function will decode the given base64 encoded data. The decoded data will be allocated, and stored into result.

You should use `gnutls_free()` to free the returned data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_base64_encode2

`int gnutls_base64_encode2` (*const gnutls_datum_t ** `data`, [Function]
*gnutls_datum_t ** `result`)

data: contains the raw data

result: will hold the newly allocated encoded data

This function will convert the given data to printable data, using the base64 encoding. This function will allocate the required memory to hold the encoded data.

You should use `gnutls_free()` to free the returned data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_buffer_append_data

`int gnutls_buffer_append_data` (*gnutls_buffer_t* `dest`, *const void* [Function]
*** `data`, *size_t* `data_size`)

dest: the buffer to append to

data: the data

data_size: the size of `data`

Appends the provided `data` to the destination buffer.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_bye

`int gnutls_bye (`*gnutls_session_t* `session`, *gnutls_close_request_t* `how`) [Function]

session: is a `gnutls_session_t` type.

how: is an integer

Terminates the current TLS/SSL connection. The connection should have been initiated using `gnutls_handshake()` . `how` should be one of `GNUTLS_SHUT_RDWR` , `GNUTLS_SHUT_WR` .

In case of `GNUTLS_SHUT_RDWR` the TLS session gets terminated and further receives and sends will be disallowed. If the return value is zero you may continue using the underlying transport layer. `GNUTLS_SHUT_RDWR` sends an alert containing a close request and waits for the peer to reply with the same message.

In case of `GNUTLS_SHUT_WR` the TLS session gets terminated and further sends will be disallowed. In order to reuse the connection you should wait for an EOF from the peer. `GNUTLS_SHUT_WR` sends an alert containing a close request.

Note that not all implementations will properly terminate a TLS connection. Some of them, usually for performance reasons, will terminate only the underlying transport layer, and thus not distinguishing between a malicious party prematurely terminating the connection and normal termination.

This function may also return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` ; cf. `gnutls_record_get_direction()` .

Returns: `GNUTLS_E_SUCCESS` on success, or an error code, see function documentation for entire semantics.

gnutls_certificate_activation_time_peers

`time_t gnutls_certificate_activation_time_peers (`*gnutls_session_t* `session`) [Function]

session: is a gnutls session

This function will return the peer's certificate activation time.

Returns: (time_t)-1 on error.

Deprecated: `gnutls_certificate_verify_peers2()` now verifies activation times.

gnutls_certificate_allocate_credentials

`int gnutls_certificate_allocate_credentials (`*gnutls_certificate_credentials_t* * `res`) [Function]

res: is a pointer to a `gnutls_certificate_credentials_t` type.

Allocate a gnutls_certificate_credentials_t structure.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_certificate_client_get_request_status

`unsigned gnutls_certificate_client_get_request_status` [Function]
(*gnutls_session_t* **session**)

session: is a gnutls session

Get whether client certificate was requested on the last handshake or not.

Returns: 0 if the peer (server) did not request client authentication or 1 otherwise.

gnutls_certificate_expiration_time_peers

`time_t gnutls_certificate_expiration_time_peers` [Function]
(*gnutls_session_t* **session**)

session: is a gnutls session

This function will return the peer's certificate expiration time.

Returns: (time_t)-1 on error.

Deprecated: `gnutls_certificate_verify_peers2()` now verifies expiration times.

gnutls_certificate_free_ca_names

`void gnutls_certificate_free_ca_names` [Function]
(*gnutls_certificate_credentials_t* **sc**)

sc: is a `gnutls_certificate_credentials_t` type.

This function will delete all the CA name in the given credentials. Clients may call this to save some memory since in client side the CA names are not used. Servers might want to use this function if a large list of trusted CAs is present and sending the names of it would just consume bandwidth without providing information to client.

CA names are used by servers to advertise the CAs they support to clients.

gnutls_certificate_free_cas

`void gnutls_certificate_free_cas` [Function]
(*gnutls_certificate_credentials_t* **sc**)

sc: is a `gnutls_certificate_credentials_t` type.

This function was operational on very early versions of gnutls. Due to internal refactorings and the fact that this was hardly ever used, it is currently a no-op.

gnutls_certificate_free_credentials

`void gnutls_certificate_free_credentials` [Function]
(*gnutls_certificate_credentials_t* **sc**)

sc: is a `gnutls_certificate_credentials_t` type.

Free a gnutls_certificate_credentials_t structure.

This function does not free any temporary parameters associated with this structure (ie RSA and DH parameters are not freed by this function).

gnutls_certificate_free_crls

`void gnutls_certificate_free_crls` [Function]
(*gnutls_certificate_credentials_t* `sc`)

sc: is a `gnutls_certificate_credentials_t` type.

This function will delete all the CRLs associated with the given credentials.

gnutls_certificate_free_keys

`void gnutls_certificate_free_keys` [Function]
(*gnutls_certificate_credentials_t* `sc`)

sc: is a `gnutls_certificate_credentials_t` type.

This function will delete all the keys and the certificates associated with the given credentials. This function must not be called when a TLS negotiation that uses the credentials is in progress.

gnutls_certificate_get_crt_raw

`int gnutls_certificate_get_crt_raw` [Function]
(*gnutls_certificate_credentials_t* `sc`, *unsigned* `idx1`, *unsigned* `idx2`, *gnutls_datum_t ** `cert`)

sc: is a `gnutls_certificate_credentials_t` type.

idx1: the index of the certificate chain if multiple are present

idx2: the index of the certificate in the chain. Zero gives the server's certificate.

cert: Will hold the DER encoded certificate.

This function will return the DER encoded certificate of the server or any other certificate on its certificate chain (based on `idx2`). The returned data should be treated as constant and only accessible during the lifetime of `sc` . The `idx1` matches the value `gnutls_certificate_set_x509_key()` and friends functions.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. In case the indexes are out of bounds `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 3.2.5

gnutls_certificate_get_issuer

`int gnutls_certificate_get_issuer` [Function]
(*gnutls_certificate_credentials_t* `sc`, *gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t ** `issuer`, *unsigned int* `flags`)

sc: is a `gnutls_certificate_credentials_t` type.

cert: is the certificate to find issuer for

issuer: Will hold the issuer if any. Should be treated as constant.

flags: Use zero or `GNUTLS_TL_GET_COPY`

This function will return the issuer of a given certificate. If the flag `GNUTLS_TL_GET_COPY` is specified a copy of the issuer will be returned which must be freed using

gnutls_x509_crt_deinit() . In that case the provided issuer must not be initialized.

As with gnutls_x509_trust_list_get_issuer() this function requires the GNUTLS_TL_GET_COPY flag in order to operate with PKCS11 trust lists in a thread-safe way.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_certificate_get_ocsp_expiration

time_t gnutls_certificate_get_ocsp_expiration (*gnutls_certificate_credentials_t* sc, *unsigned* idx, *int* oidx, *unsigned* flags) [Function]

sc: is a credentials structure.

idx: is a certificate chain index as returned by gnutls_certificate_set_key() and friends

oidx: is an OCSP response index

flags: should be zero

This function returns the validity of the loaded OCSP responses, to provide information on when to reload/refresh them.

Note that the credentials structure should be read-only when in use, thus when reloading, either the credentials structure must not be in use by any sessions, or a new credentials structure should be allocated for new sessions.

When oidx is (-1) then the minimum refresh time for all responses is returned. Otherwise the index specifies the response corresponding to the odix certificate in the certificate chain.

Returns: On success, the expiration time of the OCSP response. Otherwise (time_t)(-1) on error, or (time_t)-2 on out of bounds.

Since: 3.6.3

gnutls_certificate_get_ours

const gnutls_datum_t * gnutls_certificate_get_ours (*gnutls_session_t* session) [Function]

session: is a gnutls session

Gets the certificate as sent to the peer in the last handshake. The certificate is in raw (DER) format. No certificate list is being returned. Only the first certificate.

This function returns the certificate that was sent in the current handshake. In subsequent resumed sessions this function will return NULL . That differs from gnutls_certificate_get_peers() which always returns the peer's certificate used in the original session.

Returns: a pointer to a gnutls_datum_t containing our certificate, or NULL in case of an error or if no certificate was used.

gnutls_certificate_get_peers

`const gnutls_datum_t * gnutls_certificate_get_peers` [Function]
(*gnutls_session_t* `session`, *unsigned int* * `list_size`)

session: is a gnutls session

list_size: is the length of the certificate list (may be `NULL`)

Get the peer's raw certificate (chain) as sent by the peer. These certificates are in raw format (DER encoded for X.509). In case of a X.509 then a certificate list may be present. The list is provided as sent by the server; the server must send as first certificate in the list its own certificate, following the issuer's certificate, then the issuer's issuer etc. However, there are servers which violate this principle and thus on certain occasions this may be an unsorted list.

In resumed sessions, this function will return the peer's certificate list as used in the first/original session.

Returns: a pointer to a `gnutls_datum_t` containing the peer's certificates, or `NULL` in case of an error or if no certificate was used.

gnutls_certificate_get_peers_subkey_id

`int gnutls_certificate_get_peers_subkey_id (`*gnutls_session_t* `session`, *gnutls_datum_t* * `id`) [Function]

session: is a gnutls session

id: will contain the ID

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.1.3

gnutls_certificate_get_verify_flags

`unsigned int gnutls_certificate_get_verify_flags` [Function]
(*gnutls_certificate_credentials_t* `res`)

res: is a gnutls_certificate_credentials_t type

Returns the verification flags set with `gnutls_certificate_set_verify_flags()` .

Returns: The certificate verification flags used by `res` .

Since: 3.4.0

gnutls_certificate_get_x509_crt

`int gnutls_certificate_get_x509_crt` [Function]
(*gnutls_certificate_credentials_t* `res`, *unsigned* `index`, *gnutls_x509_crt_t* ** `crt_list`, *unsigned* * `crt_list_size`)

res: is a `gnutls_certificate_credentials_t` type.

index: The index of the certificate list to obtain.

crt_list: Where to store the certificate list.

crt_list_size: Will hold the number of certificates.

Obtains a X.509 certificate list that has been stored in `res` with one of `gnutls_certificate_set_x509_key()` , `gnutls_certificate_set_key()` , `gnutls_certificate_set_x509_key_file()` , `gnutls_certificate_set_x509_key_file2()` , `gnutls_certificate_set_x509_key_mem()` , or `gnutls_certificate_set_x509_key_mem2()` . Each certificate in the returned certificate list must be deallocated with `gnutls_x509_crt_deinit()` , and the list itself must be freed with `gnutls_free()` .

The `index` matches the return value of `gnutls_certificate_set_x509_key()` and friends functions, when the `GNUTLS_CERTIFICATE_API_V2` flag is set.

If there is no certificate with the given index, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned. If the certificate with the given index is not a X.509 certificate, `GNUTLS_E_INVALID_REQUEST` is returned. The returned certificates must be deinitialized after use, and the `crt_list` pointer must be freed using `gnutls_free()` .

Returns: `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

Since: 3.4.0

gnutls_certificate_get_x509_key

`int gnutls_certificate_get_x509_key` [Function]
(*gnutls_certificate_credentials_t* `res`, *unsigned* `index`, *gnutls_x509_privkey_t ** `key`)

res: is a `gnutls_certificate_credentials_t` type.

index: The index of the key to obtain.

key: Location to store the key.

Obtains a X.509 private key that has been stored in `res` with one of `gnutls_certificate_set_x509_key()` , `gnutls_certificate_set_key()` , `gnutls_certificate_set_x509_key_file()` , `gnutls_certificate_set_x509_key_file2()` , `gnutls_certificate_set_x509_key_mem()` , or `gnutls_certificate_set_x509_key_mem2()` . The returned key must be deallocated with `gnutls_x509_privkey_deinit()` when no longer needed.

The `index` matches the return value of `gnutls_certificate_set_x509_key()` and friends functions, when the `GNUTLS_CERTIFICATE_API_V2` flag is set.

If there is no key with the given index, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned. If the key with the given index is not a X.509 key, `GNUTLS_E_INVALID_REQUEST` is returned.

Returns: `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

Since: 3.4.0

gnutls_certificate_send_x509_rdn_sequence

`void gnutls_certificate_send_x509_rdn_sequence` [Function]
(*gnutls_session_t* `session`, *int* `status`)

session: a `gnutls_session_t` type.

status: is 0 or 1

If status is non zero, this function will order gnutls not to send the rdnSequence in the certificate request message. That is the server will not advertise its trusted CAs to the peer. If status is zero then the default behaviour will take effect, which is to advertise the server's trusted CAs.

This function has no effect in clients, and in authentication methods other than certificate with X.509 certificates.

gnutls_certificate_server_set_request

`void gnutls_certificate_server_set_request` (*gnutls_session_t* `session`, *gnutls_certificate_request_t* `req`) [Function]

session: is a `gnutls_session_t` type.

req: is one of GNUTLS_CERT_REQUEST, GNUTLS_CERT_REQUIRE, GNUTLS_CERT_IGNORE

This function specifies if we (in case of a server) are going to send a certificate request message to the client. If `req` is GNUTLS_CERT_REQUIRE then the server will return the `GNUTLS_E_NO_CERTIFICATE_FOUND` error if the peer does not provide a certificate. If you do not call this function then the client will not be asked to send a certificate. Invoking the function with `req` GNUTLS_CERT_IGNORE has the same effect.

gnutls_certificate_set_dh_params

`void gnutls_certificate_set_dh_params` (*gnutls_certificate_credentials_t* `res`, *gnutls_dh_params_t* `dh_params`) [Function]

res: is a gnutls_certificate_credentials_t type

dh_params: the Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for a certificate server to use. These parameters will be used in Ephemeral Diffie-Hellman cipher suites. Note that only a pointer to the parameters are stored in the certificate handle, so you must not deallocate the parameters before the certificate is deallocated.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_certificate_set_flags

`void gnutls_certificate_set_flags` (*gnutls_certificate_credentials_t* `res`, *unsigned int* `flags`) [Function]

res: is a gnutls_certificate_credentials_t type

flags: are the flags of `gnutls_certificate_flags` type

This function will set flags to tweak the operation of the credentials structure. See the `gnutls_certificate_flags` enumerations for more information on the available flags.

Since: 3.4.7

gnutls_certificate_set_known_dh_params

`int gnutls_certificate_set_known_dh_params` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_sec_param_t* `sec_param`)

res: is a gnutls_certificate_credentials_t type

sec_param: is an option of the `gnutls_sec_param_t` enumeration

This function will set the Diffie-Hellman parameters for a certificate server to use. These parameters will be used in Ephemeral Diffie-Hellman cipher suites and will be selected from the FFDHE set of RFC7919 according to the security level provided.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.6

gnutls_certificate_set_ocsp_status_request_file

`int gnutls_certificate_set_ocsp_status_request_file` [Function]
(*gnutls_certificate_credentials_t* `sc`, *const char ** `response_file`, *unsigned* `idx`)

sc: is a credentials structure.

response_file: a filename of the OCSP response

idx: is a certificate index as returned by `gnutls_certificate_set_key()` and friends

This function loads the provided OCSP response. It will be sent to the client if requests an OCSP certificate status for the certificate chain specified by `idx` .

Note: the ability to set multiple OCSP responses per credential structure via the index `idx` was added in version 3.5.6. To keep backwards compatibility, it requires using `gnutls_certificate_set_flags()` with the `GNUTLS_CERTIFICATE_API_V2` flag to make the set certificate functions return an index usable by this function.

This function can be called multiple times since GnuTLS 3.6.3 when multiple responses which apply to the chain are available. If the response provided does not match any certificates present in the chain, the code `GNUTLS_E_OCSP_MISMATCH_WITH_CERTS` is returned. To revert to the previous behavior set the flag `GNUTLS_CERTIFICATE_SKIP_OCSP_RESPONSE_CHECK` in the certificate credentials structure. In that case, only the end-certificate's OCSP response can be set. If the response is already expired at the time of loading the code `GNUTLS_E_EXPIRED` is returned.

To revert to the previous behavior of this function which does not return any errors, set the flag `GNUTLS_CERTIFICATE_SKIP_OCSP_RESPONSE_CHECK`

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.1.3

gnutls_certificate_set_ocsp_status_request_file2

`int gnutls_certificate_set_ocsp_status_request_file2` [Function]
(*gnutls_certificate_credentials_t* **sc**, *const char ** **response_file**, *unsigned* **idx**, *gnutls_x509_crt_fmt_t* **fmt**)

sc: is a credentials structure.

response_file: a filename of the OCSP response

idx: is a certificate index as returned by `gnutls_certificate_set_key()` and friends

fmt: is PEM or DER

This function loads the OCSP responses to be sent to the peer for the certificate chain specified by `idx` . When `fmt` is set to PEM, multiple responses can be loaded.

This function must be called after setting any certificates, and cannot be used for certificates that are provided via a callback – that is when `gnutls_certificate_set_retrieve_function()` is used. In that case consider using `gnutls_certificate_set_retrieve_function3()` .

This function can be called multiple times when multiple responses applicable to the certificate chain are available. If the response provided does not match any certificates present in the chain, the code `GNUTLS_E_OCSP_MISMATCH_WITH_CERTS` is returned. If the response is already expired at the time of loading the code `GNUTLS_E_EXPIRED` is returned.

Returns: On success, the number of loaded responses is returned, otherwise a negative error code.

Since: 3.1.3

gnutls_certificate_set_ocsp_status_request_function

`void gnutls_certificate_set_ocsp_status_request_function` [Function]
(*gnutls_certificate_credentials_t* **sc**, *gnutls_status_request_ocsp_func* **ocsp_func**, *void ** **ptr**)

sc: is a `gnutls_certificate_credentials_t` type.

ocsp_func: function pointer to OCSP status request callback.

ptr: opaque pointer passed to callback function

This function is to be used by server to register a callback to handle OCSP status requests from the client. The callback will be invoked if the client supplied a status-request OCSP extension. The callback function prototype is:

typedef int (*gnutls_status_request_ocsp_func) (gnutls_session_t session, void *ptr, gnutls_datum_t *ocsp_response);

The callback will be invoked if the client requests an OCSP certificate status. The callback may return `GNUTLS_E_NO_CERTIFICATE_STATUS` , if there is no recent OCSP response. If the callback returns `GNUTLS_E_SUCCESS` , it is expected to have the `ocsp_response` field set with a valid (DER-encoded) OCSP response. The response must be a value allocated using `gnutls_malloc()` , and will be deinitialized by the caller.

It is possible to set a specific callback for each provided certificate using `gnutls_certificate_set_ocsp_status_request_function2()` .

Since: 3.1.3

gnutls_certificate_set_ocsp_status_request_function2

`int gnutls_certificate_set_ocsp_status_request_function2` [Function]
(*gnutls_certificate_credentials_t* `sc`, *unsigned* `idx`, *gnutls_status_request_ocsp_func* `ocsp_func`, *void * *`ptr`)

sc: is a `gnutls_certificate_credentials_t` type.

idx: is a certificate index as returned by `gnutls_certificate_set_key()` and friends

ocsp_func: function pointer to OCSP status request callback.

ptr: opaque pointer passed to callback function

This function is to be used by server to register a callback to provide OCSP status requests that correspond to the indexed certificate chain from the client. The callback will be invoked if the client supplied a status-request OCSP extension.

The callback function prototype is:

typedef int (*gnutls_status_request_ocsp_func) (gnutls_session_t session, void *ptr, gnutls_datum_t *ocsp_response);

The callback will be invoked if the client requests an OCSP certificate status. The callback may return `GNUTLS_E_NO_CERTIFICATE_STATUS` , if there is no recent OCSP response. If the callback returns `GNUTLS_E_SUCCESS` , it is expected to have the `ocsp_response` field set with a valid (DER-encoded) OCSP response. The response must be a value allocated using `gnutls_malloc()` , and will be deinitialized by the caller.

Note: the ability to set multiple OCSP responses per credential structure via the index `idx` was added in version 3.5.6. To keep backwards compatibility, it requires using `gnutls_certificate_set_flags()` with the `GNUTLS_CERTIFICATE_API_V2` flag to make the set certificate functions return an index usable by this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.5.5

gnutls_certificate_set_ocsp_status_request_mem

`int gnutls_certificate_set_ocsp_status_request_mem` [Function]
(*gnutls_certificate_credentials_t* `sc`, *const gnutls_datum_t * *`resp_data`, *unsigned* `idx`, *gnutls_x509_crt_fmt_t* `fmt`)

sc: is a credentials structure.

resp_data: a memory buffer holding an OCSP response

idx: is a certificate index as returned by `gnutls_certificate_set_key()` and friends

fmt: is PEM or DER

This function sets the OCSP responses to be sent to the peer for the certificate chain specified by `idx` . When `fmt` is set to PEM, multiple responses can be loaded.

Note: the ability to set multiple OCSP responses per credential structure via the index `idx` was added in version 3.5.6. To keep backwards compatibility, it requires using `gnutls_certificate_set_flags()` with the `GNUTLS_CERTIFICATE_API_V2` flag to make the set certificate functions return an index usable by this function.

This function must be called after setting any certificates, and cannot be used for certificates that are provided via a callback – that is when `gnutls_certificate_set_retrieve_function()` is used.

This function can be called multiple times when multiple responses which apply to the certificate chain are available. If the response provided does not match any certificates present in the chain, the code `GNUTLS_E_OCSP_MISMATCH_WITH_CERTS` is returned. If the response is already expired at the time of loading the code `GNUTLS_E_EXPIRED` is returned.

Returns: On success, the number of loaded responses is returned, otherwise a negative error code.

Since: 3.6.3

gnutls_certificate_set_params_function

`void gnutls_certificate_set_params_function` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_params_function* * `func`)

res: is a gnutls_certificate_credentials_t type

func: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for certificate authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_certificate_set_pin_function

`void gnutls_certificate_set_pin_function` [Function]
(*gnutls_certificate_credentials_t* `cred`, *gnutls_pin_callback_t* `fn`, *void* * `userdata`)

cred: is a `gnutls_certificate_credentials_t` type.

fn: A PIN callback

userdata: Data to be passed in the callback

This function will set a callback function to be used when required to access a protected object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

Since: 3.1.0

gnutls_certificate_set_rawpk_key_file

`int gnutls_certificate_set_rawpk_key_file` [Function]
(*gnutls_certificate_credentials_t* `cred`, *const char** `rawpkfile`, *const char** `privkeyfile`, *gnutls_x509_crt_fmt_t* `format`, *const char ** `pass`, *unsigned int* `key_usage`, *const char ** `names`, *unsigned int* `names_length`, *unsigned int* `privkey_flags`, *unsigned int* `pkcs11_flags`)

cred: is a `gnutls_certificate_credentials_t` type.

rawpkfile: contains a raw public key in PKIX.SubjectPublicKeyInfo format.

privkeyfile: contains a file path to a private key.

format: encoding of the keys. DER or PEM.

pass: an optional password to unlock the private key privkeyfile.

key_usage: an ORed sequence of `GNUTLS_KEY_` * flags.

names: is an array of DNS names belonging to the public-key (NULL if none).

names_length: holds the length of the names list.

privkey_flags: an ORed sequence of `gnutls_pkcs_encrypt_flags_t` . These apply to the private key pkey.

pkcs11_flags: one of gnutls_pkcs11_obj_flags. These apply to URLs.

This function sets a public/private keypair read from file in the `gnutls_certificate_credentials_t` type to be used for authentication and/or encryption. `spki` and `privkey` should match otherwise set signatures cannot be validated. In case of no match this function returns `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` . This function should be called once for the client because there is currently no mechanism to determine which raw public-key to select for the peer when there are multiple present. Multiple raw public keys for the server can be distinghuished by setting the `names` .

Note here that `spki` is a raw public-key as defined in RFC7250. It means that there is no surrounding certificate that holds the public key and that there is therefore no direct mechanism to prove the authenticity of this key. The keypair can be used during a TLS handshake but its authenticity should be established via a different mechanism (e.g. TOFU or known fingerprint).

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format and will be autodetected.

If the raw public-key and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

Key usage (as defined by X.509 extension (2.5.29.15)) can be explicitly set because there is no certificate structure around the key to define this value. See for more info `gnutls_x509_crt_get_key_usage()` .

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used in other functions to refer to the added key-pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, in case the key pair does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned, in other erroneous cases a different negative error code is returned.

Since: 3.6.6

gnutls_certificate_set_rawpk_key_mem

`int gnutls_certificate_set_rawpk_key_mem` [Function]
(*gnutls_certificate_credentials_t* `cred`, *const gnutls_datum_t** `spki`, *const gnutls_datum_t** `pkey`, *gnutls_x509_crt_fmt_t* `format`, *const char** `pass`, *unsigned int* `key_usage`, *const char ** `names`, *unsigned int* `names_length`, *unsigned int* `flags`)

cred: is a `gnutls_certificate_credentials_t` type.

spki: contains a raw public key in PKIX.SubjectPublicKeyInfo format.

pkey: contains a raw private key.

format: encoding of the keys. DER or PEM.

pass: an optional password to unlock the private key pkey.

key_usage: An ORed sequence of `GNUTLS_KEY_` * flags.

names: is an array of DNS names belonging to the public-key (NULL if none).

names_length: holds the length of the names list.

flags: an ORed sequence of `gnutls_pkcs_encrypt_flags_t` . These apply to the private key pkey.

This function sets a public/private keypair in the `gnutls_certificate_credentials_t` type to be used for authentication and/or encryption. `spki` and `privkey` should match otherwise set signatures cannot be validated. In case of no match this function returns `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` . This function should be called once for the client because there is currently no mechanism to determine which raw public-key to select for the peer when there are multiple present. Multiple raw public keys for the server can be distinghuished by setting the `names` .

Note here that `spki` is a raw public-key as defined in RFC7250. It means that there is no surrounding certificate that holds the public key and that there is therefore no direct mechanism to prove the authenticity of this key. The keypair can be used during a TLS handshake but its authenticity should be established via a different mechanism (e.g. TOFU or known fingerprint).

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format and will be autodetected.

If the raw public-key and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

Key usage (as defined by X.509 extension (2.5.29.15)) can be explicitly set because there is no certificate structure around the key to define this value. See for more info `gnutls_x509_crt_get_key_usage()` .

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used in other functions to refer to the added key-pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, in case the key pair does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned, in other erroneous cases a different negative error code is returned.

Since: 3.6.6

gnutls_certificate_set_retrieve_function

`void gnutls_certificate_set_retrieve_function` [Function]
(*gnutls_certificate_credentials_t* `cred`, *gnutls_certificate_retrieve_function* * `func`)

cred: is a `gnutls_certificate_credentials_t` type.

func: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. The callback will take control only if a certificate is requested by the peer. You are advised to use `gnutls_certificate_set_retrieve_function2()` because it is much more efficient in the processing it requires from gnutls.

The callback's function prototype is: int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_retr2_st* st);

`req_ca_dn` is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. This is a hint and typically the client should send a certificate that is signed by one of these CAs. These names, when available, are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()` .

`pk_algos` contains a list with server's acceptable public key algorithms. The certificate returned should support the server's given algorithms.

`st` should contain the certificates and private keys.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received.

In server side pk_algos and req_ca_dn are NULL.

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated. If both certificates are set in the credentials and a callback is available, the callback takes predence.

Since: 3.0

gnutls_certificate_set_verify_flags

`void gnutls_certificate_set_verify_flags` [Function]
(*gnutls_certificate_credentials_t* `res`, *unsigned int* `flags`)

res: is a gnutls_certificate_credentials_t type

flags: are the flags

This function will set the flags to be used for verification of certificates and override any defaults. The provided flags must be an OR of the `gnutls_certificate_verify_flags` enumerations.

gnutls_certificate_set_verify_function

`void gnutls_certificate_set_verify_function` [Function]
(*gnutls_certificate_credentials_t* `cred`, *gnutls_certificate_verify_function ** `func`)

cred: is a `gnutls_certificate_credentials_t` type.

func: is the callback function

This function sets a callback to be called when peer's certificate has been received in order to verify it on receipt rather than doing after the handshake is completed.

The callback's function prototype is: int (*callback)(gnutls_session_t);

If the callback function is provided then gnutls will call it, in the handshake, just after the certificate message has been received. To verify or obtain the certificate the `gnutls_certificate_verify_peers2()` , `gnutls_certificate_type_get()` , `gnutls_certificate_get_peers()` functions can be used.

The callback function should return 0 for the handshake to continue or non-zero to terminate.

Since: 2.10.0

gnutls_certificate_set_verify_limits

`void gnutls_certificate_set_verify_limits` [Function]
(*gnutls_certificate_credentials_t* `res`, *unsigned int* `max_bits`, *unsigned int* `max_depth`)

res: is a gnutls_certificate_credentials type

max_bits: is the number of bits of an acceptable certificate (default 8200)

max_depth: is maximum depth of the verification of a certificate chain (default 5)

This function will set some upper limits for the default verification function, `gnutls_certificate_verify_peers2()` , to avoid denial of service attacks. You can set them to zero to disable limits.

gnutls_certificate_set_x509_crl

`int gnutls_certificate_set_x509_crl` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_x509_crl_t ** `crl_list`, *int* `crl_list_size`)

res: is a `gnutls_certificate_credentials_t` type.

crl_list: is a list of trusted CRLs. They should have been verified before.

crl_list_size: holds the size of the crl_list

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()` . This function may be called multiple times.

Returns: number of CRLs processed, or a negative error code on error.

Since: 2.4.0

gnutls_certificate_set_x509_crl_file

`int gnutls_certificate_set_x509_crl_file` [Function]
(*gnutls_certificate_credentials_t* `res`, *const char * * `crlfile`, *gnutls_x509_crt_fmt_t* `type`)

res: is a `gnutls_certificate_credentials_t` type.

crlfile: is a file containing the list of verified CRLs (DER or PEM list)

type: is PEM or DER

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()` . This function may be called multiple times.

Returns: number of CRLs processed or a negative error code on error.

gnutls_certificate_set_x509_crl_mem

`int gnutls_certificate_set_x509_crl_mem` [Function]
(*gnutls_certificate_credentials_t* `res`, *const gnutls_datum_t * `CRL`, *gnutls_x509_crt_fmt_t* `type`)

res: is a `gnutls_certificate_credentials_t` type.

CRL: is a list of trusted CRLs. They should have been verified before.

type: is DER or PEM

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()` . This function may be called multiple times.

Returns: number of CRLs processed, or a negative error code on error.

gnutls_certificate_set_x509_key

`int gnutls_certificate_set_x509_key` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_x509_crt_t * `cert_list`, *int* `cert_list_size`, *gnutls_x509_privkey_t* `key`)

res: is a `gnutls_certificate_credentials_t` type.

cert_list: contains a certificate list (path) for the specified private key

cert_list_size: holds the size of the certificate list

key: is a `gnutls_x509_privkey_t` key

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t type. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that wants to send more than their own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in `cert_list` .

Note that the certificates and keys provided, can be safely deinitialized after this function is called.

If that function fails to load the `res` type is at an undefined state, it must not be reused to load other keys or certificates.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

Since: 2.4.0

gnutls_certificate_set_x509_key_file

`int gnutls_certificate_set_x509_key_file` [Function]
(*gnutls_certificate_credentials_t* `res`, *const char ** `certfile`, *const char ** `keyfile`, *gnutls_x509_crt_fmt_t* `type`)

res: is a `gnutls_certificate_credentials_t` type.

certfile: is a file that containing the certificate list (path) for the specified private key, in PKCS7 format, or a list of certificates

keyfile: is a file that contains the private key

type: is PEM or DER

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t type. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that need to send more than its own end entity certificate, e.g., also an intermediate CA cert, then the `certfile` must contain the ordered certificate chain.

Note that the names in the certificate provided will be considered when selecting the appropriate certificate to use (in case of multiple certificate/key pairs).

This function can also accept URLs at `keyfile` and `certfile` . In that case it will use the private key and certificate indicated by the URLs. Note that the supported URLs are the ones indicated by `gnutls_url_is_supported()` .

In case the `certfile` is provided as a PKCS `11` URL, then the certificate, and its present issuers in the token are imported (i.e., forming the required trust chain).

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

Since: 3.1.11

gnutls_certificate_set_x509_key_file2

`int gnutls_certificate_set_x509_key_file2` [Function]
(*gnutls_certificate_credentials_t* `res`, *const char ** `certfile`, *const char ** `keyfile`, *gnutls_x509_crt_fmt_t* `type`, *const char ** `pass`, *unsigned int* `flags`)

res: is a `gnutls_certificate_credentials_t` type.

certfile: is a file that containing the certificate list (path) for the specified private key, in PKCS7 format, or a list of certificates

keyfile: is a file that contains the private key

type: is PEM or DER

pass: is the password of the key

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t type. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that need to send more than its own end entity certificate, e.g., also an intermediate CA cert, then the `certfile` must contain the ordered certificate chain.

Note that the names in the certificate provided will be considered when selecting the appropriate certificate to use (in case of multiple certificate/key pairs).

This function can also accept URLs at `keyfile` and `certfile` . In that case it will use the private key and certificate indicated by the URLs. Note that the supported URLs are the ones indicated by `gnutls_url_is_supported()` . Before GnuTLS 3.4.0 when a URL was specified, the `pass` part was ignored and a PIN callback had to be registered, this is no longer the case in current releases.

In case the `certfile` is provided as a PKCS `11` URL, then the certificate, and its present issuers in the token are imported (i.e., forming the required trust chain).

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

gnutls_certificate_set_x509_key_mem

`int gnutls_certificate_set_x509_key_mem` [Function]
(*gnutls_certificate_credentials_t* `res`, *const gnutls_datum_t ** `cert`, *const gnutls_datum_t ** `key`, *gnutls_x509_crt_fmt_t* `type`)

res: is a `gnutls_certificate_credentials_t` type.

cert: contains a certificate list (path) for the specified private key

key: is the private key, or `NULL`

type: is PEM or DER

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t type. This function may be called more than once, in case multiple keys/certificates exist for the server.

Note that the keyUsage (2.5.29.15) PKIX extension in X.509 certificates is supported. This means that certificates intended for signing cannot be used for ciphersuites that require encryption.

If the certificate and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

The `key` may be `NULL` if you are using a sign callback, see `gnutls_sign_callback_set()` .

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

gnutls_certificate_set_x509_key_mem2

`int gnutls_certificate_set_x509_key_mem2` [Function]
(*gnutls_certificate_credentials_t* `res`, *const gnutls_datum_t ** `cert`, *const gnutls_datum_t ** `key`, *gnutls_x509_crt_fmt_t* `type`, *const char ** `pass`, *unsigned int* `flags`)

res: is a `gnutls_certificate_credentials_t` type.

cert: contains a certificate list (path) for the specified private key

key: is the private key, or `NULL`

type: is PEM or DER

pass: is the key's password

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t type. This function may be called more than once, in case multiple keys/certificates exist for the server.

Note that the keyUsage (2.5.29.15) PKIX extension in X.509 certificates is supported. This means that certificates intended for signing cannot be used for ciphersuites that require encryption.

If the certificate and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

The `key` may be `NULL` if you are using a sign callback, see `gnutls_sign_callback_set()` .

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

gnutls_certificate_set_x509_simple_pkcs12_file

`int gnutls_certificate_set_x509_simple_pkcs12_file` [Function]
(*gnutls_certificate_credentials_t* `res`, *const char ** `pkcs12file`, *gnutls_x509_crt_fmt_t* `type`, *const char ** `password`)

res: is a `gnutls_certificate_credentials_t` type.

pkcs12file: filename of file containing PKCS12 blob.

type: is PEM or DER of the `pkcs12file` .

password: optional password used to decrypt PKCS12 file, bags and keys.

This function sets a certificate/private key pair and/or a CRL in the gnutls_certificate_credentials_t type. This function may be called more than once (in case multiple keys/certificates exist for the server).

PKCS12 files with a MAC, encrypted bags and PKCS 8 private keys are supported. However, only password based security, and the same password for all operations, are supported.

PKCS12 file may contain many keys and/or certificates, and this function will try to auto-detect based on the key ID the certificate and key pair to use. If the PKCS12 file contain the issuer of the selected certificate, it will be appended to the certificate to form a chain.

If more than one private keys are stored in the PKCS12 file, then only one key will be read (and it is undefined which one).

It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

gnutls_certificate_set_x509_simple_pkcs12_mem

`int gnutls_certificate_set_x509_simple_pkcs12_mem` [Function]
(*gnutls_certificate_credentials_t* `res`, *const gnutls_datum_t ** `p12blob`, *gnutls_x509_crt_fmt_t* `type`, *const char ** `password`)

res: is a `gnutls_certificate_credentials_t` type.

p12blob: the PKCS12 blob.

type: is PEM or DER of the `pkcs12file` .

password: optional password used to decrypt PKCS12 file, bags and keys.

This function sets a certificate/private key pair and/or a CRL in the gnutls_certificate_credentials_t type. This function may be called more than once (in case multiple keys/certificates exist for the server).

Encrypted PKCS12 bags and PKCS8 private keys are supported. However, only password based security, and the same password for all operations, are supported.

PKCS12 file may contain many keys and/or certificates, and this function will try to auto-detect based on the key ID the certificate and key pair to use. If the PKCS12 file contain the issuer of the selected certificate, it will be appended to the certificate to form a chain.

If more than one private keys are stored in the PKCS12 file, then only one key will be read (and it is undefined which one).

It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used to other functions to refer to the added key-pair.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

Since: 2.8.0

gnutls_certificate_set_x509_system_trust

`int gnutls_certificate_set_x509_system_trust` [Function]
(*gnutls_certificate_credentials_t* `cred`)

cred: is a `gnutls_certificate_credentials_t` type.

This function adds the system's default trusted CAs in order to verify client or server certificates.

In the case the system is currently unsupported `GNUTLS_E_UNIMPLEMENTED_FEATURE` is returned.

Returns: the number of certificates processed or a negative error code on error.

Since: 3.0.20

gnutls_certificate_set_x509_trust

`int gnutls_certificate_set_x509_trust` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_x509_crt_t* * `ca_list`, *int* `ca_list_size`)

res: is a `gnutls_certificate_credentials_t` type.

ca_list: is a list of trusted CAs

ca_list_size: holds the size of the CA list

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified

using gnutls_certificate_verify_peers2() . This function may be called multiple times.

In case of a server the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using gnutls_certificate_send_x509_rdn_sequence() .

Returns: the number of certificates processed or a negative error code on error.

Since: 2.4.0

gnutls_certificate_set_x509_trust_dir

int gnutls_certificate_set_x509_trust_dir [Function]
(*gnutls_certificate_credentials_t* **cred**, *const char ** **ca_dir**, *gnutls_x509_crt_fmt_t* **type**)

cred: is a gnutls_certificate_credentials_t type.

ca_dir: is a directory containing the list of trusted CAs (DER or PEM list)

type: is PEM or DER

This function adds the trusted CAs present in the directory in order to verify client or server certificates. This function is identical to gnutls_certificate_set_x509_trust_file() but loads all certificates in a directory.

Returns: the number of certificates processed

Since: 3.3.6

gnutls_certificate_set_x509_trust_file

int gnutls_certificate_set_x509_trust_file [Function]
(*gnutls_certificate_credentials_t* **cred**, *const char ** **cafile**, *gnutls_x509_crt_fmt_t* **type**)

cred: is a gnutls_certificate_credentials_t type.

cafile: is a file containing the list of trusted CAs (DER or PEM list)

type: is PEM or DER

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using gnutls_certificate_verify_peers2() . This function may be called multiple times.

In case of a server the names of the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using gnutls_certificate_send_x509_rdn_sequence() .

This function can also accept URLs. In that case it will import all certificates that are marked as trusted. Note that the supported URLs are the ones indicated by gnutls_url_is_supported() .

Returns: the number of certificates processed

gnutls_certificate_set_x509_trust_mem

`int gnutls_certificate_set_x509_trust_mem` [Function]
(*gnutls_certificate_credentials_t* `res`, *const gnutls_datum_t ** `ca`, *gnutls_x509_crt_fmt_t* `type`)

res: is a `gnutls_certificate_credentials_t` type.

ca: is a list of trusted CAs or a DER certificate

type: is DER or PEM

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()` . This function may be called multiple times.

In case of a server the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using `gnutls_certificate_send_x509_rdn_sequence()` .

Returns: the number of certificates processed or a negative error code on error.

gnutls_certificate_type_get

`gnutls_certificate_type_t gnutls_certificate_type_get` [Function]
(*gnutls_session_t* `session`)

session: is a `gnutls_session_t` type.

This function returns the type of the certificate that is negotiated for this side to send to the peer. The certificate type is by default X.509, unless an alternative certificate type is enabled by `gnutls_init()` and negotiated during the session.

Resumed sessions will return the certificate type that was negotiated and used in the original session.

As of version 3.6.4 it is recommended to use `gnutls_certificate_type_get2()` which is more fine-grained.

Returns: the currently used `gnutls_certificate_type_t` certificate type as negotiated for 'our' side of the connection.

gnutls_certificate_type_get2

`gnutls_certificate_type_t gnutls_certificate_type_get2` [Function]
(*gnutls_session_t* `session`, *gnutls_ctype_target_t* `target`)

session: is a `gnutls_session_t` type.

target: is a `gnutls_ctype_target_t` type.

This function returns the type of the certificate that a side is negotiated to use. The certificate type is by default X.509, unless an alternative certificate type is enabled by `gnutls_init()` and negotiated during the session.

The `target` parameter specifies whether to request the negotiated certificate type for the client (`GNUTLS_CTYPE_CLIENT`), or for the server (`GNUTLS_CTYPE_SERVER`). Additionally, in P2P mode connection set up where you don't know in advance who will be client and who will be server you can use the flag (`GNUTLS_CTYPE_OURS`) and (`GNUTLS_CTYPE_PEERS`) to retrieve the corresponding certificate types.

Resumed sessions will return the certificate type that was negotiated and used in the original session. That is, this function can be used to reliably determine the type of the certificate returned by `gnutls_certificate_get_peers()` .

Returns: the currently used `gnutls_certificate_type_t` certificate type for the client or the server.

Since: 3.6.4

gnutls_certificate_type_get_id

`gnutls_certificate_type_t gnutls_certificate_type_get_id` [Function]
(*const char * **name***)

name: is a certificate type name

The names are compared in a case insensitive way.

Returns: a `gnutls_certificate_type_t` for the specified in a string certificate type, or `GNUTLS_CRT_UNKNOWN` on error.

gnutls_certificate_type_get_name

`const char * gnutls_certificate_type_get_name` [Function]
(*gnutls_certificate_type_t **type***)

type: is a certificate type

Convert a `gnutls_certificate_type_t` type to a string.

Returns: a string that contains the name of the specified certificate type, or `NULL` in case of unknown types.

gnutls_certificate_type_list

`const gnutls_certificate_type_t *` [Function]
`gnutls_certificate_type_list ( void)`

Get a list of certificate types.

Returns: a (0)-terminated list of `gnutls_certificate_type_t` integers indicating the available certificate types.

gnutls_certificate_verification_status_print

`int gnutls_certificate_verification_status_print (unsigned` [Function]
*int **status**, gnutls_certificate_type_t **type**, gnutls_datum_t * **out**, unsigned int **flags***)

status: The status flags to be printed

type: The certificate type

out: Newly allocated datum with (0) terminated string.

flags: should be zero

This function will pretty print the status of a verification process – eg. the one obtained by `gnutls_certificate_verify_peers3()` .

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.1.4

gnutls_certificate_verify_peers

`int gnutls_certificate_verify_peers (`*gnutls_session_t* **session**, *gnutls_typed_vdata_st* * **data**, *unsigned int* **elements**, *unsigned int* * **status**`)` [Function]

session: is a gnutls session

data: an array of typed data

elements: the number of data elements

status: is the output of the verification

This function will verify the peer's certificate and store the the status in the status variable as a bitwise OR of gnutls_certificate_status_t values or zero if the certificate is trusted. Note that value in status is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using gnutls_certificate_set_verify_flags() . See the documentation of gnutls_certificate_verify_peers2() for details in the verification process.

This function will take into account the stapled OCSP responses sent by the server, as well as the following X.509 certificate extensions: Name Constraints, Key Usage, and Basic Constraints (pathlen).

The acceptable data types are GNUTLS_DT_DNS_HOSTNAME , GNUTLS_DT_RFC822NAME and GNUTLS_DT_KEY_PURPOSE_OID . The former two accept as data a null-terminated hostname or email address, and the latter a null-terminated object identifier (e.g., GNUTLS_KP_TLS_WWW_SERVER).

If a DNS hostname is provided then this function will compare the hostname in the certificate against the given. If names do not match the GNUTLS_CERT_UNEXPECTED_OWNER status flag will be set. If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to be have the provided key purpose or be marked for any purpose, otherwise verification status will have the GNUTLS_CERT_SIGNER_CONSTRAINTS_FAILURE flag set.

To avoid denial of service attacks some default upper limits regarding the certificate key size and chain size are set. To override them use gnutls_certificate_set_verify_limits() .

Note that when using raw public-keys verification will not work because there is no corresponding certificate body belonging to the raw key that can be verified. In that case this function will return GNUTLS_E_INVALID_REQUEST .

Returns: GNUTLS_E_SUCCESS (0) when the validation is performed, or a negative error code otherwise. A successful error code means that the status parameter must be checked to obtain the validation status.

Since: 3.3.0

gnutls_certificate_verify_peers2

`int gnutls_certificate_verify_peers2 (`*gnutls_session_t* `session`*, unsigned int * status*`)` [Function]

session: is a gnutls session

status: is the output of the verification

This function will verify the peer's certificate and store the status in the `status` variable as a bitwise OR of gnutls_certificate_status_t values or zero if the certificate is trusted. Note that value in `status` is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using `gnutls_certificate_set_verify_flags()` .

This function will take into account the stapled OCSP responses sent by the server, as well as the following X.509 certificate extensions: Name Constraints, Key Usage, and Basic Constraints (pathlen).

Note that you must also check the peer's name in order to check if the verified certificate belongs to the actual peer, see `gnutls_x509_crt_check_hostname()` , or use `gnutls_certificate_verify_peers3()` .

To avoid denial of service attacks some default upper limits regarding the certificate key size and chain size are set. To override them use `gnutls_certificate_set_verify_limits()` .

Note that when using raw public-keys verification will not work because there is no corresponding certificate body belonging to the raw key that can be verified. In that case this function will return `GNUTLS_E_INVALID_REQUEST` .

Returns: `GNUTLS_E_SUCCESS` (0) when the validation is performed, or a negative error code otherwise. A successful error code means that the `status` parameter must be checked to obtain the validation status.

gnutls_certificate_verify_peers3

`int gnutls_certificate_verify_peers3 (`*gnutls_session_t* `session`*, const char * hostname, unsigned int * status*`)` [Function]

session: is a gnutls session

hostname: is the expected name of the peer; may be `NULL`

status: is the output of the verification

This function will verify the peer's certificate and store the the status in the `status` variable as a bitwise OR of gnutls_certificate_status_t values or zero if the certificate is trusted. Note that value in `status` is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using `gnutls_certificate_set_verify_flags()` . See the documentation of `gnutls_certificate_verify_peers2()` for details in the verification process.

This function will take into account the stapled OCSP responses sent by the server, as well as the following X.509 certificate extensions: Name Constraints, Key Usage, and Basic Constraints (pathlen).

If the `hostname` provided is non-NULL then this function will compare the hostname in the certificate against it. The comparison will follow the RFC6125 recommendations. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set.

In order to verify the purpose of the end-certificate (by checking the extended key usage), use `gnutls_certificate_verify_peers()` .

To avoid denial of service attacks some default upper limits regarding the certificate key size and chain size are set. To override them use `gnutls_certificate_set_verify_limits()` .

Note that when using raw public-keys verification will not work because there is no corresponding certificate body belonging to the raw key that can be verified. In that case this function will return `GNUTLS_E_INVALID_REQUEST` .

Returns: `GNUTLS_E_SUCCESS` (0) when the validation is performed, or a negative error code otherwise. A successful error code means that the `status` parameter must be checked to obtain the validation status.

Since: 3.1.4

gnutls_check_version

`const char * gnutls_check_version` (*const char * req_version*) [Function]

req_version: version string to compare with, or `NULL` .

Check the GnuTLS Library version against the provided string. See `GNUTLS_VERSION` for a suitable `req_version` string.

See also `gnutls_check_version_numeric()` , which provides this functionality as a macro.

Returns: Check that the version of the library is at minimum the one given as a string in `req_version` and return the actual version string of the library; return `NULL` if the condition is not met. If `NULL` is passed to this function no check is done and only the version string is returned.

gnutls_cipher_get

`gnutls_cipher_algorithm_t gnutls_cipher_get` (*gnutls_session_t session*) [Function]

session: is a `gnutls_session_t` type.

Get the currently used cipher.

Returns: the currently used cipher, a `gnutls_cipher_algorithm_t` type.

gnutls_cipher_get_id

`gnutls_cipher_algorithm_t gnutls_cipher_get_id` (*const char * name*) [Function]

name: is a cipher algorithm name

The names are compared in a case insensitive way.

Returns: return a `gnutls_cipher_algorithm_t` value corresponding to the specified cipher, or `GNUTLS_CIPHER_UNKNOWN` on error.

gnutls_cipher_get_key_size

`size_t gnutls_cipher_get_key_size (`*gnutls_cipher_algorithm_t* `algorithm`) [Function]

algorithm: is an encryption algorithm

This function returns the key size of the provided algorithm.

Returns: length (in bytes) of the given cipher's key size, or 0 if the given cipher is invalid.

gnutls_cipher_get_name

`const char * gnutls_cipher_get_name (`*gnutls_cipher_algorithm_t* `algorithm`) [Function]

algorithm: is an encryption algorithm

Convert a `gnutls_cipher_algorithm_t` type to a string.

Returns: a pointer to a string that contains the name of the specified cipher, or `NULL` .

gnutls_cipher_list

`const gnutls_cipher_algorithm_t * gnutls_cipher_list (` *void*`)` [Function]

Get a list of supported cipher algorithms. Note that not necessarily all ciphers are supported as TLS cipher suites. For example, DES is not supported as a cipher suite, but is supported for other purposes (e.g., PKCS8 or similar).

This function is not thread safe.

Returns: a (0)-terminated list of `gnutls_cipher_algorithm_t` integers indicating the available ciphers.

gnutls_cipher_suite_get_name

`const char * gnutls_cipher_suite_get_name` [Function]
`(`*gnutls_kx_algorithm_t* `kx_algorithm`, *gnutls_cipher_algorithm_t* `cipher_algorithm`, *gnutls_mac_algorithm_t* `mac_algorithm`)

kx_algorithm: is a Key exchange algorithm

cipher_algorithm: is a cipher algorithm

mac_algorithm: is a MAC algorithm

This function returns the ciphersuite name under TLS1.2 or earlier versions when provided with individual algorithms. The full cipher suite name must be prepended by TLS or SSL depending of the protocol in use.

To get a description of the current ciphersuite across versions, it is recommended to use `gnutls_session_get_desc()` .

Returns: a string that contains the name of a TLS cipher suite, specified by the given algorithms, or `NULL` .

gnutls_cipher_suite_info

`const char * gnutls_cipher_suite_info (size_t idx, unsigned char * cs_id, gnutls_kx_algorithm_t * kx, gnutls_cipher_algorithm_t * cipher, gnutls_mac_algorithm_t * mac, gnutls_protocol_t * min_version)` [Function]

idx: index of cipher suite to get information about, starts on 0.

cs_id: output buffer with room for 2 bytes, indicating cipher suite value

kx: output variable indicating key exchange algorithm, or `NULL` .

cipher: output variable indicating cipher, or `NULL` .

mac: output variable indicating MAC algorithm, or `NULL` .

min_version: output variable indicating TLS protocol version, or `NULL` .

Get information about supported cipher suites. Use the function iteratively to get information about all supported cipher suites. Call with idx=0 to get information about first cipher suite, then idx=1 and so on until the function returns NULL.

Returns: the name of `idx` cipher suite, and set the information about the cipher suite in the output variables. If `idx` is out of bounds, `NULL` is returned.

gnutls_credentials_clear

`void gnutls_credentials_clear (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` type.

Clears all the credentials previously set in this session.

gnutls_credentials_get

`int gnutls_credentials_get (gnutls_session_t session, gnutls_credentials_type_t type, void ** cred)` [Function]

session: is a `gnutls_session_t` type.

type: is the type of the credentials to return

cred: will contain the credentials.

Returns the previously provided credentials structures.

For `GNUTLS_CRD_ANON` , `cred` will be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , `cred` will be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , `cred` will be `gnutls_certificate_credentials_t` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.3.3

gnutls_credentials_set

`int gnutls_credentials_set` (*gnutls_session_t* `session`, *gnutls_credentials_type_t* `type`, *void * cred*) [Function]

session: is a `gnutls_session_t` type.

type: is the type of the credentials

cred: the credentials to set

Sets the needed credentials for the specified type. E.g. username, password - or public and private keys etc. The `cred` parameter is a structure that depends on the specified type and on the current session (client or server).

In order to minimize memory usage, and share credentials between several threads gnutls keeps a pointer to cred, and not the whole cred structure. Thus you will have to keep the structure allocated until you call `gnutls_deinit()` .

For `GNUTLS_CRD_ANON` , `cred` should be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , `cred` should be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , `cred` should be `gnutls_certificate_credentials_t` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_db_check_entry

`int gnutls_db_check_entry` (*gnutls_session_t* `session`, *gnutls_datum_t* `session_entry`) [Function]

session: is a `gnutls_session_t` type.

session_entry: is the session data (not key)

This function has no effect.

Returns: Returns `GNUTLS_E_EXPIRED` , if the database entry has expired or 0 otherwise.

Deprecated: This function is deprecated.

gnutls_db_check_entry_expire_time

`time_t gnutls_db_check_entry_expire_time` (*gnutls_datum_t * entry*) [Function]

entry: is a pointer to a `gnutls_datum_t` type.

This function returns the time that this entry will expire. It can be used for database entry expiration.

Returns: The time this entry will expire, or zero on error.

Since: 3.6.5

gnutls_db_check_entry_time

`time_t gnutls_db_check_entry_time (`*gnutls_datum_t * `entry`*`)` [Function]

entry: is a pointer to a `gnutls_datum_t` type.

This function returns the time that this entry was active. It can be used for database entry expiration.

Returns: The time this entry was created, or zero on error.

gnutls_db_get_default_cache_expiration

`unsigned gnutls_db_get_default_cache_expiration (` *`void`*`)` [Function]

Returns the expiration time (in seconds) of stored sessions for resumption.

gnutls_db_get_ptr

`void * gnutls_db_get_ptr (`*gnutls_session_t `session`*`)` [Function]

session: is a `gnutls_session_t` type.

Get db function pointer.

Returns: the pointer that will be sent to db store, retrieve and delete functions, as the first argument.

gnutls_db_remove_session

`void gnutls_db_remove_session (`*gnutls_session_t `session`*`)` [Function]

session: is a `gnutls_session_t` type.

This function will remove the current session data from the session database. This will prevent future handshakes reusing these session data. This function should be called if a session was terminated abnormally, and before `gnutls_deinit()` is called.

Normally `gnutls_deinit()` will remove abnormally terminated sessions.

gnutls_db_set_cache_expiration

`void gnutls_db_set_cache_expiration (`*gnutls_session_t `session`*`,` [Function]
int `seconds``)`

session: is a `gnutls_session_t` type.

seconds: is the number of seconds.

Set the expiration time for resumed sessions. The default is 21600 (6 hours) at the time of writing.

The maximum value that can be set using this function is 604800 (7 days).

gnutls_db_set_ptr

`void gnutls_db_set_ptr (`*gnutls_session_t `session`*`, void * `ptr`*`)` [Function]

session: is a `gnutls_session_t` type.

ptr: is the pointer

Sets the pointer that will be provided to db store, retrieve and delete functions, as the first argument.

gnutls_db_set_remove_function

`void gnutls_db_set_remove_function (`*gnutls_session_t* **`session`**, *gnutls_db_remove_func* **`rem_func`**`)` [Function]

session: is a `gnutls_session_t` type.

rem_func: is the function.

Sets the function that will be used to remove data from the resumed sessions database. This function must return 0 on success.

The first argument to `rem_func` will be null unless `gnutls_db_set_ptr()` has been called.

gnutls_db_set_retrieve_function

`void gnutls_db_set_retrieve_function (`*gnutls_session_t* **`session`**, *gnutls_db_retr_func* **`retr_func`**`)` [Function]

session: is a `gnutls_session_t` type.

retr_func: is the function.

Sets the function that will be used to retrieve data from the resumed sessions database. This function must return a gnutls_datum_t containing the data on success, or a gnutls_datum_t containing null and 0 on failure.

The datum's data must be allocated using the function `gnutls_malloc()` .

The first argument to `retr_func` will be null unless `gnutls_db_set_ptr()` has been called.

gnutls_db_set_store_function

`void gnutls_db_set_store_function (`*gnutls_session_t* **`session`**, *gnutls_db_store_func* **`store_func`**`)` [Function]

session: is a `gnutls_session_t` type.

store_func: is the function

Sets the function that will be used to store data in the resumed sessions database. This function must return 0 on success.

The first argument to `store_func` will be null unless `gnutls_db_set_ptr()` has been called.

gnutls_deinit

`void gnutls_deinit (`*gnutls_session_t* **`session`**`)` [Function]

session: is a `gnutls_session_t` type.

This function clears all buffers associated with the `session` . This function will also remove session data from the session database if the session was terminated abnormally.

gnutls_dh_get_group

`int gnutls_dh_get_group` (*gnutls_session_t* **`session`**, *gnutls_datum_t* * **`raw_gen`**, *gnutls_datum_t* * **`raw_prime`**) [Function]

session: is a gnutls session

raw_gen: will hold the generator.

raw_prime: will hold the prime.

This function will return the group parameters used in the last Diffie-Hellman key exchange with the peer. These are the prime and the generator used. This function should be used for both anonymous and ephemeral Diffie-Hellman. The output parameters must be freed with `gnutls_free()` .

Note, that the prime and generator are exported as non-negative integers and may include a leading zero byte.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_dh_get_peers_public_bits

`int gnutls_dh_get_peers_public_bits` (*gnutls_session_t* **`session`**) [Function]

session: is a gnutls session

Get the Diffie-Hellman public key bit size. Can be used for both anonymous and ephemeral Diffie-Hellman.

Returns: The public key bit size used in the last Diffie-Hellman key exchange with the peer, or a negative error code in case of error.

gnutls_dh_get_prime_bits

`int gnutls_dh_get_prime_bits` (*gnutls_session_t* **`session`**) [Function]

session: is a gnutls session

This function will return the bits of the prime used in the last Diffie-Hellman key exchange with the peer. Should be used for both anonymous and ephemeral Diffie-Hellman. Note that some ciphers, like RSA and DSA without DHE, do not use a Diffie-Hellman key exchange, and then this function will return 0.

Returns: The Diffie-Hellman bit strength is returned, or 0 if no Diffie-Hellman key exchange was done, or a negative error code on failure.

gnutls_dh_get_pubkey

`int gnutls_dh_get_pubkey` (*gnutls_session_t* **`session`**, *gnutls_datum_t* * **`raw_key`**) [Function]

session: is a gnutls session

raw_key: will hold the public key.

This function will return the peer's public key used in the last Diffie-Hellman key exchange. This function should be used for both anonymous and ephemeral Diffie-Hellman. The output parameters must be freed with `gnutls_free()` .

Note, that public key is exported as non-negative integer and may include a leading zero byte.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_dh_get_secret_bits

`int gnutls_dh_get_secret_bits` (*gnutls_session_t* **`session`**) [Function]

session: is a gnutls session

This function will return the bits used in the last Diffie-Hellman key exchange with the peer. Should be used for both anonymous and ephemeral Diffie-Hellman.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_dh_params_cpy

`int gnutls_dh_params_cpy` (*gnutls_dh_params_t* **`dst`**, *gnutls_dh_params_t* **`src`**) [Function]

dst: Is the destination parameters, which should be initialized.

src: Is the source parameters

This function will copy the DH parameters structure from source to destination. The destination should be already initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_deinit

`void gnutls_dh_params_deinit` (*gnutls_dh_params_t* **`dh_params`**) [Function]

dh_params: The parameters

This function will deinitialize the DH parameters type.

gnutls_dh_params_export2_pkcs3

`int gnutls_dh_params_export2_pkcs3` (*gnutls_dh_params_t* **`params`**, *gnutls_x509_crt_fmt_t* **`format`**, *gnutls_datum_t * `out`*) [Function]

params: Holds the DH parameters

format: the format of output params. One of PEM or DER.

out: will contain a PKCS3 DHParams structure PEM or DER encoded

This function will export the given dh parameters to a PKCS3 DHParams structure. This is the format generated by "openssl dhparam" tool. The data in `out` will be allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN DH PARAMETERS".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.1.3

gnutls_dh_params_export_pkcs3

`int gnutls_dh_params_export_pkcs3` (*gnutls_dh_params_t* `params`, *gnutls_x509_crt_fmt_t* `format`, *unsigned char** `params_data`, *size_t** `params_data_size`) [Function]

params: Holds the DH parameters

format: the format of output params. One of PEM or DER.

params_data: will contain a PKCS3 DHParams structure PEM or DER encoded

params_data_size: holds the size of params_data (and will be replaced by the actual size of parameters)

This function will export the given dh parameters to a PKCS3 DHParams structure. This is the format generated by "openssl dhparam" tool. If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN DH PARAMETERS".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_export_raw

`int gnutls_dh_params_export_raw` (*gnutls_dh_params_t* `params`, *gnutls_datum_t** `prime`, *gnutls_datum_t** `generator`, *unsigned int** `bits`) [Function]

params: Holds the DH parameters

prime: will hold the new prime

generator: will hold the new generator

bits: if non null will hold the secret key's number of bits

This function will export the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_generate2

`int gnutls_dh_params_generate2` (*gnutls_dh_params_t* `dparams`, *unsigned int* `bits`) [Function]

dparams: The parameters

bits: is the prime's number of bits

This function will generate a new pair of prime and generator for use in the Diffie-Hellman key exchange. This may take long time.

It is recommended not to set the number of bits directly, but use `gnutls_sec_param_to_pk_bits()` instead. Also note that the DH parameters are only useful to servers.

Since clients use the parameters sent by the server, it's of no use to call this in client side.

The parameters generated are of the DSA form. It also is possible to generate provable parameters (following the Shawe-Taylor algorithm), using `gnutls_x509_privkey_generate2()` with DSA option and the `GNUTLS_PRIVKEY_FLAG_PROVABLE` flag set. These can the be imported with `gnutls_dh_params_import_dsa()` .

It is no longer recommended for applications to generate parameters. See the "Parameter generation" section in the manual.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_import_dsa

`int gnutls_dh_params_import_dsa` (*gnutls_dh_params_t* ***dh_params***, *gnutls_x509_privkey_t* ***key***) [Function]

dh_params: The parameters

key: holds a DSA private key

This function will import the prime and generator of the DSA key for use in the Diffie-Hellman key exchange.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_import_pkcs3

`int gnutls_dh_params_import_pkcs3` (*gnutls_dh_params_t* ***params***, *const gnutls_datum_t ** ***pkcs3_params***, *gnutls_x509_crt_fmt_t* ***format***) [Function]

params: The parameters

pkcs3_params: should contain a PKCS3 DHParams structure PEM or DER encoded

format: the format of params. PEM or DER.

This function will extract the DHParams found in a PKCS3 formatted structure. This is the format generated by "openssl dhparam" tool.

If the structure is PEM encoded, it should have a header of "BEGIN DH PARAMETERS".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_import_raw

`int gnutls_dh_params_import_raw` (*gnutls_dh_params_t* ***dh_params***, *const gnutls_datum_t ** ***prime***, *const gnutls_datum_t ** ***generator***) [Function]

dh_params: The parameters

prime: holds the new prime

generator: holds the new generator

This function will replace the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters should be stored in the appropriate gnutls_datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_import_raw2

`int gnutls_dh_params_import_raw2 (`*gnutls_dh_params_t* **`dh_params`**, *const gnutls_datum_t ** **`prime`**, *const gnutls_datum_t ** **`generator`**, *unsigned* **`key_bits`**`)` [Function]

dh_params: The parameters

prime: holds the new prime

generator: holds the new generator

key_bits: the private key bits (set to zero when unknown)

This function will replace the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters should be stored in the appropriate gnutls_datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_import_raw3

`int gnutls_dh_params_import_raw3 (`*gnutls_dh_params_t* **`dh_params`**, *const gnutls_datum_t ** **`prime`**, *const gnutls_datum_t ** **`q`**, *const gnutls_datum_t ** **`generator`**`)` [Function]

dh_params: The parameters

prime: holds the new prime

q: holds the subgroup if available, otherwise NULL

generator: holds the new generator

This function will replace the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters should be stored in the appropriate gnutls_datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_params_init

`int gnutls_dh_params_init (`*gnutls_dh_params_t ** **`dh_params`**`)` [Function]

dh_params: The parameters

This function will initialize the DH parameters type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_dh_set_prime_bits

`void gnutls_dh_set_prime_bits (`*gnutls_session_t* **`session`**, *unsigned int* **`bits`**`)` [Function]

session: is a `gnutls_session_t` type.

bits: is the number of bits

This function sets the number of bits, for use in a Diffie-Hellman key exchange. This is used both in DH ephemeral and DH anonymous cipher suites. This will set the minimum size of the prime that will be used for the handshake.

In the client side it sets the minimum accepted number of bits. If a server sends a prime with less bits than that `GNUTLS_E_DH_PRIME_UNACCEPTABLE` will be returned by the handshake.

Note that this function will warn via the audit log for value that are believed to be weak.

The function has no effect in server side.

Note that since 3.1.7 this function is deprecated. The minimum number of bits is set by the priority string level. Also this function must be called after `gnutls_priority_set_direct()` or the set value may be overridden by the selected priority options.

gnutls_digest_get_id

`gnutls_digest_algorithm_t gnutls_digest_get_id (`*const char * name*`)` [Function]

name: is a digest algorithm name

Convert a string to a `gnutls_digest_algorithm_t` value. The names are compared in a case insensitive way.

Returns: a `gnutls_digest_algorithm_t` id of the specified MAC algorithm string, or `GNUTLS_DIG_UNKNOWN` on failure.

gnutls_digest_get_name

`const char * gnutls_digest_get_name (`*gnutls_digest_algorithm_t algorithm*`)` [Function]

algorithm: is a digest algorithm

Convert a `gnutls_digest_algorithm_t` value to a string.

Returns: a string that contains the name of the specified digest algorithm, or `NULL` .

gnutls_digest_get_oid

`const char * gnutls_digest_get_oid (`*gnutls_digest_algorithm_t algorithm*`)` [Function]

algorithm: is a digest algorithm

Convert a `gnutls_digest_algorithm_t` value to its object identifier.

Returns: a string that contains the object identifier of the specified digest algorithm, or `NULL` .

Since: 3.4.3

gnutls_digest_list

`const gnutls_digest_algorithm_t * gnutls_digest_list (void)` [Function]

Get a list of hash (digest) algorithms supported by GnuTLS.

This function is not thread safe.

Returns: Return a (0)-terminated list of `gnutls_digest_algorithm_t` integers indicating the available digests.

gnutls_ecc_curve_get

`gnutls_ecc_curve_t gnutls_ecc_curve_get (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` type.

Returns the currently used elliptic curve for key exchange. Only valid when using an elliptic curve ciphersuite.

Returns: the currently used curve, a `gnutls_ecc_curve_t` type.

Since: 3.0

gnutls_ecc_curve_get_id

`gnutls_ecc_curve_t gnutls_ecc_curve_get_id (const char * name)` [Function]

name: is a curve name

The names are compared in a case insensitive way.

Returns: return a `gnutls_ecc_curve_t` value corresponding to the specified curve, or `GNUTLS_ECC_CURVE_INVALID` on error.

Since: 3.4.3

gnutls_ecc_curve_get_name

`const char * gnutls_ecc_curve_get_name (gnutls_ecc_curve_t curve)` [Function]

curve: is an ECC curve

Convert a `gnutls_ecc_curve_t` value to a string.

Returns: a string that contains the name of the specified curve or `NULL` .

Since: 3.0

gnutls_ecc_curve_get_oid

`const char * gnutls_ecc_curve_get_oid (gnutls_ecc_curve_t curve)` [Function]

curve: is an ECC curve

Convert a `gnutls_ecc_curve_t` value to its object identifier.

Returns: a string that contains the OID of the specified curve or `NULL` .

Since: 3.4.3

gnutls_ecc_curve_get_pk

`gnutls_pk_algorithm_t gnutls_ecc_curve_get_pk` [Function]
(*gnutls_ecc_curve_t* `curve`)

curve: is an ECC curve

Returns: the public key algorithm associated with the named curve or `GNUTLS_PK_UNKNOWN` .

Since: 3.5.0

gnutls_ecc_curve_get_size

`int gnutls_ecc_curve_get_size` (*gnutls_ecc_curve_t* `curve`) [Function]

curve: is an ECC curve

Returns: the size in bytes of the curve or 0 on failure.

Since: 3.0

gnutls_ecc_curve_list

`const gnutls_ecc_curve_t * gnutls_ecc_curve_list ( void)` [Function]

Get the list of supported elliptic curves.

This function is not thread safe.

Returns: Return a (0)-terminated list of `gnutls_ecc_curve_t` integers indicating the available curves.

gnutls_error_is_fatal

`int gnutls_error_is_fatal` (*int* `error`) [Function]

error: is a GnuTLS error code, a negative error code

If a GnuTLS function returns a negative error code you may feed that value to this function to see if the error condition is fatal to a TLS session (i.e., must be terminated).

Note that you may also want to check the error code manually, since some non-fatal errors to the protocol (such as a warning alert or a rehandshake request) may be fatal for your program.

This function is only useful if you are dealing with errors from functions that relate to a TLS session (e.g., record layer or handshake layer handling functions).

Returns: Non-zero value on fatal errors or zero on non-fatal.

gnutls_error_to_alert

`int gnutls_error_to_alert` (*int* `err`, *int* * `level`) [Function]

err: is a negative integer

level: the alert level will be stored there

Get an alert depending on the error code returned by a gnutls function. All alerts sent by this function should be considered fatal. The only exception is when `err` is `GNUTLS_E_REHANDSHAKE` , where a warning alert should be sent to the peer indicating that no renegotiation will be performed.

If there is no mapping to a valid alert the alert to indicate internal error (`GNUTLS_A_INTERNAL_ERROR`) is returned.

Returns: the alert code to use for a particular error code.

gnutls_est_record_overhead_size

`size_t gnutls_est_record_overhead_size` (*gnutls_protocol_t* `version`, *gnutls_cipher_algorithm_t* `cipher`, *gnutls_mac_algorithm_t* `mac`, *gnutls_compression_method_t* `comp`, *unsigned int* `flags`) [Function]

version: is a `gnutls_protocol_t` value

cipher: is a `gnutls_cipher_algorithm_t` value

mac: is a `gnutls_mac_algorithm_t` value

comp: is a `gnutls_compression_method_t` value (ignored)

flags: must be zero

This function will return the set size in bytes of the overhead due to TLS (or DTLS) per record.

Note that this function may provide inaccurate values when TLS extensions that modify the record format are negotiated. In these cases a more accurate value can be obtained using `gnutls_record_overhead_size()` after a completed handshake.

Since: 3.2.2

gnutls_ext_get_current_msg

`unsigned gnutls_ext_get_current_msg` (*gnutls_session_t* `session`) [Function]

session: a `gnutls_session_t` opaque pointer

This function allows an extension handler to obtain the message this extension is being called from. The returned value is a single entry of the `gnutls_ext_flags_t` enumeration. That is, if an extension was registered with the `GNUTLS_EXT_FLAG_HRR` and `GNUTLS_EXT_FLAG_EE` flags, the value when called during parsing of the encrypted extensions message will be `GNUTLS_EXT_FLAG_EE` .

If not called under an extension handler, its value is undefined.

Since: 3.6.3

gnutls_ext_get_data

`int gnutls_ext_get_data` (*gnutls_session_t* `session`, *unsigned* `tls_id`, *gnutls_ext_priv_data_t * * `data`) [Function]

session: a `gnutls_session_t` opaque pointer

tls_id: the numeric id of the extension

data: a pointer to the private data to retrieve

This function retrieves any data previously stored with `gnutls_ext_set_data()` .

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_ext_get_name

`const char * gnutls_ext_get_name (unsigned int ext)` [Function]

ext: is a TLS extension numeric ID

Convert a TLS extension numeric ID to a printable string.

Returns: a pointer to a string that contains the name of the specified cipher, or `NULL` .

gnutls_ext_get_name2

`const char * gnutls_ext_get_name2 (gnutls_session_t session, unsigned int tls_id, gnutls_ext_parse_type_t parse_point)` [Function]

session: a `gnutls_session_t` opaque pointer

tls_id: is a TLS extension numeric ID

parse_point: the parse type of the extension

Convert a TLS extension numeric ID to a printable string.

Returns: a pointer to a string that contains the name of the specified cipher, or `NULL` .

gnutls_ext_raw_parse

`int gnutls_ext_raw_parse (void * ctx, gnutls_ext_raw_process_func cb, const gnutls_datum_t * data, unsigned int flags)` [Function]

ctx: a pointer to pass to callback function

cb: callback function to process each extension found

data: TLS extension data

flags: should be zero or `GNUTLS_EXT_RAW_FLAG_TLS_CLIENT_HELLO` or `GNUTLS_EXT_RAW_FLAG_DTLS_CLIENT_HELLO`

This function iterates through the TLS extensions as passed in `data` , passing the individual extension data to callback. The `data` must conform to Extension extensions<0..2^16-1> format.

If flags is `GNUTLS_EXT_RAW_TLS_FLAG_CLIENT_HELLO` then this function will parse the extension data from the position, as if the packet in `data` is a client hello (without record or handshake headers) - as provided by `gnutls_handshake_set_hook_function()` .

The return value of the callback will be propagated.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code. On unknown flags it returns `GNUTLS_E_INVALID_REQUEST` .

Since: 3.6.3

gnutls_ext_register

`int gnutls_ext_register` (*const char * `name`*, *int `id`*, *gnutls_ext_parse_type_t `parse_point`*, *gnutls_ext_recv_func `recv_func`*, *gnutls_ext_send_func `send_func`*, *gnutls_ext_deinit_data_func `deinit_func`*, *gnutls_ext_pack_func `pack_func`*, *gnutls_ext_unpack_func `unpack_func`*) [Function]

name: the name of the extension to register

id: the numeric TLS id of the extension

parse_point: the parse type of the extension (see gnutls_ext_parse_type_t)

recv_func: a function to receive the data

send_func: a function to send the data

deinit_func: a function deinitialize any private data

pack_func: a function which serializes the extension's private data (used on session packing for resumption)

unpack_func: a function which will deserialize the extension's private data

This function will register a new extension type. The extension will remain registered until `gnutls_global_deinit()` is called. If the extension type is already registered then `GNUTLS_E_ALREADY_REGISTERED` will be returned.

Each registered extension can store temporary data into the gnutls_session_t structure using `gnutls_ext_set_data()` , and they can be retrieved using `gnutls_ext_get_data()` .

Any extensions registered with this function are valid for the client and TLS1.2 server hello (or encrypted extensions for TLS1.3).

This function is not thread safe.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_ext_set_data

`void gnutls_ext_set_data` (*gnutls_session_t `session`*, *unsigned `tls_id`*, *gnutls_ext_priv_data_t `data`*) [Function]

session: a `gnutls_session_t` opaque pointer

tls_id: the numeric id of the extension

data: the private data to set

This function allows an extension handler to store data in the current session and retrieve them later on. The set data will be deallocated using the gnutls_ext_deinit_data_func.

Since: 3.4.0

gnutls_fingerprint

`int gnutls_fingerprint (`*gnutls_digest_algorithm_t* `algo`, *const gnutls_datum_t ** `data`, *void ** `result`, *size_t ** `result_size`) [Function]

algo: is a digest algorithm

data: is the data

result: is the place where the result will be copied (may be null).

result_size: should hold the size of the result. The actual size of the returned result will also be copied there.

This function will calculate a fingerprint (actually a hash), of the given data. The result is not printable data. You should convert it to hex, or to something else printable.

This is the usual way to calculate a fingerprint of an X.509 DER encoded certificate. Note however that the fingerprint of an OpenPGP certificate is not just a hash and cannot be calculated with this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_fips140_mode_enabled

`unsigned gnutls_fips140_mode_enabled ( void)` [Function]

Checks whether this library is in FIPS140 mode. The returned value corresponds to the library mode as set with `gnutls_fips140_set_mode()` .

If `gnutls_fips140_set_mode()` was called with `GNUTLS_FIPS140_SET_MODE_THREAD` then this function will return the current thread's FIPS140 mode, otherwise the global value is returned.

Returns: return non-zero if true or zero if false.

Since: 3.3.0

gnutls_fips140_set_mode

`void gnutls_fips140_set_mode (`*gnutls_fips_mode_t* `mode`, *unsigned* `flags`) [Function]

mode: the FIPS140-2 mode to switch to

flags: should be zero or `GNUTLS_FIPS140_SET_MODE_THREAD`

That function is not thread-safe when changing the mode with no flags (globally), and should be called prior to creating any threads. Its behavior with no flags after threads are created is undefined.

When the flag `GNUTLS_FIPS140_SET_MODE_THREAD` is specified then this call will change the FIPS140-2 mode for this particular thread and not for the whole process. That way an application can utilize this function to set and reset mode for specific operations.

This function never fails but will be a no-op if used when the library is not in FIPS140-2 mode. When asked to switch to unknown values for `mode` or to `GNUTLS_FIPS140_SELFTESTS` mode, the library switches to `GNUTLS_FIPS140_STRICT` mode.

Since: 3.6.2

gnutls_get_system_config_file

`const char * gnutls_get_system_config_file ( void)` [Function]

Returns the filename of the system wide configuration file loaded by the library. The returned pointer is valid until the library is unloaded.

Returns: a constant pointer to the config file loaded, or `NULL` if none

Since: 3.6.9

gnutls_global_deinit

`void gnutls_global_deinit ( void)` [Function]

This function deinitializes the global data, that were initialized using `gnutls_global_init()` .

Since GnuTLS 3.3.0 this function is no longer necessary to be explicitly called. GnuTLS will automatically deinitialize on library destructor. See `gnutls_global_init()` for disabling the implicit initialization/deinitialization.

gnutls_global_init

`int gnutls_global_init ( void)` [Function]

Since GnuTLS 3.3.0 this function is no longer necessary to be explicitly called. To disable the implicit call (in a library constructor) of this function set the environment variable `GNUTLS_NO_EXPLICIT_INIT` to 1.

This function performs any required precalculations, detects the supported CPU capabilities and initializes the underlying cryptographic backend. In order to free any resources taken by this call you should `gnutls_global_deinit()` when gnutls usage is no longer needed.

This function increments a global counter, so that `gnutls_global_deinit()` only releases resources when it has been called as many times as `gnutls_global_init()` . This is useful when GnuTLS is used by more than one library in an application. This function can be called many times, but will only do something the first time. It is thread safe since GnuTLS 3.3.0.

A subsequent call of this function if the initial has failed will return the same error code.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_global_set_audit_log_function

`void gnutls_global_set_audit_log_function` [Function]
`(gnutls_audit_log_func log_func)`

log_func: it is the audit log function

This is the function to set the audit logging function. This is a function to report important issues, such as possible attacks in the protocol. This is different from `gnutls_global_set_log_function()` because it will report also session-specific events. The session parameter will be null if there is no corresponding TLS session.

`gnutls_audit_log_func` is of the form, void (*gnutls_audit_log_func)(gnutls_session_t, const char*);

Since: 3.0

gnutls_global_set_log_function

`void gnutls_global_set_log_function (`*gnutls_log_func* `log_func`) [Function]

log_func: it's a log function

This is the function where you set the logging function gnutls is going to use. This function only accepts a character array. Normally you may not use this function since it is only used for debugging purposes.

`gnutls_log_func` is of the form, void (*gnutls_log_func)(int level, const char*);

gnutls_global_set_log_level

`void gnutls_global_set_log_level (`*int* `level`) [Function]

level: it's an integer from 0 to 99.

This is the function that allows you to set the log level. The level is an integer between 0 and 9. Higher values mean more verbosity. The default value is 0. Larger values should only be used with care, since they may reveal sensitive information.

Use a log level over 10 to enable all debugging options.

gnutls_global_set_mutex

`void gnutls_global_set_mutex (`*mutex_init_func* `init`, *mutex_deinit_func* `deinit`, *mutex_lock_func* `lock`, *mutex_unlock_func* `unlock`) [Function]

init: mutex initialization function

deinit: mutex deinitialization function

lock: mutex locking function

unlock: mutex unlocking function

With this function you are allowed to override the default mutex locks used in some parts of gnutls and dependent libraries. This function should be used if you have complete control of your program and libraries. Do not call this function from a library, or preferably from any application unless really needed to. GnuTLS will use the appropriate locks for the running system.

Note that since the move to implicit initialization of GnuTLS on library load, calling this function will deinitialize the library, and re-initialize it after the new locking functions are set.

This function must be called prior to any other gnutls function.

Since: 2.12.0

gnutls_global_set_time_function

`void gnutls_global_set_time_function (`*gnutls_time_func* `time_func`) [Function]

time_func: it's the system time function, a `gnutls_time_func()` callback.

This is the function where you can override the default system time function. The application provided function should behave the same as the standard function.

Since: 2.12.0

gnutls_gost_paramset_get_name

`const char * gnutls_gost_paramset_get_name (`*gnutls_gost_paramset_t* `param`) [Function]

param: is a GOST 28147 param set

Convert a `gnutls_gost_paramset_t` value to a string.

Returns: a string that contains the name of the specified GOST param set, or `NULL` .

Since: 3.6.3

gnutls_gost_paramset_get_oid

`const char * gnutls_gost_paramset_get_oid (`*gnutls_gost_paramset_t* `param`) [Function]

param: is a GOST 28147 param set

Convert a `gnutls_gost_paramset_t` value to its object identifier.

Returns: a string that contains the object identifier of the specified GOST param set, or `NULL` .

Since: 3.6.3

gnutls_group_get

`gnutls_group_t gnutls_group_get (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Returns the currently used group for key exchange. Only valid when using an elliptic curve or DH ciphersuite.

Returns: the currently used group, a `gnutls_group_t` type.

Since: 3.6.0

gnutls_group_get_id

`gnutls_group_t gnutls_group_get_id (`*const char * name*`)` [Function]

name: is a group name

The names are compared in a case insensitive way.

Returns: return a `gnutls_group_t` value corresponding to the specified group, or `GNUTLS_GROUP_INVALID` on error.

Since: 3.6.0

gnutls_group_get_name

`const char * gnutls_group_get_name (gnutls_group_t group)` [Function]

group: is an element from `gnutls_group_t`

Convert a `gnutls_group_t` value to a string.

Returns: a string that contains the name of the specified group or `NULL` .

Since: 3.6.0

gnutls_group_list

`const gnutls_group_t * gnutls_group_list ( void)` [Function]

Get the list of supported elliptic curves.

This function is not thread safe.

Returns: Return a (0)-terminated list of `gnutls_group_t` integers indicating the available groups.

Since: 3.6.0

gnutls_handshake

`int gnutls_handshake (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` type.

This function performs the handshake of the TLS/SSL protocol, and initializes the TLS session parameters.

The non-fatal errors expected by this function are: `GNUTLS_E_INTERRUPTED` , `GNUTLS_E_AGAIN` , `GNUTLS_E_WARNING_ALERT_RECEIVED` . When this function is called for re-handshake under TLS 1.2 or earlier, the non-fatal error code `GNUTLS_E_GOT_APPLICATION_DATA` may also be returned.

The former two interrupt the handshake procedure due to the transport layer being interrupted, and the latter because of a "warning" alert that was sent by the peer (it is always a good idea to check any received alerts). On these non-fatal errors call this function again, until it returns 0; cf. `gnutls_record_get_direction()` and `gnutls_error_is_fatal()` . In DTLS sessions the non-fatal error `GNUTLS_E_LARGE_PACKET` is also possible, and indicates that the MTU should be adjusted.

When this function is called by a server after a rehandshake request under TLS 1.2 or earlier the `GNUTLS_E_GOT_APPLICATION_DATA` error code indicates that some data were pending prior to peer initiating the handshake. Under TLS 1.3 this function when called after a successful handshake, is a no-op and always succeeds in server side; in client side this function is equivalent to `gnutls_session_key_update()` with `GNUTLS_KU_PEER` flag.

This function handles both full and abbreviated TLS handshakes (resumption). For abbreviated handshakes, in client side, the `gnutls_session_set_data()` should be called prior to this function to set parameters from a previous session. In server side, resumption is handled by either setting a DB back-end, or setting up keys for session tickets.

Returns: `GNUTLS_E_SUCCESS` on a successful handshake, otherwise a negative error code.

gnutls_handshake_description_get_name

`const char * gnutls_handshake_description_get_name` [Function]
(*gnutls_handshake_description_t* `type`)

type: is a handshake message description

Convert a `gnutls_handshake_description_t` value to a string.

Returns: a string that contains the name of the specified handshake message or `NULL` .

gnutls_handshake_get_last_in

`gnutls_handshake_description_t` [Function]
`gnutls_handshake_get_last_in` (*gnutls_session_t* `session`)

session: is a `gnutls_session_t` type.

This function is only useful to check where the last performed handshake failed. If the previous handshake succeed or was not performed at all then no meaningful value will be returned.

Check `gnutls_handshake_description_t` in gnutls.h for the available handshake descriptions.

Returns: the last handshake message type received, a `gnutls_handshake_description_t` .

gnutls_handshake_get_last_out

`gnutls_handshake_description_t` [Function]
`gnutls_handshake_get_last_out` (*gnutls_session_t* `session`)

session: is a `gnutls_session_t` type.

This function is only useful to check where the last performed handshake failed. If the previous handshake succeed or was not performed at all then no meaningful value will be returned.

Check `gnutls_handshake_description_t` in gnutls.h for the available handshake descriptions.

Returns: the last handshake message type sent, a `gnutls_handshake_description_t` .

gnutls_handshake_set_hook_function

`void gnutls_handshake_set_hook_function` (*gnutls_session_t* `session`, *unsigned int* `htype`, *int* `when`, *gnutls_handshake_hook_func* `func`) [Function]

session: is a `gnutls_session_t` type

htype: the `gnutls_handshake_description_t` of the message to hook at

when: `GNUTLS_HOOK_` * depending on when the hook function should be called

func: is the function to be called

This function will set a callback to be called after or before the specified handshake message has been received or generated. This is a generalization of `gnutls_handshake_set_post_client_hello_function()` .

To call the hook function prior to the message being generated or processed use `GNUTLS_HOOK_PRE` as `when` parameter, `GNUTLS_HOOK_POST` to call after, and `GNUTLS_HOOK_BOTH` for both cases.

This callback must return 0 on success or a gnutls error code to terminate the handshake.

To hook at all handshake messages use an `htype` of `GNUTLS_HANDSHAKE_ANY` .

Warning: You should not use this function to terminate the handshake based on client input unless you know what you are doing. Before the handshake is finished there is no way to know if there is a man-in-the-middle attack being performed.

gnutls_handshake_set_max_packet_length

`void gnutls_handshake_set_max_packet_length (`*gnutls_session_t* `session`, *size_t* `max`) [Function]

session: is a `gnutls_session_t` type.

max: is the maximum number.

This function will set the maximum size of all handshake messages. Handshakes over this size are rejected with `GNUTLS_E_HANDSHAKE_TOO_LARGE` error code. The default value is 128kb which is typically large enough. Set this to 0 if you do not want to set an upper limit.

The reason for restricting the handshake message sizes are to limit Denial of Service attacks.

Note that the maximum handshake size was increased to 128kb from 48kb in GnuTLS 3.5.5.

gnutls_handshake_set_post_client_hello_function

`void gnutls_handshake_set_post_client_hello_function` [Function]
(*gnutls_session_t* `session`, *gnutls_handshake_simple_hook_func* `func`)

session: is a `gnutls_session_t` type.

func: is the function to be called

This function will set a callback to be called after the client hello has been received (callback valid in server side only). This allows the server to adjust settings based on received extensions.

Those settings could be ciphersuites, requesting certificate, or anything else except for version negotiation (this is done before the hello message is parsed).

This callback must return 0 on success or a gnutls error code to terminate the handshake.

Since GnuTLS 3.3.5 the callback is allowed to return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` to put the handshake on hold. In that case `gnutls_handshake()` will return `GNUTLS_E_INTERRUPTED` and can be resumed when needed.

Warning: You should not use this function to terminate the handshake based on client input unless you know what you are doing. Before the handshake is finished there is no way to know if there is a man-in-the-middle attack being performed.

gnutls_handshake_set_private_extensions

`void gnutls_handshake_set_private_extensions` [Function]
(*gnutls_session_t* **`session`**, *int* **`allow`**)

session: is a `gnutls_session_t` type.

allow: is an integer (0 or 1)

This function will enable or disable the use of private cipher suites (the ones that start with 0xFF). By default or if `allow` is 0 then these cipher suites will not be advertised nor used.

Currently GnuTLS does not include such cipher-suites or compression algorithms.

Enabling the private ciphersuites when talking to other than gnutls servers and clients may cause interoperability problems.

gnutls_handshake_set_random

`int gnutls_handshake_set_random` (*gnutls_session_t* **`session`**, [Function]
*const gnutls_datum_t ** **`random`**)

session: is a `gnutls_session_t` type.

random: a random value of 32-bytes

This function will explicitly set the server or client hello random value in the subsequent TLS handshake. The random value should be a 32-byte value.

Note that this function should not normally be used as gnutls will select automatically a random value for the handshake.

This function should not be used when resuming a session.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since 3.1.9

gnutls_handshake_set_timeout

`void gnutls_handshake_set_timeout` (*gnutls_session_t* **`session`**, [Function]
unsigned int **`ms`**)

session: is a `gnutls_session_t` type.

ms: is a timeout value in milliseconds

This function sets the timeout for the TLS handshake process to the provided value. Use an `ms` value of zero to disable timeout, or `GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT` for a reasonable default value. For the DTLS protocol, the more detailed `gnutls_dtls_set_timeouts()` is provided.

This function requires to set a pull timeout callback. See `gnutls_transport_set_pull_timeout_function()` .

Since: 3.1.0

gnutls_heartbeat_allowed

`unsigned gnutls_heartbeat_allowed` (*gnutls_session_t* `session`, *unsigned int* `type`) [Function]

session: is a `gnutls_session_t` type.

type: one of `GNUTLS_HB_LOCAL_ALLOWED_TO_SEND` and `GNUTLS_HB_PEER_ALLOWED_TO_SEND`

This function will check whether heartbeats are allowed to be sent or received in this session.

Returns: Non zero if heartbeats are allowed.

Since: 3.1.2

gnutls_heartbeat_enable

`void gnutls_heartbeat_enable` (*gnutls_session_t* `session`, *unsigned int* `type`) [Function]

session: is a `gnutls_session_t` type.

type: one of the GNUTLS_HB_* flags

If this function is called with the `GNUTLS_HB_PEER_ALLOWED_TO_SEND` `type` , GnuTLS will allow heartbeat messages to be received. Moreover it also request the peer to accept heartbeat messages. This function must be called prior to TLS handshake.

If the `type` used is `GNUTLS_HB_LOCAL_ALLOWED_TO_SEND` , then the peer will be asked to accept heartbeat messages but not send ones.

The function `gnutls_heartbeat_allowed()` can be used to test Whether locally generated heartbeat messages can be accepted by the peer.

Since: 3.1.2

gnutls_heartbeat_get_timeout

`unsigned int gnutls_heartbeat_get_timeout` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the milliseconds remaining for a retransmission of the previously sent ping message. This function is useful when ping is used in non-blocking mode, to estimate when to call `gnutls_heartbeat_ping()` if no packets have been received.

Returns: the remaining time in milliseconds.

Since: 3.1.2

gnutls_heartbeat_ping

`int gnutls_heartbeat_ping` (*gnutls_session_t* `session`, *size_t* `data_size`, *unsigned int* `max_tries`, *unsigned int* `flags`) [Function]

session: is a `gnutls_session_t` type.

data_size: is the length of the ping payload.

max_tries: if flags is GNUTLS_HEARTBEAT_WAIT then this sets the number of retransmissions. Use zero for indefinite (until timeout).

flags: if GNUTLS_HEARTBEAT_WAIT then wait for pong or timeout instead of returning immediately.

This function sends a ping to the peer. If the flags is set to GNUTLS_HEARTBEAT_WAIT then it waits for a reply from the peer.

Note that it is highly recommended to use this function with the flag GNUTLS_HEARTBEAT_WAIT , or you need to handle retransmissions and timeouts manually.

The total TLS data transmitted as part of the ping message are given by the following formula: MAX(16, data_size)+gnutls_record_overhead_size() +3.

Returns: GNUTLS_E_SUCCESS on success, otherwise a negative error code.

Since: 3.1.2

gnutls_heartbeat_pong

`int gnutls_heartbeat_pong (`*gnutls_session_t* ***session***, *unsigned int* ***flags***`)` [Function]

session: is a gnutls_session_t type.

flags: should be zero

This function replies to a ping by sending a pong to the peer.

Returns: GNUTLS_E_SUCCESS on success, otherwise a negative error code.

Since: 3.1.2

gnutls_heartbeat_set_timeouts

`void gnutls_heartbeat_set_timeouts (`*gnutls_session_t* ***session***, *unsigned int* ***retrans_timeout***, *unsigned int* ***total_timeout***`)` [Function]

session: is a gnutls_session_t type.

retrans_timeout: The time at which a retransmission will occur in milliseconds

total_timeout: The time at which the connection will be aborted, in milliseconds.

This function will override the timeouts for the DTLS heartbeat protocol. The retransmission timeout is the time after which a message from the peer is not received, the previous request will be retransmitted. The total timeout is the time after which the handshake will be aborted with GNUTLS_E_TIMEDOUT .

Since: 3.1.2

gnutls_hex2bin

`int gnutls_hex2bin (`*const char ** ***hex_data***, *size_t* ***hex_size***, *void ** ***bin_data***, *size_t ** ***bin_size***`)` [Function]

hex_data: string with data in hex format

hex_size: size of hex data

bin_data: output array with binary data

bin_size: when calling should hold maximum size of bin_data , on return will hold actual length of bin_data .

Convert a buffer with hex data to binary data. This function unlike `gnutls_hex_decode()` can parse hex data with separators between numbers. That is, it ignores any non-hex characters.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 2.4.0

gnutls_hex_decode

`int gnutls_hex_decode` (*const gnutls_datum_t * **hex_data***, *void * **result***, *size_t * **result_size***) [Function]

hex_data: contain the encoded data

result: the place where decoded data will be copied

result_size: holds the size of the result

This function will decode the given encoded data, using the hex encoding used by PSK password files.

Initially `result_size` must hold the maximum size available in `result` , and on return it will contain the number of bytes written.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, `GNUTLS_E_PARSING_ERROR` on invalid hex data, or 0 on success.

gnutls_hex_decode2

`int gnutls_hex_decode2` (*const gnutls_datum_t * **hex_data***, *gnutls_datum_t * **result***) [Function]

hex_data: contain the encoded data

result: the result in an allocated string

This function will decode the given encoded data, using the hex encoding used by PSK password files.

Returns: `GNUTLS_E_PARSING_ERROR` on invalid hex data, or 0 on success.

gnutls_hex_encode

`int gnutls_hex_encode` (*const gnutls_datum_t * **data***, *char * **result***, *size_t * **result_size***) [Function]

data: contain the raw data

result: the place where hex data will be copied

result_size: holds the size of the result

This function will convert the given data to printable data, using the hex encoding, as used in the PSK password files.

Note that the size of the result includes the null terminator.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, or 0 on success.

gnutls_hex_encode2

`int gnutls_hex_encode2` (*const gnutls_datum_t * `data`*, *gnutls_datum_t * `result`*) [Function]

data: contain the raw data

result: the result in an allocated string

This function will convert the given data to printable data, using the hex encoding, as used in the PSK password files.

Note that the size of the result does NOT include the null terminator.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

gnutls_idna_map

`int gnutls_idna_map` (*const char * `input`*, *unsigned `ilen`*, *gnutls_datum_t * `out`*, *unsigned `flags`*) [Function]

input: contain the UTF-8 formatted domain name

ilen: the length of the provided string

out: the result in an null-terminated allocated string

flags: should be zero

This function will convert the provided UTF-8 domain name, to its IDNA mapping in an allocated variable. Note that depending on the flags the used gnutls library was compiled with, the output of this function may vary (i.e., may be IDNA2008, or IDNA2003).

To force IDNA2008 specify the flag `GNUTLS_IDNA_FORCE_2008` . In the case GnuTLS is not compiled with the necessary dependencies, `GNUTLS_E_UNIMPLEMENTED_FEATURE` will be returned to indicate that gnutls is unable to perform the requested conversion.

Note also, that this function will return an empty string if an empty string is provided as input.

Returns: `GNUTLS_E_INVALID_UTF8_STRING` on invalid UTF-8 data, or 0 on success.

Since: 3.5.8

gnutls_idna_reverse_map

`int gnutls_idna_reverse_map` (*const char * `input`*, *unsigned `ilen`*, *gnutls_datum_t * `out`*, *unsigned `flags`*) [Function]

input: contain the ACE (IDNA) formatted domain name

ilen: the length of the provided string

out: the result in an null-terminated allocated UTF-8 string

flags: should be zero

This function will convert an ACE (ASCII-encoded) domain name to a UTF-8 domain name.

If GnuTLS is compiled without IDNA support, then this function will return `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Note also, that this function will return an empty string if an empty string is provided as input.

Returns: A negative error code on error, or 0 on success.

Since: 3.5.8

gnutls_init

`int gnutls_init` (*gnutls_session_t * session, unsigned int flags*) [Function]

session: is a pointer to a `gnutls_session_t` type.

flags: indicate if this session is to be used for server or client.

This function initializes the provided session. Every session must be initialized before use, and must be deinitialized after used by calling `gnutls_deinit()` .

`flags` can be any combination of flags from `gnutls_init_flags_t` .

Note that since version 3.1.2 this function enables some common TLS extensions such as session tickets and OCSP certificate status request in client side by default. To prevent that use the `GNUTLS_NO_EXTENSIONS` flag.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_key_generate

`int gnutls_key_generate` (*gnutls_datum_t * key, unsigned int key_size*) [Function]

key: is a pointer to a `gnutls_datum_t` which will contain a newly created key

key_size: the number of bytes of the key

Generates a random key of `key_size` bytes.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 3.0

gnutls_kx_get

`gnutls_kx_algorithm_t gnutls_kx_get` (*gnutls_session_t session*) [Function]

session: is a `gnutls_session_t` type.

Get the currently used key exchange algorithm.

This function will return `GNUTLS_KX_ECDHE_RSA` , or `GNUTLS_KX_DHE_RSA` under TLS 1.3, to indicate an elliptic curve DH key exchange or a finite field one. The precise group used is available by calling `gnutls_group_get()` instead.

Returns: the key exchange algorithm used in the last handshake, a `gnutls_kx_algorithm_t` value.

gnutls_kx_get_id

`gnutls_kx_algorithm_t gnutls_kx_get_id` (*const char * name*) [Function]

name: is a KX name

Convert a string to a `gnutls_kx_algorithm_t` value. The names are compared in a case insensitive way.

Returns: an id of the specified KX algorithm, or `GNUTLS_KX_UNKNOWN` on error.

gnutls_kx_get_name

`const char * gnutls_kx_get_name (`*gnutls_kx_algorithm_t* *algorithm*`)` [Function]

algorithm: is a key exchange algorithm

Convert a `gnutls_kx_algorithm_t` value to a string.

Returns: a pointer to a string that contains the name of the specified key exchange algorithm, or `NULL` .

gnutls_kx_list

`const gnutls_kx_algorithm_t * gnutls_kx_list ( void)` [Function]

Get a list of supported key exchange algorithms.

This function is not thread safe.

Returns: a (0)-terminated list of `gnutls_kx_algorithm_t` integers indicating the available key exchange algorithms.

gnutls_load_file

`int gnutls_load_file (`*const char * filename, gnutls_datum_t * data*`)` [Function]

filename: the name of the file to load

data: Where the file will be stored

This function will load a file into a datum. The data are zero terminated but the terminating null is not included in length. The returned data are allocated using `gnutls_malloc()` .

Note that this function is not designed for reading sensitive materials, such as private keys, on practical applications. When the reading fails in the middle, the partially loaded content might remain on memory.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since 3.1.0

gnutls_mac_get

`gnutls_mac_algorithm_t gnutls_mac_get (`*gnutls_session_t* *session*`)` [Function]

session: is a `gnutls_session_t` type.

Get the currently used MAC algorithm.

Returns: the currently used mac algorithm, a `gnutls_mac_algorithm_t` value.

gnutls_mac_get_id

`gnutls_mac_algorithm_t gnutls_mac_get_id (`*const char * name*`)` [Function]

name: is a MAC algorithm name

Convert a string to a `gnutls_mac_algorithm_t` value. The names are compared in a case insensitive way.

Returns: a `gnutls_mac_algorithm_t` id of the specified MAC algorithm string, or `GNUTLS_MAC_UNKNOWN` on failure.

gnutls_mac_get_key_size

`size_t gnutls_mac_get_key_size (`*gnutls_mac_algorithm_t* ***algorithm***`)` [Function]

algorithm: is an encryption algorithm

Returns the size of the MAC key used in TLS.

Returns: length (in bytes) of the given MAC key size, or 0 if the given MAC algorithm is invalid.

gnutls_mac_get_name

`const char * gnutls_mac_get_name (`*gnutls_mac_algorithm_t* ***algorithm***`)` [Function]

algorithm: is a MAC algorithm

Convert a `gnutls_mac_algorithm_t` value to a string.

Returns: a string that contains the name of the specified MAC algorithm, or `NULL` .

gnutls_mac_list

`const gnutls_mac_algorithm_t * gnutls_mac_list ( void)` [Function]

Get a list of hash algorithms for use as MACs. Note that not necessarily all MACs are supported in TLS cipher suites. This function is not thread safe.

Returns: Return a (0)-terminated list of `gnutls_mac_algorithm_t` integers indicating the available MACs.

gnutls_memcmp

`int gnutls_memcmp (`*const void * **s1***, *const void * **s2***, *size_t* **n**`)` [Function]

s1: the first address to compare

s2: the second address to compare

n: the size of memory to compare

This function will operate similarly to `memcmp()` , but will operate on time that depends only on the size of the string. That is will not return early if the strings don't match on the first byte.

Returns: non zero on difference and zero if the buffers are identical.

Since: 3.4.0

gnutls_memset

`void gnutls_memset (`*void * **data***, *int* **c**, *size_t* **size**`)` [Function]

data: the memory to set

c: the constant byte to fill the memory with

size: the size of memory

This function will operate similarly to `memset()` , but will not be optimized out by the compiler.

Since: 3.4.0

gnutls_ocsp_status_request_enable_client

`int gnutls_ocsp_status_request_enable_client` [Function]
(*gnutls_session_t* ***session***, *gnutls_datum_t ** ***responder_id***, *size_t* ***responder_id_size***, *gnutls_datum_t ** ***extensions***)

session: is a `gnutls_session_t` type.

responder_id: ignored, must be `NULL`

responder_id_size: ignored, must be zero

extensions: ignored, must be `NULL`

This function is to be used by clients to request OCSP response from the server, using the "status_request" TLS extension. Only OCSP status type is supported.

Previous versions of GnuTLS supported setting `responder_id` and `extensions` fields, but due to the difficult semantics of the parameter usage, and other issues, this support was removed since 3.6.0 and these parameters must be set to `NULL` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.1.3

gnutls_ocsp_status_request_get

`int gnutls_ocsp_status_request_get` (*gnutls_session_t* ***session***, [Function]
*gnutls_datum_t ** ***response***)

session: is a `gnutls_session_t` type.

response: a `gnutls_datum_t` with DER encoded OCSP response

This function returns the OCSP status response received from the TLS server. The `response` should be treated as constant. If no OCSP response is available then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.1.3

gnutls_ocsp_status_request_get2

`int gnutls_ocsp_status_request_get2` (*gnutls_session_t* ***session***, [Function]
unsigned ***idx***, *gnutls_datum_t ** ***response***)

session: is a `gnutls_session_t` type.

idx: the index of peer's certificate

response: a `gnutls_datum_t` with DER encoded OCSP response

This function returns the OCSP status response received from the TLS server for the certificate index provided. The index corresponds to certificates as returned by gnutls_certificate_get_peers. When index is zero this function operates identically to `gnutls_ocsp_status_request_get()` .

The returned `response` should be treated as constant. If no OCSP response is available for the given index then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.6.3

gnutls_ocsp_status_request_is_checked

`unsigned gnutls_ocsp_status_request_is_checked` [Function]
(*gnutls_session_t* `session`, *unsigned int* `flags`)

session: is a gnutls session

flags: should be zero or `GNUTLS_OCSP_SR_IS_AVAIL`

When flags are zero this function returns non-zero if a valid OCSP status response was included in the TLS handshake. That is, an OCSP status response which is not too old, superseded or marks the certificate as revoked. It returns zero otherwise.

When the flag `GNUTLS_OCSP_SR_IS_AVAIL` is specified, the function returns non-zero if an OCSP status response was included in the handshake even if it was invalid. Otherwise, if no OCSP status response was included, it returns zero. The `GNUTLS_OCSP_SR_IS_AVAIL` flag was introduced in GnuTLS 3.4.0.

This is a helper function when needing to decide whether to perform an explicit OCSP validity check on the peer's certificate. Should be called after any of gnutls_certificate_verify_peers*() are called.

This function is always usable on client side, but on server side only under TLS 1.3, which is the first version of TLS that allows cliend-side OCSP responses.

Returns: Non-zero if the response was valid, or a zero if it wasn't sent, or sent and was invalid.

Since: 3.1.4

gnutls_oid_to_digest

`gnutls_digest_algorithm_t gnutls_oid_to_digest` (*const char ** `oid`) [Function]

oid: is an object identifier

Converts a textual object identifier to a `gnutls_digest_algorithm_t` value.

Returns: a `gnutls_digest_algorithm_t` id of the specified digest algorithm, or `GNUTLS_DIG_UNKNOWN` on failure.

Since: 3.4.3

gnutls_oid_to_ecc_curve

`gnutls_ecc_curve_t gnutls_oid_to_ecc_curve (const char * oid)` [Function]

oid: is a curve's OID

Returns: return a `gnutls_ecc_curve_t` value corresponding to the specified OID, or `GNUTLS_ECC_CURVE_INVALID` on error.

Since: 3.4.3

gnutls_oid_to_gost_paramset

`gnutls_gost_paramset_t gnutls_oid_to_gost_paramset (const char * oid)` [Function]

oid: is an object identifier

Converts a textual object identifier to a `gnutls_gost_paramset_t` value.

Returns: a `gnutls_gost_paramset_get_oid` of the specified GOST 28147 param st, or `GNUTLS_GOST_PARAMSET_UNKNOWN` on failure.

Since: 3.6.3

gnutls_oid_to_mac

`gnutls_mac_algorithm_t gnutls_oid_to_mac (const char * oid)` [Function]

oid: is an object identifier

Converts a textual object identifier typically from PKCS5 values to a `gnutls_mac_algorithm_t` value.

Returns: a `gnutls_mac_algorithm_t` id of the specified digest algorithm, or `GNUTLS_MAC_UNKNOWN` on failure.

Since: 3.5.4

gnutls_oid_to_pk

`gnutls_pk_algorithm_t gnutls_oid_to_pk (const char * oid)` [Function]

oid: is an object identifier

Converts a textual object identifier to a `gnutls_pk_algorithm_t` value.

Returns: a `gnutls_pk_algorithm_t` id of the specified digest algorithm, or `GNUTLS_PK_UNKNOWN` on failure.

Since: 3.4.3

gnutls_oid_to_sign

`gnutls_sign_algorithm_t gnutls_oid_to_sign (const char * oid)` [Function]

oid: is an object identifier

Converts a textual object identifier to a `gnutls_sign_algorithm_t` value.

Returns: a `gnutls_sign_algorithm_t` id of the specified digest algorithm, or `GNUTLS_SIGN_UNKNOWN` on failure.

Since: 3.4.3

gnutls_openpgp_send_cert

`void gnutls_openpgp_send_cert (`*gnutls_session_t* **`session`**`,` *gnutls_openpgp_crt_status_t* **`status`**`)` [Function]

session: is a gnutls session

status: is ignored

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

gnutls_packet_deinit

`void gnutls_packet_deinit (`*gnutls_packet_t* **`packet`**`)` [Function]

packet: is a pointer to a `gnutls_packet_st` structure.

This function will deinitialize all data associated with the received packet.

Since: 3.3.5

gnutls_packet_get

`void gnutls_packet_get (`*gnutls_packet_t* **`packet`**`,` *gnutls_datum_t * data*, *unsigned char * sequence*`)` [Function]

packet: is a `gnutls_packet_t` type.

data: will contain the data present in the `packet` structure (may be `NULL`)

sequence: the 8-bytes of the packet sequence number (may be `NULL`)

This function returns the data and sequence number associated with the received packet.

Since: 3.3.5

gnutls_pem_base64_decode

`int gnutls_pem_base64_decode (`*const char * header*, *const gnutls_datum_t * b64_data*, *unsigned char * result*, *size_t * result_size*`)` [Function]

header: A null terminated string with the PEM header (eg. CERTIFICATE)

b64_data: contain the encoded data

result: the place where decoded data will be copied

result_size: holds the size of the result

This function will decode the given encoded data. If the header given is non `NULL` this function will search for "——BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

Returns: On success `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned if the buffer given is not long enough, or 0 on success.

gnutls_pem_base64_decode2

`int gnutls_pem_base64_decode2 (`*const char * `header`, const gnutls_datum_t * `b64_data`, gnutls_datum_t * `result`*`)` [Function]

header: The PEM header (eg. CERTIFICATE)

b64_data: contains the encoded data

result: the location of decoded data

This function will decode the given encoded data. The decoded data will be allocated, and stored into result. If the header given is non null this function will search for "—–BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

You should use `gnutls_free()` to free the returned data.

Note, that prior to GnuTLS 3.4.0 this function was available under the name `gnutls_pem_base64_decode_alloc()` . There is compatibility macro pointing to this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.4.0

gnutls_pem_base64_encode

`int gnutls_pem_base64_encode (`*const char * `msg`, const gnutls_datum_t * `data`, char * `result`, size_t * `result_size`*`)` [Function]

msg: is a message to be put in the header (may be `NULL`)

data: contain the raw data

result: the place where base64 data will be copied

result_size: holds the size of the result

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in PEM messages.

The output string will be null terminated, although the output size will not include the terminating null.

Returns: On success `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned if the buffer given is not long enough, or 0 on success.

gnutls_pem_base64_encode2

`int gnutls_pem_base64_encode2 (`*const char * `header`, const gnutls_datum_t * `data`, gnutls_datum_t * `result`*`)` [Function]

header: is a message to be put in the encoded header (may be `NULL`)

data: contains the raw data

result: will hold the newly allocated encoded data

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in PEM messages. This function will allocate the required memory to hold the encoded data.

You should use `gnutls_free()` to free the returned data.

Note, that prior to GnuTLS 3.4.0 this function was available under the name `gnutls_pem_base64_encode_alloc()` . There is compatibility macro pointing to this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.4.0

gnutls_perror

`void gnutls_perror (`*int* `error``)` [Function]

error: is a GnuTLS error code, a negative error code

This function is like `perror()` . The only difference is that it accepts an error number returned by a gnutls function.

gnutls_pk_algorithm_get_name

`const char * gnutls_pk_algorithm_get_name` [Function]
`(`*gnutls_pk_algorithm_t* `algorithm`)

algorithm: is a pk algorithm

Convert a `gnutls_pk_algorithm_t` value to a string.

Returns: a string that contains the name of the specified public key algorithm, or `NULL` .

gnutls_pk_bits_to_sec_param

`gnutls_sec_param_t gnutls_pk_bits_to_sec_param` [Function]
`(`*gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`)

algo: is a public key algorithm

bits: is the number of bits

This is the inverse of `gnutls_sec_param_to_pk_bits()` . Given an algorithm and the number of bits, it will return the security parameter. This is a rough indication.

Returns: The security parameter.

Since: 2.12.0

gnutls_pk_get_id

`gnutls_pk_algorithm_t gnutls_pk_get_id (`*const char ** `name``)` [Function]

name: is a string containing a public key algorithm name.

Convert a string to a `gnutls_pk_algorithm_t` value. The names are compared in a case insensitive way. For example, gnutls_pk_get_id("RSA") will return `GNUTLS_PK_RSA` .

Returns: a `gnutls_pk_algorithm_t` id of the specified public key algorithm string, or `GNUTLS_PK_UNKNOWN` on failures.

Since: 2.6.0

gnutls_pk_get_name

`const char * gnutls_pk_get_name (gnutls_pk_algorithm_t algorithm)` [Function]

algorithm: is a public key algorithm

Convert a `gnutls_pk_algorithm_t` value to a string.

Returns: a pointer to a string that contains the name of the specified public key algorithm, or `NULL` .

Since: 2.6.0

gnutls_pk_get_oid

`const char * gnutls_pk_get_oid (gnutls_pk_algorithm_t algorithm)` [Function]

algorithm: is a public key algorithm

Convert a `gnutls_pk_algorithm_t` value to its object identifier string.

Returns: a pointer to a string that contains the object identifier of the specified public key algorithm, or `NULL` .

Since: 3.4.3

gnutls_pk_list

`const gnutls_pk_algorithm_t * gnutls_pk_list ( void)` [Function]

Get a list of supported public key algorithms.

This function is not thread safe.

Returns: a (0)-terminated list of `gnutls_pk_algorithm_t` integers indicating the available ciphers.

Since: 2.6.0

gnutls_pk_to_sign

`gnutls_sign_algorithm_t gnutls_pk_to_sign (gnutls_pk_algorithm_t pk, gnutls_digest_algorithm_t hash)` [Function]

pk: is a public key algorithm

hash: a hash algorithm

This function maps public key and hash algorithms combinations to signature algorithms.

Returns: return a `gnutls_sign_algorithm_t` value, or `GNUTLS_SIGN_UNKNOWN` on error.

gnutls_prf

`int gnutls_prf (gnutls_session_t session, size_t label_size, const char * label, int server_random_first, size_t extra_size, const char * extra, size_t outsize, char * out)` [Function]

session: is a `gnutls_session_t` type.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

server_random_first: non-zero if server random field should be first in seed

extra_size: length of the `extra` variable.

extra: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-allocated buffer to hold the generated data.

Applies the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data, seeded with the client and server random fields. For the key expansion specified in RFC5705 see `gnutls_prf_rfc5705()` .

The `label` variable usually contains a string denoting the purpose for the generated data. The `server_random_first` indicates whether the client random field or the server random field should be first in the seed. Non-zero indicates that the server random field is first, 0 that the client random field is first.

The `extra` variable can be used to add more data to the seed, after the random variables. It can be used to make sure the generated output is strongly connected to some additional data (e.g., a string used in user authentication).

The output is placed in `out` , which must be pre-allocated.

Note: This function produces identical output with `gnutls_prf_rfc5705()` when `server_random_first` is set to 0 and `extra` is `NULL` . Under TLS1.3 this function will only operate when these conditions are true, or otherwise return `GNUTLS_E_INVALID_REQUEST` .

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_prf_early

`int gnutls_prf_early` (*gnutls_session_t* `session`, *size_t* `label_size`, *const char * * `label`, *size_t* `context_size`, *const char * `context`, *size_t* `outsize`, *char * `out`) [Function]

session: is a `gnutls_session_t` type.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

context_size: length of the `extra` variable.

context: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-allocated buffer to hold the generated data.

This function is similar to `gnutls_prf_rfc5705()` , but only works in TLS 1.3 or later to export early keying material.

Note that the keying material is only available after the ClientHello message is processed and before the application traffic keys are established. Therefore this function shall be called in a handshake hook function for `GNUTLS_HANDSHAKE_CLIENT_HELLO` .

The `label` variable usually contains a string denoting the purpose for the generated data.

The `context` variable can be used to add more data to the seed, after the random variables. It can be used to make sure the generated output is strongly connected to some additional data (e.g., a string used in user authentication).

The output is placed in `out` , which must be pre-allocated.

Note that, to provide the RFC5705 context, the `context` variable must be non-null.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 3.6.8

gnutls_prf_hash_get

`gnutls_digest_algorithm_t gnutls_prf_hash_get (`*const gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Get the currently used hash algorithm. In TLS 1.3, the hash algorithm is used for both the key derivation function and handshake message authentication code. In TLS 1.2, it matches the hash algorithm used for PRF.

Returns: the currently used hash algorithm, a `gnutls_digest_algorithm_t` value.

Since: 3.6.13

gnutls_prf_raw

`int gnutls_prf_raw (`*gnutls_session_t* `session`, *size_t* `label_size`, *const char ** `label`, *size_t* `seed_size`, *const char ** `seed`, *size_t* `outsize`, *char ** `out`) [Function]

session: is a `gnutls_session_t` type.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

seed_size: length of the `seed` variable.

seed: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-allocated buffer to hold the generated data.

Apply the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data.

The `label` variable usually contains a string denoting the purpose for the generated data. The `seed` usually contains data such as the client and server random, perhaps together with some additional data that is added to guarantee uniqueness of the output for a particular purpose.

Because the output is not guaranteed to be unique for a particular session unless `seed` includes the client random and server random fields (the PRF would output the same data on another connection resumed from the first one), it is not recommended to use this function directly. The `gnutls_prf()` function seeds the PRF with the client and server random fields directly, and is recommended if you want to generate pseudo random data unique for each session.

Note: This function will only operate under TLS versions prior to 1.3. In TLS1.3 the use of PRF is replaced with HKDF and the generic exporters like `gnutls_prf_rfc5705()` should be used instead. Under TLS1.3 this function returns `GNUTLS_E_INVALID_REQUEST` .

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_prf_rfc5705

`int gnutls_prf_rfc5705` (*gnutls_session_t* `session`, *size_t* `label_size`, *const char * * `label`, *size_t* `context_size`, *const char * `context`, *size_t* `outsize`, *char * * `out`) [Function]

session: is a `gnutls_session_t` type.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

context_size: length of the `extra` variable.

context: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-allocated buffer to hold the generated data.

Exports keying material from TLS/DTLS session to an application, as specified in RFC5705.

In the TLS versions prior to 1.3, it applies the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data, seeded with the client and server random fields.

In TLS 1.3, it applies HKDF on the exporter master secret derived from the master secret.

The `label` variable usually contains a string denoting the purpose for the generated data.

The `context` variable can be used to add more data to the seed, after the random variables. It can be used to make sure the generated output is strongly connected to some additional data (e.g., a string used in user authentication).

The output is placed in `out` , which must be pre-allocated.

Note that, to provide the RFC5705 context, the `context` variable must be non-null.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 3.4.4

gnutls_priority_certificate_type_list

`int gnutls_priority_certificate_type_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** `list`) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available certificate types in the priority structure.

As of version 3.6.4 this function is an alias for gnutls_priority_certificate_type_list2 with the target parameter set to: - GNUTLS_CTYPE_SERVER, if the `SERVER_PRECEDENCE` option is set - GNUTLS_CTYPE_CLIENT, otherwise.

Returns: the number of certificate types, or an error code.

Since: 3.0

gnutls_priority_certificate_type_list2

`int gnutls_priority_certificate_type_list2` (*gnutls_priority_t* `pcache`, *const unsigned int ** *`list`*, *gnutls_ctype_target_t* `target`) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list.

target: is a `gnutls_ctype_target_t` type. Valid arguments are GNUTLS_CTYPE_CLIENT and GNUTLS_CTYPE_SERVER

Get a list of available certificate types for the given target in the priority structure.

Returns: the number of certificate types, or an error code.

Since: 3.6.4

gnutls_priority_cipher_list

`int gnutls_priority_cipher_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** *`list`*) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available ciphers in the priority structure.

Returns: the number of items, or an error code.

Since: 3.2.3

gnutls_priority_deinit

`void gnutls_priority_deinit` (*gnutls_priority_t* `priority_cache`) [Function]

priority_cache: is a `gnutls_prioritity_t` type.

Deinitializes the priority cache.

gnutls_priority_ecc_curve_list

`int gnutls_priority_ecc_curve_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** *`list`*) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available elliptic curves in the priority structure.

Deprecated: This function has been replaced by `gnutls_priority_group_list()` since 3.6.0.

Returns: the number of items, or an error code.

Since: 3.0

gnutls_priority_get_cipher_suite_index

`int gnutls_priority_get_cipher_suite_index (`*gnutls_priority_t* `pcache`*, unsigned int* `idx`*, unsigned int * `sidx`*)* [Function]

pcache: is a `gnutls_prioritity_t` type.

idx: is an index number.

sidx: internal index of cipher suite to get information about.

Provides the internal ciphersuite index to be used with `gnutls_cipher_suite_info()` . The index `idx` provided is an index kept at the priorities structure. It might be that a valid priorities index does not correspond to a ciphersuite and in that case `GNUTLS_E_UNKNOWN_CIPHER_SUITE` will be returned. Once the last available index is crossed then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Returns: On success it returns `GNUTLS_E_SUCCESS` (0), or a negative error value otherwise.

Since: 3.0.9

gnutls_priority_group_list

`int gnutls_priority_group_list (`*gnutls_priority_t* `pcache`*, const unsigned int ** `list`*)* [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available groups in the priority structure.

Returns: the number of items, or an error code.

Since: 3.6.0

gnutls_priority_init

`int gnutls_priority_init (`*gnutls_priority_t * `priority_cache`, const char * `priorities`, const char ** `err_pos`*)* [Function]

priority_cache: is a `gnutls_prioritity_t` type.

priorities: is a string describing priorities (may be `NULL`)

err_pos: In case of an error this will have the position in the string the error occurred

For applications that do not modify their crypto settings per release, consider using `gnutls_priority_init2()` with `GNUTLS_PRIORITY_INIT_DEF_APPEND` flag instead. We suggest to use centralized crypto settings handled by the GnuTLS library, and applications modifying the default settings to their needs.

This function is identical to `gnutls_priority_init2()` with zero flags.

A `NULL` `priorities` string indicates the default priorities to be used (this is available since GnuTLS 3.3.0).

Returns: On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_priority_init2

`int gnutls_priority_init2 (`*gnutls_priority_t * `priority_cache*, *const char * `priorities*, *const char ** `err_pos*, *unsigned `flags*`)` [Function]

priority_cache: is a `gnutls_prioritity_t` type.

priorities: is a string describing priorities (may be `NULL`)

err_pos: In case of an error this will have the position in the string the error occurred

flags: zero or `GNUTLS_PRIORITY_INIT_DEF_APPEND`

Sets priorities for the ciphers, key exchange methods, and macs. The `priority_cache` should be deinitialized using `gnutls_priority_deinit()` .

The `priorities` option allows you to specify a colon separated list of the cipher priorities to enable. Some keywords are defined to provide quick access to common preferences.

When `flags` is set to `GNUTLS_PRIORITY_INIT_DEF_APPEND` then the `priorities` specified will be appended to the default options.

Unless there is a special need, use the "NORMAL" keyword to apply a reasonable security level, or "NORMAL:%COMPAT" for compatibility.

"PERFORMANCE" means all the "secure" ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance.

"LEGACY" the NORMAL settings for GnuTLS 3.2.x or earlier. There is no verification profile set, and the allowed DH primes are considered weak today.

"NORMAL" means all "secure" ciphersuites. The 256-bit ciphers are included as a fallback only. The ciphers are sorted by security margin.

"PFS" means all "secure" ciphersuites that support perfect forward secrecy. The 256-bit ciphers are included as a fallback only. The ciphers are sorted by security margin.

"SECURE128" means all "secure" ciphersuites of security level 128-bit or more.

"SECURE192" means all "secure" ciphersuites of security level 192-bit or more.

"SUITEB128" means all the NSA SuiteB ciphersuites with security level of 128.

"SUITEB192" means all the NSA SuiteB ciphersuites with security level of 192.

"NONE" means nothing is enabled. This disables everything, including protocols.

"@KEYWORD1,KEYWORD2,..." The system administrator imposed settings. The provided keyword(s) will be expanded from a configuration-time provided file - default is: /etc/gnutls/config. Any attributes that follow it, will be appended to the expanded string. If multiple keywords are provided, separated by commas, then the first keyword that exists in the configuration file will be used. At least one of the keywords must exist, or this function will return an error. Typical usage would be to specify an application specified keyword first, followed by "SYSTEM" as a default fallback. e.g., " `LIBVIRT` ,SYSTEM:!-VERS-SSL3.0" will first try to find a config file entry matching "LIBVIRT", but if that does not exist will use the entry for "SYSTEM". If "SYSTEM" does not exist either, an error will be returned. In all cases, the SSL3.0 protocol will be disabled. The system priority file entries should be formatted as "KEYWORD=VALUE", e.g., "SYSTEM=NORMAL:+ARCFOUR-128".

Special keywords are "!", "-" and "+". "!" or "-" appended with an algorithm will remove this algorithm. "+" appended with an algorithm will add this algorithm.

Check the GnuTLS manual section "Priority strings" for detailed information.

Examples: "NONE:+VERS-TLS-ALL:+MAC-ALL:+RSA:+AES-128-CBC:+SIGN-ALL:+COMP-NULL"

"NORMAL:+ARCFOUR-128" means normal ciphers plus ARCFOUR-128.

"SECURE128:-VERS-SSL3.0" means that only secure ciphers are and enabled, SSL3.0 is disabled.

"NONE:+VERS-TLS-ALL:+AES-128-CBC:+RSA:+SHA1:+COMP-NULL:+SIGN-RSA-SHA1",

"NONE:+VERS-TLS-ALL:+AES-128-CBC:+ECDHE-RSA:+SHA1:+COMP-NULL:+SIGN-RSA-SHA1:+CURVE-SECP256R1",

"SECURE256:+SECURE128",

Note that "NORMAL:%COMPAT" is the most compatible mode.

A `NULL` `priorities` string indicates the default priorities to be used (this is available since GnuTLS 3.3.0).

Returns: On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 3.6.3

gnutls_priority_kx_list

`int gnutls_priority_kx_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** `list`*) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available key exchange methods in the priority structure.

Returns: the number of items, or an error code.

Since: 3.2.3

gnutls_priority_mac_list

`int gnutls_priority_mac_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** `list`*) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available MAC algorithms in the priority structure.

Returns: the number of items, or an error code.

Since: 3.2.3

gnutls_priority_protocol_list

`int gnutls_priority_protocol_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** * `list`) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available TLS version numbers in the priority structure.

Returns: the number of protocols, or an error code.

Since: 3.0

gnutls_priority_set

`int gnutls_priority_set` (*gnutls_session_t* `session`, *gnutls_priority_t* `priority`) [Function]

session: is a `gnutls_session_t` type.

priority: is a `gnutls_priority_t` type.

Sets the priorities to use on the ciphers, key exchange methods, and macs. Note that this function is expected to be called once per session; when called multiple times (e.g., before a re-handshake, the caller should make sure that any new settings are not incompatible with the original session).

Returns: `GNUTLS_E_SUCCESS` on success, or an error code on error.

gnutls_priority_set_direct

`int gnutls_priority_set_direct` (*gnutls_session_t* `session`, *const char * * `priorities`, *const char ** * `err_pos`) [Function]

session: is a `gnutls_session_t` type.

priorities: is a string describing priorities

err_pos: In case of an error this will have the position in the string the error occurred

Sets the priorities to use on the ciphers, key exchange methods, and macs. This function avoids keeping a priority cache and is used to directly set string priorities to a TLS session. For documentation check the `gnutls_priority_init()` .

To use a reasonable default, consider using `gnutls_set_default_priority()` , or `gnutls_set_default_priority_append()` instead of this function.

Returns: On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

gnutls_priority_sign_list

`int gnutls_priority_sign_list` (*gnutls_priority_t* `pcache`, *const unsigned int ** * `list`) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available signature algorithms in the priority structure.

Returns: the number of algorithms, or an error code.

Since: 3.0

gnutls_priority_string_list

`const char * gnutls_priority_string_list (unsigned iter, unsigned int flags)` [Function]

iter: an integer counter starting from zero

flags: one of `GNUTLS_PRIORITY_LIST_INIT_KEYWORDS` , `GNUTLS_PRIORITY_LIST_SPECIAL`

Can be used to iterate all available priority strings. Due to internal implementation details, there are cases where this function can return the empty string. In that case that string should be ignored. When no strings are available it returns `NULL` .

Returns: a priority string

Since: 3.4.0

gnutls_protocol_get_id

`gnutls_protocol_t gnutls_protocol_get_id (const char * name)` [Function]

name: is a protocol name

The names are compared in a case insensitive way.

Returns: an id of the specified protocol, or `GNUTLS_VERSION_UNKNOWN` on error.

gnutls_protocol_get_name

`const char * gnutls_protocol_get_name (gnutls_protocol_t version)` [Function]

version: is a (gnutls) version number

Convert a `gnutls_protocol_t` value to a string.

Returns: a string that contains the name of the specified TLS version (e.g., "TLS1.0"), or `NULL` .

gnutls_protocol_get_version

`gnutls_protocol_t gnutls_protocol_get_version (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` type.

Get TLS version, a `gnutls_protocol_t` value.

Returns: The version of the currently used protocol.

gnutls_protocol_list

`const gnutls_protocol_t * gnutls_protocol_list ( void)` [Function]

Get a list of supported protocols, e.g. SSL 3.0, TLS 1.0 etc.

This function is not thread safe.

Returns: a (0)-terminated list of `gnutls_protocol_t` integers indicating the available protocols.

gnutls_psk_allocate_client_credentials

`int gnutls_psk_allocate_client_credentials` [Function]
(*gnutls_psk_client_credentials_t * sc*)

sc: is a pointer to a `gnutls_psk_server_credentials_t` type.

Allocate a gnutls_psk_client_credentials_t structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_psk_allocate_server_credentials

`int gnutls_psk_allocate_server_credentials` [Function]
(*gnutls_psk_server_credentials_t * sc*)

sc: is a pointer to a `gnutls_psk_server_credentials_t` type.

Allocate a gnutls_psk_server_credentials_t structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_psk_client_get_hint

`const char * gnutls_psk_client_get_hint (`*gnutls_session_t* [Function]
session)

session: is a gnutls session

The PSK identity hint may give the client help in deciding which username to use. This should only be called in case of PSK authentication and in case of a client.

Note: there is no hint in TLS 1.3, so this function will return `NULL` if TLS 1.3 has been negotiated.

Returns: the identity hint of the peer, or `NULL` in case of an error or if TLS 1.3 is being used.

Since: 2.4.0

gnutls_psk_free_client_credentials

`void gnutls_psk_free_client_credentials` [Function]
(*gnutls_psk_client_credentials_t sc*)

sc: is a `gnutls_psk_client_credentials_t` type.

Free a gnutls_psk_client_credentials_t structure.

gnutls_psk_free_server_credentials

`void gnutls_psk_free_server_credentials` [Function]
(*gnutls_psk_server_credentials_t sc*)

sc: is a `gnutls_psk_server_credentials_t` type.

Free a gnutls_psk_server_credentials_t structure.

gnutls_psk_server_get_username

`const char * gnutls_psk_server_get_username (gnutls_session_t session)` [Function]

session: is a gnutls session

This should only be called in case of PSK authentication and in case of a server.

The returned pointer should be considered constant (do not free) and valid for the lifetime of the session.

This function will return `NULL` if the username has embedded NULL bytes. In that case, `gnutls_psk_server_get_username2()` should be used to retrieve the username.

Returns: the username of the peer, or `NULL` in case of an error, or if the username has embedded NULLs.

gnutls_psk_server_get_username2

`int gnutls_psk_server_get_username2 (gnutls_session_t session, gnutls_datum_t * username)` [Function]

session: is a gnutls session

username: a datum that will be filled in by this function

Return a pointer to the username of the peer in the supplied datum. Does not need to be null-terminated.

This should only be called in case of PSK authentication and in case of a server.

The returned pointer should be considered constant (do not free) and valid for the lifetime of the session.

Returns: `GNUTLS_E_SUCCESS` , or a negative value in case of an error.

gnutls_psk_set_client_credentials

`int gnutls_psk_set_client_credentials (gnutls_psk_client_credentials_t res, const char * username, const gnutls_datum_t * key, gnutls_psk_key_flags flags)` [Function]

res: is a `gnutls_psk_client_credentials_t` type.

username: is the user's zero-terminated userid

key: is the user's key

flags: indicate the format of the key, either `GNUTLS_PSK_KEY_RAW` or `GNUTLS_PSK_KEY_HEX` .

This function sets the username and password, in a gnutls_psk_client_credentials_t type. Those will be used in PSK authentication. `username` should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265). The key can be either in raw byte format or in Hex format (without the 0x prefix).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_psk_set_client_credentials2

`int gnutls_psk_set_client_credentials2` [Function]
(*gnutls_psk_client_credentials_t* **res**, *const gnutls_datum_t ** **username**, *const gnutls_datum_t ** **key**, *gnutls_psk_key_flags* **flags**)

res: is a `gnutls_psk_client_credentials_t` type.

username: is the userid

key: is the user's key

flags: indicate the format of the key, either `GNUTLS_PSK_KEY_RAW` or `GNUTLS_PSK_KEY_HEX` .

This function is identical to `gnutls_psk_set_client_credentials()` , except that it allows a non-null-terminated username to be introduced.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_psk_set_client_credentials_function

`void gnutls_psk_set_client_credentials_function` [Function]
(*gnutls_psk_client_credentials_t* **cred**, *gnutls_psk_client_credentials_function ** **func**)

cred: is a `gnutls_psk_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the username and password for client PSK authentication. The callback's function form is: int (*callback)(gnutls_session_t, char** username, gnutls_datum_t* key);

The `username` and `key` ->data must be allocated using `gnutls_malloc()` . The `username` should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265).

The callback function will be called once per handshake.

The callback function should return 0 on success. -1 indicates an error.

gnutls_psk_set_client_credentials_function2

`void gnutls_psk_set_client_credentials_function2` [Function]
(*gnutls_psk_client_credentials_t* **cred**, *gnutls_psk_client_credentials_function2 ** **func**)

cred: is a `gnutls_psk_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the username and password for client PSK authentication. The callback's function form is: int (*callback)(gnutls_session_t, gnutls_datum_t* username, gnutls_datum_t* key);

This callback function has the same semantics as that of `gnutls_psk_set_client_credentials_function()` , but it allows non-string usernames to be used.

The `username` and `key` ->data must be allocated using `gnutls_malloc()` . The `username` should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265).

The callback function will be called once per handshake.

The callback function should return 0 on success. -1 indicates an error.

gnutls_psk_set_params_function

`void gnutls_psk_set_params_function` [Function]
(*gnutls_psk_server_credentials_t* **`res`**, *gnutls_params_function ** **`func`**)

res: is a gnutls_psk_server_credentials_t type

func: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for PSK authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_psk_set_server_credentials_file

`int gnutls_psk_set_server_credentials_file` [Function]
(*gnutls_psk_server_credentials_t* **`res`**, *const char ** **`password_file`**)

res: is a `gnutls_psk_server_credentials_t` type.

password_file: is the PSK password file (passwd.psk)

This function sets the password file, in a `gnutls_psk_server_credentials_t` type. This password file holds usernames and keys and will be used for PSK authentication.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_psk_set_server_credentials_function

`void gnutls_psk_set_server_credentials_function` [Function]
(*gnutls_psk_server_credentials_t* **`cred`**,
*gnutls_psk_server_credentials_function ** **`func`**)

cred: is a `gnutls_psk_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the user's PSK credentials. The callback's function form is: int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t* key);

`username` contains the actual username. The `key` must be filled in using the `gnutls_malloc()` .

In case the callback returned a negative number then gnutls will assume that the username does not exist.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

gnutls_psk_set_server_credentials_function2

`void gnutls_psk_set_server_credentials_function2` [Function]
(*gnutls_psk_server_credentials_t* `cred`, *gnutls_psk_server_credentials_function2* `func`)

cred: is a `gnutls_psk_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the user's PSK credentials. The callback's function form is: int (*callback)(gnutls_session_t, const gnutls_datum_t* username, gnutls_datum_t* key);

This callback function has the same semantics as that of `gnutls_psk_set_server_credentials_function()` , but it allows non-string usernames to be used.

`username` contains the actual username. The `key` must be filled in using the `gnutls_malloc()` .

In case the callback returned a negative number then gnutls will assume that the username does not exist.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

gnutls_psk_set_server_credentials_hint

`int gnutls_psk_set_server_credentials_hint` [Function]
(*gnutls_psk_server_credentials_t* `res`, *const char * * `hint`)

res: is a `gnutls_psk_server_credentials_t` type.

hint: is the PSK identity hint string

This function sets the identity hint, in a `gnutls_psk_server_credentials_t` type. This hint is sent to the client to help it chose a good PSK credential (i.e., username and password).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 2.4.0

gnutls_psk_set_server_dh_params

`void gnutls_psk_set_server_dh_params` [Function]
(*gnutls_psk_server_credentials_t* `res`, *gnutls_dh_params_t* `dh_params`)

res: is a gnutls_psk_server_credentials_t type

dh_params: is a structure that holds Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for an anonymous server to use. These parameters will be used in Diffie-Hellman exchange with PSK cipher suites.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_psk_set_server_known_dh_params

`int gnutls_psk_set_server_known_dh_params` [Function]
(*gnutls_psk_server_credentials_t* `res`, *gnutls_sec_param_t* `sec_param`)

res: is a gnutls_psk_server_credentials_t type

sec_param: is an option of the `gnutls_sec_param_t` enumeration

This function will set the Diffie-Hellman parameters for a PSK server to use. These parameters will be used in Ephemeral Diffie-Hellman cipher suites and will be selected from the FFDHE set of RFC7919 according to the security level provided.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.6

gnutls_psk_set_server_params_function

`void gnutls_psk_set_server_params_function` [Function]
(*gnutls_psk_server_credentials_t* `res`, *gnutls_params_function* * `func`)

res: is a `gnutls_certificate_credentials_t` type

func: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman parameters for PSK authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

gnutls_random_art

`int gnutls_random_art` (*gnutls_random_art_t* `type`, *const char* * `key_type`, *unsigned int* `key_size`, *void* * `fpr`, *size_t* `fpr_size`, *gnutls_datum_t* * `art`) [Function]

type: The type of the random art (for now only `GNUTLS_RANDOM_ART_OPENSSH` is supported)

key_type: The type of the key (RSA, DSA etc.)

key_size: The size of the key in bits

fpr: The fingerprint of the key

fpr_size: The size of the fingerprint

art: The returned random art

This function will convert a given fingerprint to an "artistic" image. The returned image is allocated using `gnutls_malloc()` , is null-terminated but art->size will not account the terminating null.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_range_split

`int gnutls_range_split` (*gnutls_session_t* **`session`**, *const gnutls_range_st * orig*, *gnutls_range_st * next*, *gnutls_range_st * remainder*) [Function]

session: is a `gnutls_session_t` type

orig: is the original range provided by the user

next: is the returned range that can be conveyed in a TLS record

remainder: is the returned remaining range

This function should be used when it is required to hide the length of very long data that cannot be directly provided to `gnutls_record_send_range()` . In that case this function should be called with the desired length hiding range in `orig` . The returned `next` value should then be used in the next call to `gnutls_record_send_range()` with the partial data. That process should be repeated until `remainder` is (0,0).

Returns: 0 in case splitting succeeds, non zero in case of error. Note that `orig` is not changed, while the values of `next` and `remainder` are modified to store the resulting values.

gnutls_reauth

`int gnutls_reauth` (*gnutls_session_t* **`session`**, *unsigned int flags*) [Function]

session: is a `gnutls_session_t` type.

flags: must be zero

This function performs the post-handshake authentication for TLS 1.3. The post-handshake authentication is initiated by the server by calling this function. Clients respond when `GNUTLS_E_REAUTH_REQUEST` has been seen while receiving data.

The non-fatal errors expected by this function are: `GNUTLS_E_INTERRUPTED` , `GNUTLS_E_AGAIN` , as well as `GNUTLS_E_GOT_APPLICATION_DATA` when called on server side.

The former two interrupt the authentication procedure due to the transport layer being interrupted, and the latter because there were pending data prior to peer initiating the re-authentication. The server should read/process that data as unauthenticated and retry calling `gnutls_reauth()` .

When this function is called under TLS1.2 or earlier or the peer didn't advertise post-handshake auth, it always fails with `GNUTLS_E_INVALID_REQUEST` . The verification of the received peers certificate is delegated to the session or credentials verification callbacks. A server can check whether post handshake authentication is supported by the client by checking the session flags with `gnutls_session_get_flags()` .

Prior to calling this function in server side, the function `gnutls_certificate_server_set_request()` must be called setting expectations for the received certificate (request or require). If none are set this function will return with `GNUTLS_E_INVALID_REQUEST` .

Note that post handshake authentication is available irrespective of the initial negotiation type (PSK or certificate). In all cases however, certificate credentials must be set to the session prior to calling this function.

Returns: `GNUTLS_E_SUCCESS` on a successful authentication, otherwise a negative error code.

gnutls_record_can_use_length_hiding

`unsigned gnutls_record_can_use_length_hiding` [Function]
(*gnutls_session_t* **`session`**)

session: is a `gnutls_session_t` type.

If the session supports length-hiding padding, you can invoke `gnutls_record_send_range()` to send a message whose length is hidden in the given range. If the session does not support length hiding padding, you can use the standard `gnutls_record_send()` function, or `gnutls_record_send_range()` making sure that the range is the same as the length of the message you are trying to send.

Returns: true (1) if the current session supports length-hiding padding, false (0) if the current session does not.

gnutls_record_check_corked

`size_t gnutls_record_check_corked` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` type.

This function checks if there pending corked data in the gnutls buffers –see `gnutls_record_cork()` .

Returns: Returns the size of the corked data or zero.

Since: 3.2.8

gnutls_record_check_pending

`size_t gnutls_record_check_pending` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` type.

This function checks if there are unread data in the gnutls buffers. If the return value is non-zero the next call to `gnutls_record_recv()` is guaranteed not to block.

Returns: Returns the size of the data or zero.

gnutls_record_cork

`void gnutls_record_cork` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` type.

If called, `gnutls_record_send()` will no longer send any records. Any sent records will be cached until `gnutls_record_uncork()` is called.

This function is safe to use with DTLS after GnuTLS 3.3.0.

Since: 3.1.9

gnutls_record_disable_padding

`void gnutls_record_disable_padding` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` type.

Used to disabled padding in TLS 1.0 and above. Normally you do not need to use this function, but there are buggy clients that complain if a server pads the encrypted data. This of course will disable protection against statistical attacks on the data.

This function is defunct since 3.1.7. Random padding is disabled by default unless requested using `gnutls_record_send_range()` .

gnutls_record_discard_queued

`size_t gnutls_record_discard_queued` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

This function discards all queued to be sent packets in a DTLS session. These are the packets queued after an interrupted `gnutls_record_send()` .

This function can only be used with transports where `send()` is an all-or-nothing operation (e.g., UDP). When partial writes are allowed this function will cause session errors.

Returns: The number of bytes discarded.

Since: 3.4.0

gnutls_record_get_direction

`int gnutls_record_get_direction` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

This function is useful to determine whether a GnuTLS function was interrupted while sending or receiving, so that `select()` or `poll()` may be called appropriately.

It provides information about the internals of the record protocol and is only useful if a prior gnutls function call, e.g. `gnutls_handshake()` , was interrupted and returned `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` . After such an interrupt applications may call `select()` or `poll()` before restoring the interrupted GnuTLS function.

This function's output is unreliable if you are using the same `session` in different threads for sending and receiving.

Returns: 0 if interrupted while trying to read data, or 1 while trying to write data.

gnutls_record_get_max_early_data_size

`size_t gnutls_record_get_max_early_data_size` [Function]
(*gnutls_session_t* ***session***)

session: is a `gnutls_session_t` type.

This function returns the maximum early data size in this connection. This property can only be set to servers. The client may be provided with the maximum allowed size through the "early_data" extension of the NewSessionTicket handshake message.

Returns: The maximum early data size in this connection.

Since: 3.6.5

gnutls_record_get_max_size

`size_t gnutls_record_get_max_size` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

Get the record size. The maximum record size is negotiated by the client after the first handshake message.

Returns: The maximum record packet size in this connection.

gnutls_record_get_state

`int gnutls_record_get_state (`*gnutls_session_t* `session`, *unsigned* `read`, *gnutls_datum_t ** `mac_key`, *gnutls_datum_t ** `IV`, *gnutls_datum_t ** `cipher_key`, *unsigned char* [*8*] `seq_number`)` [Function]

session: is a `gnutls_session_t` type

read: if non-zero the read parameters are returned, otherwise the write

mac_key: the key used for MAC (if a MAC is used)

IV: the initialization vector or nonce used

cipher_key: the cipher key

seq_number: A 64-bit sequence number

This function will return the parameters of the current record state. These are only useful to be provided to an external off-loading device or subsystem. The returned values should be considered constant and valid for the lifetime of the session.

In that case, to sync the state back you must call `gnutls_record_set_state()` .

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since 3.4.0

gnutls_record_overhead_size

`size_t gnutls_record_overhead_size (`*gnutls_session_t* `session`)` [Function]

session: is `gnutls_session_t`

This function will return the size in bytes of the overhead due to TLS (or DTLS) per record. On certain occasions (e.g., CBC ciphers) the returned value is the maximum possible overhead.

Since: 3.2.2

gnutls_record_recv

`ssize_t gnutls_record_recv (`*gnutls_session_t* `session`, *void ** `data`, *size_t* `data_size`)` [Function]

session: is a `gnutls_session_t` type.

data: the buffer that the data will be read into

data_size: the number of requested bytes

This function has the similar semantics with `recv()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. In the special case that the peer requests a renegotiation, the caller will receive an error code of `GNUTLS_E_REHANDSHAKE` . In case of a client, this message may be simply ignored, replied with an alert `GNUTLS_A_NO_RENEGOTIATION` , or replied with a new handshake, depending on the client's will. A server receiving this error code can only initiate a new handshake or terminate the session.

If `EINTR` is returned by the internal pull function (the default is `recv()`) then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again to get the data. See also `gnutls_record_get_direction()` .

Returns: The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error. The number of bytes received might be less than the requested `data_size` .

gnutls_record_recv_early_data

`ssize_t gnutls_record_recv_early_data (`*gnutls_session_t* `session`, *void ** `data`, *size_t* `data_size``)` [Function]

session: is a `gnutls_session_t` type.

data: the buffer that the data will be read into

data_size: the number of requested bytes

This function can be used by a searver to retrieve data sent early in the handshake processes when resuming a session. This is used to implement a zero-roundtrip (0-RTT) mode. It has the same semantics as `gnutls_record_recv()` .

This function can be called either in a handshake hook, or after the handshake is complete.

Returns: The number of bytes received and zero when early data reading is complete. A negative error code is returned in case of an error. If no early data is received during the handshake, this function returns `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` . The number of bytes received might be less than the requested `data_size` .

Since: 3.6.5

gnutls_record_recv_packet

`ssize_t gnutls_record_recv_packet (`*gnutls_session_t* `session`, *gnutls_packet_t ** `packet``)` [Function]

session: is a `gnutls_session_t` type.

packet: the structure that will hold the packet data

This is a lower-level function than `gnutls_record_recv()` and allows to directly receive the whole decrypted packet. That avoids a memory copy, and is intended to be used by applications seeking high performance.

The received packet is accessed using `gnutls_packet_get()` and must be deinitialized using `gnutls_packet_deinit()` . The returned packet will be `NULL` if the return value is zero (EOF).

Returns: The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error.

Since: 3.3.5

gnutls_record_recv_seq

`ssize_t gnutls_record_recv_seq (`*gnutls_session_t* `session`, *void ** `data`, *size_t* `data_size`, *unsigned char ** `seq``)` [Function]

session: is a `gnutls_session_t` type.

data: the buffer that the data will be read into

data_size: the number of requested bytes

seq: is the packet's 64-bit sequence number. Should have space for 8 bytes.

This function is the same as `gnutls_record_recv()` , except that it returns in addition to data, the sequence number of the data. This is useful in DTLS where record packets might be received out-of-order. The returned 8-byte sequence number is an integer in big-endian format and should be treated as a unique message identification.

Returns: The number of bytes received and zero on EOF. A negative error code is returned in case of an error. The number of bytes received might be less than `data_size` .

Since: 3.0

gnutls_record_send

`ssize_t gnutls_record_send (`*gnutls_session_t* `session`, *const void* * `data`, *size_t* `data_size`) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send

data_size: is the length of the data

This function has the similar semantics with `send()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. Note that if the send buffer is full, `send()` will block this function. See the `send()` documentation for more information.

You can replace the default push function which is `send()` , by using `gnutls_transport_set_push_function()` .

If the EINTR is returned by the internal push function then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again with the exact same parameters, or provide a `NULL` pointer for `data` and 0 for `data_size` , in order to write the same data as before. If you wish to discard the previous data instead of retrying, you must call `gnutls_record_discard_queued()` before calling this function with different parameters. Note that the latter works only on special transports (e.g., UDP). cf. `gnutls_record_get_direction()` .

Note that in DTLS this function will return the `GNUTLS_E_LARGE_PACKET` error code if the send data exceed the data MTU value - as returned by `gnutls_dtls_get_data_mtu()` . The errno value EMSGSIZE also maps to `GNUTLS_E_LARGE_PACKET` . Note that since 3.2.13 this function can be called under cork in DTLS mode, and will refuse to send data over the MTU size by returning `GNUTLS_E_LARGE_PACKET` .

Returns: The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

gnutls_record_send2

`ssize_t gnutls_record_send2 (`*gnutls_session_t* `session`, *const void* * `data`, *size_t* `data_size`, *size_t* `pad`, *unsigned* `flags`) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send

data_size: is the length of the data

pad: padding to be added to the record

flags: must be zero

This function is identical to `gnutls_record_send()` except that it takes an extra argument to specify padding to be added the record. To determine the maximum size of padding, use `gnutls_record_get_max_size()` and `gnutls_record_overhead_size()` .

Note that in order for GnuTLS to provide constant time processing of padding and data in TLS1.3, the flag `GNUTLS_SAFE_PADDING_CHECK` must be used in `gnutls_init()` .

Returns: The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

Since: 3.6.3

gnutls_record_send_early_data

`ssize_t gnutls_record_send_early_data` (*gnutls_session_t* `session`, *const void * * `data`, *size_t* `data_size`) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send

data_size: is the length of the data

This function can be used by a client to send data early in the handshake processes when resuming a session. This is used to implement a zero-roundtrip (0-RTT) mode. It has the same semantics as `gnutls_record_send()` .

There may be a limit to the amount of data sent as early data. Use `gnutls_record_get_max_early_data_size()` to check the limit. If the limit exceeds, this function returns `GNUTLS_E_RECORD_LIMIT_REACHED` .

Returns: The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

Since: 3.6.5

gnutls_record_send_range

`ssize_t gnutls_record_send_range` (*gnutls_session_t* `session`, *const void * * `data`, *size_t* `data_size`, *const gnutls_range_st * * `range`) [Function]

session: is a `gnutls_session_t` type.

data: contains the data to send.

data_size: is the length of the data.

range: is the range of lengths in which the real data length must be hidden.

This function operates like `gnutls_record_send()` but, while `gnutls_record_send()` adds minimal padding to each TLS record, this function uses the

TLS extra-padding feature to conceal the real data size within the range of lengths provided. Some TLS sessions do not support extra padding (e.g. stream ciphers in standard TLS or SSL3 sessions). To know whether the current session supports extra padding, and hence length hiding, use the `gnutls_record_can_use_length_hiding()` function.

Note: This function currently is limited to blocking sockets.

Returns: The number of bytes sent (that is data_size in a successful invocation), or a negative error code.

gnutls_record_set_max_early_data_size

`int gnutls_record_set_max_early_data_size (`*gnutls_session_t* ***session***, *size_t* ***size***`)` [Function]

session: is a `gnutls_session_t` type.

size: is the new size

This function sets the maximum early data size in this connection. This property can only be set to servers. The client may be provided with the maximum allowed size through the "early_data" extension of the NewSessionTicket handshake message.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.6.4

gnutls_record_set_max_recv_size

`ssize_t gnutls_record_set_max_recv_size (`*gnutls_session_t* ***session***, *size_t* ***size***`)` [Function]

session: is a `gnutls_session_t` type.

size: is the new size

This function sets the maximum amount of plaintext received in a record in this connection.

The limit is also negotiated through a TLS extension called 'record size limit'. Note that while the 'record size limit' extension is preferred, not all TLS implementations use or even understand the extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.6.8

gnutls_record_set_max_size

`ssize_t gnutls_record_set_max_size (`*gnutls_session_t* ***session***, *size_t* ***size***`)` [Function]

session: is a `gnutls_session_t` type.

size: is the new size

This function sets the maximum amount of plaintext sent and received in a record in this connection.

Prior to 3.6.4, this function was implemented using a TLS extension called 'max fragment length', which limits the acceptable values to 512(=2^9), 1024(=2^10), 2048(=2^11) and 4096(=2^12).

Since 3.6.4, the limit is also negotiated through a new TLS extension called 'record size limit', which doesn't have the limitation, as long as the value ranges between 512 and 16384. Note that while the 'record size limit' extension is preferred, not all TLS implementations use or even understand the extension.

Deprecated: if the client can assume that the 'record size limit' extension is supported by the server, we recommend using `gnutls_record_set_max_recv_size()` instead.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_record_set_state

`int gnutls_record_set_state` (*gnutls_session_t* **`session`**, *unsigned* **`read`**, *const unsigned char* [*8*] **`seq_number`**) [Function]

session: is a `gnutls_session_t` type

read: if non-zero the read parameters are returned, otherwise the write

seq_number: A 64-bit sequence number

This function will set the sequence number in the current record state. This function is useful if sending and receiving are offloaded from gnutls. That is, if `gnutls_record_get_state()` was used.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since 3.4.0

gnutls_record_set_timeout

`void gnutls_record_set_timeout` (*gnutls_session_t* **`session`**, *unsigned int* **`ms`**) [Function]

session: is a `gnutls_session_t` type.

ms: is a timeout value in milliseconds

This function sets the receive timeout for the record layer to the provided value. Use an `ms` value of zero to disable timeout (the default), or `GNUTLS_INDEFINITE_TIMEOUT`, to set an indefinite timeout.

This function requires to set a pull timeout callback. See `gnutls_transport_set_pull_timeout_function()`.

Since: 3.1.7

gnutls_record_uncork

`int gnutls_record_uncork` (*gnutls_session_t* **`session`**, *unsigned int* **`flags`**) [Function]

session: is a `gnutls_session_t` type.

flags: Could be zero or `GNUTLS_RECORD_WAIT`

This resets the effect of `gnutls_record_cork()` , and flushes any pending data. If the `GNUTLS_RECORD_WAIT` flag is specified then this function will block until the data is sent or a fatal error occurs (i.e., the function will retry on `GNUTLS_E_AGAIN` and `GNUTLS_E_INTERRUPTED`).

If the flag `GNUTLS_RECORD_WAIT` is not specified and the function is interrupted then the `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` errors will be returned. To obtain the data left in the corked buffer use `gnutls_record_check_corked()` .

Returns: On success the number of transmitted data is returned, or otherwise a negative error code.

Since: 3.1.9

gnutls_rehandshake

`int gnutls_rehandshake (`*gnutls_session_t* ***session***`)` [Function]

session: is a `gnutls_session_t` type.

This function can only be called in server side, and instructs a TLS 1.2 or earlier client to renegotiate parameters (perform a handshake), by sending a hello request message.

If this function succeeds, the calling application should call `gnutls_record_recv()` until `GNUTLS_E_REHANDSHAKE` is returned to clear any pending data. If the `GNUTLS_E_REHANDSHAKE` error code is not seen, then the handshake request was not followed by the peer (the TLS protocol does not require the client to do, and such compliance should be handled by the application protocol).

Once the `GNUTLS_E_REHANDSHAKE` error code is seen, the calling application should proceed to calling `gnutls_handshake()` to negotiate the new parameters.

If the client does not wish to renegotiate parameters he may reply with an alert message, and in that case the return code seen by subsequent `gnutls_record_recv()` will be `GNUTLS_E_WARNING_ALERT_RECEIVED` with the specific alert being `GNUTLS_A_NO_RENEGOTIATION` . A client may also choose to ignore this request.

Under TLS 1.3 this function is equivalent to `gnutls_session_key_update()` with the `GNUTLS_KU_PEER` flag. In that case subsequent calls to `gnutls_record_recv()` will not return `GNUTLS_E_REHANDSHAKE` , and calls to `gnutls_handshake()` in server side are a no-op.

This function always fails with `GNUTLS_E_INVALID_REQUEST` when called in client side.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

gnutls_safe_renegotiation_status

`unsigned gnutls_safe_renegotiation_status (`*gnutls_session_t* ***session***`)` [Function]

session: is a `gnutls_session_t` type.

Can be used to check whether safe renegotiation is being used in the current session.

Returns: 0 when safe renegotiation is not used and non (0) when safe renegotiation is used.

Since: 2.10.0

gnutls_sec_param_get_name

`const char * gnutls_sec_param_get_name` (*gnutls_sec_param_t* `param`) [Function]

param: is a security parameter

Convert a `gnutls_sec_param_t` value to a string.

Returns: a pointer to a string that contains the name of the specified security level, or `NULL` .

Since: 2.12.0

gnutls_sec_param_to_pk_bits

`unsigned int gnutls_sec_param_to_pk_bits` (*gnutls_pk_algorithm_t* `algo`, *gnutls_sec_param_t* `param`) [Function]

algo: is a public key algorithm

param: is a security parameter

When generating private and public key pairs a difficult question is which size of "bits" the modulus will be in RSA and the group size in DSA. The easy answer is 1024, which is also wrong. This function will convert a human understandable security parameter to an appropriate size for the specific algorithm.

Returns: The number of bits, or (0).

Since: 2.12.0

gnutls_sec_param_to_symmetric_bits

`unsigned int gnutls_sec_param_to_symmetric_bits` (*gnutls_sec_param_t* `param`) [Function]

param: is a security parameter

This function will return the number of bits that correspond to symmetric cipher strength for the given security parameter.

Returns: The number of bits, or (0).

Since: 3.3.0

gnutls_server_name_get

`int gnutls_server_name_get` (*gnutls_session_t* `session`, *void *` `data`, *size_t *` `data_length`, *unsigned int *` `type`, *unsigned int* `indx`) [Function]

session: is a `gnutls_session_t` type.

data: will hold the data

data_length: will hold the data length. Must hold the maximum size of data.

type: will hold the server name indicator type

indx: is the index of the server_name

This function will allow you to get the name indication (if any), a client has sent. The name indication may be any of the enumeration gnutls_server_name_type_t.

If `type` is GNUTLS_NAME_DNS, then this function is to be used by servers that support virtual hosting, and the data will be a null terminated IDNA ACE string (prior to GnuTLS 3.4.0 it was a UTF-8 string).

If `data` has not enough size to hold the server name GNUTLS_E_SHORT_MEMORY_BUFFER is returned, and `data_length` will hold the required size.

`indx` is used to retrieve more than one server names (if sent by the client). The first server name has an index of 0, the second 1 and so on. If no name with the given index exists GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, on UTF-8 decoding error `GNUTLS_E_IDNA_ERROR` is returned, otherwise a negative error code is returned.

gnutls_server_name_set

`int gnutls_server_name_set` (*gnutls_session_t* `session`, *gnutls_server_name_type_t* `type`, *const void ** `name`, *size_t* `name_length`) [Function]

session: is a `gnutls_session_t` type.

type: specifies the indicator type

name: is a string that contains the server name.

name_length: holds the length of name excluding the terminating null byte

This function is to be used by clients that want to inform (via a TLS extension mechanism) the server of the name they connected to. This should be used by clients that connect to servers that do virtual hosting.

The value of `name` depends on the `type` type. In case of `GNUTLS_NAME_DNS` , a UTF-8 null-terminated domain name string, without the trailing dot, is expected.

IPv4 or IPv6 addresses are not permitted to be set by this function. If the function is called with a name of `name_length` zero it will clear all server names set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_session_channel_binding

`int gnutls_session_channel_binding` (*gnutls_session_t* `session`, *gnutls_channel_binding_t* `cbtype`, *gnutls_datum_t ** `cb`) [Function]

session: is a `gnutls_session_t` type.

cbtype: an `gnutls_channel_binding_t` enumeration type

cb: output buffer array with data

Extract given channel binding data of the `cbtype` (e.g., `GNUTLS_CB_TLS_UNIQUE`) type.

Returns: `GNUTLS_E_SUCCESS` on success, `GNUTLS_E_UNIMPLEMENTED_FEATURE` if the `cbtype` is unsupported, `GNUTLS_E_CHANNEL_BINDING_NOT_AVAILABLE` if the data is not currently available, or an error code.

Since: 2.12.0

gnutls_session_enable_compatibility_mode

`void gnutls_session_enable_compatibility_mode` [Function]
(*gnutls_session_t* **`session`**)

session: is a `gnutls_session_t` type.

This function can be used to disable certain (security) features in TLS in order to maintain maximum compatibility with buggy clients. Because several trade-offs with security are enabled, if required they will be reported through the audit subsystem.

Normally only servers that require maximum compatibility with everything out there, need to call this function.

Note that this function must be called after any call to gnutls_priority functions.

Since: 2.1.4

gnutls_session_etm_status

`unsigned gnutls_session_etm_status` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` type.

Get the status of the encrypt-then-mac extension negotiation. This is in accordance to rfc7366

Returns: Non-zero if the negotiation was successful or zero otherwise.

gnutls_session_ext_master_secret_status

`unsigned gnutls_session_ext_master_secret_status` [Function]
(*gnutls_session_t* **`session`**)

session: is a `gnutls_session_t` type.

Get the status of the extended master secret extension negotiation. This is in accordance to RFC7627. That information is also available to the more generic `gnutls_session_get_flags()` .

Returns: Non-zero if the negotiation was successful or zero otherwise.

gnutls_session_ext_register

`int gnutls_session_ext_register` (*gnutls_session_t* **`session`**, [Function]
*const char ** **`name`**, *int* **`id`**, *gnutls_ext_parse_type_t* **`parse_point`**, *gnutls_ext_recv_func* **`recv_func`**, *gnutls_ext_send_func* **`send_func`**, *gnutls_ext_deinit_data_func* **`deinit_func`**, *gnutls_ext_pack_func* **`pack_func`**, *gnutls_ext_unpack_func* **`unpack_func`**, *unsigned* **`flags`**)

session: the session for which this extension will be set

name: the name of the extension to register

id: the numeric id of the extension

parse_point: the parse type of the extension (see gnutls_ext_parse_type_t)

recv_func: a function to receive the data

send_func: a function to send the data

deinit_func: a function deinitialize any private data

pack_func: a function which serializes the extension's private data (used on session packing for resumption)

unpack_func: a function which will deserialize the extension's private data

flags: must be zero or flags from `gnutls_ext_flags_t`

This function will register a new extension type. The extension will be only usable within the registered session. If the extension type is already registered then `GNUTLS_E_ALREADY_REGISTERED` will be returned, unless the flag `GNUTLS_EXT_FLAG_OVERRIDE_INTERNAL` is specified. The latter flag when specified can be used to override certain extensions introduced after 3.6.0. It is expected to be used by applications which handle custom extensions that are not currently supported in GnuTLS, but direct support for them may be added in the future.

Each registered extension can store temporary data into the gnutls_session_t structure using `gnutls_ext_set_data()` , and they can be retrieved using `gnutls_ext_get_data()` .

The validity of the extension registered can be given by the appropriate flags of `gnutls_ext_flags_t` . If no validity is given, then the registered extension will be valid for client and TLS1.2 server hello (or encrypted extensions for TLS1.3).

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.5.5

gnutls_session_force_valid

`void gnutls_session_force_valid (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Clears the invalid flag in a session. That means that sessions were corrupt or invalid data were received can be re-used. Use only when debugging or experimenting with the TLS protocol. Should not be used in typical applications.

gnutls_session_get_data

`int gnutls_session_get_data (`*gnutls_session_t* `session`, *void ** `session_data`, *size_t ** `session_data_size`) [Function]

session: is a `gnutls_session_t` type.

session_data: is a pointer to space to hold the session.

session_data_size: is the session_data's size, or it will be set by the function.

Returns all session parameters needed to be stored to support resumption, in a preallocated buffer.

See `gnutls_session_get_data2()` for more information.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_session_get_data2

`int gnutls_session_get_data2 (`*gnutls_session_t* `session`, *gnutls_datum_t ** `data`) [Function]

session: is a `gnutls_session_t` type.

data: is a pointer to a datum that will hold the session.

Returns necessary parameters to support resumption. The client should call this function and store the returned session data. A session can be resumed later by calling `gnutls_session_set_data()` with the returned data. Note that under TLS 1.3, it is recommended for clients to use session parameters only once, to prevent passive-observers from correlating the different connections.

The returned `data` are allocated and must be released using `gnutls_free()` .

This function will fail if called prior to handshake completion. In case of false start TLS, the handshake completes only after data have been successfully received from the peer.

Under TLS1.3 session resumption is possible only after a session ticket is received by the client. To ensure that such a ticket has been received use `gnutls_session_get_flags()` and check for flag `GNUTLS_SFLAGS_SESSION_TICKET` ; if this flag is not set, this function will wait for a new ticket within an estimated roundtrip, and if not received will return dummy data which cannot lead to resumption.

To get notified when new tickets are received by the server use `gnutls_handshake_set_hook_function()` to wait for `GNUTLS_HANDSHAKE_NEW_SESSION_TICKET` messages. Each call of `gnutls_session_get_data2()` after a ticket is received, will return session resumption data corresponding to the last received ticket.

Note that this function under TLS1.3 requires a callback to be set with `gnutls_transport_set_pull_timeout_function()` for successful operation. There was a bug before 3.6.10 which could make this function fail if that callback was not set. On later versions if not set, the function will return a successful error code, but will return dummy data that cannot lead to a resumption.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_session_get_desc

`char * gnutls_session_get_desc` (*gnutls_session_t* ***session***) [Function]

session: is a gnutls session

This function returns a string describing the current session. The string is null terminated and allocated using `gnutls_malloc()` .

If initial negotiation is not complete when this function is called, `NULL` will be returned.

Returns: a description of the protocols and algorithms in the current session.

Since: 3.1.10

gnutls_session_get_flags

`unsigned gnutls_session_get_flags` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

This function will return a series (ORed) of flags, applicable for the current session.

This replaces individual informational functions such as `gnutls_safe_renegotiation_status()` , `gnutls_session_ext_master_secret_status()` , etc.

Returns: An ORed sequence of flags (see `gnutls_session_flags_t`)

Since: 3.5.0

gnutls_session_get_id

`int gnutls_session_get_id (`*gnutls_session_t* ***session***, *void ** ***session_id***, *size_t ** ***session_id_size***`)` [Function]

session: is a `gnutls_session_t` type.

session_id: is a pointer to space to hold the session id.

session_id_size: initially should contain the maximum `session_id` size and will be updated.

Returns the TLS session identifier. The session ID is selected by the server, and in older versions of TLS was a unique identifier shared between client and server which was persistent across resumption. In the latest version of TLS (1.3) or TLS with session tickets, the notion of session identifiers is undefined and cannot be relied for uniquely identifying sessions across client and server.

In client side this function returns the identifier returned by the server, and cannot be assumed to have any relation to session resumption. In server side this function is guaranteed to return a persistent identifier of the session since GnuTLS 3.6.4, which may not necessarily map into the TLS session ID value. Prior to that version the value could only be considered a persistent identifier, under TLS1.2 or earlier and when no session tickets were in use.

The session identifier value returned is always less than `GNUTLS_MAX_SESSION_ID_SIZE` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_session_get_id2

`int gnutls_session_get_id2 (`*gnutls_session_t* ***session***, *gnutls_datum_t ** ***session_id***`)` [Function]

session: is a `gnutls_session_t` type.

session_id: will point to the session ID.

Returns the TLS session identifier. The session ID is selected by the server, and in older versions of TLS was a unique identifier shared between client and server which was persistent across resumption. In the latest version of TLS (1.3) or TLS 1.2 with session tickets, the notion of session identifiers is undefined and cannot be relied for uniquely identifying sessions across client and server.

In client side this function returns the identifier returned by the server, and cannot be assumed to have any relation to session resumption. In server side this function is guaranteed to return a persistent identifier of the session since GnuTLS 3.6.4, which may not necessarily map into the TLS session ID value. Prior to that version the value could only be considered a persistent identifier, under TLS1.2 or earlier and when no session tickets were in use.

The session identifier value returned is always less than `GNUTLS_MAX_SESSION_ID_SIZE` and should be treated as constant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.1.4

gnutls_session_get_keylog_function

`gnutls_keylog_func gnutls_session_get_keylog_function` [Function]
(*const gnutls_session_t* `session`)

session: is `gnutls_session_t` type

This function will return the callback function set using `gnutls_session_set_keylog_function()` .

Returns: The function set or `NULL` otherwise.

Since: 3.6.13

gnutls_session_get_master_secret

`void gnutls_session_get_master_secret` (*gnutls_session_t* `session`, *gnutls_datum_t* * `secret`) [Function]

session: is a `gnutls_session_t` type.

secret: the session's master secret

This function returns pointers to the master secret used in the TLS session. The pointers are not to be modified or deallocated.

This function is only applicable under TLS 1.2 or earlier versions.

Since: 3.5.0

gnutls_session_get_ptr

`void * gnutls_session_get_ptr` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Get user pointer for session. Useful in callbacks. This is the pointer set with `gnutls_session_set_ptr()` .

Returns: the user given pointer from the session structure, or `NULL` if it was never set.

gnutls_session_get_random

`void gnutls_session_get_random` (*gnutls_session_t* `session`, *gnutls_datum_t* * `client`, *gnutls_datum_t* * `server`) [Function]

session: is a `gnutls_session_t` type.

client: the client part of the random

server: the server part of the random

This function returns pointers to the client and server random fields used in the TLS handshake. The pointers are not to be modified or deallocated.

If a client random value has not yet been established, the output will be garbage.

Since: 3.0

gnutls_session_get_verify_cert_status

`unsigned int gnutls_session_get_verify_cert_status` [Function]
(*gnutls_session_t* ***session***)

session: is a gnutls session

This function returns the status of the verification when initiated via auto-verification, i.e., by `gnutls_session_set_verify_cert2()` or `gnutls_session_set_verify_cert()` . If no certificate verification was occurred then the return value would be set to ((unsigned int)-1).

The certificate verification status is the same as in `gnutls_certificate_verify_peers()` .

Returns: the certificate verification status.

Since: 3.4.6

gnutls_session_is_resumed

`int gnutls_session_is_resumed` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

Checks whether session is resumed or not. This is functional for both server and client side.

Returns: non zero if this session is resumed, or a zero if this is a new session.

gnutls_session_key_update

`int gnutls_session_key_update` (*gnutls_session_t* ***session***, [Function]
unsigned ***flags***)

session: is a `gnutls_session_t` type.

flags: zero of `GNUTLS_KU_PEER`

This function will update/refresh the session keys when the TLS protocol is 1.3 or better. The peer is notified of the update by sending a message, so this function should be treated similarly to `gnutls_record_send()` –i.e., it may return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` .

When this flag `GNUTLS_KU_PEER` is specified, this function in addition to updating the local keys, will ask the peer to refresh its keys too.

If the negotiated version is not TLS 1.3 or better this function will return `GNUTLS_E_INVALID_REQUEST` .

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.3

gnutls_session_resumption_requested

`int gnutls_session_resumption_requested` (*gnutls_session_t* ***session***) [Function]

session: is a `gnutls_session_t` type.

Check whether the client has asked for session resumption. This function is valid only on server side.

Returns: non zero if session resumption was asked, or a zero if not.

gnutls_session_set_data

`int gnutls_session_set_data` (*gnutls_session_t* `session`, *const void * * `session_data`, *size_t* `session_data_size`) [Function]

session: is a `gnutls_session_t` type.

session_data: is a pointer to space to hold the session.

session_data_size: is the session's size

Sets all session parameters, in order to resume a previously established session. The session data given must be the one returned by `gnutls_session_get_data()` . This function should be called before `gnutls_handshake()` .

Keep in mind that session resuming is advisory. The server may choose not to resume the session, thus a full handshake will be performed.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_session_set_id

`int gnutls_session_set_id` (*gnutls_session_t* `session`, *const gnutls_datum_t * `sid`) [Function]

session: is a `gnutls_session_t` type.

sid: the session identifier

This function sets the session ID to be used in a client hello. This is a function intended for exceptional uses. Do not use this function unless you are implementing a custom protocol.

To set session resumption parameters use `gnutls_session_set_data()` instead.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.2.1

gnutls_session_set_keylog_function

`void gnutls_session_set_keylog_function` (*gnutls_session_t* `session`, *gnutls_keylog_func* `func`) [Function]

session: is `gnutls_session_t` type

func: is the function to be called

This function will set a callback to be called when a new secret is derived and installed during handshake.

Since: 3.6.13

gnutls_session_set_premaster

`int gnutls_session_set_premaster` (*gnutls_session_t* `session`, *unsigned int* `entity`, *gnutls_protocol_t* `version`, *gnutls_kx_algorithm_t* `kx`, *gnutls_cipher_algorithm_t* `cipher`, *gnutls_mac_algorithm_t* `mac`, *gnutls_compression_method_t* `comp`, *const gnutls_datum_t * `master`, *const gnutls_datum_t * `session_id`) [Function]

session: is a `gnutls_session_t` type.

entity: GNUTLS_SERVER or GNUTLS_CLIENT

version: the TLS protocol version

kx: the key exchange method

cipher: the cipher

mac: the MAC algorithm

comp: the compression method (ignored)

master: the master key to use

session_id: the session identifier

This function sets the premaster secret in a session. This is a function intended for exceptional uses. Do not use this function unless you are implementing a legacy protocol. Use `gnutls_session_set_data()` instead.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

gnutls_session_set_ptr

`void gnutls_session_set_ptr (`*gnutls_session_t* ***session***, *void ** ***ptr***`)` [Function]

session: is a `gnutls_session_t` type.

ptr: is the user pointer

This function will set (associate) the user given pointer `ptr` to the session structure. This pointer can be accessed with `gnutls_session_get_ptr()` .

gnutls_session_set_verify_cert

`void gnutls_session_set_verify_cert (`*gnutls_session_t* ***session***, *const char ** ***hostname***, *unsigned* ***flags***`)` [Function]

session: is a gnutls session

hostname: is the expected name of the peer; may be `NULL`

flags: flags for certificate verification – `gnutls_certificate_verify_flags`

This function instructs GnuTLS to verify the peer's certificate using the provided hostname. If the verification fails the handshake will also fail with `GNUTLS_E_CERTIFICATE_VERIFICATION_ERROR` . In that case the verification result can be obtained using `gnutls_session_get_verify_cert_status()` .

The `hostname` pointer provided must remain valid for the lifetime of the session. More precisely it should be available during any subsequent handshakes. If no hostname is provided, no hostname verification will be performed. For a more advanced verification function check `gnutls_session_set_verify_cert2()` .

If `flags` is provided which contain a profile, this function should be called after any session priority setting functions.

The `gnutls_session_set_verify_cert()` function is intended to be used by TLS clients to verify the server's certificate.

Since: 3.4.6

gnutls_session_set_verify_cert2

`void gnutls_session_set_verify_cert2 (`*gnutls_session_t* `session`, *gnutls_typed_vdata_st* * `data`, *unsigned* `elements`, *unsigned* `flags`) [Function]

session: is a gnutls session

data: an array of typed data

elements: the number of data elements

flags: flags for certificate verification – `gnutls_certificate_verify_flags`

This function instructs GnuTLS to verify the peer's certificate using the provided typed data information. If the verification fails the handshake will also fail with `GNUTLS_E_CERTIFICATE_VERIFICATION_ERROR` . In that case the verification result can be obtained using `gnutls_session_get_verify_cert_status()` .

The acceptable typed data are the same as in `gnutls_certificate_verify_peers()` , and once set must remain valid for the lifetime of the session. More precisely they should be available during any subsequent handshakes.

If `flags` is provided which contain a profile, this function should be called after any session priority setting functions.

Since: 3.4.6

gnutls_session_set_verify_function

`void gnutls_session_set_verify_function (`*gnutls_session_t* `session`, *gnutls_certificate_verify_function* * `func`) [Function]

session: is a `gnutls_session_t` type.

func: is the callback function

This function sets a callback to be called when peer's certificate has been received in order to verify it on receipt rather than doing after the handshake is completed. This overrides any callback set using `gnutls_certificate_set_verify_function()` .

The callback's function prototype is: int (*callback)(gnutls_session_t);

If the callback function is provided then gnutls will call it, in the handshake, just after the certificate message has been received. To verify or obtain the certificate the `gnutls_certificate_verify_peers2()` , `gnutls_certificate_type_get()` , `gnutls_certificate_get_peers()` functions can be used.

The callback function should return 0 for the handshake to continue or non-zero to terminate.

Since: 3.4.6

gnutls_session_supplemental_register

`int gnutls_session_supplemental_register (`*gnutls_session_t* `session`, *const char* * `name`, *gnutls_supplemental_data_format_type_t* `type`, *gnutls_supp_recv_func* `recv_func`, *gnutls_supp_send_func* `send_func`, *unsigned* `flags`) [Function]

session: the session for which this will be registered

name: the name of the supplemental data to register

type: the type of the supplemental data format

recv_func: the function to receive the data

send_func: the function to send the data

flags: must be zero

This function will register a new supplemental data type (rfc4680). The registered supplemental functions will be used for that specific session. The provided `type` must be an unassigned type in `gnutls_supplemental_data_format_type_t` .

If the type is already registered or handled by GnuTLS internally `GNUTLS_E_ALREADY_REGISTERED` will be returned.

As supplemental data are not defined under TLS 1.3, this function will disable TLS 1.3 support for the given session.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.5.5

gnutls_session_ticket_enable_client

`int gnutls_session_ticket_enable_client (`*gnutls_session_t session*`)` [Function]

session: is a `gnutls_session_t` type.

Request that the client should attempt session resumption using SessionTicket. This call is typically unnecessary as session tickets are enabled by default.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 2.10.0

gnutls_session_ticket_enable_server

`int gnutls_session_ticket_enable_server (`*gnutls_session_t session, const gnutls_datum_t * key*`)` [Function]

session: is a `gnutls_session_t` type.

key: key to encrypt session parameters.

Request that the server should attempt session resumption using session tickets, i.e., by delegating storage to the client. `key` must be initialized using `gnutls_session_ticket_key_generate()` . To avoid leaking that key, use `gnutls_memset()` prior to releasing it.

The default ticket expiration time can be overridden using `gnutls_db_set_cache_expiration()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 2.10.0

gnutls_session_ticket_key_generate

`int gnutls_session_ticket_key_generate (gnutls_datum_t * key)` [Function]

key: is a pointer to a `gnutls_datum_t` which will contain a newly created key.

Generate a random key to encrypt security parameters within SessionTicket.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

Since: 2.10.0

gnutls_session_ticket_send

`int gnutls_session_ticket_send (gnutls_session_t session, unsigned nr, unsigned flags)` [Function]

session: is a `gnutls_session_t` type.

nr: the number of tickets to send

flags: must be zero

Sends a fresh session ticket to the peer. This is relevant only in server side under TLS1.3. This function may also return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` and in that case it must be called again.

Returns: `GNUTLS_E_SUCCESS` on success, or a negative error code.

gnutls_set_default_priority

`int gnutls_set_default_priority (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` type.

Sets the default priority on the ciphers, key exchange methods, and macs. This is the recommended method of setting the defaults, in order to promote consistency between applications using GnuTLS, and to allow GnuTLS using applications to update settings in par with the library. For client applications which require maximum compatibility consider calling `gnutls_session_enable_compatibility_mode()` after this function.

For an application to specify additional options to priority string consider using `gnutls_set_default_priority_append()` .

To allow a user to override the defaults (e.g., when a user interface or configuration file is available), the functions `gnutls_priority_set_direct()` or `gnutls_priority_set()` can be used.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 2.1.4

gnutls_set_default_priority_append

`int gnutls_set_default_priority_append (gnutls_session_t session, const char * add_prio, const char ** err_pos, unsigned flags)` [Function]

session: is a `gnutls_session_t` type.

add_prio: is a string describing priorities to be appended to default

err_pos: In case of an error this will have the position in the string the error occurred

flags: must be zero

Sets the default priority on the ciphers, key exchange methods, and macs with the additional options in `add_prio` . This is the recommended method of setting the defaults when only few additional options are to be added. This promotes consistency between applications using GnuTLS, and allows GnuTLS using applications to update settings in par with the library.

The `add_prio` string should start as a normal priority string, e.g., '-VERS-TLS-ALL:+VERS-TLS1.3:%COMPAT' or '%FORCE_ETM'. That is, it must not start with ':'.

To allow a user to override the defaults (e.g., when a user interface or configuration file is available), the functions `gnutls_priority_set_direct()` or `gnutls_priority_set()` can be used.

Returns: `GNUTLS_E_SUCCESS` on success, or an error code.

Since: 3.6.3

gnutls_sign_algorithm_get

`int gnutls_sign_algorithm_get (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Returns the signature algorithm that is (or will be) used in this session by the server to sign data. This function should be used only with TLS 1.2 or later.

Returns: The sign algorithm or `GNUTLS_SIGN_UNKNOWN` .

Since: 3.1.1

gnutls_sign_algorithm_get_client

`int gnutls_sign_algorithm_get_client (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Returns the signature algorithm that is (or will be) used in this session by the client to sign data. This function should be used only with TLS 1.2 or later.

Returns: The sign algorithm or `GNUTLS_SIGN_UNKNOWN` .

Since: 3.1.11

gnutls_sign_algorithm_get_requested

`int gnutls_sign_algorithm_get_requested (`*gnutls_session_t* `session`, *size_t* `indx`, *gnutls_sign_algorithm_t* * `algo`) [Function]

session: is a `gnutls_session_t` type.

indx: is an index of the signature algorithm to return

algo: the returned certificate type will be stored there

Returns the signature algorithm specified by index that was requested by the peer. If the specified index has no data available this function returns `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` . If the negotiated TLS version does not support signature

algorithms then GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned even for the first index. The first index is 0.

This function is useful in the certificate callback functions to assist in selecting the correct certificate.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise an error code is returned.

Since: 2.10.0

gnutls_sign_get_hash_algorithm

gnutls_digest_algorithm_t gnutls_sign_get_hash_algorithm (*gnutls_sign_algorithm_t sign*) [Function]

sign: is a signature algorithm

This function returns the digest algorithm corresponding to the given signature algorithms.

Since: 3.1.1

Returns: return a gnutls_digest_algorithm_t value, or GNUTLS_DIG_UNKNOWN on error.

gnutls_sign_get_id

gnutls_sign_algorithm_t gnutls_sign_get_id (*const char * name*) [Function]

name: is a sign algorithm name

The names are compared in a case insensitive way.

Returns: return a gnutls_sign_algorithm_t value corresponding to the specified algorithm, or GNUTLS_SIGN_UNKNOWN on error.

gnutls_sign_get_name

const char * gnutls_sign_get_name (*gnutls_sign_algorithm_t algorithm*) [Function]

algorithm: is a sign algorithm

Convert a gnutls_sign_algorithm_t value to a string.

Returns: a string that contains the name of the specified sign algorithm, or NULL .

gnutls_sign_get_oid

const char * gnutls_sign_get_oid (*gnutls_sign_algorithm_t sign*) [Function]

sign: is a sign algorithm

Convert a gnutls_sign_algorithm_t value to its object identifier.

Returns: a string that contains the object identifier of the specified sign algorithm, or NULL .

Since: 3.4.3

gnutls_sign_get_pk_algorithm

`gnutls_pk_algorithm_t gnutls_sign_get_pk_algorithm` [Function]
(*gnutls_sign_algorithm_t* `sign`)

sign: is a signature algorithm

This function returns the public key algorithm corresponding to the given signature algorithms. Note that there may be multiple public key algorithms supporting a particular signature type; when dealing with such algorithms use instead `gnutls_sign_supports_pk_algorithm()` .

Since: 3.1.1

Returns: return a `gnutls_pk_algorithm_t` value, or `GNUTLS_PK_UNKNOWN` on error.

gnutls_sign_is_secure

`unsigned gnutls_sign_is_secure` (*gnutls_sign_algorithm_t* `algorithm`) [Function]

algorithm: is a sign algorithm

Returns: Non-zero if the provided signature algorithm is considered to be secure.

gnutls_sign_is_secure2

`unsigned gnutls_sign_is_secure2` (*gnutls_sign_algorithm_t* `algorithm`, *unsigned int* `flags`) [Function]

algorithm: is a sign algorithm

flags: zero or `GNUTLS_SIGN_FLAG_SECURE_FOR_CERTS`

Returns: Non-zero if the provided signature algorithm is considered to be secure.

gnutls_sign_list

`const gnutls_sign_algorithm_t * gnutls_sign_list ( void)` [Function]

Get a list of supported public key signature algorithms. This function is not thread safe.

Returns: a (0)-terminated list of `gnutls_sign_algorithm_t` integers indicating the available ciphers.

gnutls_sign_supports_pk_algorithm

`unsigned gnutls_sign_supports_pk_algorithm` [Function]
(*gnutls_sign_algorithm_t* `sign`, *gnutls_pk_algorithm_t* `pk`)

sign: is a signature algorithm

pk: is a public key algorithm

This function returns non-zero if the public key algorithm corresponds to the given signature algorithm. That is, if that signature can be generated from the given private key algorithm.

Since: 3.6.0

Returns: return non-zero when the provided algorithms are compatible.

gnutls_srp_allocate_client_credentials

`int gnutls_srp_allocate_client_credentials` [Function]
(*gnutls_srp_client_credentials_t * sc*)

sc: is a pointer to a `gnutls_srp_server_credentials_t` type.

Allocate a gnutls_srp_client_credentials_t structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

gnutls_srp_allocate_server_credentials

`int gnutls_srp_allocate_server_credentials` [Function]
(*gnutls_srp_server_credentials_t * sc*)

sc: is a pointer to a `gnutls_srp_server_credentials_t` type.

Allocate a gnutls_srp_server_credentials_t structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

gnutls_srp_base64_decode

`int gnutls_srp_base64_decode` (*const gnutls_datum_t * b64_data*, [Function]
*char * result*, *size_t * result_size*)

b64_data: contain the encoded data

result: the place where decoded data will be copied

result_size: holds the size of the result

This function will decode the given encoded data, using the base64 encoding found in libsrp.

Note that `b64_data` should be null terminated.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, or 0 on success.

gnutls_srp_base64_decode2

`int gnutls_srp_base64_decode2` (*const gnutls_datum_t * b64_data*, *gnutls_datum_t * result*) [Function]

b64_data: contains the encoded data

result: the place where decoded data lie

This function will decode the given encoded data. The decoded data will be allocated, and stored into result. It will decode using the base64 algorithm as used in libsrp.

You should use `gnutls_free()` to free the returned data.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

Returns: 0 on success, or an error code.

gnutls_srp_base64_encode

`int gnutls_srp_base64_encode` (*const gnutls_datum_t * data*, *char * result*, *size_t * result_size*) [Function]

data: contain the raw data

result: the place where base64 data will be copied

result_size: holds the size of the result

This function will convert the given data to printable data, using the base64 encoding, as used in the libsrp. This is the encoding used in SRP password files. If the provided buffer is not long enough GNUTLS_E_SHORT_MEMORY_BUFFER is returned.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, or 0 on success.

gnutls_srp_base64_encode2

`int gnutls_srp_base64_encode2` (*const gnutls_datum_t * data*, *gnutls_datum_t * result*) [Function]

data: contains the raw data

result: will hold the newly allocated encoded data

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in SRP password files. This function will allocate the required memory to hold the encoded data.

You should use `gnutls_free()` to free the returned data.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

Returns: 0 on success, or an error code.

gnutls_srp_free_client_credentials

`void gnutls_srp_free_client_credentials` (*gnutls_srp_client_credentials_t sc*) [Function]

sc: is a `gnutls_srp_client_credentials_t` type.

Free a gnutls_srp_client_credentials_t structure.

gnutls_srp_free_server_credentials

`void gnutls_srp_free_server_credentials` (*gnutls_srp_server_credentials_t sc*) [Function]

sc: is a `gnutls_srp_server_credentials_t` type.

Free a gnutls_srp_server_credentials_t structure.

gnutls_srp_server_get_username

`const char * gnutls_srp_server_get_username (gnutls_session_t session)` [Function]

session: is a gnutls session

This function will return the username of the peer. This should only be called in case of SRP authentication and in case of a server. Returns NULL in case of an error.

Returns: SRP username of the peer, or NULL in case of error.

gnutls_srp_set_client_credentials

`int gnutls_srp_set_client_credentials (gnutls_srp_client_credentials_t res, const char * username, const char * password)` [Function]

res: is a `gnutls_srp_client_credentials_t` type.

username: is the user's userid

password: is the user's password

This function sets the username and password, in a `gnutls_srp_client_credentials_t` type. Those will be used in SRP authentication. `username` should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265). The password can be in ASCII format, or normalized using `gnutls_utf8_password_normalize()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

gnutls_srp_set_client_credentials_function

`void gnutls_srp_set_client_credentials_function (gnutls_srp_client_credentials_t cred, gnutls_srp_client_credentials_function * func)` [Function]

cred: is a `gnutls_srp_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the username and password for client SRP authentication. The callback's function form is:

int (*callback)(gnutls_session_t, char** username, char**password);

The `username` and `password` must be allocated using `gnutls_malloc()` .

The `username` should be an ASCII string or UTF-8 string. In case of a UTF-8 string it is recommended to be following the PRECIS framework for usernames (rfc8265). The password can be in ASCII format, or normalized using `gnutls_utf8_password_normalize()` .

The callback function will be called once per handshake before the initial hello message is sent.

The callback should not return a negative error code the second time called, since the handshake procedure will be aborted.

The callback function should return 0 on success. -1 indicates an error.

gnutls_srp_set_prime_bits

`void gnutls_srp_set_prime_bits (`*gnutls_session_t* `session`, *unsigned int* `bits``)` [Function]

session: is a `gnutls_session_t` type.

bits: is the number of bits

This function sets the minimum accepted number of bits, for use in an SRP key exchange. If zero, the default 2048 bits will be used.

In the client side it sets the minimum accepted number of bits. If a server sends a prime with less bits than that `GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER` will be returned by the handshake.

This function has no effect in server side.

Since: 2.6.0

gnutls_srp_set_server_credentials_file

`int gnutls_srp_set_server_credentials_file` [Function]
`(`*gnutls_srp_server_credentials_t* `res`, *const char ** `password_file`, *const char ** `password_conf_file`)

res: is a `gnutls_srp_server_credentials_t` type.

password_file: is the SRP password file (tpasswd)

password_conf_file: is the SRP password conf file (tpasswd.conf)

This function sets the password files, in a `gnutls_srp_server_credentials_t` type. Those password files hold usernames and verifiers and will be used for SRP authentication.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

gnutls_srp_set_server_credentials_function

`void gnutls_srp_set_server_credentials_function` [Function]
`(`*gnutls_srp_server_credentials_t* `cred`,
*gnutls_srp_server_credentials_function ** `func`)

cred: is a `gnutls_srp_server_credentials_t` type.

func: is the callback function

This function can be used to set a callback to retrieve the user's SRP credentials. The callback's function form is:

int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t *salt, gnutls_datum_t *verifier, gnutls_datum_t *generator, gnutls_datum_t *prime);

`username` contains the actual username. The `salt` , `verifier` , `generator` and `prime` must be filled in using the `gnutls_malloc()` . For convenience `prime` and `generator` may also be one of the static parameters defined in gnutls.h.

Initially, the data field is NULL in every `gnutls_datum_t` structure that the callback has to fill in. When the callback is done GnuTLS deallocates all of those buffers which are non-NULL, regardless of the return value.

In order to prevent attackers from guessing valid usernames, if a user does not exist, g and n values should be filled in using a random user's parameters. In that case the callback must return the special value (1). See `gnutls_srp_set_server_fake_salt_seed` too. If this is not required for your application, return a negative number from the callback to abort the handshake.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

gnutls_srp_set_server_fake_salt_seed

`void gnutls_srp_set_server_fake_salt_seed` [Function]
(*gnutls_srp_server_credentials_t* `cred`, *const gnutls_datum_t ** `seed`, *unsigned int* `salt_length`)

cred: is a `gnutls_srp_server_credentials_t` type

seed: is the seed data, only needs to be valid until the function returns; size of the seed must be greater than zero

salt_length: is the length of the generated fake salts

This function sets the seed that is used to generate salts for invalid (non-existent) usernames.

In order to prevent attackers from guessing valid usernames, when a user does not exist gnutls generates a salt and a verifier and proceeds with the protocol as usual. The authentication will ultimately fail, but the client cannot tell whether the username is valid (exists) or invalid.

If an attacker learns the seed, given a salt (which is part of the handshake) which was generated when the seed was in use, it can tell whether or not the authentication failed because of an unknown username. This seed cannot be used to reveal application data or passwords.

`salt_length` should represent the salt length your application uses. Generating fake salts longer than 20 bytes is not supported.

By default the seed is a random value, different each time a `gnutls_srp_server_credentials_t` is allocated and fake salts are 16 bytes long.

Since: 3.3.0

gnutls_srp_verifier

`int gnutls_srp_verifier` (*const char ** `username`, *const char ** `password`, *const gnutls_datum_t ** `salt`, *const gnutls_datum_t ** `generator`, *const gnutls_datum_t ** `prime`, *gnutls_datum_t ** `res`) [Function]

username: is the user's name

password: is the user's password

salt: should be some randomly generated bytes

generator: is the generator of the group

prime: is the group's prime

res: where the verifier will be stored.

This function will create an SRP verifier, as specified in RFC2945. The `prime` and `generator` should be one of the static parameters defined in gnutls/gnutls.h or may be generated.

The verifier will be allocated with `gnutls_malloc` () and will be stored in `res` using binary format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

gnutls_srtp_get_keys

`int gnutls_srtp_get_keys` (*gnutls_session_t* `session`, *void** `key_material`, *unsigned int* `key_material_size`, *gnutls_datum_t** `client_key`, *gnutls_datum_t** `client_salt`, *gnutls_datum_t** `server_key`, *gnutls_datum_t** `server_salt`) [Function]

session: is a `gnutls_session_t` type.

key_material: Space to hold the generated key material

key_material_size: The maximum size of the key material

client_key: The master client write key, pointing inside the key material

client_salt: The master client write salt, pointing inside the key material

server_key: The master server write key, pointing inside the key material

server_salt: The master server write salt, pointing inside the key material

This is a helper function to generate the keying material for SRTP. It requires the space of the key material to be pre-allocated (should be at least 2x the maximum key size and salt size). The `client_key` , `client_salt` , `server_key` and `server_salt` are convenience datums that point inside the key material. They may be `NULL` .

Returns: On success the size of the key material is returned, otherwise, `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not sufficient, or a negative error code.

Since 3.1.4

gnutls_srtp_get_mki

`int gnutls_srtp_get_mki` (*gnutls_session_t* `session`, *gnutls_datum_t** `mki`) [Function]

session: is a `gnutls_session_t` type.

mki: will hold the MKI

This function exports the negotiated Master Key Identifier, received by the peer if any. The returned value in `mki` should be treated as constant and valid only during the session's lifetime.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

gnutls_srtp_get_profile_id

`int gnutls_srtp_get_profile_id (const char * name, gnutls_srtp_profile_t * profile)` [Function]

name: The name of the profile to look up

profile: Will hold the profile id

This function allows you to look up a profile based on a string.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

gnutls_srtp_get_profile_name

`const char * gnutls_srtp_get_profile_name (gnutls_srtp_profile_t profile)` [Function]

profile: The profile to look up a string for

This function allows you to get the corresponding name for a SRTP protection profile.

Returns: On success, the name of a SRTP profile as a string, otherwise NULL.

Since 3.1.4

gnutls_srtp_get_selected_profile

`int gnutls_srtp_get_selected_profile (gnutls_session_t session, gnutls_srtp_profile_t * profile)` [Function]

session: is a `gnutls_session_t` type.

profile: will hold the profile

This function allows you to get the negotiated SRTP profile.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

gnutls_srtp_set_mki

`int gnutls_srtp_set_mki (gnutls_session_t session, const gnutls_datum_t * mki)` [Function]

session: is a `gnutls_session_t` type.

mki: holds the MKI

This function sets the Master Key Identifier, to be used by this session (if any).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

gnutls_srtp_set_profile

`int gnutls_srtp_set_profile` (*gnutls_session_t* `session`, *gnutls_srtp_profile_t* `profile`) [Function]

session: is a `gnutls_session_t` type.

profile: is the profile id to add.

This function is to be used by both clients and servers, to declare what SRTP profiles they support, to negotiate with the peer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

gnutls_srtp_set_profile_direct

`int gnutls_srtp_set_profile_direct` (*gnutls_session_t* `session`, *const char * profiles*, *const char ** err_pos*) [Function]

session: is a `gnutls_session_t` type.

profiles: is a string that contains the supported SRTP profiles, separated by colons.

err_pos: In case of an error this will have the position in the string the error occurred, may be NULL.

This function is to be used by both clients and servers, to declare what SRTP profiles they support, to negotiate with the peer.

Returns: On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

Since 3.1.4

gnutls_store_commitment

`int gnutls_store_commitment` (*const char * db_name*, *gnutls_tdb_t* `tdb`, *const char * host*, *const char * service*, *gnutls_digest_algorithm_t hash_algo*, *const gnutls_datum_t * hash*, *time_t expiration*, *unsigned int flags*) [Function]

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

hash_algo: The hash algorithm type

hash: The raw hash

expiration: The expiration time (use 0 to disable expiration)

flags: should be 0 or `GNUTLS_SCOMMIT_FLAG_ALLOW_BROKEN` .

This function will store the provided hash commitment to the list of stored public keys. The key with the given hash will be considered valid until the provided expiration time.

The `tdb` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Note that this function is not thread safe with the default backend.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_store_pubkey

`int gnutls_store_pubkey` (*const char * db_name, gnutls_tdb_t tdb, const char * host, const char * service, gnutls_certificate_type_t cert_type, const gnutls_datum_t * cert, time_t expiration, unsigned int flags*) [Function]

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

cert_type: The type of the certificate

cert: The data of the certificate

expiration: The expiration time (use 0 to disable expiration)

flags: should be 0.

This function will store a raw public-key or a public-key provided via a raw (DER-encoded) certificate to the list of stored public keys. The key will be considered valid until the provided expiration time.

The `tdb` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Unless an alternative `tdb` is provided, the storage format is a textual format consisting of a line for each host with fields separated by '`|`'. The contents of the fields are a format-identifier which is set to 'g0', the hostname that the rest of the data applies to, the numeric port or host name, the expiration time in seconds since the epoch (0 for no expiration), and a base64 encoding of the raw (DER) public key information (SPKI) of the peer.

As of GnuTLS 3.6.6 this function also accepts raw public keys.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0.13

gnutls_strerror

`const char * gnutls_strerror` (*int error*) [Function]

error: is a GnuTLS error code, a negative error code

This function is similar to strerror. The difference is that it accepts an error number returned by a gnutls function; In case of an unknown error a descriptive string is sent instead of `NULL` .

Error codes are always a negative error code.

Returns: A string explaining the GnuTLS error message.

gnutls_strerror_name

`const char * gnutls_strerror_name (int error)` [Function]

error: is an error returned by a gnutls function.

Return the GnuTLS error code define as a string. For example, gnutls_strerror_name (GNUTLS_E_DH_PRIME_UNACCEPTABLE) will return the string "GNUTLS_E_DH_PRIME_UNACCEPTABLE".

Returns: A string corresponding to the symbol name of the error code.

Since: 2.6.0

gnutls_supplemental_get_name

`const char * gnutls_supplemental_get_name (gnutls_supplemental_data_format_type_t type)` [Function]

type: is a supplemental data format type

Convert a `gnutls_supplemental_data_format_type_t` value to a string.

Returns: a string that contains the name of the specified supplemental data format type, or `NULL` for unknown types.

gnutls_supplemental_recv

`void gnutls_supplemental_recv (gnutls_session_t session, unsigned do_recv_supplemental)` [Function]

session: is a `gnutls_session_t` type.

do_recv_supplemental: non-zero in order to expect supplemental data

This function is to be called by an extension handler to instruct gnutls to attempt to receive supplemental data during the handshake process.

Since: 3.4.0

gnutls_supplemental_register

`int gnutls_supplemental_register (const char * name, gnutls_supplemental_data_format_type_t type, gnutls_supp_recv_func recv_func, gnutls_supp_send_func send_func)` [Function]

name: the name of the supplemental data to register

type: the type of the supplemental data format

recv_func: the function to receive the data

send_func: the function to send the data

This function will register a new supplemental data type (rfc4680). The registered data will remain until `gnutls_global_deinit()` is called. The provided `type` must be an unassigned type in `gnutls_supplemental_data_format_type_t` . If the type is already registered or handled by GnuTLS internally `GNUTLS_E_ALREADY_REGISTERED` will be returned.

This function is not thread safe. As supplemental data are not defined under TLS 1.3, this function will disable TLS 1.3 support globally.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_supplemental_send

`void gnutls_supplemental_send` (*gnutls_session_t* `session`, *unsigned* `do_send_supplemental`) [Function]

session: is a `gnutls_session_t` type.

do_send_supplemental: non-zero in order to send supplemental data

This function is to be called by an extension handler to instruct gnutls to send supplemental data during the handshake process.

Since: 3.4.0

gnutls_system_recv_timeout

`int gnutls_system_recv_timeout` (*gnutls_transport_ptr_t* `ptr`, *unsigned int* `ms`) [Function]

ptr: A file descriptor (wrapped in a gnutls_transport_ptr_t pointer)

ms: The number of milliseconds to wait.

Wait for data to be received from the provided socket (`ptr`) within a timeout period in milliseconds, using `select()` on the provided `ptr` .

This function is provided as a helper for constructing custom callbacks for `gnutls_transport_set_pull_timeout_function()` , which can be used if you rely on socket file descriptors.

Returns -1 on error, 0 on timeout, positive value if data are available for reading.

Since: 3.4.0

gnutls_tdb_deinit

`void gnutls_tdb_deinit` (*gnutls_tdb_t* `tdb`) [Function]

tdb: The structure to be deinitialized

This function will deinitialize a public key trust storage structure.

gnutls_tdb_init

`int gnutls_tdb_init` (*gnutls_tdb_t ** `tdb`) [Function]

tdb: A pointer to the type to be initialized

This function will initialize a public key trust storage structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_tdb_set_store_commitment_func

`void gnutls_tdb_set_store_commitment_func` (*gnutls_tdb_t* `tdb`, *gnutls_tdb_store_commitment_func* `cstore`) [Function]

tdb: The trust storage

cstore: The commitment storage function

This function will associate a commitment (hash) storage function with the trust storage structure. The function is of the following form.

int gnutls_tdb_store_commitment_func(const char* db_name, const char* host, const char* service, time_t expiration, gnutls_digest_algorithm_t, const gnutls_datum_t* hash);

The `db_name` should be used to pass any private data to this function.

gnutls_tdb_set_store_func

`void gnutls_tdb_set_store_func` (*gnutls_tdb_t* `tdb`, *gnutls_tdb_store_func* `store`) [Function]

tdb: The trust storage

store: The storage function

This function will associate a storage function with the trust storage structure. The function is of the following form.

int gnutls_tdb_store_func(const char* db_name, const char* host, const char* service, time_t expiration, const gnutls_datum_t* pubkey);

The `db_name` should be used to pass any private data to this function.

gnutls_tdb_set_verify_func

`void gnutls_tdb_set_verify_func` (*gnutls_tdb_t* `tdb`, *gnutls_tdb_verify_func* `verify`) [Function]

tdb: The trust storage

verify: The verification function

This function will associate a retrieval function with the trust storage structure. The function is of the following form.

int gnutls_tdb_verify_func(const char* db_name, const char* host, const char* service, const gnutls_datum_t* pubkey);

The verify function should return zero on a match, `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` if there is a mismatch and any other negative error code otherwise.

The `db_name` should be used to pass any private data to this function.

gnutls_transport_get_int

`int gnutls_transport_get_int` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Used to get the first argument of the transport function (like PUSH and PULL). This must have been set using `gnutls_transport_set_int()` .

Returns: The first argument of the transport function.

Since: 3.1.9

gnutls_transport_get_int2

`void gnutls_transport_get_int2 (`*gnutls_session_t* ***session***`, int *` [Function]
recv_int`, int *` ***send_int***`)`

session: is a `gnutls_session_t` type.

recv_int: will hold the value for the pull function

send_int: will hold the value for the push function

Used to get the arguments of the transport functions (like PUSH and PULL). These should have been set using `gnutls_transport_set_int2()` .

Since: 3.1.9

gnutls_transport_get_ptr

`gnutls_transport_ptr_t gnutls_transport_get_ptr` [Function]
`(`*gnutls_session_t* ***session***`)`

session: is a `gnutls_session_t` type.

Used to get the first argument of the transport function (like PUSH and PULL). This must have been set using `gnutls_transport_set_ptr()` .

Returns: The first argument of the transport function.

gnutls_transport_get_ptr2

`void gnutls_transport_get_ptr2 (`*gnutls_session_t* ***session***`,` [Function]
gnutls_transport_ptr_t ***recv_ptr***`, gnutls_transport_ptr_t *` ***send_ptr***`)`

session: is a `gnutls_session_t` type.

recv_ptr: will hold the value for the pull function

send_ptr: will hold the value for the push function

Used to get the arguments of the transport functions (like PUSH and PULL). These should have been set using `gnutls_transport_set_ptr2()` .

gnutls_transport_set_errno

`void gnutls_transport_set_errno (`*gnutls_session_t* ***session***`, int` [Function]
err`)`

session: is a `gnutls_session_t` type.

err: error value to store in session-specific errno variable.

Store `err` in the session-specific errno variable. Useful values for `err` are EINTR, EAGAIN and EMSGSIZE, other values are treated will be treated as real errors in the push/pull function.

This function is useful in replacement push and pull functions set by `gnutls_transport_set_push_function()` and `gnutls_transport_set_pull_function()` under Windows, where the replacements may not have access to the same `errno` variable that is used by GnuTLS (e.g., the application is linked to msvcr71.dll and gnutls is linked to msvcrt.dll).

This function is unreliable if you are using the same `session` in different threads for sending and receiving.

gnutls_transport_set_errno_function

`void gnutls_transport_set_errno_function` (*gnutls_session_t* `session`, *gnutls_errno_func* `errno_func`) [Function]

session: is a `gnutls_session_t` type.

errno_func: a callback function similar to `write()`

This is the function where you set a function to retrieve errno after a failed push or pull operation.

`errno_func` is of the form, int (*gnutls_errno_func)(gnutls_transport_ptr_t); and should return the errno.

Since: 2.12.0

gnutls_transport_set_int

`void gnutls_transport_set_int` (*gnutls_session_t* `session`, *int* `fd`) [Function]

session: is a `gnutls_session_t` type.

fd: is the socket descriptor for the connection.

This function sets the first argument of the transport function, such as `send()` and `recv()` for the default callbacks using the system's socket API.

This function is equivalent to calling `gnutls_transport_set_ptr()` with the descriptor, but requires no casts.

Since: 3.1.9

gnutls_transport_set_int2

`void gnutls_transport_set_int2` (*gnutls_session_t* `session`, *int* `recv_fd`, *int* `send_fd`) [Function]

session: is a `gnutls_session_t` type.

recv_fd: is socket descriptor for the pull function

send_fd: is socket descriptor for the push function

This function sets the first argument of the transport functions, such as `send()` and `recv()` for the default callbacks using the system's socket API. With this function you can set two different descriptors for receiving and sending.

This function is equivalent to calling `gnutls_transport_set_ptr2()` with the descriptors, but requires no casts.

Since: 3.1.9

gnutls_transport_set_ptr

`void gnutls_transport_set_ptr` (*gnutls_session_t* `session`, *gnutls_transport_ptr_t* `ptr`) [Function]

session: is a `gnutls_session_t` type.

ptr: is the value.

Used to set the first argument of the transport function (for push and pull callbacks). In berkeley style sockets this function will set the connection descriptor.

gnutls_transport_set_ptr2

`void gnutls_transport_set_ptr2 (`*gnutls_session_t* **`session`**, *gnutls_transport_ptr_t* **`recv_ptr`**, *gnutls_transport_ptr_t* **`send_ptr`**`)` [Function]

session: is a `gnutls_session_t` type.

recv_ptr: is the value for the pull function

send_ptr: is the value for the push function

Used to set the first argument of the transport function (for push and pull callbacks). In berkeley style sockets this function will set the connection descriptor. With this function you can use two different pointers for receiving and sending.

gnutls_transport_set_pull_function

`void gnutls_transport_set_pull_function (`*gnutls_session_t* **`session`**, *gnutls_pull_func* **`pull_func`**`)` [Function]

session: is a `gnutls_session_t` type.

pull_func: a callback function similar to `read()`

This is the function where you set a function for gnutls to receive data. Normally, if you use berkeley style sockets, do not need to use this function since the default recv(2) will probably be ok. The callback should return 0 on connection termination, a positive number indicating the number of bytes received, and -1 on error.

`gnutls_pull_func` is of the form, ssize_t (*gnutls_pull_func)(gnutls_transport_ptr_t, void*, size_t);

gnutls_transport_set_pull_timeout_function

`void gnutls_transport_set_pull_timeout_function (`*gnutls_session_t* **`session`**, *gnutls_pull_timeout_func* **`func`**`)` [Function]

session: is a `gnutls_session_t` type.

func: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()` , if the timeout value is zero the callback should return zero if no data are immediately available. The special value `GNUTLS_INDEFINITE_TIMEOUT` indicates that the callback should wait indefinitely for data.

`gnutls_pull_timeout_func` is of the form, int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);

This callback is necessary when `gnutls_handshake_set_timeout()` or `gnutls_record_set_timeout()` are set, under TLS1.3 and for enforcing the DTLS mode timeouts when in blocking mode.

For compatibility with future GnuTLS versions this callback must be set when a custom pull function is registered. The callback will not be used when the session is

in TLS mode with non-blocking sockets. That is, when `GNUTLS_NONBLOCK` is specified for a TLS session in `gnutls_init()` .

The helper function `gnutls_system_recv_timeout()` is provided to simplify writing callbacks.

Since: 3.0

gnutls_transport_set_push_function

`void gnutls_transport_set_push_function (`*gnutls_session_t* `session`, *gnutls_push_func* `push_func`) [Function]

session: is a `gnutls_session_t` type.

push_func: a callback function similar to `write()`

This is the function where you set a push function for gnutls to use in order to send data. If you are going to use berkeley style sockets, you do not need to use this function since the default send(2) will probably be ok. Otherwise you should specify this function for gnutls to be able to send data. The callback should return a positive number indicating the bytes sent, and -1 on error.

`push_func` is of the form, ssize_t (*gnutls_push_func)(gnutls_transport_ptr_t, const void*, size_t);

gnutls_transport_set_vec_push_function

`void gnutls_transport_set_vec_push_function (`*gnutls_session_t* `session`, *gnutls_vec_push_func* `vec_func`) [Function]

session: is a `gnutls_session_t` type.

vec_func: a callback function similar to `writev()`

Using this function you can override the default writev(2) function for gnutls to send data. Setting this callback instead of `gnutls_transport_set_push_function()` is recommended since it introduces less overhead in the TLS handshake process.

`vec_func` is of the form, ssize_t (*gnutls_vec_push_func) (gnutls_transport_ptr_t, const giovec_t * iov, int iovcnt);

Since: 2.12.0

gnutls_url_is_supported

`unsigned gnutls_url_is_supported (`*const char * url*`)` [Function]

url: A URI to be tested

Check whether the provided `url` is supported. Depending on the system libraries GnuTLS may support pkcs11, tpmkey or other URLs.

Returns: return non-zero if the given URL is supported, and zero if it is not known.

Since: 3.1.0

gnutls_utf8_password_normalize

`int gnutls_utf8_password_normalize (`*const unsigned char * `password`, unsigned `plen`, gnutls_datum_t * `out`, unsigned `flags`*`)` [Function]

password: contain the UTF-8 formatted password

plen: the length of the provided password

out: the result in an null-terminated allocated string

flags: should be zero

This function will convert the provided UTF-8 password according to the normalization rules in RFC7613.

If the flag `GNUTLS_UTF8_IGNORE_ERRS` is specified, any UTF-8 encoding errors will be ignored, and in that case the output will be a copy of the input.

Returns: `GNUTLS_E_INVALID_UTF8_STRING` on invalid UTF-8 data, or 0 on success.

Since: 3.5.7

gnutls_verify_stored_pubkey

`int gnutls_verify_stored_pubkey (`*const char * `db_name`, gnutls_tdb_t `tdb`, const char * `host`, const char * `service`, gnutls_certificate_type_t `cert_type`, const gnutls_datum_t * `cert`, unsigned int `flags`*`)` [Function]

db_name: A file specifying the stored keys (use NULL for the default)

tdb: A storage structure or NULL to use the default

host: The peer's name

service: non-NULL if this key is specific to a service (e.g. http)

cert_type: The type of the certificate

cert: The raw (der) data of the certificate

flags: should be 0.

This function will try to verify a raw public-key or a public-key provided via a raw (DER-encoded) certificate using a list of stored public keys. The `service` field if non-NULL should be a port number.

The `db_name` variable if non-null specifies a custom backend for the retrieval of entries. If it is NULL then the default file backend will be used. In POSIX-like systems the file backend uses the $HOME/.gnutls/known_hosts file.

Note that if the custom storage backend is provided the retrieval function should return `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` if the host/service pair is found but key doesn't match, `GNUTLS_E_NO_CERTIFICATE_FOUND` if no such host/service with the given key is found, and 0 if it was found. The storage function should return 0 on success.

As of GnuTLS 3.6.6 this function also verifies raw public keys.

Returns: If no associated public key is found then `GNUTLS_E_NO_CERTIFICATE_FOUND` will be returned. If a key is found but does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned. On success, `GNUTLS_E_SUCCESS` (0) is returned, or a negative error value on other errors.

Since: 3.0.13

E.2 Datagram TLS API

The prototypes for the following functions lie in `gnutls/dtls.h`.

gnutls_dtls_cookie_send

`int gnutls_dtls_cookie_send (`*gnutls_datum_t * key*, *void * client_data*, *size_t client_data_size*, *gnutls_dtls_prestate_st * prestate*, *gnutls_transport_ptr_t ptr*, *gnutls_push_func push_func*`)` [Function]

key: is a random key to be used at cookie generation

client_data: contains data identifying the client (i.e. address)

client_data_size: The size of client's data

prestate: The previous cookie returned by `gnutls_dtls_cookie_verify()`

ptr: A transport pointer to be used by `push_func`

push_func: A function that will be used to reply

This function can be used to prevent denial of service attacks to a DTLS server by requiring the client to reply using a cookie sent by this function. That way it can be ensured that a client we allocated resources for (i.e. `gnutls_session_t`) is the one that the original incoming packet was originated from.

This function must be called at the first incoming packet, prior to allocating any resources and must be succeeded by `gnutls_dtls_cookie_verify()` .

Returns: the number of bytes sent, or a negative error code.

Since: 3.0

gnutls_dtls_cookie_verify

`int gnutls_dtls_cookie_verify (`*gnutls_datum_t * key*, *void * client_data*, *size_t client_data_size*, *void * _msg*, *size_t msg_size*, *gnutls_dtls_prestate_st * prestate*`)` [Function]

key: is a random key to be used at cookie generation

client_data: contains data identifying the client (i.e. address)

client_data_size: The size of client's data

_msg: An incoming message that initiates a connection.

msg_size: The size of the message.

prestate: The cookie of this client.

This function will verify the received message for a valid cookie. If a valid cookie is returned then it should be associated with the session using `gnutls_dtls_prestate_set()` ;

This function must be called after `gnutls_dtls_cookie_send()` .

Returns: `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

Since: 3.0

gnutls_dtls_get_data_mtu

`unsigned int gnutls_dtls_get_data_mtu (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the actual maximum transfer unit for application data. I.e. DTLS headers are subtracted from the actual MTU which is set using `gnutls_dtls_set_mtu()` .

Returns: the maximum allowed transfer unit.

Since: 3.0

gnutls_dtls_get_mtu

`unsigned int gnutls_dtls_get_mtu (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the MTU size as set with `gnutls_dtls_set_mtu()` . This is not the actual MTU of data you can transmit. Use `gnutls_dtls_get_data_mtu()` for that reason.

Returns: the set maximum transfer unit.

Since: 3.0

gnutls_dtls_get_timeout

`unsigned int gnutls_dtls_get_timeout (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

This function will return the milliseconds remaining for a retransmission of the previously sent handshake message. This function is useful when DTLS is used in non-blocking mode, to estimate when to call `gnutls_handshake()` if no packets have been received.

Returns: the remaining time in milliseconds.

Since: 3.0

gnutls_dtls_prestate_set

`void gnutls_dtls_prestate_set (`*gnutls_session_t* `session`, *gnutls_dtls_prestate_st* * `prestate`) [Function]

session: a new session

prestate: contains the client's prestate

This function will associate the prestate acquired by the cookie authentication with the client, with the newly established session.

This functions must be called after a successful `gnutls_dtls_cookie_verify()` and should be succeeded by the actual DTLS handshake using `gnutls_handshake()` .

Since: 3.0

gnutls_dtls_set_data_mtu

`int gnutls_dtls_set_data_mtu (`*gnutls_session_t* `session`, *unsigned int* `mtu``)` [Function]

session: is a `gnutls_session_t` type.

mtu: The maximum unencrypted transfer unit of the session

This function will set the maximum size of the *unencrypted* records which will be sent over a DTLS session. It is equivalent to calculating the DTLS packet overhead with the current encryption parameters, and calling `gnutls_dtls_set_mtu()` with that value. In particular, this means that you may need to call this function again after any negotiation or renegotiation, in order to ensure that the MTU is still sufficient to account for the new protocol overhead.

In most cases you only need to call `gnutls_dtls_set_mtu()` with the maximum MTU of your transport layer.

Returns: `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

Since: 3.1

gnutls_dtls_set_mtu

`void gnutls_dtls_set_mtu (`*gnutls_session_t* `session`, *unsigned int* `mtu``)` [Function]

session: is a `gnutls_session_t` type.

mtu: The maximum transfer unit of the transport

This function will set the maximum transfer unit of the transport that DTLS packets are sent over. Note that this should exclude the IP (or IPv6) and UDP headers. So for DTLS over IPv6 on an Ethernet device with MTU 1500, the DTLS MTU set with this function would be 1500 - 40 (IPV6 header) - 8 (UDP header) = 1452.

Since: 3.0

gnutls_dtls_set_timeouts

`void gnutls_dtls_set_timeouts (`*gnutls_session_t* `session`, *unsigned int* `retrans_timeout`, *unsigned int* `total_timeout``)` [Function]

session: is a `gnutls_session_t` type.

retrans_timeout: The time at which a retransmission will occur in milliseconds

total_timeout: The time at which the connection will be aborted, in milliseconds.

This function will set the timeouts required for the DTLS handshake protocol. The retransmission timeout is the time after which a message from the peer is not received, the previous messages will be retransmitted. The total timeout is the time after which the handshake will be aborted with `GNUTLS_E_TIMEDOUT` .

The DTLS protocol recommends the values of 1 sec and 60 seconds respectively, and these are the default values.

To disable retransmissions set a `retrans_timeout` larger than the `total_timeout` .

Since: 3.0

gnutls_record_get_discarded

`unsigned int gnutls_record_get_discarded (`*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Returns the number of discarded packets in a DTLS connection.

Returns: The number of discarded packets.

Since: 3.0

E.3 X.509 certificate API

The following functions are to be used for X.509 certificate handling. Their prototypes lie in `gnutls/x509.h`.

gnutls_certificate_get_trust_list

`void gnutls_certificate_get_trust_list` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_x509_trust_list_t* * `tlist`)

res: is a `gnutls_certificate_credentials_t` type.

tlist: Location where to store the trust list.

Obtains the list of trusted certificates stored in `res` and writes a pointer to it to the location `tlist` . The pointer will point to memory internal to `res` , and must not be deinitialized. It will be automatically deallocated when the `res` structure is deinitialized.

Since: 3.4.0

gnutls_certificate_set_trust_list

`void gnutls_certificate_set_trust_list` [Function]
(*gnutls_certificate_credentials_t* `res`, *gnutls_x509_trust_list_t* `tlist`, *unsigned* `flags`)

res: is a `gnutls_certificate_credentials_t` type.

tlist: is a `gnutls_x509_trust_list_t` type

flags: must be zero

This function sets a trust list in the gnutls_certificate_credentials_t type.

Note that the `tlist` will become part of the credentials structure and must not be deallocated. It will be automatically deallocated when the `res` structure is deinitialized.

Returns: `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

Since: 3.2.2

gnutls_certificate_verification_profile_get_id

`gnutls_certificate_verification_profiles_t` [Function]
`gnutls_certificate_verification_profile_get_id (`*const char* * `name`)

name: is a profile name

Convert a string to a gnutls_certificate_verification_profiles_t value. The names are compared in a case insensitive way.

Returns: a gnutls_certificate_verification_profiles_t id of the specified profile, or GNUTLS_PROFILE_UNKNOWN on failure.

gnutls_certificate_verification_profile_get_name

const char * gnutls_certificate_verification_profile_get_name (*gnutls_certificate_verification_profiles_t* id) [Function]

id: is a profile ID

Convert a gnutls_certificate_verification_profiles_t value to a string.

Returns: a string that contains the name of the specified profile or NULL .

gnutls_pkcs8_info

int gnutls_pkcs8_info (*const gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, unsigned int * schema, unsigned int * cipher, void * salt, unsigned int * salt_size, unsigned int * iter_count, char ** oid*) [Function]

data: Holds the PKCS 8 data

format: the format of the PKCS 8 data

schema: indicate the schema as one of gnutls_pkcs_encrypt_flags_t

cipher: the cipher used as gnutls_cipher_algorithm_t

salt: PBKDF2 salt (if non-NULL then salt_size initially holds its size)

salt_size: PBKDF2 salt size

iter_count: PBKDF2 iteration count

oid: if non-NULL it will contain an allocated null-terminated variable with the OID

This function will provide information on the algorithms used in a particular PKCS 8 structure. If the structure algorithms are unknown the code GNUTLS_E_UNKNOWN_CIPHER_TYPE will be returned, and only oid , will be set. That is, oid will be set on encrypted PKCS 8 structures whether supported or not. It must be deinitialized using gnutls_free() . The other variables are only set on supported structures.

Returns: GNUTLS_E_INVALID_REQUEST if the provided structure isn't an encrypted key, GNUTLS_E_UNKNOWN_CIPHER_TYPE if the structure's encryption isn't supported, or another negative error code in case of a failure. Zero on success.

Since: 3.4.0

gnutls_pkcs_schema_get_name

const char * gnutls_pkcs_schema_get_name (*unsigned int schema*) [Function]

schema: Holds the PKCS 12 or PBES2 schema (gnutls_pkcs_encrypt_flags_t)

This function will return a human readable description of the PKCS12 or PBES2 schema.

Returns: a constraint string or `NULL` on error.

Since: 3.4.0

gnutls_pkcs_schema_get_oid

`const char * gnutls_pkcs_schema_get_oid (unsigned int schema)` [Function]

schema: Holds the PKCS 12 or PBES2 schema (`gnutls_pkcs_encrypt_flags_t`)

This function will return the object identifier of the PKCS12 or PBES2 schema.

Returns: a constraint string or `NULL` on error.

Since: 3.4.0

gnutls_subject_alt_names_deinit

`void gnutls_subject_alt_names_deinit (gnutls_subject_alt_names_t sans)` [Function]

sans: The alternative names

This function will deinitialize an alternative names structure.

Since: 3.3.0

gnutls_subject_alt_names_get

`int gnutls_subject_alt_names_get (gnutls_subject_alt_names_t sans, unsigned int seq, unsigned int * san_type, gnutls_datum_t * san, gnutls_datum_t * othername_oid)` [Function]

sans: The alternative names

seq: The index of the name to get

san_type: Will hold the type of the name (of `gnutls_subject_alt_names_t`)

san: The alternative name data (should be treated as constant)

othername_oid: The object identifier if `san_type` is `GNUTLS_SAN_OTHERNAME` (should be treated as constant)

This function will return a specific alternative name as stored in the `sans` type. The returned values should be treated as constant and valid for the lifetime of `sans` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

Since: 3.3.0

gnutls_subject_alt_names_init

`int gnutls_subject_alt_names_init (gnutls_subject_alt_names_t * sans)` [Function]

sans: The alternative names

This function will initialize an alternative names structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_subject_alt_names_set

`int gnutls_subject_alt_names_set (`*gnutls_subject_alt_names_t* `sans`, *unsigned int* `san_type`, *const gnutls_datum_t * * `san`, *const char * `othername_oid`*)* [Function]

sans: The alternative names

san_type: The type of the name (of `gnutls_subject_alt_names_t`)

san: The alternative name data

othername_oid: The object identifier if `san_type` is `GNUTLS_SAN_OTHERNAME`

This function will store the specified alternative name in the `sans` .

Since version 3.5.7 the `GNUTLS_SAN_RFC822NAME` , `GNUTLS_SAN_DNSNAME` , and `GNUTLS_SAN_OTHERNAME_XMPP` are converted to ACE format when necessary.

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aia_deinit

`void gnutls_x509_aia_deinit (`*gnutls_x509_aia_t* `aia`*)* [Function]

aia: The authority info access

This function will deinitialize an authority info access type.

Since: 3.3.0

gnutls_x509_aia_get

`int gnutls_x509_aia_get (`*gnutls_x509_aia_t* `aia`, *unsigned int* `seq`, *gnutls_datum_t * `oid`, *unsigned * `san_type`, *gnutls_datum_t * `san`*)* [Function]

aia: The authority info access

seq: specifies the sequence number of the access descriptor (0 for the first one, 1 for the second etc.)

oid: the type of available data; to be treated as constant.

san_type: Will hold the type of the name of `gnutls_subject_alt_names_t` (may be null).

san: the access location name; to be treated as constant (may be null).

This function reads from the Authority Information Access type.

The `seq` input parameter is used to indicate which member of the sequence the caller is interested in. The first member is 0, the second member 1 and so on. When the `seq` value is out of bounds, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Typically `oid` is `GNUTLS_OID_AD_CAISSUERS` or `GNUTLS_OID_AD_OCSP` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aia_init

`int gnutls_x509_aia_init (`*gnutls_x509_aia_t * `aia`*`)` [Function]

aia: The authority info access

This function will initialize an authority info access type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aia_set

`int gnutls_x509_aia_set (`*gnutls_x509_aia_t `aia`, const char * `oid`, unsigned `san_type`, const gnutls_datum_t * `san`*`)` [Function]

aia: The authority info access

oid: the type of data.

san_type: The type of the name (of `gnutls_subject_alt_names_t`)

san: The alternative name data

This function will store the specified alternative name in the `aia` type.

Typically the value for `oid` should be `GNUTLS_OID_AD_OCSP` , or `GNUTLS_OID_AD_CAISSUERS` .

Since version 3.5.7 the `GNUTLS_SAN_RFC822NAME` , and `GNUTLS_SAN_DNSNAME` , are converted to ACE format when necessary.

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aki_deinit

`void gnutls_x509_aki_deinit (`*gnutls_x509_aki_t `aki`*`)` [Function]

aki: The authority key identifier type

This function will deinitialize an authority key identifier.

Since: 3.3.0

gnutls_x509_aki_get_cert_issuer

`int gnutls_x509_aki_get_cert_issuer (`*gnutls_x509_aki_t `aki`, unsigned int `seq`, unsigned int * `san_type`, gnutls_datum_t * `san`, gnutls_datum_t * `othername_oid`, gnutls_datum_t * `serial`*`)` [Function]

aki: The authority key ID

seq: The index of the name to get

san_type: Will hold the type of the name (of `gnutls_subject_alt_names_t`)

san: The alternative name data

othername_oid: The object identifier if `san_type` is `GNUTLS_SAN_OTHERNAME`

serial: The authorityCertSerialNumber number

This function will return a specific authorityCertIssuer name as stored in the `aki` type, as well as the authorityCertSerialNumber. All the returned values should be treated as constant, and may be set to `NULL` when are not required.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aki_get_id

`int gnutls_x509_aki_get_id (`*gnutls_x509_aki_t* `aki`, *gnutls_datum_t ** `id``)` [Function]

aki: The authority key ID

id: Will hold the identifier

This function will return the key identifier as stored in the `aki` type. The identifier should be treated as constant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aki_init

`int gnutls_x509_aki_init (`*gnutls_x509_aki_t ** `aki``)` [Function]

aki: The authority key ID type

This function will initialize an authority key ID.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aki_set_cert_issuer

`int gnutls_x509_aki_set_cert_issuer (`*gnutls_x509_aki_t* `aki`, *unsigned int* `san_type`, *const gnutls_datum_t ** `san`, *const char ** `othername_oid`, *const gnutls_datum_t ** `serial`)` [Function]

aki: The authority key ID

san_type: the type of the name (of `gnutls_subject_alt_names_t`), may be null

san: The alternative name data

othername_oid: The object identifier if `san_type` is `GNUTLS_SAN_OTHERNAME`

serial: The authorityCertSerialNumber number (may be null)

This function will set the authorityCertIssuer name and the authorityCertSerialNumber to be stored in the `aki` type. When storing multiple names, the serial should be set on the first call, and subsequent calls should use a `NULL` serial.

Since version 3.5.7 the `GNUTLS_SAN_RFC822NAME` , `GNUTLS_SAN_DNSNAME` , and `GNUTLS_SAN_OTHERNAME_XMPP` are converted to ACE format when necessary.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_aki_set_id

`int gnutls_x509_aki_set_id (`*gnutls_x509_aki_t* `aki`, *const gnutls_datum_t ** `id``)` [Function]

aki: The authority key ID

id: the key identifier

This function will set the keyIdentifier to be stored in the `aki` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_cidr_to_rfc5280

`int gnutls_x509_cidr_to_rfc5280 (`*const char ** `cidr`, *gnutls_datum_t ** `cidr_rfc5280``)` [Function]

cidr: CIDR in RFC4632 format (IP/prefix), null-terminated

cidr_rfc5280: CIDR range converted to RFC5280 format

This function will convert text CIDR range with prefix (such as '10.0.0.0/8') to RFC5280 (IP address in network byte order followed by its network mask). Works for both IPv4 and IPv6.

The resulting object is directly usable for IP name constraints usage, for example in functions `gnutls_x509_name_constraints_add_permitted` or `gnutls_x509_name_constraints_add_excluded` .

The data in datum needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.4

gnutls_x509_crl_check_issuer

`unsigned gnutls_x509_crl_check_issuer (`*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_t* `issuer``)` [Function]

crl: is the CRL to be checked

issuer: is the certificate of a possible issuer

This function will check if the given CRL was issued by the given issuer certificate.

Returns: true (1) if the given CRL was issued by the given issuer, and false (0) if not.

gnutls_x509_crl_deinit

`void gnutls_x509_crl_deinit (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: The data to be deinitialized

This function will deinitialize a CRL structure.

gnutls_x509_crl_dist_points_deinit

`void gnutls_x509_crl_dist_points_deinit` [Function]
(*gnutls_x509_crl_dist_points_t* `cdp`)

cdp: The CRL distribution points

This function will deinitialize a CRL distribution points type.

Since: 3.3.0

gnutls_x509_crl_dist_points_get

`int gnutls_x509_crl_dist_points_get` [Function]
(*gnutls_x509_crl_dist_points_t* `cdp`, *unsigned int* `seq`, *unsigned int ** `type`, *gnutls_datum_t ** `san`, *unsigned int ** `reasons`)

cdp: The CRL distribution points

seq: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)

type: The name type of the corresponding name (gnutls_x509_subject_alt_name_t)

san: The distribution point names (to be treated as constant)

reasons: Revocation reasons. An ORed sequence of flags from `gnutls_x509_crl_reason_flags_t` .

This function retrieves the individual CRL distribution points (2.5.29.31), contained in provided type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

gnutls_x509_crl_dist_points_init

`int gnutls_x509_crl_dist_points_init` [Function]
(*gnutls_x509_crl_dist_points_t ** `cdp`)

cdp: The CRL distribution points

This function will initialize a CRL distribution points type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_crl_dist_points_set

`int gnutls_x509_crl_dist_points_set` [Function]
(*gnutls_x509_crl_dist_points_t* `cdp`, *gnutls_x509_subject_alt_name_t* `type`, *const gnutls_datum_t ** `san`, *unsigned int* `reasons`)

cdp: The CRL distribution points

type: The type of the name (of `gnutls_subject_alt_names_t`)

san: The point name data

reasons: Revocation reasons. An ORed sequence of flags from `gnutls_x509_crl_reason_flags_t` .

This function will store the specified CRL distribution point value the `cdp` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

Since: 3.3.0

gnutls_x509_crl_export

`int gnutls_x509_crl_export` (*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_fmt_t* `format`, *void ** `output_data`, *size_t ** `output_data_size`) [Function]

crl: Holds the revocation list

format: the format of output params. One of PEM or DER.

output_data: will contain a private key PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the revocation list to DER or PEM format.

If the buffer provided is not long enough to hold the output, then `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN X509 CRL".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_export2

`int gnutls_x509_crl_export2` (*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t ** `out`) [Function]

crl: Holds the revocation list

format: the format of output params. One of PEM or DER.

out: will contain a private key PEM or DER encoded

This function will export the revocation list to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN X509 CRL".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 3.1.3

gnutls_x509_crl_get_authority_key_gn_serial

`int gnutls_x509_crl_get_authority_key_gn_serial` (*gnutls_x509_crl_t* `crl`, *unsigned int* `seq`, *void *** `alt`, *size_t ** `alt_size`, *unsigned int ** `alt_type`, *void *** `serial`, *size_t ** `serial_size`, *unsigned int ** `critical`) [Function]

crl: should contain a `gnutls_x509_crl_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

alt: is the place where the alternative name will be copied to

alt_size: holds the size of alt.

alt_type: holds the type of the alternative name (one of gnutls_x509_subject_alt_name_t).

serial: buffer to store the serial number (may be null)

serial_size: Holds the size of the serial field (may be null)

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509 authority key identifier when stored as a general name (authorityCertIssuer) and serial number.

Because more than one general names might be stored `seq` can be used as a counter to request them all until `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Returns: Returns 0 on success, or an error code.

Since: 3.0

gnutls_x509_crl_get_authority_key_id

`int gnutls_x509_crl_get_authority_key_id` (*gnutls_x509_crl_t* `crl`, *void * id*, *size_t * id_size*, *unsigned int * critical*) [Function]

crl: should contain a `gnutls_x509_crl_t` type

id: The place where the identifier will be copied

id_size: Holds the size of the result field.

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the CRL authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension and `GNUTLS_E_X509_UNSUPPORTED_EXTENSION` , if the extension contains the name and serial number of the certificate. In that case `gnutls_x509_crl_get_authority_key_gn_serial()` may be used.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error.

Since: 2.8.0

gnutls_x509_crl_get_crt_count

`int gnutls_x509_crl_get_crt_count` (*gnutls_x509_crl_t* `crl`) [Function]

crl: should contain a `gnutls_x509_crl_t` type

This function will return the number of revoked certificates in the given CRL.

Returns: number of certificates, a negative error code on failure.

gnutls_x509_crl_get_crt_serial

`int gnutls_x509_crl_get_crt_serial` (*gnutls_x509_crl_t* `crl`, *unsigned indx*, *unsigned char * serial*, *size_t * serial_size*, *time_t * t*) [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: the index of the certificate to extract (starting from 0)

serial: where the serial number will be copied

serial_size: initially holds the size of serial

t: if non null, will hold the time this certificate was revoked

This function will retrieve the serial number of the specified, by the index, revoked certificate.

Note that this function will have performance issues in large sequences of revoked certificates. In that case use `gnutls_x509_crl_iter_crt_serial()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_get_dn_oid

`int gnutls_x509_crl_get_dn_oid` (*gnutls_x509_crl_t* `crl`, *unsigned* `indx`, *void * * `oid`, *size_t * `sizeof_oid`) [Function]

crl: should contain a gnutls_x509_crl_t type

indx: Specifies which DN OID to send. Use (0) to get the first one.

oid: a pointer to store the OID (may be null)

sizeof_oid: initially holds the size of 'oid'

This function will extract the requested OID of the name of the CRL issuer, specified by the given index.

If oid is null then only the size will be filled.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the sizeof_oid will be updated with the required size. On success 0 is returned.

gnutls_x509_crl_get_extension_data

`int gnutls_x509_crl_get_extension_data` (*gnutls_x509_crl_t* `crl`, *unsigned* `indx`, *void * * `data`, *size_t * `sizeof_data`) [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: Specifies which extension OID to send. Use (0) to get the first one.

data: a pointer to a structure to hold the data (may be null)

sizeof_data: initially holds the size of `oid`

This function will return the requested extension data in the CRL. The extension data will be stored as a string in the provided buffer.

Use `gnutls_x509_crl_get_extension_info()` to extract the OID and critical flag. Use `gnutls_x509_crl_get_extension_info()` instead, if you want to get data indexed by the extension OID rather than sequence.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crl_get_extension_data2

`int gnutls_x509_crl_get_extension_data2 (`*gnutls_x509_crl_t* `crl`, *unsigned* `indx`, *gnutls_datum_t * * `data`*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: Specifies which extension OID to read. Use (0) to get the first one.

data: will contain the extension DER-encoded data

This function will return the requested by the index extension data in the certificate revocation list. The extension data will be allocated using `gnutls_malloc()` .

Use `gnutls_x509_crt_get_extension_info()` to extract the OID.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crl_get_extension_info

`int gnutls_x509_crl_get_extension_info (`*gnutls_x509_crl_t* `crl`, *unsigned* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`, *unsigned int * `critical`*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: Specifies which extension OID to send, use (0) to get the first one.

oid: a pointer to store the OID

sizeof_oid: initially holds the maximum size of `oid` , on return holds actual size of `oid` .

critical: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the CRL, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use `gnutls_x509_crl_get_extension_data()` to extract the data.

If the buffer provided is not long enough to hold the output, then * `sizeof_oid` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crl_get_extension_oid

`int gnutls_x509_crl_get_extension_oid (`*gnutls_x509_crl_t* `crl`, *unsigned* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

indx: Specifies which extension OID to send, use (0) to get the first one.

oid: a pointer to store the OID (may be null)

sizeof_oid: initially holds the size of `oid`

This function will return the requested extension OID in the CRL. The extension OID will be stored as a string in the provided buffer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crl_get_issuer_dn

`int gnutls_x509_crl_get_issuer_dn (`*gnutls_x509_crl_t* `crl`*, char * buf, size_t * sizeof_buf*`)` [Function]

crl: should contain a gnutls_x509_crl_t type

buf: a pointer to a structure to hold the peer's name (may be null)

sizeof_buf: initially holds the size of `buf`

This function will copy the name of the CRL issuer in the provided buffer. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If buf is `NULL` then only the size will be filled.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crl_get_issuer_dn3()` .

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the sizeof_buf will be updated with the required size, and 0 on success.

gnutls_x509_crl_get_issuer_dn2

`int gnutls_x509_crl_get_issuer_dn2 (`*gnutls_x509_crl_t* `crl`*, gnutls_datum_t * dn*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the CRL issuer. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crl_get_issuer_dn3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.10

gnutls_x509_crl_get_issuer_dn3

`int gnutls_x509_crl_get_issuer_dn3 (`*gnutls_x509_crl_t* `crl`*, gnutls_datum_t * dn, unsigned flags*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

dn: a pointer to a structure to hold the name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will allocate buffer and copy the name of the CRL issuer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

When the flag GNUTLS_X509_DN_FLAG_COMPAT is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.5.7

gnutls_x509_crl_get_issuer_dn_by_oid

int gnutls_x509_crl_get_issuer_dn_by_oid (*gnutls_x509_crl_t* `crl`, *const char * oid*, *unsigned indx*, *unsigned int raw_flag*, *void * buf*, *size_t * sizeof_buf*) [Function]

crl: should contain a gnutls_x509_crl_t type

oid: holds an Object Identified in null terminated string

indx: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

raw_flag: If non-zero returns the raw DER data of the DN part.

buf: a pointer to a structure to hold the peer's name (may be null)

sizeof_buf: initially holds the size of buf

This function will extract the part of the name of the CRL issuer specified by the given OID. The output will be encoded as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using gnutls_x509_dn_oid_known() .

If buf is null then only the size will be filled.

Returns: GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and in that case the sizeof_buf will be updated with the required size, and 0 on success.

gnutls_x509_crl_get_next_update

time_t gnutls_x509_crl_get_next_update (*gnutls_x509_crl_t* `crl`) [Function]

crl: should contain a gnutls_x509_crl_t type

This function will return the time the next CRL will be issued. This field is optional in a CRL so it might be normal to get an error instead.

Returns: when the next CRL will be issued, or (time_t)-1 on error.

gnutls_x509_crl_get_number

`int gnutls_x509_crl_get_number (`*gnutls_x509_crl_t* `crl`, *void ** `ret`, *size_t ** `ret_size`, *unsigned int ** `critical``)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

ret: The place where the number will be copied

ret_size: Holds the size of the result field.

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the CRL number extension. This is obtained by the CRL Number extension field (2.5.29.20).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error.

Since: 2.8.0

gnutls_x509_crl_get_raw_issuer_dn

`int gnutls_x509_crl_get_raw_issuer_dn (`*gnutls_x509_crl_t* `crl`, *gnutls_datum_t ** `dn`*)* [Function]

crl: should contain a gnutls_x509_crl_t type

dn: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length.

Returns: a negative error code on error, and (0) on success.

Since: 2.12.0

gnutls_x509_crl_get_signature

`int gnutls_x509_crl_get_signature (`*gnutls_x509_crl_t* `crl`, *char ** `sig`, *size_t ** `sizeof_sig`*)* [Function]

crl: should contain a gnutls_x509_crl_t type

sig: a pointer where the signature part will be copied (may be null).

sizeof_sig: initially holds the size of `sig`

This function will extract the signature field of a CRL.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_get_signature_algorithm

`int gnutls_x509_crl_get_signature_algorithm (`*gnutls_x509_crl_t* `crl`*)* [Function]

crl: should contain a `gnutls_x509_crl_t` type

This function will return a value of the `gnutls_sign_algorithm_t` enumeration that is the signature algorithm.

Since 3.6.0 this function never returns a negative error code. Error cases and unknown/unsupported signature algorithms are mapped to `GNUTLS_SIGN_UNKNOWN` .

Returns: a `gnutls_sign_algorithm_t` value

gnutls_x509_crl_get_signature_oid

`int gnutls_x509_crl_get_signature_oid (`*gnutls_x509_crl_t* `crl`, *char * * `oid`*, *size_t * `oid_size`*`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will return the OID of the signature algorithm that has been used to sign this CRL. This is function is useful in the case `gnutls_x509_crl_get_signature_algorithm()` returned `GNUTLS_SIGN_UNKNOWN` .

Returns: zero or a negative error code on error.

Since: 3.5.0

gnutls_x509_crl_get_this_update

`time_t gnutls_x509_crl_get_this_update (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

This function will return the time this CRL was issued.

Returns: when the CRL was issued, or (time_t)-1 on error.

gnutls_x509_crl_get_version

`int gnutls_x509_crl_get_version (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

This function will return the version of the specified CRL.

Returns: The version number, or a negative error code on error.

gnutls_x509_crl_import

`int gnutls_x509_crl_import (`*gnutls_x509_crl_t* `crl`, *const gnutls_datum_t * `data`*, *gnutls_x509_crt_fmt_t* `format`*`)` [Function]

crl: The data to store the parsed CRL.

data: The DER or PEM encoded CRL.

format: One of DER or PEM

This function will convert the given DER or PEM encoded CRL to the native `gnutls_x509_crl_t` format. The output will be stored in 'crl'.

If the CRL is PEM encoded it should have a header of "X509 CRL".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_init

`int gnutls_x509_crl_init (`*gnutls_x509_crl_t * `crl`*`)` [Function]

crl: A pointer to the type to be initialized

This function will initialize a CRL structure. CRL stands for Certificate Revocation List. A revocation list usually contains lists of certificate serial numbers that

have been revoked by an Authority. The revocation lists are always signed with the authority's private key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_iter_crt_serial

`int gnutls_x509_crl_iter_crt_serial (`*gnutls_x509_crl_t* **`crl`**, *gnutls_x509_crl_iter_t ** **`iter`**, *unsigned char ** **`serial`**, *size_t ** **`serial_size`**, *time_t ** **`t`**`)` [Function]

crl: should contain a `gnutls_x509_crl_t` type

iter: A pointer to an iterator (initially the iterator should be `NULL`)

serial: where the serial number will be copied

serial_size: initially holds the size of serial

t: if non null, will hold the time this certificate was revoked

This function performs the same as `gnutls_x509_crl_get_crt_serial()` , but reads sequentially and keeps state in the iterator between calls. That allows it to provide better performance in sequences with many elements (50000+).

When past the last element is accessed `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned and the iterator is reset.

After use, the iterator must be deinitialized using `gnutls_x509_crl_iter_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_iter_deinit

`void gnutls_x509_crl_iter_deinit (`*gnutls_x509_crl_iter_t* **`iter`**`)` [Function]

iter: The iterator to be deinitialized

This function will deinitialize an iterator type.

gnutls_x509_crl_list_import

`int gnutls_x509_crl_list_import (`*gnutls_x509_crl_t ** **`crls`**, *unsigned int ** **`crl_max`**, *const gnutls_datum_t ** **`data`**, *gnutls_x509_crt_fmt_t* **`format`**, *unsigned int* **`flags`**`)` [Function]

crls: Indicates where the parsed CRLs will be copied to. Must not be initialized.

crl_max: Initially must hold the maximum number of crls. It will be updated with the number of crls available.

data: The PEM encoded CRLs

format: One of DER or PEM.

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded CRL list to the native gnutls_x509_crl_t format. The output will be stored in `crls` . They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CRL".

Returns: the number of certificates read or a negative error value.

Since: 3.0

gnutls_x509_crl_list_import2

`int gnutls_x509_crl_list_import2` (*gnutls_x509_crl_t ** `crls`, unsigned int * `size`, const gnutls_datum_t * `data`, gnutls_x509_crt_fmt_t `format`, unsigned int `flags`*) [Function]

crls: Will contain the parsed crl list.

size: It will contain the size of the list.

data: The PEM encoded CRL.

format: One of DER or PEM.

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded CRL list to the native gnutls_x509_crl_t format. The output will be stored in `crls` . They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CRL".

Returns: the number of certificates read or a negative error value.

Since: 3.0

gnutls_x509_crl_print

`int gnutls_x509_crl_print` (*gnutls_x509_crl_t `crl`, gnutls_certificate_print_formats_t `format`, gnutls_datum_t * `out`*) [Function]

crl: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print a X.509 certificate revocation list, suitable for display to a human.

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_set_authority_key_id

`int gnutls_x509_crl_set_authority_key_id` (*gnutls_x509_crl_t `crl`, const void * `id`, size_t `id_size`*) [Function]

crl: a CRL of type `gnutls_x509_crl_t`

id: The key ID

id_size: Holds the size of the serial field.

This function will set the CRL's authority key ID extension. Only the keyIdentifier field can be set with this function. This may be used by an authority that holds multiple private keys, to distinguish the used key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crl_set_crt

`int gnutls_x509_crl_set_crt` (*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_t* `crt`, *time_t* `revocation_time`) [Function]

crl: should contain a gnutls_x509_crl_t type

crt: a certificate of type `gnutls_x509_crt_t` with the revoked certificate

revocation_time: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_set_crt_serial

`int gnutls_x509_crl_set_crt_serial` (*gnutls_x509_crl_t* `crl`, *const void * * `serial`, *size_t* `serial_size`, *time_t* `revocation_time`) [Function]

crl: should contain a gnutls_x509_crl_t type

serial: The revoked certificate's serial number

serial_size: Holds the size of the serial field.

revocation_time: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_set_next_update

`int gnutls_x509_crl_set_next_update` (*gnutls_x509_crl_t* `crl`, *time_t* `exp_time`) [Function]

crl: should contain a gnutls_x509_crl_t type

exp_time: The actual time

This function will set the time this CRL will be updated. This is an optional value to be set on a CRL and this call can be omitted when generating a CRL.

Prior to GnuTLS 3.5.7, setting a nextUpdate field was required in order to generate a CRL.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_set_number

`int gnutls_x509_crl_set_number` (*gnutls_x509_crl_t* `crl`, *const void * * `nr`, *size_t* `nr_size`) [Function]

crl: a CRL of type `gnutls_x509_crl_t`

nr: The CRL number

nr_size: Holds the size of the nr field.

This function will set the CRL's number extension. This is to be used as a unique and monotonic number assigned to the CRL by the authority.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crl_set_this_update

`int gnutls_x509_crl_set_this_update (`*gnutls_x509_crl_t* `crl`, *time_t* `act_time``)` [Function]

crl: should contain a gnutls_x509_crl_t type

act_time: The actual time

This function will set the time this CRL was issued.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_set_version

`int gnutls_x509_crl_set_version (`*gnutls_x509_crl_t* `crl`, *unsigned int* `version``)` [Function]

crl: should contain a gnutls_x509_crl_t type

version: holds the version number. For CRLv1 crls must be 1.

This function will set the version of the CRL. This must be one for CRL version 1, and so on. The CRLs generated by gnutls should have a version number of 2.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_sign

`int gnutls_x509_crl_sign (`*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_t* `issuer`, *gnutls_x509_privkey_t* `issuer_key`)` [Function]

crl: should contain a gnutls_x509_crl_t type

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

This function is the same a `gnutls_x509_crl_sign2()` with no flags, and an appropriate hash algorithm. The hash algorithm used may vary between versions of GnuTLS, and it is tied to the security level of the issuer's public key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_sign2

`int gnutls_x509_crl_sign2` (*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_t* `issuer`, *gnutls_x509_privkey_t* `issuer_key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

crl: should contain a gnutls_x509_crl_t type

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use. GNUTLS_DIG_SHA256 is the safe choice unless you know what you're doing.

flags: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed CRL will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crl_verify

`int gnutls_x509_crl_verify` (*gnutls_x509_crl_t* `crl`, *const gnutls_x509_crt_t * * trusted_cas*, *unsigned* `tcas_size`, *unsigned int* `flags`, *unsigned int * * verify*) [Function]

crl: is the crl to be verified

trusted_cas: is a certificate list that is considered to be trusted one

tcas_size: holds the number of CA certificates in CA_list

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

verify: will hold the crl verification output.

This function will try to verify the given crl and return its verification status. See `gnutls_x509_crt_list_verify()` for a detailed description of return values. Note that since GnuTLS 3.1.4 this function includes the time checks.

Note that value in `verify` is set only when the return value of this function is success (i.e, failure to trust a CRL a certificate does not imply a negative return value).

Before GnuTLS 3.5.7 this function would return zero or a positive number on success.

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

gnutls_x509_crq_deinit

`void gnutls_x509_crq_deinit` (*gnutls_x509_crq_t* `crq`) [Function]

crq: the type to be deinitialized

This function will deinitialize a PKCS10 certificate request structure.

gnutls_x509_crq_export

`int gnutls_x509_crq_export` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_crt_fmt_t* `format`, *void * * `output_data`, *size_t * `output_data_size`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

format: the format of output params. One of PEM or DER.

output_data: will contain a certificate request PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the certificate request to a PEM or DER encoded PKCS10 structure.

If the buffer provided is not long enough to hold the output, then `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned and * `output_data_size` will be updated.

If the structure is PEM encoded, it will have a header of "BEGIN NEW CERTIFICATE REQUEST".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_export2

`int gnutls_x509_crq_export2` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t * `out`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

format: the format of output params. One of PEM or DER.

out: will contain a certificate request PEM or DER encoded

This function will export the certificate request to a PEM or DER encoded PKCS10 structure.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN NEW CERTIFICATE REQUEST".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 3.1.3

gnutls_x509_crq_get_attribute_by_oid

`int gnutls_x509_crq_get_attribute_by_oid (`*gnutls_x509_crq_t* `crq`, *const char * * `oid`, *unsigned* `indx`, *void * * `buf`, *size_t * * `buf_size`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

oid: holds an Object Identifier in null-terminated string

indx: In case multiple same OIDs exist in the attribute list, this specifies which to get, use (0) to get the first one

buf: a pointer to a structure to hold the attribute data (may be `NULL`)

buf_size: initially holds the size of `buf`

This function will return the attribute in the certificate request specified by the given Object ID. The attribute will be DER encoded.

Attributes in a certificate request is an optional set of data appended to the request. Their interpretation depends on the CA policy.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_get_attribute_data

`int gnutls_x509_crq_get_attribute_data (`*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void * * `data`, *size_t * * `sizeof_data`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

indx: Specifies which attribute number to get. Use (0) to get the first one.

data: a pointer to a structure to hold the data (may be null)

sizeof_data: initially holds the size of `oid`

This function will return the requested attribute data in the certificate request. The attribute data will be stored as a string in the provided buffer.

Use `gnutls_x509_crq_get_attribute_info()` to extract the OID. Use `gnutls_x509_crq_get_attribute_by_oid()` instead, if you want to get data indexed by the attribute OID rather than sequence.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crq_get_attribute_info

`int gnutls_x509_crq_get_attribute_info (`*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

indx: Specifies which attribute number to get. Use (0) to get the first one.

oid: a pointer to a structure to hold the OID

sizeof_oid: initially holds the maximum size of `oid` , on return holds actual size of `oid` .

This function will return the requested attribute OID in the certificate, and the critical flag for it. The attribute OID will be stored as a string in the provided buffer. Use `gnutls_x509_crq_get_attribute_data()` to extract the data.

If the buffer provided is not long enough to hold the output, then * `sizeof_oid` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crq_get_basic_constraints

`int gnutls_x509_crq_get_basic_constraints (`*gnutls_x509_crq_t* `crq`*, unsigned int * *`critical`*, unsigned int * *`ca`*, int * *`pathlen`*`)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

critical: will be non-zero if the extension is marked as critical

ca: pointer to output integer indicating CA status, may be NULL, value is 1 if the certificate CA flag is set, 0 otherwise.

pathlen: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pathLenConstraint field and the actual value, -1 indicate that the field is absent.

This function will read the certificate's basic constraints, and return the certificates CA status. It reads the basicConstraints X.509 extension (2.5.29.19).

Returns: If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the basicConstraints extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crq_get_challenge_password

`int gnutls_x509_crq_get_challenge_password (`*gnutls_x509_crq_t* `crq`*, char * *`pass`*, size_t * *`pass_size`*`)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

pass: will hold a (0)-terminated password string

pass_size: Initially holds the size of `pass` .

This function will return the challenge password in the request. The challenge password is intended to be used for requesting a revocation of the certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_get_dn

`int gnutls_x509_crq_get_dn (`*gnutls_x509_crq_t* `crq`*, char * *`buf`*, size_t * *`buf_size`*`)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

buf: a pointer to a structure to hold the name (may be `NULL`)

buf_size: initially holds the size of `buf`

This function will copy the name of the Certificate request subject to the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC 2253. The output string `buf` will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crq_get_dn3()` .

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the * `buf_size` will be updated with the required size. On success 0 is returned.

gnutls_x509_crq_get_dn2

`int gnutls_x509_crq_get_dn2` (*gnutls_x509_crq_t* ***crq***, *gnutls_datum_t * **dn***) [Function]

crq: should contain a `gnutls_x509_crq_t` type

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate request. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crq_get_dn3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. and a negative error code on error.

Since: 3.1.10

gnutls_x509_crq_get_dn3

`int gnutls_x509_crq_get_dn3` (*gnutls_x509_crq_t* ***crq***, *gnutls_datum_t * **dn***, *unsigned* ***flags***) [Function]

crq: should contain a `gnutls_x509_crq_t` type

dn: a pointer to a structure to hold the name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will allocate buffer and copy the name of the Certificate request. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. and a negative error code on error.

Since: 3.5.7

gnutls_x509_crq_get_dn_by_oid

`int gnutls_x509_crq_get_dn_by_oid` (*gnutls_x509_crq_t* `crq`, *const char * oid*, *unsigned* `indx`, *unsigned int* `raw_flag`, *void * buf*, *size_t * buf_size*) [Function]

crq: should contain a gnutls_x509_crq_t type

oid: holds an Object Identifier in a null terminated string

indx: In case multiple same OIDs exist in the RDN, this specifies which to get. Use (0) to get the first one.

raw_flag: If non-zero returns the raw DER data of the DN part.

buf: a pointer to a structure to hold the name (may be `NULL`)

buf_size: initially holds the size of `buf`

This function will extract the part of the name of the Certificate request subject, specified by the given OID. The output will be encoded as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC2253 – in hex format with a '\#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()` .

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the * `buf_size` will be updated with the required size. On success 0 is returned.

gnutls_x509_crq_get_dn_oid

`int gnutls_x509_crq_get_dn_oid` (*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void * oid*, *size_t * sizeof_oid*) [Function]

crq: should contain a gnutls_x509_crq_t type

indx: Specifies which DN OID to get. Use (0) to get the first one.

oid: a pointer to a structure to hold the name (may be `NULL`)

sizeof_oid: initially holds the size of `oid`

This function will extract the requested OID of the name of the certificate request subject, specified by the given index.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the * `sizeof_oid` will be updated with the required size. On success 0 is returned.

gnutls_x509_crq_get_extension_by_oid

`int gnutls_x509_crq_get_extension_by_oid` (*gnutls_x509_crq_t* `crq`, *const char * oid*, *unsigned* `indx`, *void * buf*, *size_t * buf_size*, *unsigned int * critical*) [Function]

crq: should contain a `gnutls_x509_crq_t` type

oid: holds an Object Identifier in a null terminated string

indx: In case multiple same OIDs exist in the extensions, this specifies which to get. Use (0) to get the first one.

buf: a pointer to a structure to hold the name (may be null)

buf_size: initially holds the size of `buf`

critical: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If the certificate does not contain the specified extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 2.8.0

gnutls_x509_crq_get_extension_by_oid2

`int gnutls_x509_crq_get_extension_by_oid2 (`*gnutls_x509_crq_t* `crq`, *const char * * `oid`*, *unsigned* `indx`, *gnutls_datum_t * * `output`*, *unsigned int * `critical`*`)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

oid: holds an Object Identifier in a null terminated string

indx: In case multiple same OIDs exist in the extensions, this specifies which to get. Use (0) to get the first one.

output: will hold the allocated extension data

critical: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If the certificate does not contain the specified extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 3.3.8

gnutls_x509_crq_get_extension_data

`int gnutls_x509_crq_get_extension_data (`*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void * * `data`*, *size_t * * `sizeof_data`*`)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

indx: Specifies which extension number to get. Use (0) to get the first one.

data: a pointer to a structure to hold the data (may be null)

sizeof_data: initially holds the size of `oid`

This function will return the requested extension data in the certificate. The extension data will be stored as a string in the provided buffer.

Use `gnutls_x509_crq_get_extension_info()` to extract the OID and critical flag. Use `gnutls_x509_crq_get_extension_by_oid()` instead, if you want to get data indexed by the extension OID rather than sequence.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

Since: 2.8.0

gnutls_x509_crq_get_extension_data2

int gnutls_x509_crq_get_extension_data2 (*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *gnutls_datum_t* * `data`) [Function]

crq: should contain a gnutls_x509_crq_t type

indx: Specifies which extension OID to read. Use (0) to get the first one.

data: will contain the extension DER-encoded data

This function will return the requested extension data in the certificate request. The extension data will be allocated using gnutls_malloc() .

Use gnutls_x509_crq_get_extension_info() to extract the OID.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

Since: 3.3.0

gnutls_x509_crq_get_extension_info

int gnutls_x509_crq_get_extension_info (*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void* * `oid`, *size_t* * `sizeof_oid`, *unsigned int* * `critical`) [Function]

crq: should contain a gnutls_x509_crq_t type

indx: Specifies which extension number to get. Use (0) to get the first one.

oid: a pointer to store the OID

sizeof_oid: initially holds the maximum size of oid , on return holds actual size of oid .

critical: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the certificate, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use gnutls_x509_crq_get_extension_data() to extract the data.

If the buffer provided is not long enough to hold the output, then * sizeof_oid is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

Since: 2.8.0

gnutls_x509_crq_get_key_id

`int gnutls_x509_crq_get_key_id` (*gnutls_x509_crq_t* `crq`, *unsigned int* `flags`, *unsigned char * * `output_data`, *size_t * * `output_data_size`) [Function]

crq: a certificate of type `gnutls_x509_crq_t`

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then * `output_data_size` is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 2.8.0

gnutls_x509_crq_get_key_purpose_oid

`int gnutls_x509_crq_get_key_purpose_oid` (*gnutls_x509_crq_t* `crq`, *unsigned* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`, *unsigned int * * `critical`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

indx: This specifies which OID to return, use (0) to get the first one

oid: a pointer to store the OID (may be `NULL`)

sizeof_oid: initially holds the size of `oid`

critical: output variable with critical flag, may be `NULL` .

This function will extract the key purpose OIDs of the Certificate specified by the given index. These are stored in the Extended Key Usage extension (2.5.29.37). See the GNUTLS_KP_* definitions for human readable names.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the * `sizeof_oid` will be updated with the required size. On success 0 is returned.

Since: 2.8.0

gnutls_x509_crq_get_key_rsa_raw

`int gnutls_x509_crq_get_key_rsa_raw` (*gnutls_x509_crq_t* `crq`, *gnutls_datum_t * * `m`, *gnutls_datum_t * * `e`) [Function]

crq: Holds the certificate

m: will hold the modulus

e: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_get_key_usage

`int gnutls_x509_crq_get_key_usage (`*gnutls_x509_crq_t* `crq`, *unsigned int ** `key_usage`, *unsigned int ** `critical``)` [Function]

crq: should contain a gnutls_x509_crq_t type

key_usage: where the key usage bits will be stored

critical: will be non-zero if the extension is marked as critical

This function will return certificate's key usage, by reading the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: GNUTLS_KEY_DIGITAL_SIGNATURE , GNUTLS_KEY_NON_REPUDIATION , GNUTLS_KEY_KEY_ENCIPHERMENT , GNUTLS_KEY_DATA_ENCIPHERMENT , GNUTLS_KEY_KEY_AGREEMENT , GNUTLS_KEY_KEY_CERT_SIGN , GNUTLS_KEY_CRL_SIGN , GNUTLS_KEY_ENCIPHER_ONLY , GNUTLS_KEY_DECIPHER_ONLY .

Returns: the certificate key usage, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

Since: 2.8.0

gnutls_x509_crq_get_pk_algorithm

`int gnutls_x509_crq_get_pk_algorithm (`*gnutls_x509_crq_t* `crq`, *unsigned int ** `bits``)` [Function]

crq: should contain a gnutls_x509_crq_t type

bits: if bits is non-NULL it will hold the size of the parameters' in bits

This function will return the public key algorithm of a PKCS10 certificate request.

If bits is non-NULL , it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

Returns: a member of the gnutls_pk_algorithm_t enumeration on success, or a negative error code on error.

gnutls_x509_crq_get_pk_oid

`int gnutls_x509_crq_get_pk_oid (`*gnutls_x509_crq_t* `crq`, *char ** `oid`, *size_t ** `oid_size`)` [Function]

crq: should contain a gnutls_x509_crq_t type

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of oid

This function will return the OID of the public key algorithm on that certificate request. This function is useful in the case gnutls_x509_crq_get_pk_algorithm() returned GNUTLS_PK_UNKNOWN .

Returns: zero or a negative error code on error.

Since: 3.5.0

gnutls_x509_crq_get_private_key_usage_period

`int gnutls_x509_crq_get_private_key_usage_period` [Function]
(*gnutls_x509_crq_t* `crq`, *time_t ** `activation`, *time_t ** `expiration`, *unsigned int ** `critical`)

crq: should contain a `gnutls_x509_crq_t` type

activation: The activation time

expiration: The expiration time

critical: the extension status

This function will return the expiration and activation times of the private key of the certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

gnutls_x509_crq_get_signature_algorithm

`int gnutls_x509_crq_get_signature_algorithm` [Function]
(*gnutls_x509_crq_t* `crq`)

crq: should contain a `gnutls_x509_cr_t` type

This function will return a value of the `gnutls_sign_algorithm_t` enumeration that is the signature algorithm that has been used to sign this certificate request.

Since 3.6.0 this function never returns a negative error code. Error cases and unknown/unsupported signature algorithms are mapped to `GNUTLS_SIGN_UNKNOWN` .

Returns: a `gnutls_sign_algorithm_t` value

Since: 3.4.0

gnutls_x509_crq_get_signature_oid

`int gnutls_x509_crq_get_signature_oid` (*gnutls_x509_crq_t* `crq`, [Function]
*char ** `oid`, *size_t ** `oid_size`)

crq: should contain a `gnutls_x509_crq_t` type

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will return the OID of the signature algorithm that has been used to sign this certificate request. This function is useful in the case `gnutls_x509_crq_get_signature_algorithm()` returned `GNUTLS_SIGN_UNKNOWN` .

Returns: zero or a negative error code on error.

Since: 3.5.0

gnutls_x509_crq_get_spki

`int gnutls_x509_crq_get_spki` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will return the public key information of a PKCS10 certificate request. The provided `spki` must be initialized.

Returns: Zero on success, or a negative error code on error.

gnutls_x509_crq_get_subject_alt_name

`int gnutls_x509_crq_get_subject_alt_name` (*gnutls_x509_crq_t* `crq`, *unsigned int* `seq`, *void * * `ret`, *size_t * * `ret_size`, *unsigned int * * `ret_type`, *unsigned int * * `critical`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

seq: specifies the sequence number of the alt name, 0 for the first one, 1 for the second etc.

ret: is the place where the alternative name will be copied to

ret_size: holds the size of ret.

ret_type: holds the `gnutls_x509_subject_alt_name_t` name type

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_crq_get_subject_alt_name()` except for the fact that it will return the type of the alternative name in `ret_type` even if the function fails for some reason (i.e. the buffer provided is not enough).

Returns: the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ret_size` is not large enough to hold the value. In that case `ret_size` will be updated with the required size. If the certificate request does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 2.8.0

gnutls_x509_crq_get_subject_alt_othername_oid

`int gnutls_x509_crq_get_subject_alt_othername_oid` (*gnutls_x509_crq_t* `crq`, *unsigned int* `seq`, *void * * `ret`, *size_t * * `ret_size`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

ret: is the place where the otherName OID will be copied to

ret_size: holds the size of ret.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

This function is only useful if `gnutls_x509_crq_get_subject_alt_name()` returned `GNUTLS_SAN_OTHERNAME` .

Returns: the alternative subject name type on success, one of the enumerated gnutls_x509_subject_alt_name_t. For supported OIDs, it will return one of the virtual (GNUTLS_SAN_OTHERNAME_*) types, e.g. `GNUTLS_SAN_OTHERNAME_XMPP` , and `GNUTLS_SAN_OTHERNAME` for unknown OIDs. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ret_size` is not large enough to hold the value. In that case `ret_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 2.8.0

gnutls_x509_crq_get_tlsfeatures

`int gnutls_x509_crq_get_tlsfeatures` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_tlsfeatures_t* `features`, *unsigned int* `flags`, *unsigned int ** `critical`) [Function]

crq: An X.509 certificate request

features: If the function succeeds, the features will be stored in this variable.

flags: zero or `GNUTLS_EXT_FLAG_APPEND`

critical: the extension status

This function will get the X.509 TLS features extension structure from the certificate request. The returned structure needs to be freed using `gnutls_x509_tlsfeatures_deinit()` .

When the `flags` is set to `GNUTLS_EXT_FLAG_APPEND` , then if the `features` structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the `features` structure then they will be merged with.

Note that `features` must be initialized prior to calling this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_crq_get_version

`int gnutls_x509_crq_get_version` (*gnutls_x509_crq_t* `crq`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

This function will return the version of the specified Certificate request.

Returns: version of certificate request, or a negative error code on error.

gnutls_x509_crq_import

`int gnutls_x509_crq_import` (*gnutls_x509_crq_t* `crq`, *const gnutls_datum_t ** `data`, *gnutls_x509_crt_fmt_t* `format`) [Function]

crq: The data to store the parsed certificate request.

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will convert the given DER or PEM encoded certificate request to a `gnutls_x509_crq_t` type. The output will be stored in `crq` .

If the Certificate is PEM encoded it should have a header of "NEW CERTIFICATE REQUEST".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_init

`int gnutls_x509_crq_init` (*gnutls_x509_crq_t * crq*) [Function]

crq: A pointer to the type to be initialized

This function will initialize a PKCS10 certificate request structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_print

`int gnutls_x509_crq_print` (*gnutls_x509_crq_t crq, gnutls_certificate_print_formats_t format, gnutls_datum_t * out*) [Function]

crq: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print a certificate request, suitable for display to a human.

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_set_attribute_by_oid

`int gnutls_x509_crq_set_attribute_by_oid` (*gnutls_x509_crq_t crq, const char * oid, void * buf, size_t buf_size*) [Function]

crq: should contain a `gnutls_x509_crq_t` type

oid: holds an Object Identifier in a null-terminated string

buf: a pointer to a structure that holds the attribute data

buf_size: holds the size of `buf`

This function will set the attribute in the certificate request specified by the given Object ID. The provided attribute must be be DER encoded.

Attributes in a certificate request is an optional set of data appended to the request. Their interpretation depends on the CA policy.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_basic_constraints

`int gnutls_x509_crq_set_basic_constraints` (*gnutls_x509_crq_t* `crq`, *unsigned int* `ca`, *int* `pathLenConstraint`) [Function]

crq: a certificate request of type `gnutls_x509_crq_t`

ca: true(1) or false(0) depending on the Certificate authority status.

pathLenConstraint: non-negative error codes indicate maximum length of path, and negative error codes indicate that the pathLenConstraints field should not be present.

This function will set the basicConstraints certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_set_challenge_password

`int gnutls_x509_crq_set_challenge_password` (*gnutls_x509_crq_t* `crq`, *const char ** `pass`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

pass: holds a (0)-terminated password

This function will set a challenge password to be used when revoking the request.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_dn

`int gnutls_x509_crq_set_dn` (*gnutls_x509_crq_t* `crq`, *const char ** `dn`, *const char *** `err`) [Function]

crq: a certificate of type `gnutls_x509_crq_t`

dn: a comma separated DN string (RFC4514)

err: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded. On DN parsing error `GNUTLS_E_PARSING_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_dn_by_oid

`int gnutls_x509_crq_set_dn_by_oid` (*gnutls_x509_crq_t* `crq`, *const char ** `oid`, *unsigned int* `raw_flag`, *const void ** `data`, *unsigned int* `sizeof_data`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

oid: holds an Object Identifier in a (0)-terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

data: a pointer to the input data

sizeof_data: holds the size of `data`

This function will set the part of the name of the Certificate request subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()` . For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_extension_by_oid

`int gnutls_x509_crq_set_extension_by_oid` (*gnutls_x509_crq_t* `crq`, *const char * oid*, *const void * buf*, *size_t sizeof_buf*, *unsigned int critical*) [Function]

crq: a certificate of type `gnutls_x509_crq_t`

oid: holds an Object Identifier in null terminated string

buf: a pointer to a DER encoded data

sizeof_buf: holds the size of `buf`

critical: should be non-zero if the extension is to be marked as critical

This function will set an the extension, by the specified OID, in the certificate request. The extension data should be binary data DER encoded.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_key

`int gnutls_x509_crq_set_key` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_privkey_t key*) [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

This function will set the public parameters from the given private key to the request.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_key_purpose_oid

`int gnutls_x509_crq_set_key_purpose_oid` (*gnutls_x509_crq_t* `crq`, *const void * oid*, *unsigned int critical*) [Function]

crq: a certificate of type `gnutls_x509_crq_t`

oid: a pointer to a null-terminated string that holds the OID

critical: Whether this extension will be critical or not

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.

Subsequent calls to this function will append OIDs to the OID list.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_set_key_rsa_raw

`int gnutls_x509_crq_set_key_rsa_raw` (*gnutls_x509_crq_t* `crq`, *const gnutls_datum_t ** `m`, *const gnutls_datum_t ** `e`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

m: holds the modulus

e: holds the public exponent

This function will set the public parameters from the given private key to the request. Only RSA keys are currently supported.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.6.0

gnutls_x509_crq_set_key_usage

`int gnutls_x509_crq_set_key_usage` (*gnutls_x509_crq_t* `crq`, *unsigned int* `usage`) [Function]

crq: a certificate request of type `gnutls_x509_crq_t`

usage: an ORed sequence of the GNUTLS_KEY_* elements.

This function will set the keyUsage certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_set_private_key_usage_period

`int gnutls_x509_crq_set_private_key_usage_period` (*gnutls_x509_crq_t* `crq`, *time_t* `activation`, *time_t* `expiration`) [Function]

crq: a certificate of type `gnutls_x509_crq_t`

activation: The activation time

expiration: The expiration time

This function will set the private key usage period extension (2.5.29.16).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_set_spki

`int gnutls_x509_crq_set_spki` (*gnutls_x509_crq_t* `crq`, *const gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

crq: a certificate request of type `gnutls_x509_crq_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will set the certificate request's subject public key information explicitly. This is intended to be used in the cases where a single public key (e.g., RSA) can be used for multiple signature algorithms (RSA PKCS1-1.5, and RSA-PSS).

To export the public key (i.e., the SubjectPublicKeyInfo part), check `gnutls_pubkey_import_x509()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_x509_crq_set_subject_alt_name

`int gnutls_x509_crq_set_subject_alt_name` (*gnutls_x509_crq_t* `crq`, *gnutls_x509_subject_alt_name_t* `nt`, *const void *` `data`, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

crq: a certificate request of type `gnutls_x509_crq_t`

nt: is one of the `gnutls_x509_subject_alt_name_t` enumerations

data: The data to be set

data_size: The size of data to be set

flags: `GNUTLS_FSAN_SET` to clear previous data or `GNUTLS_FSAN_APPEND` to append.

This function will set the subject alternative name certificate extension. It can set the following types:

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crq_set_subject_alt_othername

`int gnutls_x509_crq_set_subject_alt_othername` (*gnutls_x509_crq_t* `crq`, *const char *` `oid`, *const void *` `data`, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

crq: a certificate request of type `gnutls_x509_crq_t`

oid: is the othername OID

data: The data to be set

data_size: The size of data to be set

flags: `GNUTLS_FSAN_SET` to clear previous data or `GNUTLS_FSAN_APPEND` to append.

This function will set the subject alternative name certificate extension. It can set the following types:

The values set must be binary values and must be properly DER encoded.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_x509_crq_set_tlsfeatures

`int gnutls_x509_crq_set_tlsfeatures (`*gnutls_x509_crq_t* `crq`*, gnutls_x509_tlsfeatures_t* `features`*)* [Function]

crq: An X.509 certificate request

features: If the function succeeds, the features will be added to the certificate request.

This function will set the certificate request's X.509 TLS extension from the given structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_crq_set_version

`int gnutls_x509_crq_set_version (`*gnutls_x509_crq_t* `crq`*, unsigned int* `version`*)* [Function]

crq: should contain a `gnutls_x509_crq_t` type

version: holds the version number, for v1 Requests must be 1

This function will set the version of the certificate request. For version 1 requests this must be one.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_sign

`int gnutls_x509_crq_sign (`*gnutls_x509_crq_t* `crq`*, gnutls_x509_privkey_t* `key`*)* [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

This function is the same a `gnutls_x509_crq_sign2()` with no flags, and an appropriate hash algorithm. The hash algorithm used may vary between versions of GnuTLS, and it is tied to the security level of the issuer's public key.

A known limitation of this function is, that a newly-signed request will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crq_sign2

`int gnutls_x509_crq_sign2 (`*gnutls_x509_crq_t* `crq`*, gnutls_x509_privkey_t* `key`*, gnutls_digest_algorithm_t* `dig`*, unsigned int* `flags`*)* [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

dig: The message digest to use, i.e., `GNUTLS_DIG_SHA256`

flags: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in `gnutls_x509_crt_set_key()` since a certificate request is self signed.

This must be the last step in a certificate request generation since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed request will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code. `GNUTLS_E_ASN1_VALUE_NOT_FOUND` is returned if you didn't set all information in the certificate request (e.g., the version using `gnutls_x509_crq_set_version()`).

gnutls_x509_crq_verify

`int gnutls_x509_crq_verify` (*gnutls_x509_crq_t* `crq`, *unsigned int* `flags`) [Function]

crq: is the crq to be verified

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

This function will verify self signature in the certificate request and return its status.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

Since 2.12.0

gnutls_x509_crt_check_email

`unsigned gnutls_x509_crt_check_email` (*gnutls_x509_crt_t* `cert`, *const char * * `email`, *unsigned int* `flags`) [Function]

cert: should contain an gnutls_x509_crt_t type

email: A null terminated string that contains an email address (RFC822)

flags: should be zero

This function will check if the given certificate's subject matches the given email address.

Returns: non-zero for a successful match, and zero on failure.

gnutls_x509_crt_check_hostname

`unsigned gnutls_x509_crt_check_hostname` (*gnutls_x509_crt_t* `cert`, *const char * * `hostname`) [Function]

cert: should contain an gnutls_x509_crt_t type

hostname: A null terminated string that contains a DNS name

This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC6125, and takes into

account wildcards, and the DNSName/IPAddress subject alternative name PKIX extension.

For details see also `gnutls_x509_crt_check_hostname2()` .

Returns: non-zero for a successful match, and zero on failure.

gnutls_x509_crt_check_hostname2

`unsigned gnutls_x509_crt_check_hostname2` (*gnutls_x509_crt_t* `cert`, *const char * hostname*, *unsigned int* `flags`) [Function]

cert: should contain an gnutls_x509_crt_t type

hostname: A null terminated string that contains a DNS name

flags: gnutls_certificate_verify_flags

This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC6125, and takes into account wildcards, and the DNSName/IPAddress subject alternative name PKIX extension.

IPv4 addresses are accepted by this function in the dotted-decimal format (e.g, ddd.ddd.ddd.ddd), and IPv6 addresses in the hexadecimal x:x:x:x:x:x:x:x format. For them the IPAddress subject alternative name extension is consulted. Previous versions to 3.6.0 of GnuTLS in case of a non-match would consult (in a non-standard extension) the DNSname and CN fields. This is no longer the case.

When the flag `GNUTLS_VERIFY_DO_NOT_ALLOW_WILDCARDS` is specified no wildcards are considered. Otherwise they are only considered if the domain name consists of three components or more, and the wildcard starts at the leftmost position. When the flag `GNUTLS_VERIFY_DO_NOT_ALLOW_IP_MATCHES` is specified, the input will be treated as a DNS name, and matching of textual IP addresses against the IPAddress part of the alternative name will not be allowed.

The function `gnutls_x509_crt_check_ip()` is available for matching IP addresses.

Returns: non-zero for a successful match, and zero on failure.

Since: 3.3.0

gnutls_x509_crt_check_ip

`unsigned gnutls_x509_crt_check_ip` (*gnutls_x509_crt_t* `cert`, *const unsigned char * ip*, *unsigned int* `ip_size`, *unsigned int* `flags`) [Function]

cert: should contain an gnutls_x509_crt_t type

ip: A pointer to the raw IP address

ip_size: the number of bytes in ip (4 or 16)

flags: should be zero

This function will check if the IP allowed IP addresses in the certificate's subject alternative name match the provided IP address.

Returns: non-zero for a successful match, and zero on failure.

gnutls_x509_crt_check_issuer

`unsigned gnutls_x509_crt_check_issuer` (*gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t* `issuer`) [Function]

cert: is the certificate to be checked

issuer: is the certificate of a possible issuer

This function will check if the given certificate was issued by the given issuer. It checks the DN fields and the authority key identifier and subject key identifier fields match.

If the same certificate is provided at the `cert` and `issuer` fields, it will check whether the certificate is self-signed.

Returns: It will return true (1) if the given certificate is issued by the given issuer, and false (0) if not.

gnutls_x509_crt_check_key_purpose

`unsigned gnutls_x509_crt_check_key_purpose` (*gnutls_x509_crt_t* `cert`, *const char * purpose*, *unsigned* `flags`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

purpose: a key purpose OID (e.g., `GNUTLS_KP_CODE_SIGNING`)

flags: zero or `GNUTLS_KP_FLAG_DISALLOW_ANY`

This function will check whether the given certificate matches the provided key purpose. If `flags` contains `GNUTLS_KP_FLAG_ALLOW_ANY` then it a certificate marked for any purpose will not match.

Returns: zero if the key purpose doesn't match, and non-zero otherwise.

Since: 3.5.6

gnutls_x509_crt_check_revocation

`int gnutls_x509_crt_check_revocation` (*gnutls_x509_crt_t* `cert`, *const gnutls_x509_crl_t * crl_list*, *unsigned* `crl_list_length`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

crl_list: should contain a list of gnutls_x509_crl_t types

crl_list_length: the length of the crl_list

This function will check if the given certificate is revoked. It is assumed that the CRLs have been verified before.

Returns: 0 if the certificate is NOT revoked, and 1 if it is. A negative error code is returned on error.

gnutls_x509_crt_cpy_crl_dist_points

`int gnutls_x509_crt_cpy_crl_dist_points` (*gnutls_x509_crt_t* `dst`, *gnutls_x509_crt_t* `src`) [Function]

dst: a certificate of type `gnutls_x509_crt_t`

src: the certificate where the dist points will be copied from

This function will copy the CRL distribution points certificate extension, from the source to the destination certificate. This may be useful to copy from a CA certificate to issued ones.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_deinit

`void gnutls_x509_crt_deinit (`*gnutls_x509_crt_t* `cert``)` [Function]

cert: The data to be deinitialized

This function will deinitialize a certificate structure.

gnutls_x509_crt_equals

`unsigned gnutls_x509_crt_equals (`*gnutls_x509_crt_t* `cert1`, *gnutls_x509_crt_t* `cert2``)` [Function]

cert1: The first certificate

cert2: The second certificate

This function will compare two X.509 certificate structures.

Returns: On equality non-zero is returned, otherwise zero.

Since: 3.5.0

gnutls_x509_crt_equals2

`unsigned gnutls_x509_crt_equals2 (`*gnutls_x509_crt_t* `cert1`, *const gnutls_datum_t ** `der``)` [Function]

cert1: The first certificate

der: A DER encoded certificate

This function will compare an X.509 certificate structures, with DER encoded certificate data.

Returns: On equality non-zero is returned, otherwise zero.

Since: 3.5.0

gnutls_x509_crt_export

`int gnutls_x509_crt_export (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_fmt_t* `format`, *void ** `output_data`, *size_t ** `output_data_size`)` [Function]

cert: Holds the certificate

format: the format of output params. One of PEM or DER.

output_data: will contain a certificate PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the certificate to DER or PEM format.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_x509_crt_export2

`int gnutls_x509_crt_export2 (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t* * `out`)` [Function]

cert: Holds the certificate

format: the format of output params. One of PEM or DER.

out: will contain a certificate PEM or DER encoded

This function will export the certificate to DER or PEM format. The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 3.1.3

gnutls_x509_crt_get_activation_time

`time_t gnutls_x509_crt_get_activation_time (`*gnutls_x509_crt_t* `cert`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

This function will return the time this Certificate was or will be activated.

Returns: activation time, or (time_t)-1 on error.

gnutls_x509_crt_get_authority_info_access

`int gnutls_x509_crt_get_authority_info_access (`*gnutls_x509_crt_t* `crt`, *unsigned int* `seq`, *int* `what`, *gnutls_datum_t* * `data`, *unsigned int* * `critical`)` [Function]

crt: Holds the certificate

seq: specifies the sequence number of the access descriptor (0 for the first one, 1 for the second etc.)

what: what data to get, a `gnutls_info_access_what_t` type.

data: output data to be freed with `gnutls_free()` .

critical: pointer to output integer that is set to non-zero if the extension is marked as critical (may be `NULL`)

Note that a simpler API to access the authority info data is provided by `gnutls_x509_aia_get()` and `gnutls_x509_ext_import_aia()` .

This function extracts the Authority Information Access (AIA) extension, see RFC 5280 section 4.2.2.1 for more information. The AIA extension holds a sequence of AccessDescription (AD) data.

The `seq` input parameter is used to indicate which member of the sequence the caller is interested in. The first member is 0, the second member 1 and so on. When the `seq` value is out of bounds, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

The type of data returned in `data` is specified via `what` which should be `gnutls_info_access_what_t` values.

If `what` is `GNUTLS_IA_ACCESSMETHOD_OID` then `data` will hold the accessMethod OID (e.g., `"1.3.6.1.5.5.7.48.1"`).

If `what` is `GNUTLS_IA_ACCESSLOCATION_GENERALNAME_TYPE` , `data` will hold the accessLocation GeneralName type (e.g., "uniformResourceIdentifier").

If `what` is `GNUTLS_IA_URI` , `data` will hold the accessLocation URI data. Requesting this `what` value leads to an error if the accessLocation is not of the "uniformResourceIdentifier" type.

If `what` is `GNUTLS_IA_OCSP_URI` , `data` will hold the OCSP URI. Requesting this `what` value leads to an error if the accessMethod is not 1.3.6.1.5.5.7.48.1 aka OCSP, or if accessLocation is not of the "uniformResourceIdentifier" type. In that case `GNUTLS_E_UNKNOWN_ALGORITHM` will be returned, and `seq` should be increased and this function called again.

If `what` is `GNUTLS_IA_CAISSUERS_URI` , `data` will hold the caIssuers URI. Requesting this `what` value leads to an error if the accessMethod is not 1.3.6.1.5.5.7.48.2 aka caIssuers, or if accessLocation is not of the "uniformResourceIdentifier" type. In that case handle as in `GNUTLS_IA_OCSP_URI` .

More `what` values may be allocated in the future as needed.

If `data` is NULL, the function does the same without storing the output data, that is, it will set `critical` and do error checking as usual.

The value of the critical flag is returned in * `critical` . Supply a NULL `critical` if you want the function to make sure the extension is non-critical, as required by RFC 5280.

Returns: `GNUTLS_E_SUCCESS` on success, `GNUTLS_E_INVALID_REQUEST` on invalid `crt` , `GNUTLS_E_CONSTRAINT_ERROR` if the extension is incorrectly marked as critical (use a non-NULL `critical` to override), `GNUTLS_E_UNKNOWN_ALGORITHM` if the requested OID does not match (e.g., when using `GNUTLS_IA_OCSP_URI`), otherwise a negative error code.

Since: 3.0

gnutls_x509_crt_get_authority_key_gn_serial

`int gnutls_x509_crt_get_authority_key_gn_serial` [Function]
(*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void *` `alt`, *size_t *` `alt_size`, *unsigned int *` `alt_type`, *void *` `serial`, *size_t *` `serial_size`, *unsigned int *` `critical`)

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

alt: is the place where the alternative name will be copied to

alt_size: holds the size of alt.

alt_type: holds the type of the alternative name (one of gnutls_x509_subject_alt_name_t).

serial: buffer to store the serial number (may be null)

serial_size: Holds the size of the serial field (may be null)

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509 authority key identifier when stored as a general name (authorityCertIssuer) and serial number.

Because more than one general names might be stored `seq` can be used as a counter to request them all until `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.0

gnutls_x509_crt_get_authority_key_id

`int gnutls_x509_crt_get_authority_key_id` (*gnutls_x509_crt_t* `cert`, *void ** `id`, *size_t *** `id_size`, *unsigned int *** `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

id: The place where the identifier will be copied

id_size: Holds the size of the id field.

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509v3 certificate authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension and `GNUTLS_E_X509_UNSUPPORTED_EXTENSION` , if the extension contains the name and serial number of the certificate. In that case `gnutls_x509_crt_get_authority_key_gn_serial()` may be used.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

gnutls_x509_crt_get_basic_constraints

`int gnutls_x509_crt_get_basic_constraints` (*gnutls_x509_crt_t* `cert`, *unsigned int *** `critical`, *unsigned int *** `ca`, *int *** `pathlen`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

critical: will be non-zero if the extension is marked as critical

ca: pointer to output integer indicating CA status, may be NULL, value is 1 if the certificate CA flag is set, 0 otherwise.

pathlen: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pathLenConstraint field and the actual value, -1 indicate that the field is absent.

This function will read the certificate's basic constraints, and return the certificates CA status. It reads the basicConstraints X.509 extension (2.5.29.19).

Returns: If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the basicConstraints extension GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

gnutls_x509_crt_get_ca_status

`int gnutls_x509_crt_get_ca_status (`*gnutls_x509_crt_t* `cert`, *unsigned int ** `critical``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

critical: will be non-zero if the extension is marked as critical

This function will return certificates CA status, by reading the basicConstraints X.509 extension (2.5.29.19). If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set.

Use `gnutls_x509_crt_get_basic_constraints()` if you want to read the pathLenConstraint field too.

Returns: If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the basicConstraints extension GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

gnutls_x509_crt_get_crl_dist_points

`int gnutls_x509_crt_get_crl_dist_points (`*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void ** `san`, *size_t ** `san_size`, *unsigned int ** `reason_flags`, *unsigned int ** `critical``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)

san: is the place where the distribution point will be copied to

san_size: holds the size of ret.

reason_flags: Revocation reasons. An ORed sequence of flags from `gnutls_x509_crl_reason_flags_t` .

critical: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the CRL distribution points (2.5.29.31), contained in the given certificate in the X509v3 Certificate Extensions.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` and updates `ret_size` if `ret_size` is not enough to hold the distribution point, or the type of the distribution point if everything was ok. The type is one of the enumerated `gnutls_x509_subject_alt_name_t` . If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

gnutls_x509_crt_get_dn

`int gnutls_x509_crt_get_dn (`*gnutls_x509_crt_t* `cert`, *char ** `buf`, *size_t ** `buf_size`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

buf: a pointer to a structure to hold the name (may be null)

buf_size: initially holds the size of `buf`

This function will copy the name of the Certificate in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If `buf` is null then only the size will be filled.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crt_get_dn3()` .

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the DN does not exist, or another error value on error. On success 0 is returned.

gnutls_x509_crt_get_dn2

`int gnutls_x509_crt_get_dn2` (*gnutls_x509_crt_t* `cert`, *gnutls_datum_t * dn*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crt_get_dn3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.10

gnutls_x509_crt_get_dn3

`int gnutls_x509_crt_get_dn3` (*gnutls_x509_crt_t* `cert`, *gnutls_datum_t * dn, unsigned flags*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: a pointer to a structure to hold the name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will allocate buffer and copy the name of the Certificate. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.7

gnutls_x509_crt_get_dn_by_oid

`int gnutls_x509_crt_get_dn_by_oid` (*gnutls_x509_crt_t* `cert`, *const char * oid*, *unsigned* `indx`, *unsigned int* `raw_flag`, *void * buf*, *size_t * buf_size*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: holds an Object Identified in null terminated string

indx: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

raw_flag: If non-zero returns the raw DER data of the DN part.

buf: a pointer where the DN part will be copied (may be null).

buf_size: initially holds the size of `buf`

This function will extract the part of the name of the Certificate subject specified by the given OID. The output, if the raw flag is not used, will be encoded as described in RFC4514. Thus a string that is ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()` .

If `buf` is null then only the size will be filled. If the `raw_flag` is not specified the output is always null terminated, although the `buf_size` will not include the null character.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

gnutls_x509_crt_get_dn_oid

`int gnutls_x509_crt_get_dn_oid` (*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void * oid*, *size_t * oid_size*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: This specifies which OID to return. Use (0) to get the first one.

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will extract the OIDs of the name of the Certificate subject specified by the given index.

If `oid` is null then only the size will be filled. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

gnutls_x509_crt_get_expiration_time

`time_t gnutls_x509_crt_get_expiration_time` (*gnutls_x509_crt_t* `cert`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

This function will return the time this certificate was or will be expired.

Returns: expiration time, or (time_t)-1 on error.

gnutls_x509_crt_get_extension_by_oid

`int gnutls_x509_crt_get_extension_by_oid` (*gnutls_x509_crt_t* `cert`, *const char * oid*, *unsigned* `indx`, *void * buf*, *size_t * buf_size*, *unsigned int * critical*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: holds an Object Identified in null terminated string

indx: In case multiple same OIDs exist in the extensions, this specifies which to send. Use (0) to get the first one.

buf: a pointer to a structure to hold the name (may be null)

buf_size: initially holds the size of `buf`

critical: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If the certificate does not contain the specified extension GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

gnutls_x509_crt_get_extension_by_oid2

`int gnutls_x509_crt_get_extension_by_oid2` (*gnutls_x509_crt_t* `cert`, *const char * oid*, *unsigned* `indx`, *gnutls_datum_t * output*, *unsigned int * critical*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: holds an Object Identified in null terminated string

indx: In case multiple same OIDs exist in the extensions, this specifies which to send. Use (0) to get the first one.

output: will hold the allocated extension data

critical: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If the certificate does not contain the specified extension GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

Since: 3.3.8

gnutls_x509_crt_get_extension_data

`int gnutls_x509_crt_get_extension_data (`*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void * *`data`, *size_t * *`sizeof_data`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: Specifies which extension OID to send. Use (0) to get the first one.

data: a pointer to a structure to hold the data (may be null)

sizeof_data: initially holds the size of `data`

This function will return the requested extension data in the certificate. The extension data will be stored in the provided buffer.

Use `gnutls_x509_crt_get_extension_info()` to extract the OID and critical flag. Use `gnutls_x509_crt_get_extension_by_oid()` instead, if you want to get data indexed by the extension OID rather than sequence.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crt_get_extension_data2

`int gnutls_x509_crt_get_extension_data2 (`*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *gnutls_datum_t * *`data`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: Specifies which extension OID to read. Use (0) to get the first one.

data: will contain the extension DER-encoded data

This function will return the requested by the index extension data in the certificate. The extension data will be allocated using `gnutls_malloc()` .

Use `gnutls_x509_crt_get_extension_info()` to extract the OID.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crt_get_extension_info

`int gnutls_x509_crt_get_extension_info (`*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void * *`oid`, *size_t * *`oid_size`, *unsigned int * *`critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: Specifies which extension OID to send. Use (0) to get the first one.

oid: a pointer to a structure to hold the OID

oid_size: initially holds the maximum size of `oid` , on return holds actual size of `oid` .

critical: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the certificate, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use `gnutls_x509_crt_get_extension()` to extract the data.

If the buffer provided is not long enough to hold the output, then `oid_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null (the latter is not true for GnuTLS prior to 3.6.0).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crt_get_extension_oid

`int gnutls_x509_crt_get_extension_oid` (*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void * * `oid`, *size_t * * `oid_size`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: Specifies which extension OID to send. Use (0) to get the first one.

oid: a pointer to a structure to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will return the requested extension OID in the certificate. The extension OID will be stored as a string in the provided buffer.

The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crt_get_fingerprint

`int gnutls_x509_crt_get_fingerprint` (*gnutls_x509_crt_t* `cert`, *gnutls_digest_algorithm_t* `algo`, *void * * `buf`, *size_t * * `buf_size`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

algo: is a digest algorithm

buf: a pointer to a structure to hold the fingerprint (may be null)

buf_size: initially holds the size of `buf`

This function will calculate and copy the certificate's fingerprint in the provided buffer. The fingerprint is a hash of the DER-encoded data of the certificate.

If the buffer is null then only the size will be filled.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *buf_size will be updated with the required size. On success 0 is returned.

gnutls_x509_crt_get_inhibit_anypolicy

`int gnutls_x509_crt_get_inhibit_anypolicy` (*gnutls_x509_crt_t* `cert`, *unsigned int * * `skipcerts`, *unsigned int * * `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

skipcerts: will hold the number of certificates after which anypolicy is no longer acceptable.

critical: will be non-zero if the extension is marked as critical

This function will return certificate's value of the SkipCerts, i.e., the Inhibit anyPolicy X.509 extension (2.5.29.54).

The returned value is the number of additional certificates that may appear in the path before the anyPolicy is no longer acceptable.

Returns: zero on success, or a negative error code in case of parsing error. If the certificate does not contain the Inhibit anyPolicy extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 3.6.0

gnutls_x509_crt_get_issuer

`int gnutls_x509_crt_get_issuer (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_dn_t ** `dn``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: output variable with pointer to uint8_t DN

Return the Certificate's Issuer DN as a `gnutls_x509_dn_t` data type, that can be decoded using `gnutls_x509_dn_get_rdn_ava()` .

Note that `dn` should be treated as constant. Because it points into the `cert` object, you should not use `dn` after `cert` is deallocated.

Returns: Returns 0 on success, or an error code.

gnutls_x509_crt_get_issuer_alt_name

`int gnutls_x509_crt_get_issuer_alt_name (`*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void ** `ian`, *size_t ** `ian_size`, *unsigned int ** `critical``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

ian: is the place where the alternative name will be copied to

ian_size: holds the size of ian.

critical: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the Issuer Alternative Name (2.5.29.18), contained in the given certificate in the X509v3 Certificate Extensions.

When the SAN type is otherName, it will extract the data in the otherName's value field, and `GNUTLS_SAN_OTHERNAME` is returned. You may use `gnutls_x509_crt_get_subject_alt_othername_oid()` to get the corresponding OID and the "virtual" SAN types (e.g., `GNUTLS_SAN_OTHERNAME_XMPP`).

If an otherName OID is known, the data will be decoded. Otherwise the returned data will be DER encoded, and you will have to decode it yourself. Currently, only the RFC 3920 id-on-xmppAddr Issuer AltName is recognized.

Returns: the alternative issuer name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ian_size` is not large enough to hold the value. In that case `ian_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 2.10.0

gnutls_x509_crt_get_issuer_alt_name2

`int gnutls_x509_crt_get_issuer_alt_name2` (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * * `ian`, *size_t * `ian_size`, *unsigned int * `ian_type`, *unsigned int * `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

ian: is the place where the alternative name will be copied to

ian_size: holds the size of ret.

ian_type: holds the type of the alternative name (one of gnutls_x509_subject_alt_name_t).

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_crt_get_issuer_alt_name()` except for the fact that it will return the type of the alternative name in `ian_type` even if the function fails for some reason (i.e. the buffer provided is not enough).

Returns: the alternative issuer name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ian_size` is not large enough to hold the value. In that case `ian_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 2.10.0

gnutls_x509_crt_get_issuer_alt_othername_oid

`int gnutls_x509_crt_get_issuer_alt_othername_oid` (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * `ret`, *size_t * `ret_size`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

ret: is the place where the otherName OID will be copied to

ret_size: holds the size of ret.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

If `oid` is null then only the size will be filled. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

This function is only useful if `gnutls_x509_crt_get_issuer_alt_name()` returned `GNUTLS_SAN_OTHERNAME` .

Returns: the alternative issuer name type on success, one of the enumerated gnutls_x509_subject_alt_name_t. For supported OIDs, it will return one of the virtual (GNUTLS_SAN_OTHERNAME_*) types, e.g. `GNUTLS_SAN_OTHERNAME_XMPP` , and `GNUTLS_SAN_OTHERNAME` for unknown OIDs. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ret_size` is not large enough to hold the value. In that case `ret_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Since: 2.10.0

gnutls_x509_crt_get_issuer_dn

`int gnutls_x509_crt_get_issuer_dn (`*gnutls_x509_crt_t* `cert`*, char * buf, size_t * buf_size*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

buf: a pointer to a structure to hold the name (may be null)

buf_size: initially holds the size of `buf`

This function will copy the name of the Certificate issuer in the provided buffer. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If `buf` is null then only the size will be filled.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crt_get_issuer_dn3()` .

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the DN does not exist, or another error value on error. On success 0 is returned.

gnutls_x509_crt_get_issuer_dn2

`int gnutls_x509_crt_get_issuer_dn2 (`*gnutls_x509_crt_t* `cert`*, gnutls_datum_t * dn*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of issuer of the Certificate. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_crt_get_issuer_dn3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.10

gnutls_x509_crt_get_issuer_dn3

`int gnutls_x509_crt_get_issuer_dn3 (`*gnutls_x509_crt_t* `cert`, *gnutls_datum_t * dn, unsigned* `flags``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: a pointer to a structure to hold the name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will allocate buffer and copy the name of issuer of the Certificate. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.7

gnutls_x509_crt_get_issuer_dn_by_oid

`int gnutls_x509_crt_get_issuer_dn_by_oid (`*gnutls_x509_crt_t* `cert`, *const char * oid, unsigned* `indx`, *unsigned int* `raw_flag`, *void * buf, size_t * buf_size*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: holds an Object Identified in null terminated string

indx: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

raw_flag: If non-zero returns the raw DER data of the DN part.

buf: a pointer to a structure to hold the name (may be null)

buf_size: initially holds the size of `buf`

This function will extract the part of the name of the Certificate issuer specified by the given OID. The output, if the raw flag is not used, will be encoded as described in RFC4514. Thus a string that is ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()` .

If `buf` is null then only the size will be filled. If the `raw_flag` is not specified the output is always null terminated, although the `buf_size` will not include the null character.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

gnutls_x509_crt_get_issuer_dn_oid

`int gnutls_x509_crt_get_issuer_dn_oid (`*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void ** `oid`, *size_t ** `oid_size``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: This specifies which OID to return. Use (0) to get the first one.

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will extract the OIDs of the name of the Certificate issuer specified by the given index.

If `oid` is null then only the size will be filled. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

gnutls_x509_crt_get_issuer_unique_id

`int gnutls_x509_crt_get_issuer_unique_id (`*gnutls_x509_crt_t* `crt`, *char ** `buf`, *size_t ** `buf_size`)` [Function]

crt: Holds the certificate

buf: user allocated memory buffer, will hold the unique id

buf_size: size of user allocated memory buffer (on input), will hold actual size of the unique ID on return.

This function will extract the issuerUniqueID value (if present) for the given certificate.

If the user allocated memory buffer is not large enough to hold the full subjectUniqueID, then a GNUTLS_E_SHORT_MEMORY_BUFFER error will be returned, and buf_size will be set to the actual length.

This function had a bug prior to 3.4.8 that prevented the setting of `NULL buf` to discover the `buf_size` . To use this function safely with the older versions the `buf` must be a valid buffer that can hold at least a single byte if `buf_size` is zero.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 2.12.0

gnutls_x509_crt_get_key_id

`int gnutls_x509_crt_get_key_id (`*gnutls_x509_crt_t* `crt`, *unsigned int* `flags`, *unsigned char ** `output_data`, *size_t ** `output_data_size`)` [Function]

crt: Holds the certificate

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_x509_crt_get_key_purpose_oid

`int gnutls_x509_crt_get_key_purpose_oid` (*gnutls_x509_crt_t* `cert`, *unsigned* `indx`, *void *` `oid`, *size_t *` `oid_size`, *unsigned int *` `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

indx: This specifies which OID to return. Use (0) to get the first one.

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

critical: output flag to indicate criticality of extension

This function will extract the key purpose OIDs of the Certificate specified by the given index. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.

If `oid` is null then only the size will be filled. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

Returns: `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *oid_size will be updated with the required size. On success 0 is returned.

gnutls_x509_crt_get_key_usage

`int gnutls_x509_crt_get_key_usage` (*gnutls_x509_crt_t* `cert`, *unsigned int *` `key_usage`, *unsigned int *` `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

key_usage: where the key usage bits will be stored

critical: will be non-zero if the extension is marked as critical

This function will return certificate's key usage, by reading the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: `GNUTLS_KEY_DIGITAL_SIGNATURE` , `GNUTLS_KEY_NON_REPUDIATION` , `GNUTLS_KEY_KEY_ENCIPHERMENT` , `GNUTLS_KEY_DATA_ENCIPHERMENT` , `GNUTLS_KEY_KEY_AGREEMENT` , `GNUTLS_KEY_KEY_CERT_SIGN` , `GNUTLS_KEY_CRL_SIGN` , `GNUTLS_KEY_ENCIPHER_ONLY` , `GNUTLS_KEY_DECIPHER_ONLY` .

Returns: zero on success, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_x509_crt_get_name_constraints

`int gnutls_x509_crt_get_name_constraints` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_name_constraints_t* `nc`, *unsigned int* `flags`, *unsigned int * critical*) [Function]

crt: should contain a `gnutls_x509_crt_t` type

nc: The nameconstraints intermediate type

flags: zero or `GNUTLS_EXT_FLAG_APPEND`

critical: the extension status

This function will return an intermediate type containing the name constraints of the provided CA certificate. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

When the `flags` is set to `GNUTLS_EXT_FLAG_APPEND` , then if the `nc` structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the `nc` structure then the constraints will be merged with the existing constraints following RFC5280 p6.1.4 (excluded constraints will be appended, permitted will be intersected).

Note that `nc` must be initialized prior to calling this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_crt_get_pk_algorithm

`int gnutls_x509_crt_get_pk_algorithm` (*gnutls_x509_crt_t* `cert`, *unsigned int * bits*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

bits: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of an X.509 certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

Unknown/unsupported algorithms are mapped to `GNUTLS_PK_UNKNOWN` .

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

gnutls_x509_crt_get_pk_dsa_raw

`int gnutls_x509_crt_get_pk_dsa_raw` (*gnutls_x509_crt_t* `crt`, *gnutls_datum_t * p*, *gnutls_datum_t * q*, *gnutls_datum_t * g*, *gnutls_datum_t * y*) [Function]

crt: Holds the certificate

p: will hold the p

q: will hold the q

g: will hold the g

y: will hold the y

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

gnutls_x509_crt_get_pk_ecc_raw

`int gnutls_x509_crt_get_pk_ecc_raw` (*gnutls_x509_crt_t* `crt`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`) [Function]

crt: Holds the certificate

curve: will hold the curve

x: will hold the x-coordinate

y: will hold the y-coordinate

This function will export the ECC public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.1

gnutls_x509_crt_get_pk_gost_raw

`int gnutls_x509_crt_get_pk_gost_raw` (*gnutls_x509_crt_t* `crt`, *gnutls_ecc_curve_t* * `curve`, *gnutls_digest_algorithm_t* * `digest`, *gnutls_gost_paramset_t* * `paramset`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`) [Function]

crt: Holds the certificate

curve: will hold the curve

digest: will hold the digest

paramset: will hold the GOST parameter set ID

x: will hold the x-coordinate

y: will hold the y-coordinate

This function will export the GOST public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.3

gnutls_x509_crt_get_pk_oid

`int gnutls_x509_crt_get_pk_oid (`*gnutls_x509_crt_t* `cert`, *char ** `oid`, *size_t ** `oid_size``)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will return the OID of the public key algorithm on that certificate. This is function is useful in the case `gnutls_x509_crt_get_pk_algorithm()` returned `GNUTLS_PK_UNKNOWN` .

Returns: zero or a negative error code on error.

Since: 3.5.0

gnutls_x509_crt_get_pk_rsa_raw

`int gnutls_x509_crt_get_pk_rsa_raw (`*gnutls_x509_crt_t* `crt`, *gnutls_datum_t ** `m`, *gnutls_datum_t ** `e`)` [Function]

crt: Holds the certificate

m: will hold the modulus

e: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

gnutls_x509_crt_get_policy

`int gnutls_x509_crt_get_policy (`*gnutls_x509_crt_t* `crt`, *unsigned* `indx`, *struct gnutls_x509_policy_st ** `policy`, *unsigned int ** `critical`)` [Function]

crt: should contain a `gnutls_x509_crt_t` type

indx: This specifies which policy to return. Use (0) to get the first one.

policy: A pointer to a policy structure.

critical: will be non-zero if the extension is marked as critical

This function will extract the certificate policy (extension 2.5.29.32) specified by the given index.

The policy returned by this function must be deinitialized by using `gnutls_x509_policy_release()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.1.5

gnutls_x509_crt_get_private_key_usage_period

`int gnutls_x509_crt_get_private_key_usage_period` [Function]
(*gnutls_x509_crt_t* `cert`, *time_t ** `activation`, *time_t ** `expiration`, *unsigned int ** `critical`)

cert: should contain a `gnutls_x509_crt_t` type

activation: The activation time

expiration: The expiration time

critical: the extension status

This function will return the expiration and activation times of the private key of the certificate. It relies on the PKIX extension 2.5.29.16 being present.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

gnutls_x509_crt_get_proxy

`int gnutls_x509_crt_get_proxy` (*gnutls_x509_crt_t* `cert`, *unsigned int ** `critical`, *int ** `pathlen`, *char *** `policyLanguage`, *char *** `policy`, *size_t ** `sizeof_policy`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

critical: will be non-zero if the extension is marked as critical

pathlen: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pCPathLenConstraint field and the actual value, -1 indicate that the field is absent.

policyLanguage: output variable with OID of policy language

policy: output variable with policy data

sizeof_policy: output variable size of policy data

This function will get information from a proxy certificate. It reads the ProxyCertInfo X.509 extension (1.3.6.1.5.5.7.1.14).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_x509_crt_get_raw_dn

`int gnutls_x509_crt_get_raw_dn` (*gnutls_x509_crt_t* `cert`, *gnutls_datum_t ** `dn`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length. This points to allocated data that must be free'd using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. or a negative error code on error.

gnutls_x509_crt_get_raw_issuer_dn

`int gnutls_x509_crt_get_raw_issuer_dn (`*gnutls_x509_crt_t* `cert`, *gnutls_datum_t * dn*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length. This points to allocated data that must be free'd using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.or a negative error code on error.

gnutls_x509_crt_get_serial

`int gnutls_x509_crt_get_serial (`*gnutls_x509_crt_t* `cert`, *void * result*, *size_t * result_size*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

result: The place where the serial number will be copied

result_size: Holds the size of the result field.

This function will return the X.509 certificate's serial number. This is obtained by the X509 Certificate serialNumber field. Serial is not always a 32 or 64bit number. Some CAs use large serial numbers, thus it may be wise to handle it as something uint8_t.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_get_signature

`int gnutls_x509_crt_get_signature (`*gnutls_x509_crt_t* `cert`, *char * sig*, *size_t * sig_size*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

sig: a pointer where the signature part will be copied (may be null).

sig_size: initially holds the size of `sig`

This function will extract the signature field of a certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_get_signature_algorithm

`int gnutls_x509_crt_get_signature_algorithm (`*gnutls_x509_crt_t* `cert`*)* [Function]

cert: should contain a `gnutls_x509_crt_t` type

This function will return a value of the `gnutls_sign_algorithm_t` enumeration that is the signature algorithm that has been used to sign this certificate.

Since 3.6.0 this function never returns a negative error code. Error cases and unknown/unsupported signature algorithms are mapped to `GNUTLS_SIGN_UNKNOWN` .

Returns: a `gnutls_sign_algorithm_t` value

gnutls_x509_crt_get_signature_oid

`int gnutls_x509_crt_get_signature_oid (`*gnutls_x509_crt_t* `cert`, *char * oid*, *size_t * oid_size*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

oid: a pointer to a buffer to hold the OID (may be null)

oid_size: initially holds the size of `oid`

This function will return the OID of the signature algorithm that has been used to sign this certificate. This is function is useful in the case `gnutls_x509_crt_get_signature_algorithm()` returned `GNUTLS_SIGN_UNKNOWN` .

Returns: zero or a negative error code on error.

Since: 3.5.0

gnutls_x509_crt_get_spki

`int gnutls_x509_crt_get_spki (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_spki_t* `spki`, *unsigned int* `flags`*)* [Function]

cert: a certificate of type `gnutls_x509_crt_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will return the public key information of an X.509 certificate. The provided `spki` must be initialized.

Since: 3.6.0

gnutls_x509_crt_get_subject

`int gnutls_x509_crt_get_subject (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_dn_t * dn*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

dn: output variable with pointer to uint8_t DN.

Return the Certificate's Subject DN as a `gnutls_x509_dn_t` data type, that can be decoded using `gnutls_x509_dn_get_rdn_ava()` .

Note that `dn` should be treated as constant. Because it points into the `cert` object, you should not use `dn` after `cert` is deallocated.

Returns: Returns 0 on success, or an error code.

gnutls_x509_crt_get_subject_alt_name

`int gnutls_x509_crt_get_subject_alt_name (`*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * san*, *size_t * san_size*, *unsigned int * critical*`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

san: is the place where the alternative name will be copied to

san_size: holds the size of san.

critical: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the Alternative Name (2.5.29.17), contained in the given certificate in the X509v3 Certificate Extensions.

When the SAN type is otherName, it will extract the data in the otherName's value field, and `GNUTLS_SAN_OTHERNAME` is returned. You may use `gnutls_x509_crt_get_subject_alt_othername_oid()` to get the corresponding OID and the "virtual" SAN types (e.g., `GNUTLS_SAN_OTHERNAME_XMPP`).

If an otherName OID is known, the data will be decoded. Otherwise the returned data will be DER encoded, and you will have to decode it yourself. Currently, only the RFC 3920 id-on-xmppAddr SAN is recognized.

Returns: the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `san_size` is not large enough to hold the value. In that case `san_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

gnutls_x509_crt_get_subject_alt_name2

`int gnutls_x509_crt_get_subject_alt_name2` (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * * `san`, *size_t * `san_size`, *unsigned int * `san_type`, *unsigned int * `critical`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

san: is the place where the alternative name will be copied to

san_size: holds the size of ret.

san_type: holds the type of the alternative name (one of gnutls_x509_subject_alt_name_t).

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_crt_get_subject_alt_name()` except for the fact that it will return the type of the alternative name in `san_type` even if the function fails for some reason (i.e. the buffer provided is not enough).

Returns: the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `san_size` is not large enough to hold the value. In that case `san_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

gnutls_x509_crt_get_subject_alt_othername_oid

`int gnutls_x509_crt_get_subject_alt_othername_oid` (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * `oid`, *size_t * `oid_size`) [Function]

cert: should contain a `gnutls_x509_crt_t` type

seq: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

oid: is the place where the otherName OID will be copied to

oid_size: holds the size of ret.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

This function is only useful if `gnutls_x509_crt_get_subject_alt_name()` returned `GNUTLS_SAN_OTHERNAME` .

If `oid` is null then only the size will be filled. The `oid` returned will be null terminated, although `oid_size` will not account for the trailing null.

Returns: the alternative subject name type on success, one of the enumerated gnutls_x509_subject_alt_name_t. For supported OIDs, it will return one of the virtual (GNUTLS_SAN_OTHERNAME_*) types, e.g. `GNUTLS_SAN_OTHERNAME_XMPP` , and `GNUTLS_SAN_OTHERNAME` for unknown OIDs. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ian_size` is not large enough to hold the value. In that case `ian_size` will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

gnutls_x509_crt_get_subject_key_id

`int gnutls_x509_crt_get_subject_key_id` (*gnutls_x509_crt_t* `cert`, *void * ret*, *size_t * ret_size*, *unsigned int * critical*) [Function]

cert: should contain a `gnutls_x509_crt_t` type

ret: The place where the identifier will be copied

ret_size: Holds the size of the result field.

critical: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509v3 certificate's subject key identifier. This is obtained by the X.509 Subject Key identifier extension field (2.5.29.14).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

gnutls_x509_crt_get_subject_unique_id

`int gnutls_x509_crt_get_subject_unique_id` (*gnutls_x509_crt_t* `crt`, *char * buf*, *size_t * buf_size*) [Function]

crt: Holds the certificate

buf: user allocated memory buffer, will hold the unique id

buf_size: size of user allocated memory buffer (on input), will hold actual size of the unique ID on return.

This function will extract the subjectUniqueID value (if present) for the given certificate.

If the user allocated memory buffer is not large enough to hold the full subjectUniqueID, then a GNUTLS_E_SHORT_MEMORY_BUFFER error will be returned, and buf_size will be set to the actual length.

This function had a bug prior to 3.4.8 that prevented the setting of `NULL buf` to discover the `buf_size` . To use this function safely with the older versions the `buf` must be a valid buffer that can hold at least a single byte if `buf_size` is zero.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

gnutls_x509_crt_get_tlsfeatures

`int gnutls_x509_crt_get_tlsfeatures (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_tlsfeatures_t* `features`, *unsigned int* `flags`, *unsigned int ** `critical``)` [Function]

crt: A X.509 certificate

features: If the function succeeds, the features will be stored in this variable.

flags: zero or `GNUTLS_EXT_FLAG_APPEND`

critical: the extension status

This function will get the X.509 TLS features extension structure from the certificate. The returned structure needs to be freed using `gnutls_x509_tlsfeatures_deinit()` .

When the `flags` is set to `GNUTLS_EXT_FLAG_APPEND` , then if the `features` structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the `features` structure then they will be merged with.

Note that `features` must be initialized prior to calling this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_crt_get_version

`int gnutls_x509_crt_get_version (`*gnutls_x509_crt_t* `cert`)` [Function]

cert: should contain a `gnutls_x509_crt_t` type

This function will return the version of the specified Certificate.

Returns: version of certificate, or a negative error code on error.

gnutls_x509_crt_import

`int gnutls_x509_crt_import (`*gnutls_x509_crt_t* `cert`, *const gnutls_datum_t ** `data`, *gnutls_x509_crt_fmt_t* `format`)` [Function]

cert: The data to store the parsed certificate.

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will convert the given DER or PEM encoded Certificate to the native gnutls_x509_crt_t format. The output will be stored in `cert` .

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_import_url

`int gnutls_x509_crt_import_url` (*gnutls_x509_crt_t* `crt`, *const char * * `url`, *unsigned int* `flags`) [Function]

crt: A certificate of type `gnutls_x509_crt_t`

url: A PKCS 11 url

flags: One of GNUTLS_PKCS11_OBJ_* flags for PKCS11 URLs or zero otherwise

This function will import a certificate present in a PKCS11 token or any type of back-end that supports URLs.

In previous versions of gnutls this function was named gnutls_x509_crt_import_pkcs11_url, and the old name is an alias to this one.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_x509_crt_init

`int gnutls_x509_crt_init` (*gnutls_x509_crt_t * `cert`) [Function]

cert: A pointer to the type to be initialized

This function will initialize an X.509 certificate structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_list_import

`int gnutls_x509_crt_list_import` (*gnutls_x509_crt_t * `certs`, *unsigned int * `cert_max`, *const gnutls_datum_t * `data`, *gnutls_x509_crt_fmt_t* `format`, *unsigned int* `flags`) [Function]

certs: Indicates where the parsed list will be copied to. Must not be initialized.

cert_max: Initially must hold the maximum number of certs. It will be updated with the number of certs available.

data: The PEM encoded certificate.

format: One of DER or PEM.

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded certificate list to the native gnutls_x509_crt_t format. The output will be stored in `certs` . They will be automatically initialized.

The flag `GNUTLS_X509_CRT_LIST_IMPORT_FAIL_IF_EXCEED` will cause import to fail if the certificates in the provided buffer are more than the available structures. The `GNUTLS_X509_CRT_LIST_FAIL_IF_UNSORTED` flag will cause the function to fail if the provided list is not sorted from subject to issuer.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

Returns: the number of certificates read or a negative error value.

gnutls_x509_crt_list_import2

`int gnutls_x509_crt_list_import2` (*gnutls_x509_crt_t ** `certs`*, *unsigned int * `size`*, *const gnutls_datum_t * `data`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `flags`*) [Function]

certs: Will hold the parsed certificate list.

size: It will contain the size of the list.

data: The PEM encoded certificate.

format: One of DER or PEM.

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded certificate list to the native gnutls_x509_crt_t format. The output will be stored in `certs` which will be allocated and initialized.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

To deinitialize `certs` , you need to deinitialize each crt structure independently, and use `gnutls_free()` at `certs` .

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.0

gnutls_x509_crt_list_import_url

`int gnutls_x509_crt_list_import_url` (*gnutls_x509_crt_t ** `certs`*, *unsigned int * `size`*, *const char * `url`*, *gnutls_pin_callback_t `pin_fn`*, *void * `pin_fn_userdata`*, *unsigned int `flags`*) [Function]

certs: Will hold the allocated certificate list.

size: It will contain the size of the list.

url: A PKCS 11 url

pin_fn: a PIN callback if not globally set

pin_fn_userdata: parameter for the PIN callback

flags: One of GNUTLS_PKCS11_OBJ_* flags for PKCS11 URLs or zero otherwise

This function will import a certificate chain present in a PKCS11 token or any type of back-end that supports URLs. The certificates must be deinitialized afterwards using `gnutls_x509_crt_deinit()` and the returned pointer must be freed using `gnutls_free()` .

The URI provided must be the first certificate in the chain; subsequent certificates will be retrieved using `gnutls_pkcs11_get_raw_issuer()` or equivalent functionality for the supported URI.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_x509_crt_list_verify

`int gnutls_x509_crt_list_verify` (*const gnutls_x509_crt_t * `cert_list`, unsigned `cert_list_length`, const gnutls_x509_crt_t * `CA_list`, unsigned `CA_list_length`, const gnutls_x509_crl_t * `CRL_list`, unsigned `CRL_list_length`, unsigned int `flags`, unsigned int * `verify`*) [Function]

cert_list: is the certificate list to be verified

cert_list_length: holds the number of certificate in cert_list

CA_list: is the CA list which will be used in verification

CA_list_length: holds the number of CA certificate in CA_list

CRL_list: holds a list of CRLs.

CRL_list_length: the length of CRL list.

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

verify: will hold the certificate verification output.

This function will try to verify the given certificate list and return its status. The details of the verification are the same as in `gnutls_x509_trust_list_verify_crt2()` .

You must check the peer's name in order to check if the verified certificate belongs to the actual peer.

The certificate verification output will be put in `verify` and will be one or more of the gnutls_certificate_status_t enumerated elements bitwise or'd. For a more detailed verification status use `gnutls_x509_crt_verify()` per list element.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_print

`int gnutls_x509_crt_print` (*gnutls_x509_crt_t `cert`, gnutls_certificate_print_formats_t `format`, gnutls_datum_t * `out`*) [Function]

cert: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print a X.509 certificate, suitable for display to a human.

If the format is `GNUTLS_CRT_PRINT_FULL` then all fields of the certificate will be output, on multiple lines. The `GNUTLS_CRT_PRINT_ONELINE` format will generate one line with some selected fields, which is useful for logging purposes.

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_activation_time

`int gnutls_x509_crt_set_activation_time` (*gnutls_x509_crt_t* `cert`, *time_t* `act_time`) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

act_time: The actual time

This function will set the time this certificate was or will be activated.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_authority_info_access

`int gnutls_x509_crt_set_authority_info_access` (*gnutls_x509_crt_t* `crt`, *int* `what`, *gnutls_datum_t ** `data`) [Function]

crt: Holds the certificate

what: what data to get, a `gnutls_info_access_what_t` type.

data: output data to be freed with `gnutls_free()` .

This function sets the Authority Information Access (AIA) extension, see RFC 5280 section 4.2.2.1 for more information.

The type of data stored in `data` is specified via `what` which should be `gnutls_info_access_what_t` values.

If `what` is `GNUTLS_IA_OCSP_URI` , `data` will hold the OCSP URI. If `what` is `GNUTLS_IA_CAISSUERS_URI` , `data` will hold the caIssuers URI.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_x509_crt_set_authority_key_id

`int gnutls_x509_crt_set_authority_key_id` (*gnutls_x509_crt_t* `cert`, *const void ** `id`, *size_t* `id_size`) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

id: The key ID

id_size: Holds the size of the key ID field.

This function will set the X.509 certificate's authority key ID extension. Only the keyIdentifier field can be set with this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_basic_constraints

`int gnutls_x509_crt_set_basic_constraints` (*gnutls_x509_crt_t* `crt`, *unsigned int* `ca`, *int* `pathLenConstraint`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

ca: true(1) or false(0). Depending on the Certificate authority status.

pathLenConstraint: non-negative error codes indicate maximum length of path, and negative error codes indicate that the pathLenConstraints field should not be present.

This function will set the basicConstraints certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_ca_status

`int gnutls_x509_crt_set_ca_status (`*gnutls_x509_crt_t* `crt`, *unsigned int* `ca``)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

ca: true(1) or false(0). Depending on the Certificate authority status.

This function will set the basicConstraints certificate extension. Use `gnutls_x509_crt_set_basic_constraints()` if you want to control the pathLenConstraint field too.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_crl_dist_points

`int gnutls_x509_crt_set_crl_dist_points (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const void ** `data_string`, *unsigned int* `reason_flags``)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data_string: The data to be set

reason_flags: revocation reasons

This function will set the CRL distribution points certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_crl_dist_points2

`int gnutls_x509_crt_set_crl_dist_points2 (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const void ** `data`, *unsigned int* `data_size`, *unsigned int* `reason_flags``)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data: The data to be set

data_size: The data size

reason_flags: revocation reasons

This function will set the CRL distribution points certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.6.0

gnutls_x509_crt_set_crq

`int gnutls_x509_crt_set_crq` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_crq_t* `crq`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

crq: holds a certificate request

This function will set the name and public parameters as well as the extensions from the given certificate request to the certificate. Only RSA keys are currently supported.

Note that this function will only set the `crq` if it is self signed and the signature is correct. See `gnutls_x509_crq_sign2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_crq_extension_by_oid

`int gnutls_x509_crt_set_crq_extension_by_oid` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_crq_t* `crq`, *const char * oid*, *unsigned* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

crq: holds a certificate request

oid: the object identifier of the OID to copy

flags: should be zero

This function will set the extension specify by `oid` from the given request to the certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_crt_set_crq_extensions

`int gnutls_x509_crt_set_crq_extensions` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_crq_t* `crq`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

crq: holds a certificate request

This function will set the extensions from the given request to the certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.8.0

gnutls_x509_crt_set_dn

`int gnutls_x509_crt_set_dn` (*gnutls_x509_crt_t* `crt`, *const char * dn*, *const char ** `err`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

dn: a comma separated DN string (RFC4514)

err: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded. On DN parsing error `GNUTLS_E_PARSING_ERROR` is returned.

Note that DNs are not expected to hold DNS information, and thus no automatic IDNA conversions are attempted when using this function. If that is required (e.g., store a domain in CN), process the corresponding input with `gnutls_idna_map()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_dn_by_oid

`int gnutls_x509_crt_set_dn_by_oid` (*gnutls_x509_crt_t* `crt`, *const char * oid*, *unsigned int* `raw_flag`, *const void * name*, *unsigned int* `sizeof_name`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

oid: holds an Object Identifier in a null terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

name: a pointer to the name

sizeof_name: holds the size of `name`

This function will set the part of the name of the Certificate subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()` . For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with `raw_flag` set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_expiration_time

`int gnutls_x509_crt_set_expiration_time` (*gnutls_x509_crt_t* `cert`, *time_t* `exp_time`) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

exp_time: The actual time

This function will set the time this Certificate will expire. Setting an expiration time to (time_t)-1 will set to the no well-defined expiration date value.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_extension_by_oid

`int gnutls_x509_crt_set_extension_by_oid` (*gnutls_x509_crt_t* `crt`, *const char * oid*, *const void * buf*, *size_t* `sizeof_buf`, *unsigned int* `critical`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

oid: holds an Object Identifier in null terminated string

buf: a pointer to a DER encoded data

sizeof_buf: holds the size of `buf`

critical: should be non-zero if the extension is to be marked as critical

This function will set an the extension, by the specified OID, in the certificate. The extension data should be binary data DER encoded.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_flags

`void gnutls_x509_crt_set_flags` (*gnutls_x509_crt_t* `cert`, *unsigned int* `flags`) [Function]

cert: A type `gnutls_x509_crt_t`

flags: flags from the `gnutls_x509_crt_flags`

This function will set flags for the specified certificate. Currently this is useful for the `GNUTLS_X509_CRT_FLAG_IGNORE_SANITY` which allows importing certificates even if they have known issues.

Since: 3.6.0

gnutls_x509_crt_set_inhibit_anypolicy

`int gnutls_x509_crt_set_inhibit_anypolicy` (*gnutls_x509_crt_t* `crt`, *unsigned int* `skipcerts`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

skipcerts: number of certificates after which anypolicy is no longer acceptable.

This function will set the Inhibit anyPolicy certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_issuer_alt_name

`int gnutls_x509_crt_set_issuer_alt_name` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const void * * `data`, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data: The data to be set

data_size: The size of data to be set

flags: GNUTLS_FSAN_SET to clear previous data or GNUTLS_FSAN_APPEND to append.

This function will set the issuer alternative name certificate extension. It can set the same types as `gnutls_x509_crt_set_subject_alt_name()` .

Since version 3.5.7 the `GNUTLS_SAN_RFC822NAME` , `GNUTLS_SAN_DNSNAME` , and `GNUTLS_SAN_OTHERNAME_XMPP` are converted to ACE format when necessary.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_crt_set_issuer_alt_othername

`int gnutls_x509_crt_set_issuer_alt_othername` [Function]
(*gnutls_x509_crt_t* `crt`, *const char * oid*, *const void * data*, *unsigned int data_size*, *unsigned int flags*)

crt: a certificate of type gnutls_x509_crt_t

oid: The other name OID

data: The data to be set

data_size: The size of data to be set

flags: GNUTLS_FSAN_SET to clear previous data or GNUTLS_FSAN_APPEND to append.

This function will set an "othername" to the issuer alternative name certificate extension.

The values set are set as binary values and are expected to have the proper DER encoding. For convenience the flags GNUTLS_FSAN_ENCODE_OCTET_STRING and GNUTLS_FSAN_ENCODE_UTF8_STRING can be used to encode the provided data.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_x509_crt_set_issuer_dn

`int gnutls_x509_crt_set_issuer_dn` (*gnutls_x509_crt_t* `crt`, *const char * dn*, *const char ** err*) [Function]

crt: a certificate of type gnutls_x509_crt_t

dn: a comma separated DN string (RFC4514)

err: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded. On DN parsing error GNUTLS_E_PARSING_ERROR is returned.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_issuer_dn_by_oid

`int gnutls_x509_crt_set_issuer_dn_by_oid` (*gnutls_x509_crt_t* `crt`, *const char * oid*, *unsigned int raw_flag*, *const void * name*, *unsigned int sizeof_name*) [Function]

crt: a certificate of type gnutls_x509_crt_t

oid: holds an Object Identifier in a null terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

name: a pointer to the name

sizeof_name: holds the size of `name`

This function will set the part of the name of the Certificate issuer, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()` . For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with `raw_flag` set.

Normally you do not need to call this function, since the signing operation will copy the signer's name as the issuer of the certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_issuer_unique_id

`int gnutls_x509_crt_set_issuer_unique_id (`*gnutls_x509_crt_t* `cert`*, const void * id, size_t id_size*`)` [Function]

cert: a certificate of type `gnutls_x509_crt_t`

id: The unique ID

id_size: Holds the size of the unique ID.

This function will set the X.509 certificate's issuer unique ID field.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.7

gnutls_x509_crt_set_key

`int gnutls_x509_crt_set_key (`*gnutls_x509_crt_t* `crt`*, gnutls_x509_privkey_t key*`)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

key: holds a private key

This function will set the public parameters from the given private key to the certificate.

To export the public key (i.e., the SubjectPublicKeyInfo part), check `gnutls_pubkey_import_x509()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_key_purpose_oid

`int gnutls_x509_crt_set_key_purpose_oid (`*gnutls_x509_crt_t* `cert`*, const void * oid, unsigned int critical*`)` [Function]

cert: a certificate of type `gnutls_x509_crt_t`

oid: a pointer to a null terminated string that holds the OID

critical: Whether this extension will be critical or not

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.

Subsequent calls to this function will append OIDs to the OID list.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_x509_crt_set_key_usage

`int gnutls_x509_crt_set_key_usage` (*gnutls_x509_crt_t* `crt`, *unsigned int* `usage`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

usage: an ORed sequence of the GNUTLS_KEY_* elements.

This function will set the keyUsage certificate extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_name_constraints

`int gnutls_x509_crt_set_name_constraints` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_name_constraints_t* `nc`, *unsigned int* `critical`) [Function]

crt: The certificate

nc: The nameconstraints structure

critical: whether this extension will be critical

This function will set the provided name constraints to the certificate extension list. This extension is always marked as critical.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_crt_set_pin_function

`void gnutls_x509_crt_set_pin_function` (*gnutls_x509_crt_t* `crt`, *gnutls_pin_callback_t* `fn`, *void * * `userdata`) [Function]

crt: The certificate structure

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when it is required to access a protected object. This function overrides the global function set using `gnutls_pkcs11_set_pin_function()` .

Note that this callback is currently used only during the import of a PKCS `11` certificate with `gnutls_x509_crt_import_url()` .

Since: 3.1.0

gnutls_x509_crt_set_policy

`int gnutls_x509_crt_set_policy (`*gnutls_x509_crt_t* `crt`, *const struct gnutls_x509_policy_st ** `policy`, *unsigned int* `critical``)` [Function]

crt: should contain a `gnutls_x509_crt_t` type

policy: A pointer to a policy

critical: use non-zero if the extension is marked as critical

This function will set the certificate policy extension (2.5.29.32). Multiple calls to this function append a new policy.

Note the maximum text size for the qualifier `GNUTLS_X509_QUALIFIER_NOTICE` is 200 characters. This function will fail with `GNUTLS_E_INVALID_REQUEST` if this is exceeded.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.5

gnutls_x509_crt_set_private_key_usage_period

`int gnutls_x509_crt_set_private_key_usage_period (`*gnutls_x509_crt_t* `crt`, *time_t* `activation`, *time_t* `expiration``)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

activation: The activation time

expiration: The expiration time

This function will set the private key usage period extension (2.5.29.16).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_proxy

`int gnutls_x509_crt_set_proxy (`*gnutls_x509_crt_t* `crt`, *int* `pathLenConstraint`, *const char ** `policyLanguage`, *const char ** `policy`, *size_t* `sizeof_policy``)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

pathLenConstraint: non-negative error codes indicate maximum length of path, and negative error codes indicate that the pathLenConstraints field should not be present.

policyLanguage: OID describing the language of `policy` .

policy: uint8_t byte array with policy language, can be `NULL`

sizeof_policy: size of `policy` .

This function will set the proxyCertInfo extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_proxy_dn

`int gnutls_x509_crt_set_proxy_dn (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_crt_t* `eecrt`, *unsigned int* `raw_flag`, *const void ** `name`, *unsigned int* `sizeof_name`) [Function]

crt: a gnutls_x509_crt_t type with the new proxy cert

eecrt: the end entity certificate that will be issuing the proxy

raw_flag: must be 0, or 1 if the CN is DER encoded

name: a pointer to the CN name, may be NULL (but MUST then be added later)

sizeof_name: holds the size of `name`

This function will set the subject in `crt` to the end entity's `eecrt` subject name, and add a single Common Name component `name` of size `sizeof_name` . This corresponds to the required proxy certificate naming style. Note that if `name` is `NULL` , you MUST set it later by using `gnutls_x509_crt_set_dn_by_oid()` or similar.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_serial

`int gnutls_x509_crt_set_serial (`*gnutls_x509_crt_t* `cert`, *const void ** `serial`, *size_t* `serial_size`) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

serial: The serial number

serial_size: Holds the size of the serial field.

This function will set the X.509 certificate's serial number. While the serial number is an integer, it is often handled as an opaque field by several CAs. For this reason this function accepts any kind of data as a serial number. To be consistent with the X.509/PKIX specifications the provided `serial` should be a big-endian positive number (i.e. it's leftmost bit should be zero).

The size of the serial is restricted to 20 bytes maximum by RFC5280. This function allows writing more than 20 bytes but the generated certificates in that case may be rejected by other implementations.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_spki

`int gnutls_x509_crt_set_spki (`*gnutls_x509_crt_t* `crt`, *const gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will set the certificate's subject public key information explicitly. This is intended to be used in the cases where a single public key (e.g., RSA) can be used for multiple signature algorithms (RSA PKCS1-1.5, and RSA-PSS).

To export the public key (i.e., the SubjectPublicKeyInfo part), check `gnutls_pubkey_import_x509()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_x509_crt_set_subject_alt_name

`int gnutls_x509_crt_set_subject_alt_name` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const void *` `data`, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data: The data to be set

data_size: The size of data to be set

flags: GNUTLS_FSAN_SET to clear previous data or GNUTLS_FSAN_APPEND to append.

This function will set the subject alternative name certificate extension. It can set the following types: `GNUTLS_SAN_DNSNAME` as a text string, `GNUTLS_SAN_RFC822NAME` as a text string, `GNUTLS_SAN_URI` as a text string, `GNUTLS_SAN_IPADDRESS` as a binary IP address (4 or 16 bytes), `GNUTLS_SAN_OTHERNAME_XMPP` as a UTF8 string (since 3.5.0).

Since version 3.5.7 the `GNUTLS_SAN_RFC822NAME` , `GNUTLS_SAN_DNSNAME` , and `GNUTLS_SAN_OTHERNAME_XMPP` are converted to ACE format when necessary.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.6.0

gnutls_x509_crt_set_subject_alt_othername

`int gnutls_x509_crt_set_subject_alt_othername` (*gnutls_x509_crt_t* `crt`, *const char *` `oid`, *const void *` `data`, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

oid: The other name OID

data: The data to be set

data_size: The size of data to be set

flags: GNUTLS_FSAN_SET to clear previous data or GNUTLS_FSAN_APPEND to append.

This function will set an "othername" to the subject alternative name certificate extension.

The values set are set as binary values and are expected to have the proper DER encoding. For convenience the flags `GNUTLS_FSAN_ENCODE_OCTET_STRING` and `GNUTLS_FSAN_ENCODE_UTF8_STRING` can be used to encode the provided data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_x509_crt_set_subject_alternative_name

`int gnutls_x509_crt_set_subject_alternative_name` [Function]
(*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const char * data_string*)

crt: a certificate of type `gnutls_x509_crt_t`

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data_string: The data to be set, a (0) terminated string

This function will set the subject alternative name certificate extension. This function assumes that data can be expressed as a null terminated string.

The name of the function is unfortunate since it is inconsistent with `gnutls_x509_crt_get_subject_alt_name()` .

See `gnutls_x509_crt_set_subject_alt_name()` for more information.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_subject_key_id

`int gnutls_x509_crt_set_subject_key_id (`*gnutls_x509_crt_t* `cert`, *const void * id*, *size_t id_size*) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

id: The key ID

id_size: Holds the size of the subject key ID field.

This function will set the X.509 certificate's subject key ID extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_subject_unique_id

`int gnutls_x509_crt_set_subject_unique_id (`*gnutls_x509_crt_t* `cert`, *const void * id*, *size_t id_size*) [Function]

cert: a certificate of type `gnutls_x509_crt_t`

id: The unique ID

id_size: Holds the size of the unique ID.

This function will set the X.509 certificate's subject unique ID field.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.7

gnutls_x509_crt_set_tlsfeatures

`int gnutls_x509_crt_set_tlsfeatures (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_tlsfeatures_t* `features`) [Function]

crt: A X.509 certificate

features: If the function succeeds, the features will be added to the certificate.

This function will set the certificates X.509 TLS extension from the given structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_crt_set_version

`int gnutls_x509_crt_set_version (`*gnutls_x509_crt_t* `crt`, *unsigned int* `version`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

version: holds the version number. For X.509v1 certificates must be 1.

This function will set the version of the certificate. This must be one for X.509 version 1, and so on. Plain certificates without extensions must have version set to one.

To create well-formed certificates, you must specify version 3 if you use any certificate extensions. Extensions are created by functions such as `gnutls_x509_crt_set_subject_alt_name()` or `gnutls_x509_crt_set_key_usage()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_sign

`int gnutls_x509_crt_sign (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_crt_t* `issuer`, *gnutls_x509_privkey_t* `issuer_key`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

This function is the same a `gnutls_x509_crt_sign2()` with no flags, and an appropriate hash algorithm. The hash algorithm used may vary between versions of GnuTLS, and it is tied to the security level of the issuer's public key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_sign2

`int gnutls_x509_crt_sign2 (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_crt_t* `issuer`, *gnutls_x509_privkey_t* `issuer_key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

crt: a certificate of type `gnutls_x509_crt_t`

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use, `GNUTLS_DIG_SHA256` is a safe choice

flags: must be 0

This function will sign the certificate with the issuer's private key, and will copy the issuer's information into the certificate.

This must be the last step in a certificate generation since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed certificate will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_verify

`int gnutls_x509_crt_verify` (*gnutls_x509_crt_t* `cert`, *const gnutls_x509_crt_t * CA_list*, *unsigned* `CA_list_length`, *unsigned int* `flags`, *unsigned int * verify*) [Function]

cert: is the certificate to be verified

CA_list: is one certificate that is considered to be trusted one

CA_list_length: holds the number of CA certificate in CA_list

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

verify: will hold the certificate verification output.

This function will try to verify the given certificate and return its status. Note that a verification error does not imply a negative return status. In that case the `verify` status is set.

The details of the verification are the same as in `gnutls_x509_trust_list_verify_crt2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_verify_data2

`int gnutls_x509_crt_verify_data2` (*gnutls_x509_crt_t* `crt`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t * data*, *const gnutls_datum_t * signature*) [Function]

crt: Holds the certificate to verify with

algo: The signature algorithm used

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

data: holds the signed data

signature: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, `GNUTLS_E_EXPIRED` or `GNUTLS_E_NOT_YET_ACTIVATED` on expired or not yet activated certificate and zero or positive code on success.

Note that since GnuTLS 3.5.6 this function introduces checks in the end certificate (`crt`), including time checks and key usage checks.

Since: 3.4.0

gnutls_x509_dn_deinit

`void gnutls_x509_dn_deinit` (*gnutls_x509_dn_t* `dn`) [Function]

dn: a DN uint8_t object pointer.

This function deallocates the DN object as returned by `gnutls_x509_dn_import()` .

Since: 2.4.0

gnutls_x509_dn_export

`int gnutls_x509_dn_export` (*gnutls_x509_dn_t* `dn`, *gnutls_x509_crt_fmt_t* `format`, *void * * `output_data`, *size_t * `output_data_size`) [Function]

dn: Holds the uint8_t DN object

format: the format of output params. One of PEM or DER.

output_data: will contain a DN PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the DN to DER or PEM format.

If the buffer provided is not long enough to hold the output, then * `output_data_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN NAME".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_dn_export2

`int gnutls_x509_dn_export2` (*gnutls_x509_dn_t* `dn`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t * `out`) [Function]

dn: Holds the uint8_t DN object

format: the format of output params. One of PEM or DER.

out: will contain a DN PEM or DER encoded

This function will export the DN to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN NAME".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.3

gnutls_x509_dn_get_rdn_ava

`int gnutls_x509_dn_get_rdn_ava` (*gnutls_x509_dn_t* `dn`, *int* `irdn`, *int* `iava`, *gnutls_x509_ava_st* * `ava`) [Function]

dn: a pointer to DN

irdn: index of RDN

iava: index of AVA.

ava: Pointer to structure which will hold output information.

Get pointers to data within the DN. The format of the `ava` structure is shown below.

struct gnutls_x509_ava_st { gnutls_datum_t oid; gnutls_datum_t value; unsigned long value_tag; };

The X.509 distinguished name is a sequence of sequences of strings and this is what the `irdn` and `iava` indexes model.

Note that `ava` will contain pointers into the `dn` structure which in turns points to the original certificate. Thus you should not modify any data or deallocate any of those.

This is a low-level function that requires the caller to do the value conversions when necessary (e.g. from UCS-2).

Returns: Returns 0 on success, or an error code.

gnutls_x509_dn_get_str

`int gnutls_x509_dn_get_str` (*gnutls_x509_dn_t* `dn`, *gnutls_datum_t* * `str`) [Function]

dn: a pointer to DN

str: a datum that will hold the name

This function will allocate buffer and copy the name in the provided DN. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.2

gnutls_x509_dn_get_str2

`int gnutls_x509_dn_get_str2` (*gnutls_x509_dn_t* `dn`, *gnutls_datum_t* * `str`, *unsigned* `flags`) [Function]

dn: a pointer to DN

str: a datum that will hold the name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will allocate buffer and copy the name in the provided DN. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.7

gnutls_x509_dn_import

`int gnutls_x509_dn_import (`*gnutls_x509_dn_t* `dn`, *const gnutls_datum_t ** `data``)` [Function]

dn: the structure that will hold the imported DN

data: should contain a DER encoded RDN sequence

This function parses an RDN sequence and stores the result to a `gnutls_x509_dn_t` type. The data must have been initialized with `gnutls_x509_dn_init()` . You may use `gnutls_x509_dn_get_rdn_ava()` to decode the DN.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.4.0

gnutls_x509_dn_init

`int gnutls_x509_dn_init (`*gnutls_x509_dn_t ** `dn``)` [Function]

dn: the object to be initialized

This function initializes a `gnutls_x509_dn_t` type.

The object returned must be deallocated using `gnutls_x509_dn_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.4.0

gnutls_x509_dn_oid_known

`int gnutls_x509_dn_oid_known (`*const char ** `oid``)` [Function]

oid: holds an Object Identifier in a null terminated string

This function will inform about known DN OIDs. This is useful since functions like `gnutls_x509_crt_set_dn_by_oid()` use the information on known OIDs to properly encode their input. Object Identifiers that are not known are not encoded by these functions, and their input is stored directly into the ASN.1 structure. In that case of unknown OIDs, you have the responsibility of DER encoding your data.

Returns: 1 on known OIDs and 0 otherwise.

gnutls_x509_dn_oid_name

`const char * gnutls_x509_dn_oid_name (`*const char ** `oid`, *unsigned int* `flags``)` [Function]

oid: holds an Object Identifier in a null terminated string

flags: 0 or GNUTLS_X509_DN_OID_*

This function will return the name of a known DN OID. If `GNUTLS_X509_DN_OID_RETURN_OID` is specified this function will return the given OID if no descriptive name has been found.

Returns: A null terminated string or NULL otherwise.

Since: 3.0

gnutls_x509_dn_set_str

`int gnutls_x509_dn_set_str` (*gnutls_x509_dn_t* `dn`, *const char *** `str`, *const char ** * `err`) [Function]

dn: a pointer to DN

str: a comma separated DN string (RFC4514)

err: indicates the error position (if any)

This function will set the DN on the provided DN structure. The input string should be plain ASCII or UTF-8 encoded. On DN parsing error `GNUTLS_E_PARSING_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.3

gnutls_x509_ext_deinit

`void gnutls_x509_ext_deinit` (*gnutls_x509_ext_st ** `ext`) [Function]

ext: The extensions structure

This function will deinitialize an extensions structure.

Since: 3.3.8

gnutls_x509_ext_export_aia

`int gnutls_x509_ext_export_aia` (*gnutls_x509_aia_t* `aia`, *gnutls_datum_t ** `ext`) [Function]

aia: The authority info access

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will DER encode the Authority Information Access (AIA) extension; see RFC 5280 section 4.2.2.1 for more information on the extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_authority_key_id

`int gnutls_x509_ext_export_authority_key_id` (*gnutls_x509_aki_t* `aki`, *gnutls_datum_t ** `ext`) [Function]

aki: An initialized authority key identifier

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided key identifier to a DER-encoded PKIX AuthorityKeyIdentifier extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_basic_constraints

`int gnutls_x509_ext_export_basic_constraints (`*unsigned int* `ca`*, int* `pathlen`*, gnutls_datum_t * ext*`)` [Function]

ca: non-zero for a CA

pathlen: The path length constraint (set to -1 for no constraint)

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the parameters provided to a basic constraints DER encoded extension (2.5.29.19). The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_crl_dist_points

`int gnutls_x509_ext_export_crl_dist_points` [Function]
`(`*gnutls_x509_crl_dist_points_t* `cdp`*, gnutls_datum_t * ext*`)`

cdp: A pointer to an initialized CRL distribution points.

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided policies, to a certificate policy DER encoded extension (2.5.29.31).

The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_inhibit_anypolicy

`int gnutls_x509_ext_export_inhibit_anypolicy (`*unsigned int* `skipcerts`*, gnutls_datum_t * ext*`)` [Function]

skipcerts: number of certificates after which anypolicy is no longer acceptable.

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the `skipcerts` value to a DER encoded Inhibit AnyPolicy PKIX extension. The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_x509_ext_export_key_purposes

`int gnutls_x509_ext_export_key_purposes` [Function]
(*gnutls_x509_key_purposes_t* `p`, *gnutls_datum_t* * `ext`)

p: The key purposes

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the key purposes type to a DER-encoded PKIX ExtKeyUsageSyntax (2.5.29.37) extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_key_usage

`int gnutls_x509_ext_export_key_usage` (*unsigned int* `usage`, [Function]
gnutls_datum_t * `ext`)

usage: an ORed sequence of the GNUTLS_KEY_* elements.

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the keyUsage bit string to a DER encoded PKIX extension. The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_name_constraints

`int gnutls_x509_ext_export_name_constraints` [Function]
(*gnutls_x509_name_constraints_t* `nc`, *gnutls_datum_t* * `ext`)

nc: The nameconstraints

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided name constraints type to a DER-encoded PKIX NameConstraints (2.5.29.30) extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_policies

`int gnutls_x509_ext_export_policies` (*gnutls_x509_policies_t* [Function]
`policies`, *gnutls_datum_t* * `ext`)

policies: A pointer to an initialized policies.

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided policies, to a certificate policy DER encoded extension (2.5.29.32).

The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_private_key_usage_period

`int gnutls_x509_ext_export_private_key_usage_period (`*time_t* `activation`, *time_t* `expiration`, *gnutls_datum_t * * `ext`*`)` [Function]

activation: The activation time

expiration: The expiration time

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the periods provided to a private key usage DER encoded extension (2.5.29.16). The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_proxy

`int gnutls_x509_ext_export_proxy (`*int* `pathLenConstraint`, *const char * * `policyLanguage`*, *const char * * `policy`*, *size_t* `sizeof_policy`, *gnutls_datum_t * * `ext`*`)` [Function]

pathLenConstraint: A negative value will remove the path length constraint, while non-negative values will be set as the length of the pathLenConstraints field.

policyLanguage: OID describing the language of `policy` .

policy: uint8_t byte array with policy language, can be `NULL`

sizeof_policy: size of `policy` .

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the parameters provided to a proxyCertInfo extension.

The `ext` data will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_subject_alt_names

`int gnutls_x509_ext_export_subject_alt_names (`*gnutls_subject_alt_names_t* `sans`, *gnutls_datum_t * * `ext`*`)` [Function]

sans: The alternative names

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided alternative names structure to a DER-encoded SubjectAltName PKIX extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_subject_key_id

`int gnutls_x509_ext_export_subject_key_id` (*const gnutls_datum_t * `id`, gnutls_datum_t * `ext`*) [Function]

id: The key identifier

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided key identifier to a DER-encoded PKIX SubjectKeyIdentifier extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_export_tlsfeatures

`int gnutls_x509_ext_export_tlsfeatures` (*gnutls_x509_tlsfeatures_t `f`, gnutls_datum_t * `ext`*) [Function]

f: The features structure

ext: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided TLS features structure structure to a DER-encoded TLS features PKIX extension. The output data in `ext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_ext_import_aia

`int gnutls_x509_ext_import_aia` (*const gnutls_datum_t * `ext`, gnutls_x509_aia_t `aia`, unsigned int `flags`*) [Function]

ext: The DER-encoded extension data

aia: The authority info access

flags: should be zero

This function extracts the Authority Information Access (AIA) extension from the provided DER-encoded data; see RFC 5280 section 4.2.2.1 for more information on the extension. The AIA extension holds a sequence of AccessDescription (AD) data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_authority_key_id

`int gnutls_x509_ext_import_authority_key_id (`*const gnutls_datum_t * `ext`, gnutls_x509_aki_t `aki`, unsigned int `flags`*`)` [Function]

ext: a DER encoded extension

aki: An initialized authority key identifier type

flags: should be zero

This function will return the subject key ID stored in the provided AuthorityKeyIdentifier extension.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_basic_constraints

`int gnutls_x509_ext_import_basic_constraints (`*const gnutls_datum_t * `ext`, unsigned int * `ca`, int * `pathlen`*`)` [Function]

ext: the DER encoded extension data

ca: will be non zero if the CA status is true

pathlen: the path length constraint; will be set to -1 for no limit

This function will return the CA status and path length constraint as written in the PKIX extension 2.5.29.19.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_crl_dist_points

`int gnutls_x509_ext_import_crl_dist_points (`*const gnutls_datum_t * `ext`, gnutls_x509_crl_dist_points_t `cdp`, unsigned int `flags`*`)` [Function]

ext: the DER encoded extension data

cdp: A pointer to an initialized CRL distribution points.

flags: should be zero

This function will extract the CRL distribution points extension (2.5.29.31) and store it into the provided type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_inhibit_anypolicy

`int gnutls_x509_ext_import_inhibit_anypolicy (`*const gnutls_datum_t * `ext`, unsigned int * `skipcerts`*`)` [Function]

ext: the DER encoded extension data

skipcerts: will hold the number of certificates after which anypolicy is no longer acceptable.

This function will return certificate's value of SkipCerts, by reading the DER data of the Inhibit anyPolicy X.509 extension (2.5.29.54).

The `skipcerts` value is the number of additional certificates that may appear in the path before the anyPolicy (`GNUTLS_X509_OID_POLICY_ANY`) is no longer acceptable.

Returns: zero, or a negative error code in case of parsing error. If the certificate does not contain the Inhibit anyPolicy extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 3.6.0

gnutls_x509_ext_import_key_purposes

`int gnutls_x509_ext_import_key_purposes` (*const gnutls_datum_t * ext, gnutls_x509_key_purposes_t p, unsigned int flags*) [Function]

ext: The DER-encoded extension data

p: The key purposes

flags: should be zero

This function will extract the key purposes in the provided DER-encoded ExtKeyUsageSyntax PKIX extension, to a `gnutls_x509_key_purposes_t` type. The data must be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_key_usage

`int gnutls_x509_ext_import_key_usage` (*const gnutls_datum_t * ext, unsigned int * key_usage*) [Function]

ext: the DER encoded extension data

key_usage: where the key usage bits will be stored

This function will return certificate's key usage, by reading the DER data of the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: `GNUTLS_KEY_DIGITAL_SIGNATURE` , `GNUTLS_KEY_NON_REPUDIATION` , `GNUTLS_KEY_KEY_ENCIPHERMENT` , `GNUTLS_KEY_DATA_ENCIPHERMENT` , `GNUTLS_KEY_KEY_AGREEMENT` , `GNUTLS_KEY_KEY_CERT_SIGN` , `GNUTLS_KEY_CRL_SIGN` , `GNUTLS_KEY_ENCIPHER_ONLY` , `GNUTLS_KEY_DECIPHER_ONLY` .

Returns: the certificate key usage, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 3.3.0

gnutls_x509_ext_import_name_constraints

`int gnutls_x509_ext_import_name_constraints (`*const gnutls_datum_t * `ext`, gnutls_x509_name_constraints_t `nc`, unsigned int `flags`*`)` [Function]

ext: a DER encoded extension

nc: The nameconstraints

flags: zero or `GNUTLS_NAME_CONSTRAINTS_FLAG_APPEND`

This function will return an intermediate type containing the name constraints of the provided NameConstraints extension. That can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

When the `flags` is set to `GNUTLS_NAME_CONSTRAINTS_FLAG_APPEND` , then if the `nc` type is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the `nc` structure then the constraints will be merged with the existing constraints following RFC5280 p6.1.4 (excluded constraints will be appended, permitted will be intersected).

Note that `nc` must be initialized prior to calling this function.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_policies

`int gnutls_x509_ext_import_policies (`*const gnutls_datum_t * `ext`, gnutls_x509_policies_t `policies`, unsigned int `flags`*`)` [Function]

ext: the DER encoded extension data

policies: A pointer to an initialized policies.

flags: should be zero

This function will extract the certificate policy extension (2.5.29.32) and store it the provided policies.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_private_key_usage_period

`int gnutls_x509_ext_import_private_key_usage_period (`*const gnutls_datum_t * `ext`, time_t * `activation`, time_t * `expiration`*`)` [Function]

ext: the DER encoded extension data

activation: Will hold the activation time

expiration: Will hold the expiration time

This function will return the expiration and activation times of the private key as written in the PKIX extension 2.5.29.16.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_proxy

`int gnutls_x509_ext_import_proxy (`*const gnutls_datum_t * `ext`*, *int * `pathlen`*, *char ** `policyLanguage`*, *char ** `policy`*, *size_t * `sizeof_policy`*)` [Function]

ext: the DER encoded extension data

pathlen: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pCPathLenConstraint field and the actual value, -1 indicate that the field is absent.

policyLanguage: output variable with OID of policy language

policy: output variable with policy data

sizeof_policy: output variable with size of policy data

This function will return the information from a proxy certificate extension. It reads the ProxyCertInfo X.509 extension (1.3.6.1.5.5.7.1.14). The `policyLanguage` and `policy` values must be deinitialized using `gnutls_free()` after use.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_subject_alt_names

`int gnutls_x509_ext_import_subject_alt_names (`*const gnutls_datum_t * `ext`*, *gnutls_subject_alt_names_t `sans`*, *unsigned int `flags`*)` [Function]

ext: The DER-encoded extension data

sans: The alternative names

flags: should be zero

This function will export the alternative names in the provided DER-encoded SubjectAltName PKIX extension, to a `gnutls_subject_alt_names_t` type. `sans` must be initialized.

This function will succeed even if there no subject alternative names in the structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_subject_key_id

`int gnutls_x509_ext_import_subject_key_id (`*const gnutls_datum_t * `ext`*, *gnutls_datum_t * `id`*)` [Function]

ext: a DER encoded extension

id: will contain the subject key ID

This function will return the subject key ID stored in the provided SubjectKeyIdentifier extension. The ID will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_ext_import_tlsfeatures

`int gnutls_x509_ext_import_tlsfeatures (`*const gnutls_datum_t * `ext`, gnutls_x509_tlsfeatures_t `f`, unsigned int `flags`*`)` [Function]

ext: The DER-encoded extension data

f: The features structure

flags: zero or `GNUTLS_EXT_FLAG_APPEND`

This function will export the features in the provided DER-encoded TLS Features PKIX extension, to a `gnutls_x509_tlsfeatures_t` type. `f` must be initialized.

When the `flags` is set to `GNUTLS_EXT_FLAG_APPEND` , then if the `features` structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the `features` structure then they will be merged with.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_ext_print

`int gnutls_x509_ext_print (`*gnutls_x509_ext_st * `exts`, unsigned int `exts_size`, gnutls_certificate_print_formats_t `format`, gnutls_datum_t * `out`*`)` [Function]

exts: The data to be printed

exts_size: the number of available structures

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print X.509 certificate extensions, suitable for display to a human.

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_key_purpose_deinit

`void gnutls_x509_key_purpose_deinit (`*gnutls_x509_key_purposes_t `p`*`)` [Function]

p: The key purposes

This function will deinitialize a key purposes type.

Since: 3.3.0

gnutls_x509_key_purpose_get

`int gnutls_x509_key_purpose_get` (*gnutls_x509_key_purposes_t* `p`, *unsigned* `idx`, *gnutls_datum_t * * `oid`*) [Function]

p: The key purposes

idx: The index of the key purpose to retrieve

oid: Will hold the object identifier of the key purpose (to be treated as constant)

This function will retrieve the specified by the index key purpose in the purposes type. The object identifier will be a null terminated string.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_key_purpose_init

`int gnutls_x509_key_purpose_init` (*gnutls_x509_key_purposes_t * `p`*) [Function]

p: The key purposes

This function will initialize an alternative names type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_key_purpose_set

`int gnutls_x509_key_purpose_set` (*gnutls_x509_key_purposes_t* `p`, *const char * `oid`*) [Function]

p: The key purposes

oid: The object identifier of the key purpose

This function will store the specified key purpose in the purposes.

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

Since: 3.3.0

gnutls_x509_name_constraints_add_excluded

`int gnutls_x509_name_constraints_add_excluded` (*gnutls_x509_name_constraints_t* `nc`, *gnutls_x509_subject_alt_name_t* `type`, *const gnutls_datum_t * `name`*) [Function]

nc: The nameconstraints

type: The type of the constraints

name: The data of the constraints

This function will add a name constraint to the list of excluded constraints. The constraints `type` can be any of the following types: `GNUTLS_SAN_DNSNAME` , `GNUTLS_SAN_RFC822NAME` , `GNUTLS_SAN_DN` , `GNUTLS_SAN_URI` , `GNUTLS_SAN_IPADDRESS` . For

the latter, an IP address in network byte order is expected, followed by its network mask (which is 4 bytes in IPv4 or 16-bytes in IPv6).

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_name_constraints_add_permitted

int gnutls_x509_name_constraints_add_permitted [Function]
(*gnutls_x509_name_constraints_t* nc, *gnutls_x509_subject_alt_name_t* type, *const gnutls_datum_t ** name)

nc: The nameconstraints

type: The type of the constraints

name: The data of the constraints

This function will add a name constraint to the list of permitted constraints. The constraints type can be any of the following types: GNUTLS_SAN_DNSNAME , GNUTLS_SAN_RFC822NAME , GNUTLS_SAN_DN , GNUTLS_SAN_URI , GNUTLS_SAN_IPADDRESS . For the latter, an IP address in network byte order is expected, followed by its network mask.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_name_constraints_check

unsigned gnutls_x509_name_constraints_check [Function]
(*gnutls_x509_name_constraints_t* nc, *gnutls_x509_subject_alt_name_t* type, *const gnutls_datum_t ** name)

nc: the extracted name constraints

type: the type of the constraint to check (of type gnutls_x509_subject_alt_name_t)

name: the name to be checked

This function will check the provided name against the constraints in nc using the RFC5280 rules. Currently this function is limited to DNS names, emails and IP addresses (of type GNUTLS_SAN_DNSNAME , GNUTLS_SAN_RFC822NAME and GNUTLS_SAN_IPADDRESS).

Returns: zero if the provided name is not acceptable, and non-zero otherwise.

Since: 3.3.0

gnutls_x509_name_constraints_check_crt

unsigned gnutls_x509_name_constraints_check_crt [Function]
(*gnutls_x509_name_constraints_t* nc, *gnutls_x509_subject_alt_name_t* type, *gnutls_x509_crt_t* cert)

nc: the extracted name constraints

type: the type of the constraint to check (of type gnutls_x509_subject_alt_name_t)

cert: the certificate to be checked

This function will check the provided certificate names against the constraints in `nc` using the RFC5280 rules. It will traverse all the certificate's names and alternative names.

Currently this function is limited to DNS names and emails (of type `GNUTLS_SAN_DNSNAME` and `GNUTLS_SAN_RFC822NAME`).

Returns: zero if the provided name is not acceptable, and non-zero otherwise.

Since: 3.3.0

gnutls_x509_name_constraints_deinit

`void gnutls_x509_name_constraints_deinit` [Function]
(*gnutls_x509_name_constraints_t* `nc`)

nc: The nameconstraints

This function will deinitialize a name constraints type.

Since: 3.3.0

gnutls_x509_name_constraints_get_excluded

`int gnutls_x509_name_constraints_get_excluded` [Function]
(*gnutls_x509_name_constraints_t* `nc`, *unsigned* `idx`, *unsigned * * `type`, *gnutls_datum_t * * `name`)

nc: the extracted name constraints

idx: the index of the constraint

type: the type of the constraint (of type gnutls_x509_subject_alt_name_t)

name: the name in the constraint (of the specific type)

This function will return an intermediate type containing the name constraints of the provided CA certificate. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

The name should be treated as constant and valid for the lifetime of `nc` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_name_constraints_get_permitted

`int gnutls_x509_name_constraints_get_permitted` [Function]
(*gnutls_x509_name_constraints_t* `nc`, *unsigned* `idx`, *unsigned * * `type`, *gnutls_datum_t * * `name`)

nc: the extracted name constraints

idx: the index of the constraint

type: the type of the constraint (of type gnutls_x509_subject_alt_name_t)

name: the name in the constraint (of the specific type)

This function will return an intermediate type containing the name constraints of the provided CA certificate. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

The name should be treated as constant and valid for the lifetime of `nc` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_name_constraints_init

`int gnutls_x509_name_constraints_init` [Function]
(*gnutls_x509_name_constraints_t * `nc`*)

nc: The nameconstraints

This function will initialize a name constraints type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_othername_to_virtual

`int gnutls_x509_othername_to_virtual` (*const char * `oid`, const gnutls_datum_t * `othername`, unsigned int * `virt_type`, gnutls_datum_t * `virt`*) [Function]

oid: The othername object identifier

othername: The othername data

virt_type: GNUTLS_SAN_OTHERNAME_XXX

virt: allocated printable data

This function will parse and convert the othername data to a virtual type supported by gnutls.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.8

gnutls_x509_policies_deinit

`void gnutls_x509_policies_deinit` (*gnutls_x509_policies_t `policies`*) [Function]

policies: The authority key identifier

This function will deinitialize an authority key identifier type.

Since: 3.3.0

gnutls_x509_policies_get

`int gnutls_x509_policies_get (`*gnutls_x509_policies_t* `policies`, *unsigned int* `seq`, *struct gnutls_x509_policy_st ** `policy`)` [Function]

policies: The policies

seq: The index of the name to get

policy: Will hold the policy

This function will return a specific policy as stored in the `policies` type. The returned values should be treated as constant and valid for the lifetime of `policies` .

The any policy OID is available as the `GNUTLS_X509_OID_POLICY_ANY` macro.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_policies_init

`int gnutls_x509_policies_init (`*gnutls_x509_policies_t ** `policies`)` [Function]

policies: The authority key ID

This function will initialize an authority key ID type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_x509_policies_set

`int gnutls_x509_policies_set (`*gnutls_x509_policies_t* `policies`, *const struct gnutls_x509_policy_st ** `policy`)` [Function]

policies: An initialized policies

policy: Contains the policy to set

This function will store the specified policy in the provided `policies` .

Returns: On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

Since: 3.3.0

gnutls_x509_policy_release

`void gnutls_x509_policy_release (`*struct gnutls_x509_policy_st ** `policy`)` [Function]

policy: a certificate policy

This function will deinitialize all memory associated with the provided `policy` . The policy is allocated using `gnutls_x509_crt_get_policy()` .

Since: 3.1.5

gnutls_x509_privkey_cpy

`int gnutls_x509_privkey_cpy (`*gnutls_x509_privkey_t* `dst`, *gnutls_x509_privkey_t* `src`)` [Function]

dst: The destination key, which should be initialized.

src: The source key

This function will copy a private key from source to destination key. Destination has to be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_deinit

`void gnutls_x509_privkey_deinit (`*gnutls_x509_privkey_t* `key`)` [Function]

key: The key to be deinitialized

This function will deinitialize a private key structure.

gnutls_x509_privkey_export

`int gnutls_x509_privkey_export (`*gnutls_x509_privkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *void ** `output_data`, *size_t ** `output_data_size`)` [Function]

key: Holds the key

format: the format of output params. One of PEM or DER.

output_data: will contain a private key PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the private key to a PKCS1 structure for RSA or RSA-PSS keys, and integer sequence for DSA keys. Other keys types will be exported in PKCS8 form.

If the structure is PEM encoded, it will have a header of "BEGIN RSA PRIVATE KEY".

It is recommended to use `gnutls_x509_privkey_export_pkcs8()` instead of this function, when a consistent output format is required.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_export2

`int gnutls_x509_privkey_export2 (`*gnutls_x509_privkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t ** `out`)` [Function]

key: Holds the key

format: the format of output params. One of PEM or DER.

out: will contain a private key PEM or DER encoded

This function will export the private key to a PKCS1 structure for RSA or RSA-PSS keys, and integer sequence for DSA keys. Other keys types will be exported in PKCS8 form.

The output buffer is allocated using `gnutls_malloc()` .

It is recommended to use `gnutls_x509_privkey_export2_pkcs8()` instead of this function, when a consistent output format is required.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 3.1.3

gnutls_x509_privkey_export2_pkcs8

`int gnutls_x509_privkey_export2_pkcs8` (*gnutls_x509_privkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *const char *** `password`, *unsigned int* `flags`, *gnutls_datum_t ** `out`) [Function]

key: Holds the key

format: the format of output params. One of PEM or DER.

password: the password that will be used to encrypt the key.

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

out: will contain a private key PEM or DER encoded

This function will export the private key to a PKCS8 structure. Both RSA and DSA keys can be exported. For DSA keys we use PKCS 11 definitions. If the flags do not specify the encryption cipher, then the default 3DES (PBES2) will be used.

The `password` can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN ENCRYPTED PRIVATE KEY" or "BEGIN PRIVATE KEY" if encryption is not used.

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since 3.1.3

gnutls_x509_privkey_export_dsa_raw

`int gnutls_x509_privkey_export_dsa_raw` (*gnutls_x509_privkey_t* `key`, *gnutls_datum_t ** `p`, *gnutls_datum_t ** `q`, *gnutls_datum_t ** `g`, *gnutls_datum_t ** `y`, *gnutls_datum_t ** `x`) [Function]

key: a key

p: will hold the p

q: will hold the q

g: will hold the g

y: will hold the y

x: will hold the x

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_export_ecc_raw

`int gnutls_x509_privkey_export_ecc_raw` (*gnutls_x509_privkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `k`) [Function]

key: a key

curve: will hold the curve

x: will hold the x-coordinate

y: will hold the y-coordinate

k: will hold the private key

This function will export the ECC private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_x509_privkey_export_gost_raw

`int gnutls_x509_privkey_export_gost_raw` (*gnutls_x509_privkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_digest_algorithm_t* * `digest`, *gnutls_gost_paramset_t* * `paramset`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `k`) [Function]

key: a key

curve: will hold the curve

digest: will hold the digest

paramset: will hold the GOST parameter set ID

x: will hold the x-coordinate

y: will hold the y-coordinate

k: will hold the private key

This function will export the GOST private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Note: parameters will be stored with least significant byte first. On version 3.6.3 this was incorrectly returned in big-endian format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_x509_privkey_export_pkcs8

`int gnutls_x509_privkey_export_pkcs8` (*gnutls_x509_privkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *const char *** `password`, *unsigned int* `flags`, *void *** `output_data`, *size_t *** `output_data_size`) [Function]

key: Holds the key

format: the format of output params. One of PEM or DER.

password: the password that will be used to encrypt the key.

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

output_data: will contain a private key PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the private key to a PKCS8 structure. Both RSA and DSA keys can be exported. For DSA keys we use PKCS `11` definitions. If the flags do not specify the encryption cipher, then the default 3DES (PBES2) will be used.

The `password` can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN ENCRYPTED PRIVATE KEY" or "BEGIN PRIVATE KEY" if encryption is not used.

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_x509_privkey_export_rsa_raw

`int gnutls_x509_privkey_export_rsa_raw` (*gnutls_x509_privkey_t* `key`, *gnutls_datum_t *** `m`, *gnutls_datum_t *** `e`, *gnutls_datum_t *** `d`, *gnutls_datum_t *** `p`, *gnutls_datum_t *** `q`, *gnutls_datum_t *** `u`) [Function]

key: a key

m: will hold the modulus

e: will hold the public exponent

d: will hold the private exponent

p: will hold the first prime (p)

q: will hold the second prime (q)

u: will hold the coefficient

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_export_rsa_raw2

`int gnutls_x509_privkey_export_rsa_raw2` [Function]
(*gnutls_x509_privkey_t* `key`, *gnutls_datum_t* * `m`, *gnutls_datum_t* * `e`, *gnutls_datum_t* * `d`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `u`, *gnutls_datum_t* * `e1`, *gnutls_datum_t* * `e2`)

key: a key

m: will hold the modulus

e: will hold the public exponent

d: will hold the private exponent

p: will hold the first prime (p)

q: will hold the second prime (q)

u: will hold the coefficient

e1: will hold e1 = d mod (p-1)

e2: will hold e2 = d mod (q-1)

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_x509_privkey_fix

`int gnutls_x509_privkey_fix` (*gnutls_x509_privkey_t* `key`) [Function]

key: a key

This function will recalculate the secondary parameters in a key. In RSA keys, this can be the coefficient and exponent1,2.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_generate

`int gnutls_x509_privkey_generate` (*gnutls_x509_privkey_t* `key`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`) [Function]

key: an initialized key

algo: is one of the algorithms in `gnutls_pk_algorithm_t` .

bits: the size of the parameters to generate

flags: Must be zero or flags from `gnutls_privkey_flags_t` .

This function will generate a random private key. Note that this function must be called on an initialized private key.

The flag `GNUTLS_PRIVKEY_FLAG_PROVABLE` instructs the key generation process to use algorithms like Shawe-Taylor (from FIPS PUB186-4) which generate provable parameters out of a seed for RSA and DSA keys. See `gnutls_x509_privkey_generate2()` for more information.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro. The input to the macro is any curve from `gnutls_ecc_curve_t` .

For DSA keys, if the subgroup size needs to be specified check the `GNUTLS_SUBGROUP_TO_BITS()` macro.

It is recommended to do not set the number of `bits` directly, use `gnutls_sec_param_to_pk_bits()` instead .

See also `gnutls_privkey_generate()` , `gnutls_x509_privkey_generate2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_generate2

`int gnutls_x509_privkey_generate2` (*gnutls_x509_privkey_t* `key`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`, *const gnutls_keygen_data_st ** `data`, *unsigned* `data_size`) [Function]

key: a key

algo: is one of the algorithms in `gnutls_pk_algorithm_t` .

bits: the size of the modulus

flags: Must be zero or flags from `gnutls_privkey_flags_t` .

data: Allow specifying `gnutls_keygen_data_st` types such as the seed to be used.

data_size: The number of `data` available.

This function will generate a random private key. Note that this function must be called on an initialized private key.

The flag `GNUTLS_PRIVKEY_FLAG_PROVABLE` instructs the key generation process to use algorithms like Shawe-Taylor (from FIPS PUB186-4) which generate provable parameters out of a seed for RSA and DSA keys. On DSA keys the PQG parameters are generated using the seed, while on RSA the two primes. To specify an explicit seed (by default a random seed is used), use the `data` with a `GNUTLS_KEYGEN_SEED` type.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

To export the generated keys in memory or in files it is recommended to use the PKCS8 form as it can handle all key types, and can store additional parameters such as the seed, in case of provable RSA or DSA keys. Generated keys can be exported in memory using `gnutls_privkey_export_x509()` , and then with `gnutls_x509_privkey_export2_pkcs8()` .

If key generation is part of your application, avoid setting the number of bits directly, and instead use `gnutls_sec_param_to_pk_bits()` . That way the generated keys will adapt to the security levels of the underlying GnuTLS library.

See also `gnutls_privkey_generate2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_get_key_id

`int gnutls_x509_privkey_get_key_id` (*gnutls_x509_privkey_t* `key`, *unsigned int* `flags`, *unsigned char * * `output_data`, *size_t * `output_data_size`) [Function]

key: a key

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given key.

If the buffer provided is not long enough to hold the output, then * `output_data_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_get_pk_algorithm

`int gnutls_x509_privkey_get_pk_algorithm` (*gnutls_x509_privkey_t* `key`) [Function]

key: should contain a `gnutls_x509_privkey_t` type

This function will return the public key algorithm of a private key.

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

gnutls_x509_privkey_get_pk_algorithm2

`int gnutls_x509_privkey_get_pk_algorithm2` (*gnutls_x509_privkey_t* `key`, *unsigned int * * `bits`) [Function]

key: should contain a `gnutls_x509_privkey_t` type

bits: The number of bits in the public key algorithm

This function will return the public key algorithm of a private key.

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

gnutls_x509_privkey_get_seed

`int gnutls_x509_privkey_get_seed` (*gnutls_x509_privkey_t* `key`, *gnutls_digest_algorithm_t * `digest`, *void * `seed`, *size_t * `seed_size`) [Function]

key: should contain a `gnutls_x509_privkey_t` type

digest: if non-NULL it will contain the digest algorithm used for key generation (if applicable)

seed: where seed will be copied to

seed_size: originally holds the size of `seed` , will be updated with actual size

This function will return the seed that was used to generate the given private key. That function will succeed only if the key was generated as a provable key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_x509_privkey_get_spki

`int gnutls_x509_privkey_get_spki` (*gnutls_x509_privkey_t* `key`, *gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

key: should contain a `gnutls_x509_privkey_t` type

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will return the public key information of a private key. The provided `spki` must be initialized.

Returns: Zero on success, or a negative error code on error.

gnutls_x509_privkey_import

`int gnutls_x509_privkey_import` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t* `format`) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will convert the given DER or PEM encoded key to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` .

If the key is PEM encoded it should have a header that contains "PRIVATE KEY". Note that this function falls back to PKCS `8` decoding without password, if the default format fails to import.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import2

`int gnutls_x509_privkey_import2` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t* `format`, *const char * password*, *unsigned int* `flags`) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded key.

format: One of DER or PEM

password: A password (optional)

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function will import the given DER or PEM encoded key, to the native `gnutls_x509_privkey_t` format, irrespective of the input format. The input format is auto-detected.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

If the provided key is encrypted but no password was given, then `GNUTLS_E_DECRYPTION_FAILED` is returned. Since GnuTLS 3.4.0 this function will utilize the PIN callbacks if any.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import_dsa_raw

`int gnutls_x509_privkey_import_dsa_raw` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * p*, *const gnutls_datum_t * q*, *const gnutls_datum_t * g*, *const gnutls_datum_t * y*, *const gnutls_datum_t * x*) [Function]

key: The data to store the parsed key

p: holds the p

q: holds the q

g: holds the g

y: holds the y

x: holds the x

This function will convert the given DSA raw parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import_ecc_raw

`int gnutls_x509_privkey_import_ecc_raw` (*gnutls_x509_privkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *const gnutls_datum_t * x*, *const gnutls_datum_t * y*, *const gnutls_datum_t * k*) [Function]

key: The data to store the parsed key

curve: holds the curve

x: holds the x-coordinate

y: holds the y-coordinate

k: holds the k

This function will convert the given elliptic curve parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` . For EdDSA keys, the `x` and `k` values must be in the native to curve format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_x509_privkey_import_gost_raw

`int gnutls_x509_privkey_import_gost_raw` [Function]
(*gnutls_x509_privkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *gnutls_digest_algorithm_t* `digest`, *gnutls_gost_paramset_t* `paramset`, *const gnutls_datum_t * x*, *const gnutls_datum_t * y*, *const gnutls_datum_t * k*)

key: The data to store the parsed key

curve: holds the curve

digest: will hold the digest

paramset: will hold the GOST parameter set ID

x: holds the x-coordinate

y: holds the y-coordinate

k: holds the k (private key)

This function will convert the given GOST private key's parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` . `digest` should be one of GNUTLS_DIG_GOSR_94, GNUTLS_DIG_STREEBOG_256 or GNUTLS_DIG_STREEBOG_512. If `paramset` is set to GNUTLS_GOST_PARAMSET_UNKNOWN default one will be selected depending on `digest` .

Note: parameters should be stored with least significant byte first. On version 3.6.3 big-endian format was used incorrectly.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_x509_privkey_import_openssl

`int gnutls_x509_privkey_import_openssl` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * data*, *const char * password*) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded key.

password: the password to decrypt the key (if it is encrypted).

This function will convert the given PEM encrypted to the native gnutls_x509_privkey_t format. The output will be stored in `key` .

The `password` should be in ASCII. If the password is not provided or wrong then `GNUTLS_E_DECRYPTION_FAILED` will be returned.

If the Certificate is PEM encoded it should have a header of "PRIVATE KEY" and the "DEK-Info" header.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import_pkcs8

`int gnutls_x509_privkey_import_pkcs8` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t* `format`, *const char * password*, *unsigned int* `flags`) [Function]

key: The data to store the parsed key

data: The DER or PEM encoded key.

format: One of DER or PEM

password: the password to decrypt the key (if it is encrypted).

flags: 0 if encrypted or GNUTLS_PKCS_PLAIN if not encrypted.

This function will convert the given DER or PEM encoded PKCS8 2.0 encrypted key to the native gnutls_x509_privkey_t format. The output will be stored in `key` . Both RSA and DSA keys can be imported, and flags can only be used to indicate an unencrypted key.

The `password` can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

If the Certificate is PEM encoded it should have a header of "ENCRYPTED PRIVATE KEY", or "PRIVATE KEY". You only need to specify the flags if the key is DER encoded, since in that case the encryption status cannot be auto-detected.

If the `GNUTLS_PKCS_PLAIN` flag is specified and the supplied data are encrypted then `GNUTLS_E_DECRYPTION_FAILED` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import_rsa_raw

`int gnutls_x509_privkey_import_rsa_raw` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * m*, *const gnutls_datum_t * e*, *const gnutls_datum_t * d*, *const gnutls_datum_t * p*, *const gnutls_datum_t * q*, *const gnutls_datum_t * u*) [Function]

key: The data to store the parsed key

m: holds the modulus

e: holds the public exponent

d: holds the private exponent

p: holds the first prime (p)

q: holds the second prime (q)

u: holds the coefficient

This function will convert the given RSA raw parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_import_rsa_raw2

`int gnutls_x509_privkey_import_rsa_raw2` [Function]
(*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t ** `m`, *const gnutls_datum_t ** `e`, *const gnutls_datum_t ** `d`, *const gnutls_datum_t ** `p`, *const gnutls_datum_t ** `q`, *const gnutls_datum_t ** `u`, *const gnutls_datum_t ** `e1`, *const gnutls_datum_t ** `e2`)

key: The data to store the parsed key

m: holds the modulus

e: holds the public exponent

d: holds the private exponent

p: holds the first prime (p)

q: holds the second prime (q)

u: holds the coefficient (optional)

e1: holds e1 = d mod (p-1) (optional)

e2: holds e2 = d mod (q-1) (optional)

This function will convert the given RSA raw parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_init

`int gnutls_x509_privkey_init` (*gnutls_x509_privkey_t ** `key`) [Function]

key: A pointer to the type to be initialized

This function will initialize a private key type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_sec_param

`gnutls_sec_param_t gnutls_x509_privkey_sec_param` [Function]
(*gnutls_x509_privkey_t* `key`)

key: a key

This function will return the security parameter appropriate with this private key.

Returns: On success, a valid security parameter is returned otherwise `GNUTLS_SEC_PARAM_UNKNOWN` is returned.

Since: 2.12.0

gnutls_x509_privkey_set_flags

`void gnutls_x509_privkey_set_flags` (*gnutls_x509_privkey_t* `key`, *unsigned int* `flags`) [Function]

key: A key of type `gnutls_x509_privkey_t`

flags: flags from the `gnutls_privkey_flags`

This function will set flags for the specified private key, after it is generated. Currently this is useful for the `GNUTLS_PRIVKEY_FLAG_EXPORT_COMPAT` to allow exporting a "provable" private key in backwards compatible way.

Since: 3.5.0

gnutls_x509_privkey_set_pin_function

`void gnutls_x509_privkey_set_pin_function` [Function]
(*gnutls_x509_privkey_t* `privkey`, *gnutls_pin_callback_t* `fn`, *void ** `userdata`)

privkey: The certificate structure

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when it is required to access a protected object. This function overrides the global function set using `gnutls_pkcs11_set_pin_function()` .

Note that this callback is used when decrypting a key.

Since: 3.4.0

gnutls_x509_privkey_set_spki

`int gnutls_x509_privkey_set_spki` (*gnutls_x509_privkey_t* `key`, [Function]
const gnutls_x509_spki_t `spki`, *unsigned int* `flags`)

key: should contain a `gnutls_x509_privkey_t` type

spki: a SubjectPublicKeyInfo structure of type `gnutls_x509_spki_t`

flags: must be zero

This function will return the public key information of a private key. The provided `spki` must be initialized.

Returns: Zero on success, or a negative error code on error.

gnutls_x509_privkey_sign_data

`int gnutls_x509_privkey_sign_data` (*gnutls_x509_privkey_t* `key`, [Function]
gnutls_digest_algorithm_t `digest`, *unsigned int* `flags`, *const gnutls_datum_t ** `data`, *void ** `signature`, *size_t ** `signature_size`)

key: a key

digest: should be a digest algorithm

flags: should be 0 for now

data: holds the data to be signed

signature: will contain the signature

signature_size: holds the size of signature (and will be replaced by the new size)

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-1 for the DSA keys.

If the buffer provided is not long enough to hold the output, then * `signature_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Use `gnutls_x509_crt_get_preferred_hash_algorithm()` to determine the hash algorithm.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_verify_params

`int gnutls_x509_privkey_verify_params (`*gnutls_x509_privkey_t* `key`) [Function]

key: a key

This function will verify the private key parameters.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_privkey_verify_seed

`int gnutls_x509_privkey_verify_seed (`*gnutls_x509_privkey_t* `key`, *gnutls_digest_algorithm_t* `digest`, *const void * * `seed`, *size_t* `seed_size`) [Function]

key: should contain a `gnutls_x509_privkey_t` type

digest: it contains the digest algorithm used for key generation (if applicable)

seed: the seed of the key to be checked with

seed_size: holds the size of `seed`

This function will verify that the given private key was generated from the provided seed. If `seed` is `NULL` then the seed stored in the `key` 's structure will be used for verification.

Returns: In case of a verification failure `GNUTLS_E_PRIVKEY_VERIFICATION_ERROR` is returned, and zero or positive code on success.

Since: 3.5.0

gnutls_x509_rdn_get

`int gnutls_x509_rdn_get (`*const gnutls_datum_t * `idn`, *char * `buf`, *size_t * `buf_size`) [Function]

idn: should contain a DER encoded RDN sequence

buf: a pointer to a structure to hold the peer's name

buf_size: holds the size of `buf`

This function will return the name of the given RDN sequence. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_x509_rdn_get2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `buf_size` is updated if the provided buffer is not long enough, otherwise a negative error value.

gnutls_x509_rdn_get2

`int gnutls_x509_rdn_get2` (*const gnutls_datum_t * idn, gnutls_datum_t * str, unsigned flags*) [Function]

idn: should contain a DER encoded RDN sequence

str: a datum that will hold the name

flags: zero of `GNUTLS_X509_DN_FLAG_COMPAT`

This function will return the name of the given RDN sequence. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC4514.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `buf_size` is updated if the provided buffer is not long enough, otherwise a negative error value.

gnutls_x509_rdn_get_by_oid

`int gnutls_x509_rdn_get_by_oid` (*const gnutls_datum_t * idn, const char * oid, unsigned indx, unsigned int raw_flag, void * buf, size_t * buf_size*) [Function]

idn: should contain a DER encoded RDN sequence

oid: an Object Identifier

indx: In case multiple same OIDs exist in the RDN indicates which to send. Use 0 for the first one.

raw_flag: If non-zero then the raw DER data are returned.

buf: a pointer to a structure to hold the peer's name

buf_size: holds the size of `buf`

This function will return the name of the given Object identifier, of the RDN sequence. The name will be encoded using the rules from RFC4514.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `buf_size` is updated if the provided buffer is not long enough, otherwise a negative error value.

gnutls_x509_rdn_get_oid

`int gnutls_x509_rdn_get_oid` (*const gnutls_datum_t * idn, unsigned indx, void * buf, size_t * buf_size*) [Function]

idn: should contain a DER encoded RDN sequence

indx: Indicates which OID to return. Use 0 for the first one.

buf: a pointer to a structure to hold the peer's name OID

buf_size: holds the size of `buf`

This function will return the specified Object identifier, of the RDN sequence.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `buf_size` is updated if the provided buffer is not long enough, otherwise a negative error value.

Since: 2.4.0

gnutls_x509_spki_deinit

`void gnutls_x509_spki_deinit (`*gnutls_x509_spki_t* `spki``)` [Function]

spki: the SubjectPublicKeyInfo structure

This function will deinitialize a SubjectPublicKeyInfo structure.

Since: 3.6.0

gnutls_x509_spki_get_rsa_pss_params

`int gnutls_x509_spki_get_rsa_pss_params (`*gnutls_x509_spki_t* `spki`, *gnutls_digest_algorithm_t* * `dig`, *unsigned int* * `salt_size``)` [Function]

spki: the SubjectPublicKeyInfo structure

dig: if non-NULL, it will hold the digest algorithm

salt_size: if non-NULL, it will hold the salt size

This function will get the public key algorithm parameters of RSA-PSS type.

Returns: zero if the parameters are present or a negative value on error.

Since: 3.6.0

gnutls_x509_spki_init

`int gnutls_x509_spki_init (`*gnutls_x509_spki_t* * `spki``)` [Function]

spki: A pointer to the type to be initialized

This function will initialize a SubjectPublicKeyInfo structure used in PKIX. The structure is used to set additional parameters in the public key information field of a certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_x509_spki_set_rsa_pss_params

`void gnutls_x509_spki_set_rsa_pss_params (`*gnutls_x509_spki_t* `spki`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `salt_size``)` [Function]

spki: the SubjectPublicKeyInfo structure

dig: a digest algorithm of type `gnutls_digest_algorithm_t`

salt_size: the size of salt string

This function will set the public key parameters for an RSA-PSS algorithm, in the SubjectPublicKeyInfo structure.

Since: 3.6.0

gnutls_x509_tlsfeatures_add

`int gnutls_x509_tlsfeatures_add (`*gnutls_x509_tlsfeatures_t* `f`, *unsigned int* `feature``)` [Function]

f: The TLS features

feature: The feature to add

This function will append a feature to the X.509 TLS features extension structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_tlsfeatures_check_crt

`unsigned gnutls_x509_tlsfeatures_check_crt` [Function]
`(`*gnutls_x509_tlsfeatures_t* `feat`, *gnutls_x509_crt_t* `cert`)

feat: a set of TLSFeatures

cert: the certificate to be checked

This function will check the provided certificate against the TLSFeatures set in `feat` using the RFC7633 p.4.2.2 rules. It will check whether the certificate contains the features in `feat` or a superset.

Returns: non-zero if the provided certificate complies, and zero otherwise.

Since: 3.5.1

gnutls_x509_tlsfeatures_deinit

`void gnutls_x509_tlsfeatures_deinit (`*gnutls_x509_tlsfeatures_t* `f`[Function]`)`

f: The TLS features

This function will deinitialize a X.509 TLS features extension structure

Since: 3.5.1

gnutls_x509_tlsfeatures_get

`int gnutls_x509_tlsfeatures_get (`*gnutls_x509_tlsfeatures_t* `f`, [Function]
unsigned `idx`, *unsigned int* * `feature`)

f: The TLS features

idx: The index of the feature to get

feature: If the function succeeds, the feature will be stored in this variable

This function will get a feature from the X.509 TLS features extension structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_tlsfeatures_init

`int gnutls_x509_tlsfeatures_init (`*gnutls_x509_tlsfeatures_t * f*`)` [Function]

f: The TLS features

This function will initialize a X.509 TLS features extension structure

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.1

gnutls_x509_trust_list_add_cas

`int gnutls_x509_trust_list_add_cas (`*gnutls_x509_trust_list_t list, const gnutls_x509_crt_t * clist, unsigned clist_size, unsigned int flags*`)` [Function]

list: The list

clist: A list of CAs

clist_size: The length of the CA list

flags: flags from `gnutls_trust_list_flags_t`

This function will add the given certificate authorities to the trusted list. The CAs in `clist` must not be deinitialized during the lifetime of `list` .

If the flag `GNUTLS_TL_NO_DUPLICATES` is specified, then this function will ensure that no duplicates will be present in the final trust list.

If the flag `GNUTLS_TL_NO_DUPLICATE_KEY` is specified, then this function will ensure that no certificates with the same key are present in the final trust list.

If either `GNUTLS_TL_NO_DUPLICATE_KEY` or `GNUTLS_TL_NO_DUPLICATES` are given, `gnutls_x509_trust_list_deinit()` must be called with parameter `all` being 1.

Returns: The number of added elements is returned; that includes duplicate entries.

Since: 3.0.0

gnutls_x509_trust_list_add_crls

`int gnutls_x509_trust_list_add_crls (`*gnutls_x509_trust_list_t list, const gnutls_x509_crl_t * crl_list, unsigned crl_size, unsigned int flags, unsigned int verification_flags*`)` [Function]

list: The list

crl_list: A list of CRLs

crl_size: The length of the CRL list

flags: flags from `gnutls_trust_list_flags_t`

verification_flags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate revocation lists to the trusted list. The CRLs in `crl_list` must not be deinitialized during the lifetime of `list` .

This function must be called after `gnutls_x509_trust_list_add_cas()` to allow verifying the CRLs for validity. If the flag `GNUTLS_TL_NO_DUPLICATES` is given, then the final CRL list will not contain duplicate entries.

If the flag GNUTLS_TL_NO_DUPLICATES is given, gnutls_x509_trust_list_deinit() must be called with parameter all being 1.

If flag GNUTLS_TL_VERIFY_CRL is given the CRLs will be verified before being added, and if verification fails, they will be skipped.

Returns: The number of added elements is returned; that includes duplicate entries.

Since: 3.0

gnutls_x509_trust_list_add_named_crt

int gnutls_x509_trust_list_add_named_crt [Function]
(*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* `cert`, *const void ** `name`, *size_t* `name_size`, *unsigned int* `flags`)

list: The list

cert: A certificate

name: An identifier for the certificate

name_size: The size of the identifier

flags: should be 0.

This function will add the given certificate to the trusted list and associate it with a name. The certificate will not be be used for verification with gnutls_x509_trust_list_verify_crt() but with gnutls_x509_trust_list_verify_named_crt() or gnutls_x509_trust_list_verify_crt2() - the latter only since GnuTLS 3.4.0 and if a hostname is provided.

In principle this function can be used to set individual "server" certificates that are trusted by the user for that specific server but for no other purposes.

The certificate cert must not be deinitialized during the lifetime of the list .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.0.0

gnutls_x509_trust_list_add_system_trust

int gnutls_x509_trust_list_add_system_trust [Function]
(*gnutls_x509_trust_list_t* `list`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The structure of the list

tl_flags: GNUTLS_TL_*

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function adds the system's default trusted certificate authorities to the trusted list. Note that on unsupported systems this function returns GNUTLS_E_UNIMPLEMENTED_FEATURE .

This function implies the flag GNUTLS_TL_NO_DUPLICATES .

Returns: The number of added elements or a negative error code on error.

Since: 3.1

gnutls_x509_trust_list_add_trust_dir

`int gnutls_x509_trust_list_add_trust_dir` [Function]
(*gnutls_x509_trust_list_t* `list`, *const char ** `ca_dir`, *const char ** `crl_dir`, *gnutls_x509_crt_fmt_t* `type`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The list

ca_dir: A directory containing the CAs (optional)

crl_dir: A directory containing a list of CRLs (optional)

type: The format of the certificates

tl_flags: flags from `gnutls_trust_list_flags_t`

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate authorities to the trusted list. Only directories are accepted by this function.

Returns: The number of added elements is returned.

Since: 3.3.6

gnutls_x509_trust_list_add_trust_file

`int gnutls_x509_trust_list_add_trust_file` [Function]
(*gnutls_x509_trust_list_t* `list`, *const char ** `ca_file`, *const char ** `crl_file`, *gnutls_x509_crt_fmt_t* `type`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The list

ca_file: A file containing a list of CAs (optional)

crl_file: A file containing a list of CRLs (optional)

type: The format of the certificates

tl_flags: flags from `gnutls_trust_list_flags_t`

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate authorities to the trusted list. PKCS `11` URLs are also accepted, instead of files, by this function. A PKCS `11` URL implies a trust database (a specially marked module in p11-kit); the URL `"pkcs11:"` implies all trust databases in the system. Only a single URL specifying trust databases can be set; they cannot be stacked with multiple calls.

Returns: The number of added elements is returned.

Since: 3.1

gnutls_x509_trust_list_add_trust_mem

`int gnutls_x509_trust_list_add_trust_mem` [Function]
(*gnutls_x509_trust_list_t* `list`, *const gnutls_datum_t ** `cas`, *const gnutls_datum_t ** `crls`, *gnutls_x509_crt_fmt_t* `type`, *unsigned int* `tl_flags`, *unsigned int* `tl_vflags`)

list: The list

cas: A buffer containing a list of CAs (optional)

crls: A buffer containing a list of CRLs (optional)

type: The format of the certificates

tl_flags: flags from `gnutls_trust_list_flags_t`

tl_vflags: gnutls_certificate_verify_flags if flags specifies GNUTLS_TL_VERIFY_CRL

This function will add the given certificate authorities to the trusted list.

If this function is used `gnutls_x509_trust_list_deinit()` must be called with parameter `all` being 1.

Returns: The number of added elements is returned.

Since: 3.1

gnutls_x509_trust_list_deinit

`void gnutls_x509_trust_list_deinit (`*gnutls_x509_trust_list_t* `list`, *unsigned int* `all``)` [Function]

list: The list to be deinitialized

all: if non-zero it will deinitialize all the certificates and CRLs contained in the structure.

This function will deinitialize a trust list. Note that the `all` flag should be typically non-zero unless you have specified your certificates using `gnutls_x509_trust_list_add_cas()` and you want to prevent them from being deinitialized by this function.

Since: 3.0.0

gnutls_x509_trust_list_get_issuer

`int gnutls_x509_trust_list_get_issuer (`*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t ** `issuer`, *unsigned int* `flags``)` [Function]

list: The list

cert: is the certificate to find issuer for

issuer: Will hold the issuer if any. Should be treated as constant.

flags: flags from `gnutls_trust_list_flags_t` (`GNUTLS_TL_GET_COPY` is applicable)

This function will find the issuer of the given certificate. If the flag `GNUTLS_TL_GET_COPY` is specified a copy of the issuer will be returned which must be freed using `gnutls_x509_crt_deinit()` . In that case the provided `issuer` must not be initialized.

Note that the flag `GNUTLS_TL_GET_COPY` is required for this function to work with PKCS11 trust lists in a thread-safe way.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_x509_trust_list_get_issuer_by_dn

`int gnutls_x509_trust_list_get_issuer_by_dn` [Function]
(*gnutls_x509_trust_list_t* `list`, *const gnutls_datum_t ** `dn`, *gnutls_x509_crt_t ** `issuer`, *unsigned int* `flags`)

list: The list

dn: is the issuer's DN

issuer: Will hold the issuer if any. Should be deallocated after use.

flags: Use zero

This function will find the issuer with the given name, and return a copy of the issuer, which must be freed using `gnutls_x509_crt_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_x509_trust_list_get_issuer_by_subject_key_id

`int gnutls_x509_trust_list_get_issuer_by_subject_key_id` [Function]
(*gnutls_x509_trust_list_t* `list`, *const gnutls_datum_t ** `dn`, *const gnutls_datum_t ** `spki`, *gnutls_x509_crt_t ** `issuer`, *unsigned int* `flags`)

list: The list

dn: is the issuer's DN (may be `NULL`)

spki: is the subject key ID

issuer: Will hold the issuer if any. Should be deallocated after use.

flags: Use zero

This function will find the issuer with the given name and subject key ID, and return a copy of the issuer, which must be freed using `gnutls_x509_crt_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.2

gnutls_x509_trust_list_init

`int gnutls_x509_trust_list_init` (*gnutls_x509_trust_list_t ** `list`, *unsigned int* `size`) [Function]

list: A pointer to the type to be initialized

size: The size of the internal hash table. Use (0) for default size.

This function will initialize an X.509 trust list structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0.0

gnutls_x509_trust_list_iter_deinit

`void gnutls_x509_trust_list_iter_deinit` [Function]
(*gnutls_x509_trust_list_iter_t* `iter`)

iter: The iterator structure to be deinitialized

This function will deinitialize an iterator structure.

Since: 3.4.0

gnutls_x509_trust_list_iter_get_ca

`int gnutls_x509_trust_list_iter_get_ca` [Function]
(*gnutls_x509_trust_list_t* `list`, *gnutls_x509_trust_list_iter_t ** `iter`, *gnutls_x509_crt_t ** `crt`)

list: The list

iter: A pointer to an iterator (initially the iterator should be `NULL`)

crt: where the certificate will be copied

This function obtains a certificate in the trust list and advances the iterator to the next certificate. The certificate returned in `crt` must be deallocated with `gnutls_x509_crt_deinit()` .

When past the last element is accessed `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned and the iterator is reset.

The iterator is deinitialized and reset to `NULL` automatically by this function after iterating through all elements until `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned. If the iteration is aborted early, it must be manually deinitialized using `gnutls_x509_trust_list_iter_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_x509_trust_list_remove_cas

`int gnutls_x509_trust_list_remove_cas` (*gnutls_x509_trust_list_t* `list`, *const gnutls_x509_crt_t ** `clist`, *unsigned* `clist_size`) [Function]

list: The list

clist: A list of CAs

clist_size: The length of the CA list

This function will remove the given certificate authorities from the trusted list.

Note that this function can accept certificates and authorities not yet known. In that case they will be kept in a separate black list that will be used during certificate verification. Unlike `gnutls_x509_trust_list_add_cas()` there is no deinitialization restriction for certificate list provided in this function.

Returns: The number of removed elements is returned.

Since: 3.1.10

gnutls_x509_trust_list_remove_trust_file

`int gnutls_x509_trust_list_remove_trust_file` [Function]
(*gnutls_x509_trust_list_t* `list`, *const char ** `ca_file`, *gnutls_x509_crt_fmt_t* `type`)

list: The list

ca_file: A file containing a list of CAs

type: The format of the certificates

This function will remove the given certificate authorities from the trusted list, and add them into a black list when needed. PKCS 11 URLs are also accepted, instead of files, by this function.

See also `gnutls_x509_trust_list_remove_cas()` .

Returns: The number of added elements is returned.

Since: 3.1.10

gnutls_x509_trust_list_remove_trust_mem

`int gnutls_x509_trust_list_remove_trust_mem` [Function]
(*gnutls_x509_trust_list_t* `list`, *const gnutls_datum_t ** `cas`, *gnutls_x509_crt_fmt_t* `type`)

list: The list

cas: A buffer containing a list of CAs (optional)

type: The format of the certificates

This function will remove the provided certificate authorities from the trusted list, and add them into a black list when needed.

See also `gnutls_x509_trust_list_remove_cas()` .

Returns: The number of removed elements is returned.

Since: 3.1.10

gnutls_x509_trust_list_verify_crt

`int gnutls_x509_trust_list_verify_crt` (*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t ** `cert_list`, *unsigned int* `cert_list_size`, *unsigned int* `flags`, *unsigned int ** `voutput`, *gnutls_verify_output_function* `func`) [Function]

list: The list

cert_list: is the certificate list to be verified

cert_list_size: is the certificate list size

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `voutput` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

The details of the verification are the same as in `gnutls_x509_trust_list_verify_crt2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_x509_trust_list_verify_crt2

`int gnutls_x509_trust_list_verify_crt2` [Function]
(*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* * `cert_list`, *unsigned int* `cert_list_size`, *gnutls_typed_vdata_st* * `data`, *unsigned int* `elements`, *unsigned int* `flags`, *unsigned int* * `voutput`, *gnutls_verify_output_function* `func`)

list: The list

cert_list: is the certificate list to be verified

cert_list_size: is the certificate list size

data: an array of typed data

elements: the number of data elements

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will attempt to verify the given certificate chain and return its status. The `voutput` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

When a certificate chain of `cert_list_size` with more than one certificates is provided, the verification status will apply to the first certificate in the chain that failed verification. The verification process starts from the end of the chain (from CA to end certificate). The first certificate in the chain must be the end-certificate while the rest of the members may be sorted or not.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

Additional verification parameters are possible via the `data` types; the acceptable types are `GNUTLS_DT_DNS_HOSTNAME` , `GNUTLS_DT_IP_ADDRESS` and `GNUTLS_DT_KEY_PURPOSE_OID` . The former accepts as data a null-terminated hostname, and the latter a null-terminated object identifier (e.g., `GNUTLS_KP_TLS_WWW_SERVER`). If a DNS hostname is provided then this function will compare the hostname in the end certificate against the given. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set. In addition it will consider certificates provided with `gnutls_x509_trust_list_add_named_crt()` .

If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to match the provided OID or be marked

for any purpose, otherwise verification will fail with GNUTLS_CERT_PURPOSE_MISMATCH status.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value. Note that verification failure will not result to an error code, only voutput will be updated.

Since: 3.3.8

gnutls_x509_trust_list_verify_named_crt

int gnutls_x509_trust_list_verify_named_crt (*gnutls_x509_trust_list_t* list, *gnutls_x509_crt_t* cert, *const void* * name, *size_t* name_size, *unsigned int* flags, *unsigned int* * voutput, *gnutls_verify_output_function* func) [Function]

list: The list

cert: is the certificate to be verified

name: is the certificate's name

name_size: is the certificate's name size

flags: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.

voutput: will hold the certificate verification output.

func: If non-null will be called on each chain element verification with the output.

This function will try to find a certificate that is associated with the provided name – see gnutls_x509_trust_list_add_named_crt() . If a match is found the certificate is considered valid. In addition to that this function will also check CRLs. The voutput parameter will hold an OR'ed sequence of gnutls_certificate_status_t flags.

Additionally a certificate verification profile can be specified from the ones in gnutls_certificate_verification_profiles_t by ORing the result of GNUTLS_PROFILE_TO_VFLAGS() to the verification flags.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.0.0

E.4 PKCS 7 API

The following functions are to be used for PKCS 7 structures handling. Their prototypes lie in gnutls/pkcs7.h.

gnutls_pkcs7_add_attr

int gnutls_pkcs7_add_attr (*gnutls_pkcs7_attrs_t* * list, *const char* * oid, *gnutls_datum_t* * data, *unsigned* flags) [Function]

list: A list of existing attributes or pointer to NULL for the first one

oid: the OID of the attribute to be set

data: the raw (DER-encoded) data of the attribute to be set

flags: zero or `GNUTLS_PKCS7_ATTR_ENCODE_OCTET_STRING`

This function will set a PKCS 7 attribute in the provided list. If this function fails, the previous list would be deallocated.

Note that any attributes set with this function must either be DER or BER encoded, unless a special flag is present.

Returns: On success, the new list head, otherwise `NULL` .

Since: 3.4.2

gnutls_pkcs7_attrs_deinit

`void gnutls_pkcs7_attrs_deinit` (*gnutls_pkcs7_attrs_t* `list`) [Function]

list: A list of existing attributes

This function will clear a PKCS 7 attribute list.

Since: 3.4.2

gnutls_pkcs7_deinit

`void gnutls_pkcs7_deinit` (*gnutls_pkcs7_t* `pkcs7`) [Function]

pkcs7: the type to be deinitialized

This function will deinitialize a PKCS7 type.

gnutls_pkcs7_delete_crl

`int gnutls_pkcs7_delete_crl` (*gnutls_pkcs7_t* `pkcs7`, *int* `indx`) [Function]

pkcs7: The pkcs7 type

indx: the index of the crl to delete

This function will delete a crl from a PKCS7 or RFC2630 crl set. Index starts from 0. Returns 0 on success.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_delete_crt

`int gnutls_pkcs7_delete_crt` (*gnutls_pkcs7_t* `pkcs7`, *int* `indx`) [Function]

pkcs7: The pkcs7 type

indx: the index of the certificate to delete

This function will delete a certificate from a PKCS7 or RFC2630 certificate set. Index starts from 0. Returns 0 on success.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_export

`int gnutls_pkcs7_export` (*gnutls_pkcs7_t* `pkcs7`, *gnutls_x509_crt_fmt_t* `format`, *void * * `output_data`, *size_t * `output_data_size`) [Function]

pkcs7: The pkcs7 type

format: the format of output params. One of PEM or DER.

output_data: will contain a structure PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the pkcs7 structure to DER or PEM format.

If the buffer provided is not long enough to hold the output, then * `output_data_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_export2

`int gnutls_pkcs7_export2` (*gnutls_pkcs7_t* `pkcs7`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t* * `out`) [Function]

pkcs7: The pkcs7 type

format: the format of output params. One of PEM or DER.

out: will contain a structure PEM or DER encoded

This function will export the pkcs7 structure to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.3

gnutls_pkcs7_get_attr

`int gnutls_pkcs7_get_attr` (*gnutls_pkcs7_attrs_t* `list`, *unsigned* `idx`, *char ** `oid`, *gnutls_datum_t* * `data`, *unsigned* `flags`) [Function]

list: A list of existing attributes or `NULL` for the first one

idx: the index of the attribute to get

oid: the OID of the attribute (read-only)

data: the raw data of the attribute

flags: zero or `GNUTLS_PKCS7_ATTR_ENCODE_OCTET_STRING`

This function will get a PKCS 7 attribute from the provided list. The OID is a constant string, but data will be allocated and must be deinitialized by the caller.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned if there are no data in the current index.

Since: 3.4.2

gnutls_pkcs7_get_crl_count

`int gnutls_pkcs7_get_crl_count (`*gnutls_pkcs7_t* `pkcs7`*`)`* [Function]

pkcs7: The pkcs7 type

This function will return the number of certificates in the PKCS7 or RFC2630 crl set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_get_crl_raw

`int gnutls_pkcs7_get_crl_raw (`*gnutls_pkcs7_t* `pkcs7`*, unsigned* `indx`*, void * crl, size_t * crl_size`*`)`* [Function]

pkcs7: The pkcs7 type

indx: contains the index of the crl to extract

crl: the contents of the crl will be copied there (may be null)

crl_size: should hold the size of the crl

This function will return a crl of the PKCS7 or RFC2630 crl set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. If the provided buffer is not long enough, then `crl_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned. After the last crl has been read `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_pkcs7_get_crl_raw2

`int gnutls_pkcs7_get_crl_raw2 (`*gnutls_pkcs7_t* `pkcs7`*, unsigned* `indx`*, gnutls_datum_t * crl`*`)`* [Function]

pkcs7: The pkcs7 type

indx: contains the index of the crl to extract

crl: will contain the contents of the CRL in an allocated buffer

This function will return a DER encoded CRL of the PKCS7 or RFC2630 crl set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. After the last crl has been read `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Since: 3.4.2

gnutls_pkcs7_get_crt_count

`int gnutls_pkcs7_get_crt_count (`*gnutls_pkcs7_t* `pkcs7`*`)`* [Function]

pkcs7: should contain a `gnutls_pkcs7_t` type

This function will return the number of certificates in the PKCS7 or RFC2630 certificate set.

Returns: On success, a positive number is returned, otherwise a negative error value.

gnutls_pkcs7_get_crt_raw

`int gnutls_pkcs7_get_crt_raw (`*gnutls_pkcs7_t* `pkcs7`, *unsigned* `indx`, *void ** `certificate`, *size_t ** `certificate_size`)` [Function]

pkcs7: should contain a gnutls_pkcs7_t type

indx: contains the index of the certificate to extract

certificate: the contents of the certificate will be copied there (may be null)

certificate_size: should hold the size of the certificate

This function will return a certificate of the PKCS7 or RFC2630 certificate set.

After the last certificate has been read `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. If the provided buffer is not long enough, then `certificate_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned.

gnutls_pkcs7_get_crt_raw2

`int gnutls_pkcs7_get_crt_raw2 (`*gnutls_pkcs7_t* `pkcs7`, *unsigned* `indx`, *gnutls_datum_t ** `cert`)` [Function]

pkcs7: should contain a gnutls_pkcs7_t type

indx: contains the index of the certificate to extract

cert: will hold the contents of the certificate; must be deallocated with `gnutls_free()`

This function will return a certificate of the PKCS7 or RFC2630 certificate set.

After the last certificate has been read `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. If the provided buffer is not long enough, then `certificate_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned.

Since: 3.4.2

gnutls_pkcs7_get_embedded_data

`int gnutls_pkcs7_get_embedded_data (`*gnutls_pkcs7_t* `pkcs7`, *unsigned* `flags`, *gnutls_datum_t ** `data`)` [Function]

pkcs7: should contain a gnutls_pkcs7_t type

flags: must be zero or `GNUTLS_PKCS7_EDATA_GET_RAW`

data: will hold the embedded data in the provided structure

This function will return the data embedded in the signature of the PKCS7 structure. If no data are available then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

The returned data must be de-allocated using `gnutls_free()` .

Note, that this function returns the exact same data that are authenticated. If the `GNUTLS_PKCS7_EDATA_GET_RAW` flag is provided, the returned data will be including the wrapping tag/value as they are encoded in the structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.8

gnutls_pkcs7_get_embedded_data_oid

`const char * gnutls_pkcs7_get_embedded_data_oid` [Function]
(*gnutls_pkcs7_t pkcs7*)

pkcs7: should contain a gnutls_pkcs7_t type

This function will return the OID of the data embedded in the signature of the PKCS7 structure. If no data are available then `NULL` will be returned. The returned value will be valid during the lifetime of the `pkcs7` structure.

Returns: On success, a pointer to an OID string, `NULL` on error.

Since: 3.5.5

gnutls_pkcs7_get_signature_count

`int gnutls_pkcs7_get_signature_count` (*gnutls_pkcs7_t pkcs7*) [Function]

pkcs7: should contain a `gnutls_pkcs7_t` type

This function will return the number of signatures in the PKCS7 structure.

Returns: On success, a positive number is returned, otherwise a negative error value.

Since: 3.4.3

gnutls_pkcs7_get_signature_info

`int gnutls_pkcs7_get_signature_info` (*gnutls_pkcs7_t pkcs7, unsigned idx, gnutls_pkcs7_signature_info_st * info*) [Function]

pkcs7: should contain a `gnutls_pkcs7_t` type

idx: the index of the signature info to check

info: will contain the output signature

This function will return information about the signature identified by idx in the provided PKCS 7 structure. The information should be deinitialized using `gnutls_pkcs7_signature_info_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.2

gnutls_pkcs7_import

`int gnutls_pkcs7_import` (*gnutls_pkcs7_t pkcs7, const gnutls_datum_t * data, gnutls_x509_crt_fmt_t format*) [Function]

pkcs7: The data to store the parsed PKCS7.

data: The DER or PEM encoded PKCS7.

format: One of DER or PEM

This function will convert the given DER or PEM encoded PKCS7 to the native `gnutls_pkcs7_t` format. The output will be stored in `pkcs7` .

If the PKCS7 is PEM encoded it should have a header of "PKCS7".

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_pkcs7_init

`int gnutls_pkcs7_init` (*gnutls_pkcs7_t * pkcs7*) [Function]

pkcs7: A pointer to the type to be initialized

This function will initialize a PKCS7 structure. PKCS7 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_pkcs7_print

`int gnutls_pkcs7_print` (*gnutls_pkcs7_t pkcs7*, *gnutls_certificate_print_formats_t format*, *gnutls_datum_t * out*) [Function]

pkcs7: The PKCS7 struct to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print a signed PKCS 7 structure, suitable for display to a human.

Currently the supported formats are GNUTLS_CRT_PRINT_FULL and GNUTLS_CRT_PRINT_COMPACT .

The output out needs to be deallocated using gnutls_free() .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_pkcs7_print_signature_info

`int gnutls_pkcs7_print_signature_info` (*gnutls_pkcs7_signature_info_st * info*, *gnutls_certificate_print_formats_t format*, *gnutls_datum_t * out*) [Function]

info: The PKCS7 signature info struct to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print a PKCS 7 signature info structure, suitable for display to a human.

Currently the supported formats are GNUTLS_CRT_PRINT_FULL and GNUTLS_CRT_PRINT_COMPACT .

The output out needs to be deallocated using gnutls_free() .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.6.14

gnutls_pkcs7_set_crl

`int gnutls_pkcs7_set_crl` (*gnutls_pkcs7_t* `pkcs7`, *gnutls_x509_crl_t* `crl`) [Function]

pkcs7: The pkcs7 type

crl: the DER encoded crl to be added

This function will add a parsed CRL to the PKCS7 or RFC2630 crl set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_set_crl_raw

`int gnutls_pkcs7_set_crl_raw` (*gnutls_pkcs7_t* `pkcs7`, *const gnutls_datum_t* * `crl`) [Function]

pkcs7: The pkcs7 type

crl: the DER encoded crl to be added

This function will add a crl to the PKCS7 or RFC2630 crl set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_set_crt

`int gnutls_pkcs7_set_crt` (*gnutls_pkcs7_t* `pkcs7`, *gnutls_x509_crt_t* `crt`) [Function]

pkcs7: The pkcs7 type

crt: the certificate to be copied.

This function will add a parsed certificate to the PKCS7 or RFC2630 certificate set. This is a wrapper function over `gnutls_pkcs7_set_crt_raw()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_set_crt_raw

`int gnutls_pkcs7_set_crt_raw` (*gnutls_pkcs7_t* `pkcs7`, *const gnutls_datum_t* * `crt`) [Function]

pkcs7: The pkcs7 type

crt: the DER encoded certificate to be added

This function will add a certificate to the PKCS7 or RFC2630 certificate set.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs7_sign

`int gnutls_pkcs7_sign` (*gnutls_pkcs7_t* `pkcs7`, *gnutls_x509_crt_t* `signer`, *gnutls_privkey_t* `signer_key`, *const gnutls_datum_t* * `data`, *gnutls_pkcs7_attrs_t* `signed_attrs`, *gnutls_pkcs7_attrs_t* `unsigned_attrs`, *gnutls_digest_algorithm_t* `dig`, *unsigned* `flags`) [Function]

pkcs7: should contain a `gnutls_pkcs7_t` type

signer: the certificate to sign the structure

signer_key: the key to sign the structure

data: The data to be signed or `NULL` if the data are already embedded

signed_attrs: Any additional attributes to be included in the signed ones (or `NULL`)

unsigned_attrs: Any additional attributes to be included in the unsigned ones (or `NULL`)

dig: The digest algorithm to use for signing

flags: Should be zero or one of `GNUTLS_PKCS7` flags

This function will add a signature in the provided PKCS `7` structure for the provided data. Multiple signatures can be made with different signers.

The available flags are: `GNUTLS_PKCS7_EMBED_DATA` , `GNUTLS_PKCS7_INCLUDE_TIME` , `GNUTLS_PKCS7_INCLUDE_CERT` , and `GNUTLS_PKCS7_WRITE_SPKI` . They are explained in the `gnutls_pkcs7_sign_flags` definition.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.2

gnutls_pkcs7_signature_info_deinit

`void gnutls_pkcs7_signature_info_deinit` [Function]
(*gnutls_pkcs7_signature_info_st * info*)

info: should point to a `gnutls_pkcs7_signature_info_st` structure

This function will deinitialize any allocated value in the provided `gnutls_pkcs7_signature_info_st` .

Since: 3.4.2

gnutls_pkcs7_verify

`int gnutls_pkcs7_verify` (*gnutls_pkcs7_t pkcs7*, [Function]
gnutls_x509_trust_list_t tl, *gnutls_typed_vdata_st * vdata*, *unsigned int vdata_size*, *unsigned idx*, *const gnutls_datum_t * data*, *unsigned flags*)

pkcs7: should contain a `gnutls_pkcs7_t` type

tl: A list of trusted certificates

vdata: an array of typed data

vdata_size: the number of data elements

idx: the index of the signature info to check

data: The data to be verified or `NULL`

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

This function will verify the provided data against the signature present in the SignedData of the PKCS `7` structure. If the data provided are NULL then the data in the encapsulatedContent field will be used instead.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. A verification error results to a `GNUTLS_E_PK_SIG_VERIFY_FAILED` and the

lack of encapsulated data to verify to a GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE .

Since: 3.4.2

gnutls_pkcs7_verify_direct

int gnutls_pkcs7_verify_direct (*gnutls_pkcs7_t* pkcs7, *gnutls_x509_crt_t* signer, *unsigned* idx, *const gnutls_datum_t* * data, *unsigned* flags) [Function]

pkcs7: should contain a gnutls_pkcs7_t type

signer: the certificate believed to have signed the structure

idx: the index of the signature info to check

data: The data to be verified or NULL

flags: Zero or an OR list of gnutls_certificate_verify_flags

This function will verify the provided data against the signature present in the SignedData of the PKCS 7 structure. If the data provided are NULL then the data in the encapsulatedContent field will be used instead.

Note that, unlike gnutls_pkcs7_verify() this function does not verify the key purpose of the signer. It is expected for the caller to verify the intended purpose of the signer -e.g., via gnutls_x509_crt_get_key_purpose_oid() , or gnutls_x509_crt_check_key_purpose() .

Note also, that since GnuTLS 3.5.6 this function introduces checks in the end certificate (signer), including time checks and key usage checks.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value. A verification error results to a GNUTLS_E_PK_SIG_VERIFY_FAILED and the lack of encapsulated data to verify to a GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE .

Since: 3.4.2

E.5 OCSP API

The following functions are for OCSP certificate status checking. Their prototypes lie in gnutls/ocsp.h.

gnutls_ocsp_req_add_cert

int gnutls_ocsp_req_add_cert (*gnutls_ocsp_req_t* req, *gnutls_digest_algorithm_t* digest, *gnutls_x509_crt_t* issuer, *gnutls_x509_crt_t* cert) [Function]

req: should contain a gnutls_ocsp_req_t type

digest: hash algorithm, a gnutls_digest_algorithm_t value

issuer: issuer of subject certificate

cert: certificate to request status for

This function will add another request to the OCSP request for a particular certificate. The issuer name hash, issuer key hash, and serial number fields is populated as follows.

The issuer name and the serial number is taken from `cert` . The issuer key is taken from `issuer` . The hashed values will be hashed using the `digest` algorithm, normally `GNUTLS_DIG_SHA1` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_req_add_cert_id

`int gnutls_ocsp_req_add_cert_id` (*gnutls_ocsp_req_t* **`req`**, *gnutls_digest_algorithm_t* **`digest`**, *const gnutls_datum_t ** **`issuer_name_hash`**, *const gnutls_datum_t ** **`issuer_key_hash`**, *const gnutls_datum_t ** **`serial_number`**) [Function]

req: should contain a `gnutls_ocsp_req_t` type

digest: hash algorithm, a `gnutls_digest_algorithm_t` value

issuer_name_hash: hash of issuer's DN

issuer_key_hash: hash of issuer's public key

serial_number: serial number of certificate to check

This function will add another request to the OCSP request for a particular certificate having the issuer name hash of `issuer_name_hash` and issuer key hash of `issuer_key_hash` (both hashed using `digest`) and serial number `serial_number` .

The information needed corresponds to the CertID structure:

<informalexample><programlisting> CertID ::= SEQUENCE { hashAlgorithm AlgorithmIdentifier, issuerNameHash OCTET STRING, – Hash of Issuer's DN issuerKeyHash OCTET STRING, – Hash of Issuers public key serialNumber CertificateSerialNumber } </programlisting></informalexample>

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_req_deinit

`void gnutls_ocsp_req_deinit` (*gnutls_ocsp_req_t* **`req`**) [Function]

req: The data to be deinitialized

This function will deinitialize a OCSP request structure.

gnutls_ocsp_req_export

`int gnutls_ocsp_req_export` (*gnutls_ocsp_req_const_t* **`req`**, *gnutls_datum_t ** **`data`**) [Function]

req: Holds the OCSP request

data: newly allocate buffer holding DER encoded OCSP request

This function will export the OCSP request to DER format.

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_ocsp_req_get_cert_id

`int gnutls_ocsp_req_get_cert_id` (*gnutls_ocsp_req_const_t* **req**, *unsigned* **indx**, *gnutls_digest_algorithm_t * **digest***, *gnutls_datum_t * **issuer_name_hash***, *gnutls_datum_t * **issuer_key_hash***, *gnutls_datum_t * **serial_number***) [Function]

req: should contain a `gnutls_ocsp_req_t` type

indx: Specifies which extension OID to get. Use (0) to get the first one.

digest: output variable with `gnutls_digest_algorithm_t` hash algorithm

issuer_name_hash: output buffer with hash of issuer's DN

issuer_key_hash: output buffer with hash of issuer's public key

serial_number: output buffer with serial number of certificate to check

This function will return the certificate information of the `indx` 'ed request in the OCSP request. The information returned corresponds to the CertID structure:

<informalexample><programlisting> CertID ::= SEQUENCE { hashAlgorithm AlgorithmIdentifier, issuerNameHash OCTET STRING, – Hash of Issuer's DN issuerKeyHash OCTET STRING, – Hash of Issuers public key serialNumber CertificateSerialNumber } </programlisting></informalexample>

Each of the pointers to output variables may be NULL to indicate that the caller is not interested in that value.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_ocsp_req_get_extension

`int gnutls_ocsp_req_get_extension` (*gnutls_ocsp_req_const_t* **req**, *unsigned* **indx**, *gnutls_datum_t * **oid***, *unsigned int * **critical***, *gnutls_datum_t * **data***) [Function]

req: should contain a `gnutls_ocsp_req_t` type

indx: Specifies which extension OID to get. Use (0) to get the first one.

oid: will hold newly allocated buffer with OID of extension, may be NULL

critical: output variable with critical flag, may be NULL.

data: will hold newly allocated buffer with extension data, may be NULL

This function will return all information about the requested extension in the OCSP request. The information returned is the OID, the critical flag, and the data itself. The extension OID will be stored as a string. Any of `oid` , `critical` , and `data` may be NULL which means that the caller is not interested in getting that information back.

The caller needs to deallocate memory by calling `gnutls_free()` on `oid` ->data and `data` ->data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_ocsp_req_get_nonce

`int gnutls_ocsp_req_get_nonce (`*gnutls_ocsp_req_const_t* **`req`**, *unsigned int ** **`critical`**, *gnutls_datum_t ** **`nonce`**`)` [Function]

req: should contain a `gnutls_ocsp_req_t` type

critical: whether nonce extension is marked critical, or NULL

nonce: will hold newly allocated buffer with nonce data

This function will return the OCSP request nonce extension data.

The caller needs to deallocate memory by calling `gnutls_free()` on `nonce ->`data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_req_get_version

`int gnutls_ocsp_req_get_version (`*gnutls_ocsp_req_const_t* **`req`**`)` [Function]

req: should contain a `gnutls_ocsp_req_t` type

This function will return the version of the OCSP request. Typically this is always 1 indicating version 1.

Returns: version of OCSP request, or a negative error code on error.

gnutls_ocsp_req_import

`int gnutls_ocsp_req_import (`*gnutls_ocsp_req_t* **`req`**, *const gnutls_datum_t ** **`data`**`)` [Function]

req: The data to store the parsed request.

data: DER encoded OCSP request.

This function will convert the given DER encoded OCSP request to the native `gnutls_ocsp_req_t` format. The output will be stored in `req` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_req_init

`int gnutls_ocsp_req_init (`*gnutls_ocsp_req_t ** **`req`**`)` [Function]

req: A pointer to the type to be initialized

This function will initialize an OCSP request structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_req_print

`int gnutls_ocsp_req_print (`*gnutls_ocsp_req_const_t* **`req`**, *gnutls_ocsp_print_formats_t* **`format`**, *gnutls_datum_t ** **`out`**`)` [Function]

req: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with (0) terminated string.

This function will pretty print a OCSP request, suitable for display to a human.

If the format is `GNUTLS_OCSP_PRINT_FULL` then all fields of the request will be output, on multiple lines.

The output `out ->`data needs to be deallocate using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_req_randomize_nonce

`int gnutls_ocsp_req_randomize_nonce` (*gnutls_ocsp_req_t* `req`) [Function]

req: should contain a `gnutls_ocsp_req_t` type

This function will add or update an nonce extension to the OCSP request with a newly generated random value.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_req_set_extension

`int gnutls_ocsp_req_set_extension` (*gnutls_ocsp_req_t* `req`, *const char * oid*, *unsigned int* `critical`, *const gnutls_datum_t * data*) [Function]

req: should contain a `gnutls_ocsp_req_t` type

oid: buffer with OID of extension as a string.

critical: critical flag, normally false.

data: the extension data

This function will add an extension to the OCSP request. Calling this function multiple times for the same OID will overwrite values from earlier calls.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_req_set_nonce

`int gnutls_ocsp_req_set_nonce` (*gnutls_ocsp_req_t* `req`, *unsigned int* `critical`, *const gnutls_datum_t * nonce*) [Function]

req: should contain a `gnutls_ocsp_req_t` type

critical: critical flag, normally false.

nonce: the nonce data

This function will add an nonce extension to the OCSP request. Calling this function multiple times will overwrite values from earlier calls.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_resp_check_crt

`int gnutls_ocsp_resp_check_crt (`*gnutls_ocsp_resp_const_t* **`resp`**, *unsigned int* **`indx`**, *gnutls_x509_crt_t* **`crt`**`)` [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

indx: Specifies response number to get. Use (0) to get the first one.

crt: The certificate to check

This function will check whether the OCSP response is about the provided certificate.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since: 3.1.3

gnutls_ocsp_resp_deinit

`void gnutls_ocsp_resp_deinit (`*gnutls_ocsp_resp_t* **`resp`**`)` [Function]

resp: The data to be deinitialized

This function will deinitialize a OCSP response structure.

gnutls_ocsp_resp_export

`int gnutls_ocsp_resp_export (`*gnutls_ocsp_resp_const_t* **`resp`**, *gnutls_datum_t ** `data`**`)` [Function]

resp: Holds the OCSP response

data: newly allocate buffer holding DER encoded OCSP response

This function will export the OCSP response to DER format.

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_ocsp_resp_export2

`int gnutls_ocsp_resp_export2 (`*gnutls_ocsp_resp_const_t* **`resp`**, *gnutls_datum_t ** `data`**, *gnutls_x509_crt_fmt_t* **`fmt`**`)` [Function]

resp: Holds the OCSP response

data: newly allocate buffer holding DER or PEM encoded OCSP response

fmt: DER or PEM

This function will export the OCSP response to DER or PEM format.

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 3.6.3

gnutls_ocsp_resp_get_certs

`int gnutls_ocsp_resp_get_certs (`*gnutls_ocsp_resp_const_t* **`resp`**, *gnutls_x509_crt_t *** `certs`**, *size_t ** `ncerts`**`)` [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

certs: newly allocated array with `gnutls_x509_crt_t` certificates

ncerts: output variable with number of allocated certs.

This function will extract the X.509 certificates found in the Basic OCSP Response. The `certs` output variable will hold a newly allocated zero-terminated array with X.509 certificates.

Every certificate in the array needs to be de-allocated with `gnutls_x509_crt_deinit()` and the array itself must be freed using `gnutls_free()` .

Both the `certs` and `ncerts` variables may be NULL. Then the function will work as normal but will not return the NULL:d information. This can be used to get the number of certificates only, or to just get the certificate array without its size.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_get_extension

`int gnutls_ocsp_resp_get_extension` (*gnutls_ocsp_resp_const_t* `resp`, *unsigned* `indx`, *gnutls_datum_t * * `oid`, *unsigned int * `critical`, *gnutls_datum_t * `data`) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

indx: Specifies which extension OID to get. Use (0) to get the first one.

oid: will hold newly allocated buffer with OID of extension, may be NULL

critical: output variable with critical flag, may be NULL.

data: will hold newly allocated buffer with extension data, may be NULL

This function will return all information about the requested extension in the OCSP response. The information returned is the OID, the critical flag, and the data itself. The extension OID will be stored as a string. Any of `oid` , `critical` , and `data` may be NULL which means that the caller is not interested in getting that information back.

The caller needs to deallocate memory by calling `gnutls_free()` on `oid` ->data and `data` ->data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_ocsp_resp_get_nonce

`int gnutls_ocsp_resp_get_nonce` (*gnutls_ocsp_resp_const_t* `resp`, *unsigned int * `critical`, *gnutls_datum_t * `nonce`) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

critical: whether nonce extension is marked critical

nonce: will hold newly allocated buffer with nonce data

This function will return the Basic OCSP Response nonce extension data.

The caller needs to deallocate memory by calling `gnutls_free()` on `nonce` ->data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_resp_get_produced

`time_t gnutls_ocsp_resp_get_produced (`*gnutls_ocsp_resp_const_t* ***resp***`)` [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

This function will return the time when the OCSP response was signed.

Returns: signing time, or (time_t)-1 on error.

gnutls_ocsp_resp_get_responder

`int gnutls_ocsp_resp_get_responder (`*gnutls_ocsp_resp_const_t* ***resp***`, `*gnutls_datum_t ** ***dn***`)` [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

dn: newly allocated buffer with name

This function will extract the name of the Basic OCSP Response in the provided buffer. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If the responder ID is not a name but a hash, this function will return zero and the `dn` elements will be set to `NULL` .

The caller needs to deallocate memory by calling `gnutls_free()` on `dn ->`data.

This function does not output a fully RFC4514 compliant string, if that is required see `gnutls_ocsp_resp_get_responder2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. When no data exist it will return success and set `dn` elements to zero.

gnutls_ocsp_resp_get_responder2

`int gnutls_ocsp_resp_get_responder2 (`*gnutls_ocsp_resp_const_t* ***resp***`, `*gnutls_datum_t ** ***dn***`, `*unsigned* ***flags***`)` [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

dn: newly allocated buffer with name

flags: zero or `GNUTLS_X509_DN_FLAG_COMPAT`

This function will extract the name of the Basic OCSP Response in the provided buffer. The name will be in the form `"C=xxxx,O=yyyy,CN=zzzz"` as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If the responder ID is not a name but a hash, this function will return zero and the `dn` elements will be set to `NULL` .

The caller needs to deallocate memory by calling `gnutls_free()` on `dn ->`data.

When the flag `GNUTLS_X509_DN_FLAG_COMPAT` is specified, the output format will match the format output by previous to 3.5.6 versions of GnuTLS which was not not fully RFC4514-compliant.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code is returned. When no data exist it will return GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE .

gnutls_ocsp_resp_get_responder_raw_id

`int gnutls_ocsp_resp_get_responder_raw_id` [Function]
(*gnutls_ocsp_resp_const_t* **resp**, *unsigned* **type**, *gnutls_datum_t* * **raw**)

resp: should contain a gnutls_ocsp_resp_t type

type: should be GNUTLS_OCSP_RESP_ID_KEY or GNUTLS_OCSP_RESP_ID_DN

raw: newly allocated buffer with the raw ID

This function will extract the raw key (or DN) ID of the Basic OCSP Response in the provided buffer. If the responder ID is not a key ID then this function will return GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE .

The caller needs to deallocate memory by calling gnutls_free() on dn ->data.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error code is returned.

gnutls_ocsp_resp_get_response

`int gnutls_ocsp_resp_get_response` (*gnutls_ocsp_resp_const_t* **resp**, *gnutls_datum_t* * **response_type_oid**, *gnutls_datum_t* * **response**) [Function]

resp: should contain a gnutls_ocsp_resp_t type

response_type_oid: newly allocated output buffer with response type OID

response: newly allocated output buffer with DER encoded response

This function will extract the response type OID in and the response data from an OCSP response. Normally the response_type_oid is always "1.3.6.1.5.5.7.48.1.1" which means the response should be decoded as a Basic OCSP Response, but technically other response types could be used.

This function is typically only useful when you want to extract the response type OID of an response for diagnostic purposes. Otherwise gnutls_ocsp_resp_import() will decode the basic OCSP response part and the caller need not worry about that aspect.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_get_signature

`int gnutls_ocsp_resp_get_signature` (*gnutls_ocsp_resp_const_t* **resp**, *gnutls_datum_t* * **sig**) [Function]

resp: should contain a gnutls_ocsp_resp_t type

sig: newly allocated output buffer with signature data

This function will extract the signature field of a OCSP response.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_get_signature_algorithm

`int gnutls_ocsp_resp_get_signature_algorithm` [Function]
(*gnutls_ocsp_resp_const_t* ***resp***)

resp: should contain a `gnutls_ocsp_resp_t` type

This function will return a value of the `gnutls_sign_algorithm_t` enumeration that is the signature algorithm that has been used to sign the OCSP response.

Returns: a `gnutls_sign_algorithm_t` value, or a negative error code on error.

gnutls_ocsp_resp_get_single

`int gnutls_ocsp_resp_get_single` (*gnutls_ocsp_resp_const_t* ***resp***, [Function]
unsigned ***indx***, *gnutls_digest_algorithm_t ** ***digest***, *gnutls_datum_t ** ***issuer_name_hash***, *gnutls_datum_t ** ***issuer_key_hash***, *gnutls_datum_t ** ***serial_number***, *unsigned int ** ***cert_status***, *time_t ** ***this_update***, *time_t ** ***next_update***, *time_t ** ***revocation_time***, *unsigned int ** ***revocation_reason***)

resp: should contain a `gnutls_ocsp_resp_t` type

indx: Specifies response number to get. Use (0) to get the first one.

digest: output variable with `gnutls_digest_algorithm_t` hash algorithm

issuer_name_hash: output buffer with hash of issuer's DN

issuer_key_hash: output buffer with hash of issuer's public key

serial_number: output buffer with serial number of certificate to check

cert_status: a certificate status, a `gnutls_ocsp_cert_status_t` enum.

this_update: time at which the status is known to be correct.

next_update: when newer information will be available, or (time_t)-1 if unspecified

revocation_time: when `cert_status` is `GNUTLS_OCSP_CERT_REVOKED` , holds time of revocation.

revocation_reason: revocation reason, a `gnutls_x509_crl_reason_t` enum.

This function will return the certificate information of the `indx` 'ed response in the Basic OCSP Response `resp` . The information returned corresponds to the OCSP SingleResponse structure except the final singleExtensions.

Each of the pointers to output variables may be NULL to indicate that the caller is not interested in that value.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

gnutls_ocsp_resp_get_status

`int gnutls_ocsp_resp_get_status` (*gnutls_ocsp_resp_const_t* ***resp***) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

This function will return the status of a OCSP response, an `gnutls_ocsp_resp_status_t` enumeration.

Returns: status of OCSP request as a `gnutls_ocsp_resp_status_t` , or a negative error code on error.

gnutls_ocsp_resp_get_version

`int gnutls_ocsp_resp_get_version` (*gnutls_ocsp_resp_const_t* ***resp***) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

This function will return the version of the Basic OCSP Response. Typically this is always 1 indicating version 1.

Returns: version of Basic OCSP response, or a negative error code on error.

gnutls_ocsp_resp_import

`int gnutls_ocsp_resp_import` (*gnutls_ocsp_resp_t* ***resp***, *const gnutls_datum_t * data*) [Function]

resp: The data to store the parsed response.

data: DER encoded OCSP response.

This function will convert the given DER encoded OCSP response to the native `gnutls_ocsp_resp_t` format. It also decodes the Basic OCSP Response part, if any. The output will be stored in `resp` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_import2

`int gnutls_ocsp_resp_import2` (*gnutls_ocsp_resp_t* ***resp***, *const gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t fmt*) [Function]

resp: The data to store the parsed response.

data: DER or PEM encoded OCSP response.

fmt: DER or PEM

This function will convert the given OCSP response to the native `gnutls_ocsp_resp_t` format. It also decodes the Basic OCSP Response part, if any. The output will be stored in `resp` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_ocsp_resp_init

`int gnutls_ocsp_resp_init` (*gnutls_ocsp_resp_t * resp*) [Function]

resp: A pointer to the type to be initialized

This function will initialize an OCSP response structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_list_import2

`int gnutls_ocsp_resp_list_import2` (*gnutls_ocsp_resp_t ** `ocsps`, unsigned int * `size`, const gnutls_datum_t * `resp_data`, gnutls_x509_crt_fmt_t `format`, unsigned int `flags`*) [Function]

ocsps: Will hold the parsed OCSP response list.

size: It will contain the size of the list.

resp_data: The PEM encoded OCSP list.

format: One of `GNUTLS_X509_FMT_PEM` or `GNUTLS_X509_FMT_DER`

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded OCSP response list to the native gnutls_ocsp_resp_t format. The output will be stored in `ocsps` which will be allocated and initialized.

The OCSP responses should have a header of "OCSP RESPONSE".

To deinitialize responses, you need to deinitialize each `gnutls_ocsp_resp_t` structure independently, and use `gnutls_free()` at `ocsps` .

In PEM files, when no OCSP responses are detected `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Returns: the number of responses read or a negative error value.

Since: 3.6.3

gnutls_ocsp_resp_print

`int gnutls_ocsp_resp_print` (*gnutls_ocsp_resp_const_t `resp`, gnutls_ocsp_print_formats_t `format`, gnutls_datum_t * `out`*) [Function]

resp: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with (0) terminated string.

This function will pretty print a OCSP response, suitable for display to a human.

If the format is `GNUTLS_OCSP_PRINT_FULL` then all fields of the response will be output, on multiple lines.

The output `out` ->data needs to be deallocate using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_verify

`int gnutls_ocsp_resp_verify` (*gnutls_ocsp_resp_const_t `resp`, gnutls_x509_trust_list_t `trustlist`, unsigned int * `verify`, unsigned int `flags`*) [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

trustlist: trust anchors as a `gnutls_x509_trust_list_t` type

verify: output variable with verification status, an `gnutls_ocsp_verify_reason_t`

flags: verification flags from `gnutls_certificate_verify_flags`

Verify signature of the Basic OCSP Response against the public key in the certificate of a trusted signer. The `trustlist` should be populated with trust anchors. The function will extract the signer certificate from the Basic OCSP Response and will verify it against the `trustlist` . A trusted signer is a certificate that is either in `trustlist` , or it is signed directly by a certificate in `trustlist` and has the id-ad-ocspSigning Extended Key Usage bit set.

The output `verify` variable will hold verification status codes (e.g., `GNUTLS_OCSP_VERIFY_SIGNER_NOT_FOUND` , `GNUTLS_OCSP_VERIFY_INSECURE_ALGORITHM`) which are only valid if the function returned `GNUTLS_E_SUCCESS` .

Note that the function returns `GNUTLS_E_SUCCESS` even when verification failed. The caller must always inspect the `verify` variable to find out the verification status.

The `flags` variable should be 0 for now.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_ocsp_resp_verify_direct

`int gnutls_ocsp_resp_verify_direct (`*gnutls_ocsp_resp_const_t* `resp`*, gnutls_x509_crt_t* `issuer`*, unsigned int * verify, unsigned int* `flags`*)* [Function]

resp: should contain a `gnutls_ocsp_resp_t` type

issuer: certificate believed to have signed the response

verify: output variable with verification status, an `gnutls_ocsp_verify_reason_t`

flags: verification flags from `gnutls_certificate_verify_flags`

Verify signature of the Basic OCSP Response against the public key in the `issuer` certificate.

The output `verify` variable will hold verification status codes (e.g., `GNUTLS_OCSP_VERIFY_SIGNER_NOT_FOUND` , `GNUTLS_OCSP_VERIFY_INSECURE_ALGORITHM`) which are only valid if the function returned `GNUTLS_E_SUCCESS` .

Note that the function returns `GNUTLS_E_SUCCESS` even when verification failed. The caller must always inspect the `verify` variable to find out the verification status.

The `flags` variable should be 0 for now.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

E.6 PKCS 12 API

The following functions are to be used for PKCS 12 handling. Their prototypes lie in `gnutls/pkcs12.h`.

gnutls_pkcs12_bag_decrypt

`int gnutls_pkcs12_bag_decrypt (`*gnutls_pkcs12_bag_t* `bag`*, const char * pass*`)` [Function]

bag: The bag

pass: The password used for encryption, must be ASCII.

This function will decrypt the given encrypted bag and return 0 on success.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_pkcs12_bag_deinit

`void gnutls_pkcs12_bag_deinit` (*gnutls_pkcs12_bag_t* `bag`) [Function]

bag: A pointer to the type to be initialized

This function will deinitialize a PKCS12 Bag structure.

gnutls_pkcs12_bag_enc_info

`int gnutls_pkcs12_bag_enc_info` (*gnutls_pkcs12_bag_t* `bag`, *unsigned int ** `schema`, *unsigned int ** `cipher`, *void ** `salt`, *unsigned int ** `salt_size`, *unsigned int ** `iter_count`, *char *** `oid`) [Function]

bag: The bag

schema: indicate the schema as one of `gnutls_pkcs_encrypt_flags_t`

cipher: the cipher used as `gnutls_cipher_algorithm_t`

salt: PBKDF2 salt (if non-NULL then `salt_size` initially holds its size)

salt_size: PBKDF2 salt size

iter_count: PBKDF2 iteration count

oid: if non-NULL it will contain an allocated null-terminated variable with the OID

This function will provide information on the encryption algorithms used in an encrypted bag. If the structure algorithms are unknown the code `GNUTLS_E_UNKNOWN_CIPHER_TYPE` will be returned, and only `oid` , will be set. That is, `oid` will be set on encrypted bags whether supported or not. It must be deinitialized using `gnutls_free()` . The other variables are only set on supported structures.

Returns: `GNUTLS_E_INVALID_REQUEST` if the provided bag isn't encrypted, `GNUTLS_E_UNKNOWN_CIPHER_TYPE` if the structure's encryption isn't supported, or another negative error code in case of a failure. Zero on success.

gnutls_pkcs12_bag_encrypt

`int gnutls_pkcs12_bag_encrypt` (*gnutls_pkcs12_bag_t* `bag`, *const char ** `pass`, *unsigned int* `flags`) [Function]

bag: The bag

pass: The password used for encryption, must be ASCII

flags: should be one of `gnutls_pkcs_encrypt_flags_t` elements bitwise or'd

This function will encrypt the given bag.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

gnutls_pkcs12_bag_get_count

`int gnutls_pkcs12_bag_get_count (`*gnutls_pkcs12_bag_t* `bag``)` [Function]

bag: The bag

This function will return the number of the elements within the bag.

Returns: Number of elements in bag, or an negative error code on error.

gnutls_pkcs12_bag_get_data

`int gnutls_pkcs12_bag_get_data (`*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`, *gnutls_datum_t ** `data``)` [Function]

bag: The bag

indx: The element of the bag to get the data from

data: where the bag's data will be. Should be treated as constant.

This function will return the bag's data. The data is a constant that is stored into the bag. Should not be accessed after the bag is deleted.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_bag_get_friendly_name

`int gnutls_pkcs12_bag_get_friendly_name (`*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`, *char *** `name``)` [Function]

bag: The bag

indx: The bag's element to add the id

name: will hold a pointer to the name (to be treated as const)

This function will return the friendly name, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. or a negative error code on error.

gnutls_pkcs12_bag_get_key_id

`int gnutls_pkcs12_bag_get_key_id (`*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`, *gnutls_datum_t ** `id``)` [Function]

bag: The bag

indx: The bag's element to add the id

id: where the ID will be copied (to be treated as const)

This function will return the key ID, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. or a negative error code on error.

gnutls_pkcs12_bag_get_type

`int gnutls_pkcs12_bag_get_type (`*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`) [Function]

bag: The bag

indx: The element of the bag to get the type

This function will return the bag's type.

Returns: On error a negative error value or one of the `gnutls_pkcs12_bag_type_t` enumerations.

gnutls_pkcs12_bag_init

`int gnutls_pkcs12_bag_init (`*gnutls_pkcs12_bag_t* * `bag`) [Function]

bag: A pointer to the type to be initialized

This function will initialize a PKCS12 bag structure. PKCS12 Bags usually contain private keys, lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_bag_set_crl

`int gnutls_pkcs12_bag_set_crl (`*gnutls_pkcs12_bag_t* `bag`, *gnutls_x509_crl_t* `crl`) [Function]

bag: The bag

crl: the CRL to be copied.

This function will insert the given CRL into the bag. This is just a wrapper over `gnutls_pkcs12_bag_set_data()` .

Returns: the index of the added bag on success, or a negative error code on failure.

gnutls_pkcs12_bag_set_crt

`int gnutls_pkcs12_bag_set_crt (`*gnutls_pkcs12_bag_t* `bag`, *gnutls_x509_crt_t* `crt`) [Function]

bag: The bag

crt: the certificate to be copied.

This function will insert the given certificate into the bag. This is just a wrapper over `gnutls_pkcs12_bag_set_data()` .

Returns: the index of the added bag on success, or a negative value on failure.

gnutls_pkcs12_bag_set_data

`int gnutls_pkcs12_bag_set_data (`*gnutls_pkcs12_bag_t* `bag`, *gnutls_pkcs12_bag_type_t* `type`, *const gnutls_datum_t* * `data`) [Function]

bag: The bag

type: The data's type

data: the data to be copied.

This function will insert the given data of the given type into the bag.

Returns: the index of the added bag on success, or a negative value on error.

gnutls_pkcs12_bag_set_friendly_name

`int gnutls_pkcs12_bag_set_friendly_name` (*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`, *const char ** `name`) [Function]

bag: The bag

indx: The bag's element to add the id

name: the name

This function will add the given key friendly name, to the specified, by the index, bag element. The name will be encoded as a 'Friendly name' bag attribute, which is usually used to set a user name to the local private key and the certificate pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. or a negative error code on error.

gnutls_pkcs12_bag_set_key_id

`int gnutls_pkcs12_bag_set_key_id` (*gnutls_pkcs12_bag_t* `bag`, *unsigned* `indx`, *const gnutls_datum_t ** `id`) [Function]

bag: The bag

indx: The bag's element to add the id

id: the ID

This function will add the given key ID, to the specified, by the index, bag element. The key ID will be encoded as a 'Local key identifier' bag attribute, which is usually used to distinguish the local private key and the certificate pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. or a negative error code on error.

gnutls_pkcs12_bag_set_privkey

`int gnutls_pkcs12_bag_set_privkey` (*gnutls_pkcs12_bag_t* `bag`, *gnutls_x509_privkey_t* `privkey`, *const char ** `password`, *unsigned* `flags`) [Function]

bag: The bag

privkey: the private key to be copied.

password: the password to protect the key with (may be `NULL`)

flags: should be one of `gnutls_pkcs_encrypt_flags_t` elements bitwise or'd

This function will insert the given private key into the bag. This is just a wrapper over `gnutls_pkcs12_bag_set_data()` .

Returns: the index of the added bag on success, or a negative value on failure.

gnutls_pkcs12_deinit

`void gnutls_pkcs12_deinit` (*gnutls_pkcs12_t* `pkcs12`) [Function]

pkcs12: The type to be initialized

This function will deinitialize a PKCS12 type.

gnutls_pkcs12_export

`int gnutls_pkcs12_export` (*gnutls_pkcs12_t* `pkcs12`, *gnutls_x509_crt_fmt_t* `format`, *void *** `output_data`, *size_t ** `output_data_size`) [Function]

pkcs12: A pkcs12 type

format: the format of output params. One of PEM or DER.

output_data: will contain a structure PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the pkcs12 structure to DER or PEM format.

If the buffer provided is not long enough to hold the output, then *output_data_size will be updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN PKCS12".

Returns: In case of failure a negative error code will be returned, and 0 on success.

gnutls_pkcs12_export2

`int gnutls_pkcs12_export2` (*gnutls_pkcs12_t* `pkcs12`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t ** `out`) [Function]

pkcs12: A pkcs12 type

format: the format of output params. One of PEM or DER.

out: will contain a structure PEM or DER encoded

This function will export the pkcs12 structure to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN PKCS12".

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 3.1.3

gnutls_pkcs12_generate_mac

`int gnutls_pkcs12_generate_mac` (*gnutls_pkcs12_t* `pkcs12`, *const char ** `pass`) [Function]

pkcs12: A pkcs12 type

pass: The password for the MAC

This function will generate a MAC for the PKCS12 structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_generate_mac2

`int gnutls_pkcs12_generate_mac2` (*gnutls_pkcs12_t* `pkcs12`, *gnutls_mac_algorithm_t* `mac`, *const char ** `pass`) [Function]

pkcs12: A pkcs12 type

mac: the MAC algorithm to use

pass: The password for the MAC

This function will generate a MAC for the PKCS12 structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_get_bag

`int gnutls_pkcs12_get_bag` (*gnutls_pkcs12_t* `pkcs12`, *int* `indx`, *gnutls_pkcs12_bag_t* `bag`) [Function]

pkcs12: A pkcs12 type

indx: contains the index of the bag to extract

bag: An initialized bag, where the contents of the bag will be copied

This function will return a Bag from the PKCS12 structure.

After the last Bag has been read `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_import

`int gnutls_pkcs12_import` (*gnutls_pkcs12_t* `pkcs12`, *const gnutls_datum_t * * `data`, *gnutls_x509_crt_fmt_t* `format`, *unsigned int* `flags`) [Function]

pkcs12: The data to store the parsed PKCS12.

data: The DER or PEM encoded PKCS12.

format: One of DER or PEM

flags: an ORed sequence of gnutls_privkey_pkcs8_flags

This function will convert the given DER or PEM encoded PKCS12 to the native gnutls_pkcs12_t format. The output will be stored in 'pkcs12'.

If the PKCS12 is PEM encoded it should have a header of `"PKCS12"`.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_init

`int gnutls_pkcs12_init` (*gnutls_pkcs12_t * * `pkcs12`) [Function]

pkcs12: A pointer to the type to be initialized

This function will initialize a PKCS12 type. PKCS12 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_mac_info

`int gnutls_pkcs12_mac_info` (*gnutls_pkcs12_t* `pkcs12`, *unsigned int ** `mac`, *void ** `salt`, *unsigned int ** `salt_size`, *unsigned int ** `iter_count`, *char *** `oid`) [Function]

pkcs12: A pkcs12 type

mac: the MAC algorithm used as `gnutls_mac_algorithm_t`

salt: the salt used for string to key (if non-NULL then `salt_size` initially holds its size)

salt_size: string to key salt size

iter_count: string to key iteration count

oid: if non-NULL it will contain an allocated null-terminated variable with the OID

This function will provide information on the MAC algorithm used in a PKCS `12` structure. If the structure algorithms are unknown the code `GNUTLS_E_UNKNOWN_HASH_ALGORITHM` will be returned, and only `oid` , will be set. That is, `oid` will be set on structures with a MAC whether supported or not. It must be deinitialized using `gnutls_free()` . The other variables are only set on supported structures.

Returns: `GNUTLS_E_INVALID_REQUEST` if the provided structure doesn't contain a MAC, `GNUTLS_E_UNKNOWN_HASH_ALGORITHM` if the structure's MAC isn't supported, or another negative error code in case of a failure. Zero on success.

gnutls_pkcs12_set_bag

`int gnutls_pkcs12_set_bag` (*gnutls_pkcs12_t* `pkcs12`, *gnutls_pkcs12_bag_t* `bag`) [Function]

pkcs12: should contain a gnutls_pkcs12_t type

bag: An initialized bag

This function will insert a Bag into the PKCS12 structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs12_simple_parse

`int gnutls_pkcs12_simple_parse` (*gnutls_pkcs12_t* `p12`, *const char ** `password`, *gnutls_x509_privkey_t ** `key`, *gnutls_x509_crt_t *** `chain`, *unsigned int ** `chain_len`, *gnutls_x509_crt_t *** `extra_certs`, *unsigned int ** `extra_certs_len`, *gnutls_x509_crl_t ** `crl`, *unsigned int* `flags`) [Function]

p12: A pkcs12 type

password: optional password used to decrypt the structure, bags and keys.

key: a structure to store the parsed private key.

chain: the corresponding to key certificate chain (may be `NULL`)

chain_len: will be updated with the number of additional (may be `NULL`)

extra_certs: optional pointer to receive an array of additional certificates found in the PKCS12 structure (may be `NULL`).

extra_certs_len: will be updated with the number of additional certs (may be `NULL`).

crl: an optional structure to store the parsed CRL (may be `NULL`).

flags: should be zero or one of GNUTLS_PKCS12_SP_*

This function parses a PKCS12 structure in `pkcs12` and extracts the private key, the corresponding certificate chain, any additional certificates and a CRL. The structures in `key` , `chain crl` , and `extra_certs` must not be initialized.

The `extra_certs` and `extra_certs_len` parameters are optional and both may be set to `NULL` . If either is non-`NULL` , then both must be set. The value for `extra_certs` is allocated using `gnutls_malloc()` .

Encrypted PKCS12 bags and PKCS8 private keys are supported, but only with password based security and the same password for all operations.

Note that a PKCS12 structure may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. For this reason this function is useful for PKCS12 files that contain only one key/certificate pair and/or one CRL.

If the provided structure has encrypted fields but no password is provided then this function returns `GNUTLS_E_DECRYPTION_FAILED` .

Note that normally the chain constructed does not include self signed certificates, to comply with TLS' requirements. If, however, the flag `GNUTLS_PKCS12_SP_INCLUDE_SELF_SIGNED` is specified then self signed certificates will be included in the chain.

Prior to using this function the PKCS `12` structure integrity must be verified using `gnutls_pkcs12_verify_mac()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_pkcs12_verify_mac

`int gnutls_pkcs12_verify_mac` (*gnutls_pkcs12_t pkcs12, const char * pass*) [Function]

pkcs12: should contain a gnutls_pkcs12_t type

pass: The password for the MAC

This function will verify the MAC for the PKCS12 structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

E.7 Hardware token via PKCS 11 API

The following functions are to be used for PKCS 11 handling. Their prototypes lie in `gnutls/pkcs11.h`.

gnutls_pkcs11_add_provider

`int gnutls_pkcs11_add_provider` (*const char * name, const char * params*) [Function]

name: The filename of the module

params: should be NULL or a known string (see description)

This function will load and add a PKCS 11 module to the module list used in gnutls. After this function is called the module will be used for PKCS 11 operations.

When loading a module to be used for certificate verification, use the string 'trusted' as `params` .

Note that this function is not thread safe.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_copy_attached_extension

`int gnutls_pkcs11_copy_attached_extension` (*const char * `token_url`, gnutls_x509_crt_t `crt`, gnutls_datum_t * `data`, const char * `label`, unsigned int `flags`*) [Function]

token_url: A PKCS `11` URL specifying a token

crt: An X.509 certificate object

data: the attached extension

label: A name to be used for the attached extension (may be `NULL`)

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy an the attached extension in `data` for the certificate provided in `crt` in the PKCS `11` token specified by the URL (typically a trust module). The extension must be in RFC5280 Extension format.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.8

gnutls_pkcs11_copy_pubkey

`int gnutls_pkcs11_copy_pubkey` (*const char * `token_url`, gnutls_pubkey_t `pubkey`, const char * `label`, const gnutls_datum_t * `cid`, unsigned int `key_usage`, unsigned int `flags`*) [Function]

token_url: A PKCS `11` URL specifying a token

pubkey: The public key to copy

label: The name to be used for the stored data

cid: The CKA_ID to set for the object -if NULL, the ID will be derived from the public key

key_usage: One of GNUTLS_KEY_*

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy a public key object into a PKCS `11` token specified by a URL. Valid flags to mark the key: `GNUTLS_PKCS11_OBJ_FLAG_MARK_TRUSTED` , `GNUTLS_PKCS11_OBJ_FLAG_MARK_PRIVATE` , `GNUTLS_PKCS11_OBJ_FLAG_MARK_CA` , `GNUTLS_PKCS11_OBJ_FLAG_MARK_ALWAYS_AUTH` .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.6

gnutls_pkcs11_copy_secret_key

`int gnutls_pkcs11_copy_secret_key (`*const char * token_url*, *gnutls_datum_t * key*, *const char * label*, *unsigned int key_usage*, *unsigned int flags*`)` [Function]

token_url: A PKCS 11 URL specifying a token

key: The raw key

label: A name to be used for the stored data

key_usage: One of GNUTLS_KEY_*

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy a raw secret (symmetric) key into a PKCS 11 token specified by a URL. The key can be marked as sensitive or not.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_copy_x509_crt

`int gnutls_pkcs11_copy_x509_crt (`*const char * token_url*, *gnutls_x509_crt_t crt*, *const char * label*, *unsigned int flags*`)` [Function]

token_url: A PKCS 11 URL specifying a token

crt: A certificate

label: A name to be used for the stored data

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy a certificate into a PKCS 11 token specified by a URL. The certificate can be marked as trusted or not.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_copy_x509_crt2

`int gnutls_pkcs11_copy_x509_crt2 (`*const char * token_url*, *gnutls_x509_crt_t crt*, *const char * label*, *const gnutls_datum_t * cid*, *unsigned int flags*`)` [Function]

token_url: A PKCS 11 URL specifying a token

crt: The certificate to copy

label: The name to be used for the stored data

cid: The CKA_ID to set for the object -if NULL, the ID will be derived from the public key

flags: One of GNUTLS_PKCS11_OBJ_FLAG_*

This function will copy a certificate into a PKCS 11 token specified by a URL. Valid flags to mark the certificate: GNUTLS_PKCS11_OBJ_FLAG_MARK_TRUSTED , GNUTLS_PKCS11_OBJ_FLAG_MARK_PRIVATE , GNUTLS_PKCS11_OBJ_FLAG_MARK_CA , GNUTLS_PKCS11_OBJ_FLAG_MARK_ALWAYS_AUTH .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_copy_x509_privkey

`int gnutls_pkcs11_copy_x509_privkey` (*const char * token_url*, *gnutls_x509_privkey_t key*, *const char * label*, *unsigned int key_usage*, *unsigned int flags*) [Function]

token_url: A PKCS 11 URL specifying a token

key: A private key

label: A name to be used for the stored data

key_usage: One of GNUTLS_KEY_*

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will copy a private key into a PKCS 11 token specified by a URL.

Since 3.6.3 the objects are marked as sensitive by default unless GNUTLS_PKCS11_OBJ_FLAG_MARK_NOT_SENSITIVE is specified.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_copy_x509_privkey2

`int gnutls_pkcs11_copy_x509_privkey2` (*const char * token_url*, *gnutls_x509_privkey_t key*, *const char * label*, *const gnutls_datum_t * cid*, *unsigned int key_usage*, *unsigned int flags*) [Function]

token_url: A PKCS 11 URL specifying a token

key: A private key

label: A name to be used for the stored data

cid: The CKA_ID to set for the object -if NULL, the ID will be derived from the public key

key_usage: One of GNUTLS_KEY_*

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will copy a private key into a PKCS 11 token specified by a URL.

Since 3.6.3 the objects are marked as sensitive by default unless GNUTLS_PKCS11_OBJ_FLAG_MARK_NOT_SENSITIVE is specified.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_crt_is_known

`unsigned gnutls_pkcs11_crt_is_known (const char * url, gnutls_x509_crt_t cert, unsigned int flags)` [Function]

url: A PKCS 11 url identifying a token

cert: is the certificate to find issuer for

flags: Use zero or flags from `GNUTLS_PKCS11_OBJ_FLAG` .

This function will check whether the provided certificate is stored in the specified token. This is useful in combination with `GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_TRUSTED` or `GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_DISTRUSTED` , to check whether a CA is present or a certificate is blacklisted in a trust PKCS `11` module.

This function can be used with a `url` of "pkcs11:", and in that case all modules will be searched. To restrict the modules to the marked as trusted in p11-kit use the `GNUTLS_PKCS11_OBJ_FLAG_PRESENT_IN_TRUSTED_MODULE` flag.

Note that the flag `GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_DISTRUSTED` is specific to p11-kit trust modules.

Returns: If the certificate exists non-zero is returned, otherwise zero.

Since: 3.3.0

gnutls_pkcs11_deinit

`void gnutls_pkcs11_deinit ( void)` [Function]

This function will deinitialize the PKCS 11 subsystem in gnutls. This function is only needed if you need to deinitialize the subsystem without calling `gnutls_global_deinit()` .

Since: 2.12.0

gnutls_pkcs11_delete_url

`int gnutls_pkcs11_delete_url (const char * object_url, unsigned int flags)` [Function]

object_url: The URL of the object to delete.

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will delete objects matching the given URL. Note that not all tokens support the delete operation.

Returns: On success, the number of objects deleted is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_get_pin_function

`gnutls_pin_callback_t gnutls_pkcs11_get_pin_function (void ** userdata)` [Function]

userdata: data to be supplied to callback

This function will return the callback function set using `gnutls_pkcs11_set_pin_function()` .

Returns: The function set or NULL otherwise.

Since: 3.1.0

gnutls_pkcs11_get_raw_issuer

`int gnutls_pkcs11_get_raw_issuer (`*const char * `url`, gnutls_x509_crt_t `cert`, gnutls_datum_t * `issuer`, gnutls_x509_crt_fmt_t `fmt`, unsigned int `flags`*`)` [Function]

url: A PKCS 11 url identifying a token

cert: is the certificate to find issuer for

issuer: Will hold the issuer if any in an allocated buffer.

fmt: The format of the exported issuer.

flags: Use zero or flags from `GNUTLS_PKCS11_OBJ_FLAG` .

This function will return the issuer of a given certificate, if it is stored in the token. By default only marked as trusted issuers are returned. If any issuer should be returned specify `GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_ANY` in `flags` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.2.7

gnutls_pkcs11_get_raw_issuer_by_dn

`int gnutls_pkcs11_get_raw_issuer_by_dn (`*const char * `url`, const gnutls_datum_t * `dn`, gnutls_datum_t * `issuer`, gnutls_x509_crt_fmt_t `fmt`, unsigned int `flags`*`)` [Function]

url: A PKCS 11 url identifying a token

dn: is the DN to search for

issuer: Will hold the issuer if any in an allocated buffer.

fmt: The format of the exported issuer.

flags: Use zero or flags from `GNUTLS_PKCS11_OBJ_FLAG` .

This function will return the certificate with the given DN, if it is stored in the token. By default only marked as trusted issuers are returned. If any issuer should be returned specify `GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_ANY` in `flags` .

The name of the function includes issuer because it can be used to discover issuers of certificates.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_get_raw_issuer_by_subject_key_id

`int gnutls_pkcs11_get_raw_issuer_by_subject_key_id (`*const char * `url`, const gnutls_datum_t * `dn`, const gnutls_datum_t * `spki`, gnutls_datum_t * `issuer`, gnutls_x509_crt_fmt_t `fmt`, unsigned int `flags`*`)` [Function]

url: A PKCS 11 url identifying a token

dn: is the DN to search for (may be NULL)

spki: is the subject key ID to search for

issuer: Will hold the issuer if any in an allocated buffer.

fmt: The format of the exported issuer.

flags: Use zero or flags from GNUTLS_PKCS11_OBJ_FLAG .

This function will return the certificate with the given DN and spki , if it is stored in the token. By default only marked as trusted issuers are returned. If any issuer should be returned specify GNUTLS_PKCS11_OBJ_FLAG_RETRIEVE_ANY in flags .

The name of the function includes issuer because it can be used to discover issuers of certificates.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.2

gnutls_pkcs11_init

`int gnutls_pkcs11_init (`*unsigned int* `flags`, *const char ** `deprecated_config_file``)` [Function]

flags: An ORed sequence of GNUTLS_PKCS11_FLAG_ *

deprecated_config_file: either NULL or the location of a deprecated configuration file

This function will initialize the PKCS 11 subsystem in gnutls. It will read configuration files if GNUTLS_PKCS11_FLAG_AUTO is used or allow you to independently load PKCS 11 modules using gnutls_pkcs11_add_provider() if GNUTLS_PKCS11_FLAG_MANUAL is specified.

You don't need to call this function since GnuTLS 3.3.0 because it is being called during the first request PKCS 11 operation. That call will assume the GNUTLS_PKCS11_FLAG_AUTO flag. If another flags are required then it must be called independently prior to any PKCS 11 operation.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_obj_deinit

`void gnutls_pkcs11_obj_deinit (`*gnutls_pkcs11_obj_t* `obj`)` [Function]

obj: The type to be deinitialized

This function will deinitialize a certificate structure.

Since: 2.12.0

gnutls_pkcs11_obj_export

`int gnutls_pkcs11_obj_export (`*gnutls_pkcs11_obj_t* `obj`, *void ** `output_data`, *size_t ** `output_data_size`)` [Function]

obj: Holds the object

output_data: will contain the object data

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case `GNUTLS_E_INVALID_REQUEST` will be returned.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

Returns: In case of failure a negative error code will be returned, and `GNUTLS_E_SUCCESS` (0) on success.

Since: 2.12.0

gnutls_pkcs11_obj_export2

`int gnutls_pkcs11_obj_export2 (`*gnutls_pkcs11_obj_t* `obj`, *gnutls_datum_t* * `out``)` [Function]

obj: Holds the object

out: will contain the object data

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case `GNUTLS_E_INVALID_REQUEST` will be returned.

The output buffer is allocated using `gnutls_malloc()` .

Returns: In case of failure a negative error code will be returned, and `GNUTLS_E_SUCCESS` (0) on success.

Since: 3.1.3

gnutls_pkcs11_obj_export3

`int gnutls_pkcs11_obj_export3 (`*gnutls_pkcs11_obj_t* `obj`, *gnutls_x509_crt_fmt_t* `fmt`, *gnutls_datum_t* * `out``)` [Function]

obj: Holds the object

fmt: The format of the exported data

out: will contain the object data

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case `GNUTLS_E_INVALID_REQUEST` will be returned.

The output buffer is allocated using `gnutls_malloc()` .

Returns: In case of failure a negative error code will be returned, and `GNUTLS_E_SUCCESS` (0) on success.

Since: 3.2.7

gnutls_pkcs11_obj_export_url

`int gnutls_pkcs11_obj_export_url (`*gnutls_pkcs11_obj_t* `obj`, *gnutls_pkcs11_url_type_t* `detailed`, *char* ** `url``)` [Function]

obj: Holds the PKCS 11 certificate

detailed: non zero if a detailed URL is required

url: will contain an allocated url

This function will export a URL identifying the given object.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_obj_flags_get_str

`char * gnutls_pkcs11_obj_flags_get_str (unsigned int flags)` [Function]

flags: holds the flags

This function given an or-sequence of `GNUTLS_PKCS11_OBJ_FLAG_MARK` , will return an allocated string with its description. The string needs to be deallocated using `gnutls_free()` .

Returns: If flags is zero `NULL` is returned, otherwise an allocated string.

Since: 3.3.7

gnutls_pkcs11_obj_get_exts

`int gnutls_pkcs11_obj_get_exts (gnutls_pkcs11_obj_t obj, gnutls_x509_ext_st ** exts, unsigned int * exts_size, unsigned int flags)` [Function]

obj: should contain a `gnutls_pkcs11_obj_t` type

exts: a pointer to a `gnutls_x509_ext_st` pointer

exts_size: will be updated with the number of `exts`

flags: Or sequence of `GNUTLS_PKCS11_OBJ_` * flags

This function will return information about attached extensions that associate to the provided object (which should be a certificate). The extensions are the attached p11-kit trust module extensions.

Each element of `exts` must be deinitialized using `gnutls_x509_ext_deinit()` while `exts` should be deallocated using `gnutls_free()` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.3.8

gnutls_pkcs11_obj_get_flags

`int gnutls_pkcs11_obj_get_flags (gnutls_pkcs11_obj_t obj, unsigned int * oflags)` [Function]

obj: The pkcs11 object

oflags: Will hold the output flags

This function will return the flags of the object. The `oflags` will be flags from `gnutls_pkcs11_obj_flags` . That is, the `GNUTLS_PKCS11_OBJ_FLAG_MARK_` * flags.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.7

gnutls_pkcs11_obj_get_info

`int gnutls_pkcs11_obj_get_info (`*gnutls_pkcs11_obj_t* `obj`, *gnutls_pkcs11_obj_info_t* `itype`, *void ** `output`, *size_t ** `output_size``)` [Function]

obj: should contain a `gnutls_pkcs11_obj_t` type

itype: Denotes the type of information requested

output: where output will be stored

output_size: contains the maximum size of the output buffer and will be overwritten with the actual size.

This function will return information about the PKCS11 certificate such as the label, id as well as token information where the key is stored.

When output is text, a null terminated string is written to `output` and its string length is written to `output_size` (without null terminator). If the buffer is too small, `output_size` will contain the expected buffer size (with null terminator for text) and return `GNUTLS_E_SHORT_MEMORY_BUFFER` .

In versions previously to 3.6.0 this function included the null terminator to `output_size` . After 3.6.0 the output size doesn't include the terminator character.

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 2.12.0

gnutls_pkcs11_obj_get_ptr

`int gnutls_pkcs11_obj_get_ptr (`*gnutls_pkcs11_obj_t* `obj`, *void *** `ptr`, *void *** `session`, *void *** `ohandle`, *unsigned long ** `slot_id`, *unsigned int* `flags`)` [Function]

obj: should contain a `gnutls_pkcs11_obj_t` type

ptr: will contain the CK_FUNCTION_LIST_PTR pointer (may be `NULL`)

session: will contain the CK_SESSION_HANDLE of the object

ohandle: will contain the CK_OBJECT_HANDLE of the object

slot_id: the identifier of the slot (may be `NULL`)

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

Obtains the PKCS11 session handles of an object. `session` and `ohandle` must be deinitialized by the caller. The returned pointers are independent of the `obj` lifetime.

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.6.3

gnutls_pkcs11_obj_get_type

`gnutls_pkcs11_obj_type_t gnutls_pkcs11_obj_get_type (`*gnutls_pkcs11_obj_t* `obj`)` [Function]

obj: Holds the PKCS 11 object

This function will return the type of the object being stored in the structure.

Returns: The type of the object

Since: 2.12.0

gnutls_pkcs11_obj_import_url

`int gnutls_pkcs11_obj_import_url (`*gnutls_pkcs11_obj_t* `obj`, *const char ** `url`, *unsigned int* `flags``)` [Function]

obj: The structure to store the object

url: a PKCS 11 url identifying the key

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

This function will "import" a PKCS 11 URL identifying an object (e.g. certificate) to the `gnutls_pkcs11_obj_t` type. This does not involve any parsing (such as X.509 or OpenPGP) since the `gnutls_pkcs11_obj_t` is format agnostic. Only data are transferred.

If the flag `GNUTLS_PKCS11_OBJ_FLAG_OVERWRITE_TRUSTMOD_EXT` is specified any certificate read, will have its extensions overwritten by any stapled extensions in the trust module.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_obj_init

`int gnutls_pkcs11_obj_init (`*gnutls_pkcs11_obj_t ** `obj``)` [Function]

obj: A pointer to the type to be initialized

This function will initialize a pkcs11 certificate structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_obj_list_import_url3

`int gnutls_pkcs11_obj_list_import_url3 (`*gnutls_pkcs11_obj_t ** `p_list`, *unsigned int ** `n_list`, *const char ** `url`, *unsigned int* `flags``)` [Function]

p_list: An uninitialized object list (may be `NULL`)

n_list: Initially should hold the maximum size of the list. Will contain the actual size.

url: A PKCS 11 url identifying a set of objects

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

This function will initialize and set values to an object list by using all objects identified by a PKCS 11 URL.

This function will enumerate all the objects specified by the PKCS11 URL provided. It expects an already allocated `p_list` which has * `n_list` elements, and that value will be updated to the actual number of present objects. The `p_list` objects will be initialized and set by this function. To obtain a list of all available objects use a `url` of 'pkcs11:'.

All returned objects must be deinitialized using `gnutls_pkcs11_obj_deinit()` .

The supported in this function flags are GNUTLS_PKCS11_OBJ_FLAG_LOGIN , GNUTLS_PKCS11_OBJ_FLAG_LOGIN_SO , GNUTLS_PKCS11_OBJ_FLAG_PRESENT_IN_TRUSTED_MODULE , GNUTLS_PKCS11_OBJ_FLAG_CRT , GNUTLS_PKCS11_OBJ_FLAG_PUBKEY , GNUTLS_PKCS11_OBJ_FLAG_PRIVKEY , GNUTLS_PKCS11_OBJ_FLAG_WITH_PRIVKEY , GNUTLS_PKCS11_OBJ_FLAG_MARK_CA , GNUTLS_PKCS11_OBJ_FLAG_MARK_TRUSTED , and since 3.5.1 the GNUTLS_PKCS11_OBJ_FLAG_OVERWRITE_TRUSTMOD_EXT .

On versions of GnuTLS prior to 3.4.0 the equivalent function was gnutls_pkcs11_obj_list_import_url() . That is also available on this version as a macro which maps to this function.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_obj_list_import_url4

int gnutls_pkcs11_obj_list_import_url4 (*gnutls_pkcs11_obj_t ** p_list*, *unsigned int * n_list*, *const char * url*, *unsigned int flags*) [Function]

p_list: An uninitialized object list (may be NULL)

n_list: It will contain the size of the list.

url: A PKCS 11 url identifying a set of objects

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

This function will enumerate all the objects specified by the PKCS11 URL provided. It will initialize and set values to the object pointer list (p_list) provided. To obtain a list of all available objects use a url of 'pkcs11:'.

All returned objects must be deinitialized using gnutls_pkcs11_obj_deinit() , and p_list must be deinitialized using gnutls_free() .

The supported in this function flags are GNUTLS_PKCS11_OBJ_FLAG_LOGIN , GNUTLS_PKCS11_OBJ_FLAG_LOGIN_SO , GNUTLS_PKCS11_OBJ_FLAG_PRESENT_IN_TRUSTED_MODULE , GNUTLS_PKCS11_OBJ_FLAG_CRT , GNUTLS_PKCS11_OBJ_FLAG_PUBKEY , GNUTLS_PKCS11_OBJ_FLAG_PRIVKEY , GNUTLS_PKCS11_OBJ_FLAG_WITH_PRIVKEY , GNUTLS_PKCS11_OBJ_FLAG_MARK_CA , GNUTLS_PKCS11_OBJ_FLAG_MARK_TRUSTED , and since 3.5.1 the GNUTLS_PKCS11_OBJ_FLAG_OVERWRITE_TRUSTMOD_EXT .

On versions of GnuTLS prior to 3.4.0 the equivalent function was gnutls_pkcs11_obj_list_import_url2() . That is also available on this version as a macro which maps to this function.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_obj_set_info

int gnutls_pkcs11_obj_set_info (*gnutls_pkcs11_obj_t obj*, *gnutls_pkcs11_obj_info_t itype*, *const void * data*, *size_t data_size*, *unsigned flags*) [Function]

obj: should contain a gnutls_pkcs11_obj_t type

itype: Denotes the type of information to be set

data: the data to set

data_size: the size of data

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

This function will set attributes on the provided object. Available options for `itype` are `GNUTLS_PKCS11_OBJ_LABEL` , `GNUTLS_PKCS11_OBJ_ID_HEX` , and `GNUTLS_PKCS11_OBJ_ID` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.4.0

gnutls_pkcs11_obj_set_pin_function

`void gnutls_pkcs11_obj_set_pin_function` (*gnutls_pkcs11_obj_t* `obj`, *gnutls_pin_callback_t* `fn`, *void * * `userdata`) [Function]

obj: The object structure

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides the global set using `gnutls_pkcs11_set_pin_function()` .

Since: 3.1.0

gnutls_pkcs11_privkey_cpy

`int gnutls_pkcs11_privkey_cpy` (*gnutls_pkcs11_privkey_t* `dst`, *gnutls_pkcs11_privkey_t* `src`) [Function]

dst: The destination key, which should be initialized.

src: The source key

This function will copy a private key from source to destination key. Destination has to be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_privkey_deinit

`void gnutls_pkcs11_privkey_deinit` (*gnutls_pkcs11_privkey_t* `key`) [Function]

key: the key to be deinitialized

This function will deinitialize a private key structure.

gnutls_pkcs11_privkey_export_pubkey

`int gnutls_pkcs11_privkey_export_pubkey` (*gnutls_pkcs11_privkey_t* `pkey`, *gnutls_x509_crt_fmt_t* `fmt`, *gnutls_datum_t * `data`, *unsigned int* `flags`) [Function]

pkey: The private key

fmt: the format of output params. PEM or DER.

data: will hold the public key

flags: should be zero

This function will extract the public key (modulus and public exponent) from the private key specified by the `url` private key. This public key will be stored in `pubkey` in the format specified by `fmt` . `pubkey` should be deinitialized using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.7

gnutls_pkcs11_privkey_export_url

`int gnutls_pkcs11_privkey_export_url` (*gnutls_pkcs11_privkey_t* `key`, *gnutls_pkcs11_url_type_t* `detailed`, *char ** * `url`) [Function]

key: Holds the PKCS 11 key

detailed: non zero if a detailed URL is required

url: will contain an allocated url

This function will export a URL identifying the given key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_privkey_generate

`int gnutls_pkcs11_privkey_generate` (*const char * * `url`, *gnutls_pk_algorithm_t* `pk`, *unsigned int* `bits`, *const char * * `label`, *unsigned int* `flags`) [Function]

url: a token URL

pk: the public key algorithm

bits: the security bits

label: a label

flags: should be zero

This function will generate a private key in the specified by the `url` token. The private key will be generate within the token and will not be exportable.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pkcs11_privkey_generate2

`int gnutls_pkcs11_privkey_generate2` (*const char * * `url`, *gnutls_pk_algorithm_t* `pk`, *unsigned int* `bits`, *const char * * `label`, *gnutls_x509_crt_fmt_t* `fmt`, *gnutls_datum_t * * `pubkey`, *unsigned int* `flags`) [Function]

url: a token URL

pk: the public key algorithm

bits: the security bits

label: a label

fmt: the format of output params. PEM or DER

pubkey: will hold the public key (may be `NULL`)

flags: zero or an OR'ed sequence of `GNUTLS_PKCS11_OBJ_FLAGs`

This function will generate a private key in the specified by the `url` token. The private key will be generate within the token and will not be exportable. This function will store the DER-encoded public key in the SubjectPublicKeyInfo format in `pubkey` . The `pubkey` should be deinitialized using `gnutls_free()` .

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.5

gnutls_pkcs11_privkey_generate3

`int gnutls_pkcs11_privkey_generate3` (*const char * `url`, gnutls_pk_algorithm_t `pk`, unsigned int `bits`, const char * `label`, const gnutls_datum_t * `cid`, gnutls_x509_crt_fmt_t `fmt`, gnutls_datum_t * `pubkey`, unsigned int `key_usage`, unsigned int `flags`*) [Function]

url: a token URL

pk: the public key algorithm

bits: the security bits

label: a label

cid: The CKA_ID to use for the new object

fmt: the format of output params. PEM or DER

pubkey: will hold the public key (may be `NULL`)

key_usage: One of GNUTLS_KEY_*

flags: zero or an OR'ed sequence of `GNUTLS_PKCS11_OBJ_FLAGs`

This function will generate a private key in the specified by the `url` token. The private key will be generate within the token and will not be exportable. This function will store the DER-encoded public key in the SubjectPublicKeyInfo format in `pubkey` . The `pubkey` should be deinitialized using `gnutls_free()` .

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

Since 3.6.3 the objects are marked as sensitive by default unless `GNUTLS_PKCS11_OBJ_FLAG_MARK_NOT_SENSITIVE` is specified.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pkcs11_privkey_get_info

`int gnutls_pkcs11_privkey_get_info` (*gnutls_pkcs11_privkey_t* `pkey`, *gnutls_pkcs11_obj_info_t* `itype`, *void *** `output`, *size_t ** `output_size`) [Function]

pkey: should contain a `gnutls_pkcs11_privkey_t` type

itype: Denotes the type of information requested

output: where output will be stored

output_size: contains the maximum size of the output and will be overwritten with actual

This function will return information about the PKCS 11 private key such as the label, id as well as token information where the key is stored. When output is text it returns null terminated string although `output_size` contains the size of the actual data only.

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

gnutls_pkcs11_privkey_get_pk_algorithm

`int gnutls_pkcs11_privkey_get_pk_algorithm` (*gnutls_pkcs11_privkey_t* `key`, *unsigned int ** `bits`) [Function]

key: should contain a `gnutls_pkcs11_privkey_t` type

bits: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of a private key.

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

gnutls_pkcs11_privkey_import_url

`int gnutls_pkcs11_privkey_import_url` (*gnutls_pkcs11_privkey_t* `pkey`, *const char ** `url`, *unsigned int* `flags`) [Function]

pkey: The private key

url: a PKCS 11 url identifying the key

flags: Or sequence of GNUTLS_PKCS11_OBJ_* flags

This function will "import" a PKCS 11 URL identifying a private key to the `gnutls_pkcs11_privkey_t` type. In reality since in most cases keys cannot be exported, the private key structure is being associated with the available operations on the token.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_privkey_init

`int gnutls_pkcs11_privkey_init` (*gnutls_pkcs11_privkey_t ** `key`) [Function]

key: A pointer to the type to be initialized

This function will initialize an private key structure. This structure can be used for accessing an underlying PKCS11 object.

In versions of GnuTLS later than 3.5.11 the object is protected using locks and a single `gnutls_pkcs11_privkey_t` can be re-used by many threads. However, for performance it is recommended to utilize one object per key per thread.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_privkey_set_pin_function

`void gnutls_pkcs11_privkey_set_pin_function` [Function]
(*gnutls_pkcs11_privkey_t* `key`, *gnutls_pin_callback_t* `fn`, *void ** `userdata`)

key: The private key

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides the global set using `gnutls_pkcs11_set_pin_function()` .

Since: 3.1.0

gnutls_pkcs11_privkey_status

`unsigned gnutls_pkcs11_privkey_status` [Function]
(*gnutls_pkcs11_privkey_t* `key`)

key: Holds the key

Checks the status of the private key token.

Returns: this function will return non-zero if the token holding the private key is still available (inserted), and zero otherwise.

Since: 3.1.9

gnutls_pkcs11_reinit

`int gnutls_pkcs11_reinit ( void)` [Function]

This function will reinitialize the PKCS 11 subsystem in gnutls. This is required by PKCS 11 when an application uses `fork()` . The reinitialization function must be called on the child.

Note that since GnuTLS 3.3.0, the reinitialization of the PKCS `11` subsystem occurs automatically after fork.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pkcs11_set_pin_function

`void gnutls_pkcs11_set_pin_function (`*gnutls_pin_callback_t* `fn`, [Function]
*void ** `userdata`)

fn: The PIN callback, a `gnutls_pin_callback_t()` function.

userdata: data to be supplied to callback

This function will set a callback function to be used when a PIN is required for PKCS 11 operations. See `gnutls_pin_callback_t()` on how the callback should behave.

Since: 2.12.0

gnutls_pkcs11_set_token_function

`void gnutls_pkcs11_set_token_function` [Function]
(*gnutls_pkcs11_token_callback_t* `fn`, *void ** `userdata`)

fn: The token callback

userdata: data to be supplied to callback

This function will set a callback function to be used when a token needs to be inserted to continue PKCS 11 operations.

Since: 2.12.0

gnutls_pkcs11_token_check_mechanism

`unsigned gnutls_pkcs11_token_check_mechanism` (*const char ** `url`, *unsigned long* `mechanism`, *void ** `ptr`, *unsigned* `psize`, *unsigned* `flags`) [Function]

url: should contain a PKCS 11 URL

mechanism: The PKCS `11` mechanism ID

ptr: if set it should point to a CK_MECHANISM_INFO struct

psize: the size of CK_MECHANISM_INFO struct (for safety)

flags: must be zero

This function will return whether a mechanism is supported by the given token. If the mechanism is supported and `ptr` is set, it will be updated with the token information.

Returns: Non-zero if the mechanism is supported or zero otherwise.

Since: 3.6.0

gnutls_pkcs11_token_get_flags

`int gnutls_pkcs11_token_get_flags` (*const char ** `url`, *unsigned int ** `flags`) [Function]

url: should contain a PKCS 11 URL

flags: The output flags (GNUTLS_PKCS11_TOKEN_*)

This function will return information about the PKCS 11 token flags.

The supported flags are: `GNUTLS_PKCS11_TOKEN_HW` and `GNUTLS_PKCS11_TOKEN_TRUSTED` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 2.12.0

gnutls_pkcs11_token_get_info

`int gnutls_pkcs11_token_get_info (`*const char * `url`*, *gnutls_pkcs11_token_info_t `ttype`*, *void * `output`*, *size_t * `output_size`*)` [Function]

url: should contain a PKCS 11 URL

ttype: Denotes the type of information requested

output: where output will be stored

output_size: contains the maximum size of the output buffer and will be overwritten with the actual size.

This function will return information about the PKCS 11 token such as the label, id, etc.

When output is text, a null terminated string is written to `output` and its string length is written to `output_size` (without null terminator). If the buffer is too small, `output_size` will contain the expected buffer size (with null terminator for text) and return `GNUTLS_E_SHORT_MEMORY_BUFFER` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 2.12.0

gnutls_pkcs11_token_get_mechanism

`int gnutls_pkcs11_token_get_mechanism (`*const char * `url`*, *unsigned int `idx`*, *unsigned long * `mechanism`*)` [Function]

url: should contain a PKCS 11 URL

idx: The index of the mechanism

mechanism: The PKCS 11 mechanism ID

This function will return the names of the supported mechanisms by the token. It should be called with an increasing index until it return GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE.

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 2.12.0

gnutls_pkcs11_token_get_ptr

`int gnutls_pkcs11_token_get_ptr (`*const char * `url`*, *void ** `ptr`*, *unsigned long * `slot_id`*, *unsigned int `flags`*)` [Function]

url: should contain a PKCS11 URL identifying a token

ptr: will contain the CK_FUNCTION_LIST_PTR pointer

slot_id: will contain the slot_id (may be `NULL`)

flags: should be zero

This function will return the function pointer of the specified token by the URL. The returned pointers are valid until gnutls is deinitialized, c.f. `_global_deinit()` .

Returns: `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

Since: 3.6.3

gnutls_pkcs11_token_get_random

`int gnutls_pkcs11_token_get_random (`*const char * `token_url`*, *void * `rnddata`*, *size_t `len`*)` [Function]

token_url: A PKCS 11 URL specifying a token

rnddata: A pointer to the memory area to be filled with random data

len: The number of bytes of randomness to request

This function will get random data from the given token. It will store rnddata and fill the memory pointed to by rnddata with len random bytes from the token.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_token_get_url

`int gnutls_pkcs11_token_get_url (`*unsigned int `seq`*, *gnutls_pkcs11_url_type_t `detailed`*, *char ** `url`*)` [Function]

seq: sequence number starting from 0

detailed: non zero if a detailed URL is required

url: will contain an allocated url

This function will return the URL for each token available in system. The url has to be released using `gnutls_free()`

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the sequence number exceeds the available tokens, otherwise a negative error value.

Since: 2.12.0

gnutls_pkcs11_token_init

`int gnutls_pkcs11_token_init (`*const char * `token_url`*, *const char * `so_pin`*, *const char * `label`*)` [Function]

token_url: A PKCS 11 URL specifying a token

so_pin: Security Officer's PIN

label: A name to be used for the token

This function will initialize (format) a token. If the token is at a factory defaults state the security officer's PIN given will be set to be the default. Otherwise it should match the officer's PIN.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_token_set_pin

`int gnutls_pkcs11_token_set_pin (`*const char * `token_url`*, *const char * `oldpin`*, *const char * `newpin`*, *unsigned int `flags`*)` [Function]

token_url: A PKCS 11 URL specifying a token

oldpin: old user's PIN

newpin: new user's PIN

flags: one of `gnutls_pin_flag_t` .

This function will modify or set a user or administrator's PIN for the given token. If it is called to set a PIN for first time the oldpin must be `NULL` . When setting the admin's PIN with the `GNUTLS_PIN_SO` flag, the `oldpin` value must be provided (this requirement is relaxed after GnuTLS 3.6.5 since which the PIN will be requested if missing).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_pkcs11_type_get_name

`const char * gnutls_pkcs11_type_get_name` [Function]
(*gnutls_pkcs11_obj_type_t* `type`)

type: Holds the PKCS 11 object type, a `gnutls_pkcs11_obj_type_t` .

This function will return a human readable description of the PKCS11 object type `obj` . It will return `"Unknown"` for unknown types.

Returns: human readable string labeling the PKCS11 object type `type` .

Since: 2.12.0

gnutls_x509_crt_import_pkcs11

`int gnutls_x509_crt_import_pkcs11` (*gnutls_x509_crt_t* `crt`, [Function]
gnutls_pkcs11_obj_t `pkcs11_crt`)

crt: A certificate of type `gnutls_x509_crt_t`

pkcs11_crt: A PKCS 11 object that contains a certificate

This function will import a PKCS 11 certificate to a `gnutls_x509_crt_t` structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_x509_crt_list_import_pkcs11

`int gnutls_x509_crt_list_import_pkcs11` (*gnutls_x509_crt_t ** [Function]
`certs`, *unsigned int* `cert_max`, *gnutls_pkcs11_obj_t * const* `objs`, *unsigned int* `flags`)

certs: A list of certificates of type `gnutls_x509_crt_t`

cert_max: The maximum size of the list

objs: A list of PKCS 11 objects

flags: 0 for now

This function will import a PKCS 11 certificate list to a list of `gnutls_x509_crt_t` type. These must not be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

E.8 TPM API

The following functions are to be used for TPM handling. Their prototypes lie in `gnutls/tpm.h`.

gnutls_tpm_get_registered

`int gnutls_tpm_get_registered (gnutls_tpm_key_list_t * list)` [Function]

list: a list to store the keys

This function will get a list of stored keys in the TPM. The uuid of those keys

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_tpm_key_list_deinit

`void gnutls_tpm_key_list_deinit (gnutls_tpm_key_list_t list)` [Function]

list: a list of the keys

This function will deinitialize the list of stored keys in the TPM.

Since: 3.1.0

gnutls_tpm_key_list_get_url

`int gnutls_tpm_key_list_get_url (gnutls_tpm_key_list_t list, unsigned int idx, char ** url, unsigned int flags)` [Function]

list: a list of the keys

idx: The index of the key (starting from zero)

url: The URL to be returned

flags: should be zero

This function will return for each given index a URL of the corresponding key. If the provided index is out of bounds then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_tpm_privkey_delete

`int gnutls_tpm_privkey_delete (const char * url, const char * srk_password)` [Function]

url: the URL describing the key

srk_password: a password for the SRK key

This function will unregister the private key from the TPM chip.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_tpm_privkey_generate

`int gnutls_tpm_privkey_generate (`*gnutls_pk_algorithm_t* `pk`, *unsigned int* `bits`, *const char ** `srk_password`, *const char ** `key_password`, *gnutls_tpmkey_fmt_t* `format`, *gnutls_x509_crt_fmt_t* `pub_format`, *gnutls_datum_t ** `privkey`, *gnutls_datum_t ** `pubkey`, *unsigned int* `flags`)` [Function]

pk: the public key algorithm

bits: the security bits

srk_password: a password to protect the exported key (optional)

key_password: the password for the TPM (optional)

format: the format of the private key

pub_format: the format of the public key

privkey: the generated key

pubkey: the corresponding public key (may be null)

flags: should be a list of GNUTLS_TPM_* flags

This function will generate a private key in the TPM chip. The private key will be generated within the chip and will be exported in a wrapped with TPM's master key form. Furthermore the wrapped key can be protected with the provided `password` .

Note that bits in TPM is quantized value. If the input value is not one of the allowed values, then it will be quantized to one of 512, 1024, 2048, 4096, 8192 and 16384.

Allowed flags are:

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

E.9 Abstract key API

The following functions are to be used for abstract key handling. Their prototypes lie in `gnutls/abstract.h`.

gnutls_certificate_set_key

`int gnutls_certificate_set_key (`*gnutls_certificate_credentials_t* `res`, *const char *** `names`, *int* `names_size`, *gnutls_pcert_st ** `pcert_list`, *int* `pcert_list_size`, *gnutls_privkey_t* `key`)` [Function]

res: is a `gnutls_certificate_credentials_t` type.

names: is an array of DNS names belonging to the public-key (NULL if none)

names_size: holds the size of the names list

pcert_list: contains a certificate list (chain) or raw public-key

pcert_list_size: holds the size of the certificate list

key: is a `gnutls_privkey_t` key corresponding to the first public-key in pcert_list

This function sets a public/private key pair in the gnutls_certificate_credentials_t type. The given public key may be encapsulated in a certificate or can be given

as a raw key. This function may be called more than once, in case multiple key pairs exist for the server. For clients that want to send more than their own end-entity certificate (e.g., also an intermediate CA cert), the full certificate chain must be provided in `pcert_list` .

Note that the `key` will become part of the credentials structure and must not be deallocated. It will be automatically deallocated when the `res` structure is deinitialized.

If this function fails, the `res` structure is at an undefined state and it must not be reused to load other keys or certificates.

Note that, this function by default returns zero on success and a negative value on error. Since 3.5.6, when the flag `GNUTLS_CERTIFICATE_API_V2` is set using `gnutls_certificate_set_flags()` it returns an index (greater or equal to zero). That index can be used for other functions to refer to the added key-pair.

Since GnuTLS 3.6.6 this function also handles raw public keys.

Returns: On success this functions returns zero, and otherwise a negative value on error (see above for modifying that behavior).

Since: 3.0

gnutls_certificate_set_retrieve_function2

`void gnutls_certificate_set_retrieve_function2` [Function]
(*gnutls_certificate_credentials_t* ***cred***,
*gnutls_certificate_retrieve_function2 ** ***func***)

cred: is a `gnutls_certificate_credentials_t` type.

func: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. The callback will take control only if a certificate is requested by the peer.

The callback's function prototype is: int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_pcert_st** pcert, unsigned int *pcert_length, gnutls_privkey_t * pkey);

`req_ca_dn` is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. This is a hint and typically the client should send a certificate that is signed by one of these CAs. These names, when available, are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()` .

`pk_algos` contains a list with server's acceptable public key algorithms. The certificate returned should support the server's given algorithms.

`pcert` should contain a single certificate and public key or a list of them.

`pcert_length` is the size of the previous list.

`pkey` is the private key.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received. All the provided by the callback values will not be released or modified by gnutls.

In server side pk_algos and req_ca_dn are NULL.

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated. If both certificates are set in the credentials and a callback is available, the callback takes predence.

Since: 3.0

gnutls_certificate_set_retrieve_function3

`void gnutls_certificate_set_retrieve_function3` [Function]
(*gnutls_certificate_credentials_t* `cred`,
*gnutls_certificate_retrieve_function3 ** `func`)

cred: is a `gnutls_certificate_credentials_t` type.

func: is the callback function

This function sets a callback to be called in order to retrieve the certificate and OCSP responses to be used in the handshake. `func` will be called only if the peer requests a certificate either during handshake or during post-handshake authentication.

The callback's function prototype is defined in 'abstract.h':

int gnutls_certificate_retrieve_function3(gnutls_session_t, const struct gnutls_cert_retr_st *info, gnutls_pcert_st **certs, unsigned int *pcert_length, gnutls_ocsp_data_st **ocsp, unsigned int *ocsp_length, gnutls_privkey_t *privkey, unsigned int *flags);

The info field of the callback contains: `req_ca_dn` which is a list with the CA names that the server considers trusted. This is a hint and typically the client should send a certificate that is signed by one of these CAs. These names, when available, are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()` . `pk_algos` contains a list with server's acceptable public key algorithms. The certificate returned should support the server's given algorithms.

The callback should fill-in the following values.

`pcert` should contain an allocated list of certificates and public keys. `pcert_length` is the size of the previous list. `ocsp` should contain an allocated list of OCSP responses. `ocsp_length` is the size of the previous list. `pkey` is the private key.

If flags in the callback are set to `GNUTLS_CERT_RETR_DEINIT_ALL` then all provided values must be allocated using `gnutls_malloc()` , and will be released by gnutls; otherwise they will not be touched by gnutls.

The callback function should set the certificate and OCSP response list to be sent, and return 0 on success. If no certificates are available, the `pcert_length` and `ocsp_length` should be set to zero. The return value (-1) indicates error and the handshake will be terminated. If both certificates are set in the credentials and a callback is available, the callback takes predence.

Since: 3.6.3

gnutls_pcert_deinit

`void gnutls_pcert_deinit (gnutls_pcert_st * pcert)` [Function]

pcert: The structure to be deinitialized

This function will deinitialize a pcert structure.

Since: 3.0

gnutls_pcert_export_openpgp

`int gnutls_pcert_export_openpgp (gnutls_pcert_st * pcert, gnutls_openpgp_crt_t * crt)` [Function]

pcert: The pcert structure.

crt: An initialized `gnutls_openpgp_crt_t` .

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.4.0

gnutls_pcert_export_x509

`int gnutls_pcert_export_x509 (gnutls_pcert_st * pcert, gnutls_x509_crt_t * crt)` [Function]

pcert: The pcert structure.

crt: An initialized `gnutls_x509_crt_t` .

Converts the given `gnutls_pcert_t` type into a `gnutls_x509_crt_t` . This function only works if the type of `pcert` is `GNUTLS_CRT_X509` . When successful, the value written to `crt` must be freed with `gnutls_x509_crt_deinit()` when no longer needed.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pcert_import_openpgp

`int gnutls_pcert_import_openpgp (gnutls_pcert_st * pcert, gnutls_openpgp_crt_t crt, unsigned int flags)` [Function]

pcert: The pcert structure

crt: The raw certificate to be imported

flags: zero for now

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.0

gnutls_pcert_import_openpgp_raw

`int gnutls_pcert_import_openpgp_raw (`*gnutls_pcert_st * `pcert`*, *const gnutls_datum_t * `cert`*, *gnutls_openpgp_crt_fmt_t `format`*, *gnutls_openpgp_keyid_t `keyid`*, *unsigned int `flags`*`)` [Function]

pcert: The pcert structure

cert: The raw certificate to be imported

format: The format of the certificate

keyid: The key ID to use (NULL for the master key)

flags: zero for now

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.0

gnutls_pcert_import_rawpk

`int gnutls_pcert_import_rawpk (`*gnutls_pcert_st* `pcert`*, *gnutls_pubkey_t `pubkey`*, *unsigned int `flags`*`)` [Function]

pcert: The pcert structure to import the data into.

pubkey: The raw public-key in `gnutls_pubkey_t` format to be imported

flags: zero for now

This convenience function will import (i.e. convert) the given raw public key `pubkey` into a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` . The given `pubkey` must not be deinitialized because it will be associated with the given `pcert` structure and will be deinitialized with it.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.6

gnutls_pcert_import_rawpk_raw

`int gnutls_pcert_import_rawpk_raw (`*gnutls_pcert_st* `pcert`*, *const gnutls_datum_t* `rawpubkey`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `key_usage`*, *unsigned int `flags`*`)` [Function]

pcert: The pcert structure to import the data into.

rawpubkey: The raw public-key in `gnutls_datum_t` format to be imported.

format: The format of the raw public-key. DER or PEM.

key_usage: An ORed sequence of `GNUTLS_KEY_` * flags.

flags: zero for now

This convenience function will import (i.e. convert) the given raw public key `rawpubkey` into a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` . Note that the caller is responsible for freeing `rawpubkey` . All necessary values will be copied into `pcert` .

Key usage (as defined by X.509 extension (2.5.29.15)) can be explicitly set because there is no certificate structure around the key to define this value. See for more info `gnutls_x509_crt_get_key_usage()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.6

gnutls_pcert_import_x509

`int gnutls_pcert_import_x509 (`*gnutls_pcert_st * `pcert`*, *gnutls_x509_crt_t `crt`*, *unsigned int `flags`*`)` [Function]

pcert: The pcert structure

crt: The certificate to be imported

flags: zero for now

This convenience function will import the given certificate to a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` ;

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pcert_import_x509_list

`int gnutls_pcert_import_x509_list (`*gnutls_pcert_st * `pcert_list`*, *gnutls_x509_crt_t * `crt`*, *unsigned * `ncrt`*, *unsigned int `flags`*`)` [Function]

pcert_list: The structures to store the certificates; must not contain initialized `gnutls_pcert_st` structures.

crt: The certificates to be imported

ncrt: The number of certificates in `crt` ; will be updated if necessary

flags: zero or `GNUTLS_X509_CRT_LIST_SORT`

This convenience function will import the given certificates to an already allocated set of `gnutls_pcert_st` structures. The structures must be deinitialized afterwards using `gnutls_pcert_deinit()` . `pcert_list` should contain space for at least `ncrt` elements.

In the case `GNUTLS_X509_CRT_LIST_SORT` is specified and that function cannot sort the list, `GNUTLS_E_CERTIFICATE_LIST_UNSORTED` will be returned. Currently sorting can fail if the list size exceeds an internal constraint (16).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_pcert_import_x509_raw

`int gnutls_pcert_import_x509_raw (`*gnutls_pcert_st * `pcert`*, *const gnutls_datum_t * `cert`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `flags`*`)` [Function]

pcert: The pcert structure

cert: The raw certificate to be imported

format: The format of the certificate

flags: zero for now

This convenience function will import the given certificate to a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` ;

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pcert_list_import_x509_file

`int gnutls_pcert_list_import_x509_file (`*gnutls_pcert_st * `pcert_list`*, *unsigned * `pcert_list_size`*, *const char * `file`*, *gnutls_x509_crt_fmt_t `format`*, *gnutls_pin_callback_t `pin_fn`*, *void * `pin_fn_userdata`*, *unsigned int `flags`*`)` [Function]

pcert_list: The structures to store the certificates; must not contain initialized `gnutls_pcert_st` structures.

pcert_list_size: Initially must hold the maximum number of certs. It will be updated with the number of certs available.

file: A file or supported URI with the certificates to load

format: `GNUTLS_X509_FMT_DER` or `GNUTLS_X509_FMT_PEM` if a file is given

pin_fn: a PIN callback if not globally set

pin_fn_userdata: parameter for the PIN callback

flags: zero or flags from `gnutls_certificate_import_flags`

This convenience function will import a certificate chain from the given file or supported URI to `gnutls_pcert_st` structures. The structures must be deinitialized afterwards using `gnutls_pcert_deinit()` .

This function will always return a sorted certificate chain.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value; if the `pcert` list doesn't have enough space `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Since: 3.6.3

gnutls_pcert_list_import_x509_raw

`int gnutls_pcert_list_import_x509_raw (`*gnutls_pcert_st * `pcert_list`, unsigned int * `pcert_list_size`, const gnutls_datum_t * `data`, gnutls_x509_crt_fmt_t `format`, unsigned int `flags`*`)` [Function]

pcert_list: The structures to store the certificates; must not contain initialized `gnutls_pcert_st` structures.

pcert_list_size: Initially must hold the maximum number of certs. It will be updated with the number of certs available.

data: The certificates.

format: One of DER or PEM.

flags: must be (0) or an OR'd sequence of gnutls_certificate_import_flags.

This function will import the provided DER or PEM encoded certificates to an already allocated set of `gnutls_pcert_st` structures. The structures must be deinitialized afterwards using `gnutls_pcert_deinit()` . `pcert_list` should contain space for at least `pcert_list_size` elements.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value; if the `pcert` list doesn't have enough space `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Since: 3.0

gnutls_privkey_decrypt_data

`int gnutls_privkey_decrypt_data (`*gnutls_privkey_t `key`, unsigned int `flags`, const gnutls_datum_t * `ciphertext`, gnutls_datum_t * `plaintext`*`)` [Function]

key: Holds the key

flags: zero for now

ciphertext: holds the data to be decrypted

plaintext: will contain the decrypted data, allocated with `gnutls_malloc()`

This function will decrypt the given data using the algorithm supported by the private key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_decrypt_data2

`int gnutls_privkey_decrypt_data2 (`*gnutls_privkey_t `key`, unsigned int `flags`, const gnutls_datum_t * `ciphertext`, unsigned char * `plaintext`, size_t `plaintext_size`*`)` [Function]

key: Holds the key

flags: zero for now

ciphertext: holds the data to be decrypted

plaintext: a preallocated buffer that will be filled with the plaintext

plaintext_size: in/out size of the plaintext

This function will decrypt the given data using the algorithm supported by the private key. Unlike with `gnutls_privkey_decrypt_data()` this function operates in constant time and constant memory access.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.5

gnutls_privkey_deinit

`void gnutls_privkey_deinit` (*gnutls_privkey_t* `key`) [Function]

key: The key to be deinitialized

This function will deinitialize a private key structure.

Since: 2.12.0

gnutls_privkey_export_dsa_raw

`int gnutls_privkey_export_dsa_raw` (*gnutls_privkey_t* `key`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `g`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `x`) [Function]

key: Holds the public key

p: will hold the p

q: will hold the q

g: will hold the g

y: will hold the y

x: will hold the x

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_privkey_export_dsa_raw2

`int gnutls_privkey_export_dsa_raw2` (*gnutls_privkey_t* `key`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `g`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `x`, *unsigned int* `flags`) [Function]

key: Holds the public key

p: will hold the p

q: will hold the q

g: will hold the g

y: will hold the y

x: will hold the x

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_privkey_export_ecc_raw

`int gnutls_privkey_export_ecc_raw` (*gnutls_privkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `k`) [Function]

key: Holds the public key

curve: will hold the curve

x: will hold the x-coordinate

y: will hold the y-coordinate

k: will hold the private key

This function will export the ECC private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_privkey_export_ecc_raw2

`int gnutls_privkey_export_ecc_raw2` (*gnutls_privkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `k`, *unsigned int* `flags`) [Function]

key: Holds the public key

curve: will hold the curve

x: will hold the x-coordinate

y: will hold the y-coordinate

k: will hold the private key

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the ECC private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_privkey_export_gost_raw2

`int gnutls_privkey_export_gost_raw2` (*gnutls_privkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_digest_algorithm_t* * `digest`, *gnutls_gost_paramset_t* * `paramset`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *gnutls_datum_t* * `k`, *unsigned int* `flags`) [Function]

key: Holds the public key

curve: will hold the curve

digest: will hold the digest

paramset: will hold the GOST parameter set ID

x: will hold the x-coordinate

y: will hold the y-coordinate

k: will hold the private key

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the GOST private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Note: parameters will be stored with least significant byte first. On version 3.6.3 this was incorrectly returned in big-endian format.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.3

gnutls_privkey_export_openpgp

`int gnutls_privkey_export_openpgp` (*gnutls_privkey_t* `pkey`, *gnutls_openpgp_privkey_t* * `key`) [Function]

pkey: The private key

key: Location for the key to be exported.

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.4.0

gnutls_privkey_export_pkcs11

`int gnutls_privkey_export_pkcs11` (*gnutls_privkey_t* `pkey`, *gnutls_pkcs11_privkey_t* * `key`) [Function]

pkey: The private key

key: Location for the key to be exported.

Converts the given abstract private key to a `gnutls_pkcs11_privkey_t` type. The key must be of type `GNUTLS_PRIVKEY_PKCS11` . The key returned in `key` must be deinitialized with `gnutls_pkcs11_privkey_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_privkey_export_rsa_raw

`int gnutls_privkey_export_rsa_raw (`*gnutls_privkey_t* `key`, *gnutls_datum_t* * `m`, *gnutls_datum_t* * `e`, *gnutls_datum_t* * `d`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `u`, *gnutls_datum_t* * `e1`, *gnutls_datum_t* * `e2`)` [Function]

key: Holds the certificate

m: will hold the modulus

e: will hold the public exponent

d: will hold the private exponent

p: will hold the first prime (p)

q: will hold the second prime (q)

u: will hold the coefficient

e1: will hold e1 = d mod (p-1)

e2: will hold e2 = d mod (q-1)

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum. For EdDSA keys, the `y` value should be `NULL` .

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_privkey_export_rsa_raw2

`int gnutls_privkey_export_rsa_raw2 (`*gnutls_privkey_t* `key`, *gnutls_datum_t* * `m`, *gnutls_datum_t* * `e`, *gnutls_datum_t* * `d`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `u`, *gnutls_datum_t* * `e1`, *gnutls_datum_t* * `e2`, *unsigned int* `flags`)` [Function]

key: Holds the certificate

m: will hold the modulus

e: will hold the public exponent

d: will hold the private exponent

p: will hold the first prime (p)

q: will hold the second prime (q)

u: will hold the coefficient

e1: will hold e1 = d mod (p-1)

e2: will hold e2 = d mod (q-1)

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_privkey_export_x509

`int gnutls_privkey_export_x509` (*gnutls_privkey_t* `pkey`, *gnutls_x509_privkey_t* * `key`) [Function]

pkey: The private key

key: Location for the key to be exported.

Converts the given abstract private key to a `gnutls_x509_privkey_t` type. The abstract key must be of type `GNUTLS_PRIVKEY_X509` . The input `key` must not be initialized. The key returned in `key` should be deinitialized using `gnutls_x509_privkey_deinit()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_privkey_generate

`int gnutls_privkey_generate` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`) [Function]

pkey: An initialized private key

algo: is one of the algorithms in `gnutls_pk_algorithm_t` .

bits: the size of the parameters to generate

flags: Must be zero or flags from `gnutls_privkey_flags_t` .

This function will generate a random private key. Note that this function must be called on an initialized private key.

The flag `GNUTLS_PRIVKEY_FLAG_PROVABLE` instructs the key generation process to use algorithms like Shawe-Taylor (from FIPS PUB186-4) which generate provable parameters out of a seed for RSA and DSA keys. See `gnutls_privkey_generate2()` for more information.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro. The input to the macro is any curve from `gnutls_ecc_curve_t` .

For DSA keys, if the subgroup size needs to be specified check the `GNUTLS_SUBGROUP_TO_BITS()` macro.

It is recommended to do not set the number of `bits` directly, use `gnutls_sec_param_to_pk_bits()` instead .

See also `gnutls_privkey_generate2()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_privkey_generate2

`int gnutls_privkey_generate2` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`, *const gnutls_keygen_data_st* * `data`, *unsigned* `data_size`) [Function]

pkey: The private key

algo: is one of the algorithms in `gnutls_pk_algorithm_t` .

bits: the size of the modulus

flags: Must be zero or flags from `gnutls_privkey_flags_t` .

data: Allow specifying `gnutls_keygen_data_st` types such as the seed to be used.

data_size: The number of `data` available.

This function will generate a random private key. Note that this function must be called on an initialized private key.

The flag `GNUTLS_PRIVKEY_FLAG_PROVABLE` instructs the key generation process to use algorithms like Shawe-Taylor (from FIPS PUB186-4) which generate provable parameters out of a seed for RSA and DSA keys. On DSA keys the PQG parameters are generated using the seed, while on RSA the two primes. To specify an explicit seed (by default a random seed is used), use the `data` with a `GNUTLS_KEYGEN_SEED` type.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

To export the generated keys in memory or in files it is recommended to use the PKCS8 form as it can handle all key types, and can store additional parameters such as the seed, in case of provable RSA or DSA keys. Generated keys can be exported in memory using `gnutls_privkey_export_x509()` , and then with `gnutls_x509_privkey_export2_pkcs8()` .

If key generation is part of your application, avoid setting the number of bits directly, and instead use `gnutls_sec_param_to_pk_bits()` . That way the generated keys will adapt to the security levels of the underlying GnuTLS library.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_privkey_get_pk_algorithm

`int gnutls_privkey_get_pk_algorithm (`*gnutls_privkey_t* `key`, *unsigned int ** `bits``)` [Function]

key: should contain a `gnutls_privkey_t` type

bits: If set will return the number of bits of the parameters (may be NULL)

This function will return the public key algorithm of a private key and if possible will return a number of bits that indicates the security parameter of the key.

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

Since: 2.12.0

gnutls_privkey_get_seed

`int gnutls_privkey_get_seed (`*gnutls_privkey_t* `key`, *gnutls_digest_algorithm_t ** `digest`, *void ** `seed`, *size_t ** `seed_size``)` [Function]

key: should contain a `gnutls_privkey_t` type

digest: if non-NULL it will contain the digest algorithm used for key generation (if applicable)

seed: where seed will be copied to

seed_size: originally holds the size of `seed` , will be updated with actual size

This function will return the seed that was used to generate the given private key. That function will succeed only if the key was generated as a provable key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.5.0

gnutls_privkey_get_spki

`int gnutls_privkey_get_spki` (*gnutls_privkey_t* `privkey`, *gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

privkey: a public key of type `gnutls_privkey_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_privkey_spki_t`

flags: must be zero

This function will return the public key information if available. The provided `spki` must be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_privkey_get_type

`gnutls_privkey_type_t gnutls_privkey_get_type` (*gnutls_privkey_t* `key`) [Function]

key: should contain a `gnutls_privkey_t` type

This function will return the type of the private key. This is actually the type of the subsystem used to set this private key.

Returns: a member of the `gnutls_privkey_type_t` enumeration on success, or a negative error code on error.

Since: 2.12.0

gnutls_privkey_import_dsa_raw

`int gnutls_privkey_import_dsa_raw` (*gnutls_privkey_t* `key`, *const gnutls_datum_t * p*, *const gnutls_datum_t * q*, *const gnutls_datum_t * g*, *const gnutls_datum_t * y*, *const gnutls_datum_t * x*) [Function]

key: The structure to store the parsed key

p: holds the p

q: holds the q

g: holds the g

y: holds the y

x: holds the x

This function will convert the given DSA raw parameters to the native `gnutls_privkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_privkey_import_ecc_raw

`int gnutls_privkey_import_ecc_raw` (*gnutls_privkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *const gnutls_datum_t * x*, *const gnutls_datum_t * y*, *const gnutls_datum_t * k*) [Function]

key: The key

curve: holds the curve

x: holds the x-coordinate

y: holds the y-coordinate

k: holds the k (private key)

This function will convert the given elliptic curve parameters to the native `gnutls_privkey_t` format. The output will be stored in `key` .

In EdDSA curves the `y` parameter should be `NULL` and the `x` and `k` parameters must be in the native format for the curve.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_privkey_import_ext

`int gnutls_privkey_import_ext` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `pk`, *void * userdata*, *gnutls_privkey_sign_func* `sign_func`, *gnutls_privkey_decrypt_func* `decrypt_func`, *unsigned int* `flags`) [Function]

pkey: The private key

pk: The public key algorithm

userdata: private data to be provided to the callbacks

sign_func: callback for signature operations

decrypt_func: callback for decryption operations

flags: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` type. At least one of the two callbacks must be non-null.

Note that the signing function is supposed to "raw" sign data, i.e., without any hashing or preprocessing. In case of RSA the DigestInfo will be provided, and the signing function is expected to do the PKCS `1` 1.5 padding and the exponentiation.

See also `gnutls_privkey_import_ext3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_privkey_import_ext2

`int gnutls_privkey_import_ext2` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `pk`, *void ** `userdata`, *gnutls_privkey_sign_func* `sign_fn`, *gnutls_privkey_decrypt_func* `decrypt_fn`, *gnutls_privkey_deinit_func* `deinit_fn`, *unsigned int* `flags`) [Function]

pkey: The private key

pk: The public key algorithm

userdata: private data to be provided to the callbacks

sign_fn: callback for signature operations

decrypt_fn: callback for decryption operations

deinit_fn: a deinitialization function

flags: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` type. At least one of the two callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` .

Note that the signing function is supposed to "raw" sign data, i.e., without any hashing or preprocessing. In case of RSA the DigestInfo will be provided, and the signing function is expected to do the PKCS `1` 1.5 padding and the exponentiation.

See also `gnutls_privkey_import_ext3()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1

gnutls_privkey_import_ext3

`int gnutls_privkey_import_ext3` (*gnutls_privkey_t* `pkey`, *void ** `userdata`, *gnutls_privkey_sign_func* `sign_fn`, *gnutls_privkey_decrypt_func* `decrypt_fn`, *gnutls_privkey_deinit_func* `deinit_fn`, *gnutls_privkey_info_func* `info_fn`, *unsigned int* `flags`) [Function]

pkey: The private key

userdata: private data to be provided to the callbacks

sign_fn: callback for signature operations

decrypt_fn: callback for decryption operations

deinit_fn: a deinitialization function

info_fn: returns info about the public key algorithm (should not be `NULL`)

flags: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` type. At least one of the two callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` .

Note that the signing function is supposed to "raw" sign data, i.e., without any hashing or preprocessing. In case of RSA the DigestInfo will be provided, and the signing function is expected to do the PKCS `1` 1.5 padding and the exponentiation.

The `info_fn` must provide information on the algorithms supported by this private key, and should support the flags `GNUTLS_PRIVKEY_INFO_PK_ALGO` and `GNUTLS_PRIVKEY_INFO_SIGN_ALGO` . It must return -1 on unknown flags.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_privkey_import_ext4

`int gnutls_privkey_import_ext4` (*gnutls_privkey_t* `pkey`, *void ** `userdata`, *gnutls_privkey_sign_data_func* `sign_data_fn`, *gnutls_privkey_sign_hash_func* `sign_hash_fn`, *gnutls_privkey_decrypt_func* `decrypt_fn`, *gnutls_privkey_deinit_func* `deinit_fn`, *gnutls_privkey_info_func* `info_fn`, *unsigned int* `flags`) [Function]

pkey: The private key

userdata: private data to be provided to the callbacks

sign_data_fn: callback for signature operations (may be `NULL`)

sign_hash_fn: callback for signature operations (may be `NULL`)

decrypt_fn: callback for decryption operations (may be `NULL`)

deinit_fn: a deinitialization function

info_fn: returns info about the public key algorithm (should not be `NULL`)

flags: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` type. At least one of the callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` .

Note that in contrast with the signing function of `gnutls_privkey_import_ext3()` , the signing functions provided to this function take explicitly the signature algorithm as parameter and different functions are provided to sign the data and hashes.

The `sign_hash_fn` is to be called to sign pre-hashed data. The input to the callback is the output of the hash (such as SHA256) corresponding to the signature algorithm. For RSA PKCS1 signatures, the signature algorithm can be set to `GNUTLS_SIGN_RSA_RAW` , and in that case the data should be handled as if they were an RSA PKCS1 DigestInfo structure.

The `sign_data_fn` is to be called to sign data. The input data will be he data to be signed (and hashed), with the provided signature algorithm. This function is to be used for signature algorithms like Ed25519 which cannot take pre-hashed data as input.

When both `sign_data_fn` and `sign_hash_fn` functions are provided they must be able to operate on all the supported signature algorithms, unless prohibited by the type of the algorithm (e.g., as with Ed25519).

The `info_fn` must provide information on the signature algorithms supported by this private key, and should support the flags `GNUTLS_PRIVKEY_INFO_PK_ALGO` , `GNUTLS_PRIVKEY_INFO_HAVE_SIGN_ALGO` and `GNUTLS_PRIVKEY_INFO_PK_ALGO_BITS` . It must return -1 on unknown flags.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_privkey_import_gost_raw

`int gnutls_privkey_import_gost_raw` (*gnutls_privkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *gnutls_digest_algorithm_t* `digest`, *gnutls_gost_paramset_t* `paramset`, *const gnutls_datum_t * x*, *const gnutls_datum_t * y*, *const gnutls_datum_t * k*) [Function]

key: The key

curve: holds the curve

digest: holds the digest

paramset: holds the GOST parameter set ID

x: holds the x-coordinate

y: holds the y-coordinate

k: holds the k (private key)

This function will convert the given GOST private key's parameters to the native `gnutls_privkey_t` format. The output will be stored in `key` . `digest` should be one of GNUTLS_DIG_GOSR_94, GNUTLS_DIG_STREEBOG_256 or GNUTLS_DIG_STREEBOG_512. If `paramset` is set to GNUTLS_GOST_PARAMSET_UNKNOWN default one will be selected depending on `digest` .

Note: parameters should be stored with least significant byte first. On version 3.6.3 big-endian format was used incorrectly.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_privkey_import_openpgp

`int gnutls_privkey_import_openpgp` (*gnutls_privkey_t* `pkey`, *gnutls_openpgp_privkey_t* `key`, *unsigned int* `flags`) [Function]

pkey: The private key

key: The private key to be imported

flags: Flags for the import

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 2.12.0

gnutls_privkey_import_openpgp_raw

`int gnutls_privkey_import_openpgp_raw` (*gnutls_privkey_t* `pkey`, *const gnutls_datum_t ** `data`, *gnutls_openpgp_crt_fmt_t* `format`, *const gnutls_openpgp_keyid_t* `keyid`, *const char ** `password`) [Function]

pkey: The private key

data: The private key data to be imported

format: The format of the private key

keyid: The key id to use (optional)

password: A password (optional)

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 3.1.0

gnutls_privkey_import_pkcs11

`int gnutls_privkey_import_pkcs11` (*gnutls_privkey_t* `pkey`, *gnutls_pkcs11_privkey_t* `key`, *unsigned int* `flags`) [Function]

pkey: The private key

key: The private key to be imported

flags: Flags for the import

This function will import the given private key to the abstract `gnutls_privkey_t` type.

The `gnutls_pkcs11_privkey_t` object must not be deallocated during the lifetime of this structure.

`flags` might be zero or one of `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` and `GNUTLS_PRIVKEY_IMPORT_COPY` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_import_pkcs11_url

`int gnutls_privkey_import_pkcs11_url` (*gnutls_privkey_t* `key`, *const char ** `url`) [Function]

key: A key of type `gnutls_pubkey_t`

url: A PKCS 11 url

This function will import a PKCS 11 private key to a `gnutls_private_key_t` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_privkey_import_rsa_raw

`int gnutls_privkey_import_rsa_raw` (*gnutls_privkey_t* `key`, *const gnutls_datum_t * m, const gnutls_datum_t * e, const gnutls_datum_t * d, const gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t * u, const gnutls_datum_t * e1, const gnutls_datum_t * e2*) [Function]

key: The structure to store the parsed key

m: holds the modulus

e: holds the public exponent

d: holds the private exponent

p: holds the first prime (p)

q: holds the second prime (q)

u: holds the coefficient (optional)

e1: holds e1 = d mod (p-1) (optional)

e2: holds e2 = d mod (q-1) (optional)

This function will convert the given RSA raw parameters to the native `gnutls_privkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_privkey_import_tpm_raw

`int gnutls_privkey_import_tpm_raw` (*gnutls_privkey_t* `pkey`, *const gnutls_datum_t * fdata, gnutls_tpmkey_fmt_t format, const char * srk_password, const char * key_password, unsigned int flags*) [Function]

pkey: The private key

fdata: The TPM key to be imported

format: The format of the private key

srk_password: The password for the SRK key (optional)

key_password: A password for the key (optional)

flags: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` type.

With respect to passwords the same as in `gnutls_privkey_import_tpm_url()` apply.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_privkey_import_tpm_url

`int gnutls_privkey_import_tpm_url (`*gnutls_privkey_t* `pkey`, *const char ** `url`, *const char ** `srk_password`, *const char ** `key_password`, *unsigned int* `flags`) [Function]

pkey: The private key

url: The URL of the TPM key to be imported

srk_password: The password for the SRK key (optional)

key_password: A password for the key (optional)

flags: One of the GNUTLS_PRIVKEY_* flags

This function will import the given private key to the abstract `gnutls_privkey_t` type.

Note that unless `GNUTLS_PRIVKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned and if the key password is wrong or not provided then `GNUTLS_E_TPM_KEY_PASSWORD_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_privkey_import_url

`int gnutls_privkey_import_url (`*gnutls_privkey_t* `key`, *const char ** `url`, *unsigned int* `flags`) [Function]

key: A key of type `gnutls_privkey_t`

url: A PKCS 11 url

flags: should be zero

This function will import a PKCS11 or TPM URL as a private key. The supported URL types can be checked using `gnutls_url_is_supported()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_privkey_import_x509

`int gnutls_privkey_import_x509 (`*gnutls_privkey_t* `pkey`, *gnutls_x509_privkey_t* `key`, *unsigned int* `flags`) [Function]

pkey: The private key

key: The private key to be imported

flags: Flags for the import

This function will import the given private key to the abstract `gnutls_privkey_t` type.

The gnutls_x509_privkey_t object must not be deallocated during the lifetime of this structure.

flags might be zero or one of GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE and GNUTLS_PRIVKEY_IMPORT_COPY .

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_import_x509_raw

int gnutls_privkey_import_x509_raw (*gnutls_privkey_t* `pkey`, *const gnutls_datum_t* * `data`, *gnutls_x509_crt_fmt_t* `format`, *const char* * `password`, *unsigned int* `flags`) [Function]

pkey: The private key

data: The private key data to be imported

format: The format of the private key

password: A password (optional)

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

This function will import the given private key to the abstract gnutls_privkey_t type.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_privkey_init

int gnutls_privkey_init (*gnutls_privkey_t* * `key`) [Function]

key: A pointer to the type to be initialized

This function will initialize a private key object. The object can be used to generate, import, and perform cryptographic operations on the associated private key.

Note that when the underlying private key is a PKCS11 key (i.e., when imported with a PKCS11 URI), the limitations of gnutls_pkcs11_privkey_init() apply to this object as well. In versions of GnuTLS later than 3.5.11 the object is protected using locks and a single gnutls_privkey_t can be re-used by many threads. However, for performance it is recommended to utilize one object per key per thread.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_set_flags

`void gnutls_privkey_set_flags (`*gnutls_privkey_t* `key`*, unsigned int* `flags``)` [Function]

key: A key of type `gnutls_privkey_t`

flags: flags from the `gnutls_privkey_flags`

This function will set flags for the specified private key, after it is generated. Currently this is useful for the `GNUTLS_PRIVKEY_FLAG_EXPORT_COMPAT` to allow exporting a "provable" private key in backwards compatible way.

Since: 3.5.0

gnutls_privkey_set_pin_function

`void gnutls_privkey_set_pin_function (`*gnutls_privkey_t* `key`*, gnutls_pin_callback_t* `fn`*, void * userdata*`)` [Function]

key: A key of type `gnutls_privkey_t`

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

Since: 3.1.0

gnutls_privkey_set_spki

`int gnutls_privkey_set_spki (`*gnutls_privkey_t* `privkey`*, const gnutls_x509_spki_t* `spki`*, unsigned int* `flags``)` [Function]

privkey: a public key of type `gnutls_privkey_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_privkey_spki_t`

flags: must be zero

This function will set the public key information. The provided `spki` must be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_privkey_sign_data

`int gnutls_privkey_sign_data (`*gnutls_privkey_t* `signer`*, gnutls_digest_algorithm_t* `hash`*, unsigned int* `flags`*, const gnutls_datum_t * data, gnutls_datum_t * signature*`)` [Function]

signer: Holds the key

hash: should be a digest algorithm

flags: Zero or one of `gnutls_privkey_flags_t`

data: holds the data to be signed

signature: will contain the signature allocated with `gnutls_malloc()`

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only the SHA family for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_sign_data2

`int gnutls_privkey_sign_data2` (*gnutls_privkey_t* `signer`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t* * `data`, *gnutls_datum_t* * `signature`) [Function]

signer: Holds the key

algo: The signature algorithm used

flags: Zero or one of `gnutls_privkey_flags_t`

data: holds the data to be signed

signature: will contain the signature allocated with `gnutls_malloc()`

This function will sign the given data using the specified signature algorithm. This function is an enhancement of `gnutls_privkey_sign_data()` , as it allows utilizing a alternative signature algorithm where possible (e.g, use an RSA key with RSA-PSS).

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_privkey_sign_hash

`int gnutls_privkey_sign_hash` (*gnutls_privkey_t* `signer`, *gnutls_digest_algorithm_t* `hash_algo`, *unsigned int* `flags`, *const gnutls_datum_t* * `hash_data`, *gnutls_datum_t* * `signature`) [Function]

signer: Holds the signer's key

hash_algo: The hash algorithm used

flags: Zero or one of `gnutls_privkey_flags_t`

hash_data: holds the data to be signed

signature: will contain newly allocated signature

This function will sign the given hashed data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-XXX for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

The flags may be GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA or GNUTLS_PRIVKEY_SIGN_FLAG_RSA_PSS . In the former case this function will ignore hash_algo and perform a raw PKCS1 signature, and in the latter an RSA-PSS signature will be generated.

Note that, not all algorithm support signing already hashed data. When signing with Ed25519, gnutls_privkey_sign_data() should be used.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_privkey_sign_hash2

int gnutls_privkey_sign_hash2 (*gnutls_privkey_t* ***signer***, *gnutls_sign_algorithm_t* **algo**, *unsigned int* **flags**, *const gnutls_datum_t * **hash_data***, *gnutls_datum_t * **signature***) [Function]

signer: Holds the signer's key

algo: The signature algorithm used

flags: Zero or one of gnutls_privkey_flags_t

hash_data: holds the data to be signed

signature: will contain newly allocated signature

This function will sign the given hashed data using the specified signature algorithm. This function is an enhancement of gnutls_privkey_sign_hash() , as it allows utilizing a alternative signature algorithm where possible (e.g, use an RSA key with RSA-PSS).

The flags may be GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA . In that case this function will ignore hash_algo and perform a raw PKCS1 signature. Note that this flag is supported since 3.6.9.

Note also that, not all algorithm support signing already hashed data. When signing with Ed25519, gnutls_privkey_sign_data2() should be used instead.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_privkey_status

int gnutls_privkey_status (*gnutls_privkey_t* ***key***) [Function]

key: Holds the key

Checks the status of the private key token. This function is an actual wrapper over gnutls_pkcs11_privkey_status() , and if the private key is a PKCS 11 token it will check whether it is inserted or not.

Returns: this function will return non-zero if the token holding the private key is still available (inserted), and zero otherwise.

Since: 3.1.10

gnutls_privkey_verify_params

`int gnutls_privkey_verify_params (`*gnutls_privkey_t* `key`) [Function]

key: should contain a `gnutls_privkey_t` type

This function will verify the private key parameters.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_privkey_verify_seed

`int gnutls_privkey_verify_seed (`*gnutls_privkey_t* `key`, *gnutls_digest_algorithm_t* `digest`, *const void ** `seed`, *size_t* `seed_size`) [Function]

key: should contain a `gnutls_privkey_t` type

digest: it contains the digest algorithm used for key generation (if applicable)

seed: the seed of the key to be checked with

seed_size: holds the size of `seed`

This function will verify that the given private key was generated from the provided seed.

Returns: In case of a verification failure `GNUTLS_E_PRIVKEY_VERIFICATION_ERROR` is returned, and zero or positive code on success.

Since: 3.5.0

gnutls_pubkey_deinit

`void gnutls_pubkey_deinit (`*gnutls_pubkey_t* `key`) [Function]

key: The key to be deinitialized

This function will deinitialize a public key structure.

Since: 2.12.0

gnutls_pubkey_encrypt_data

`int gnutls_pubkey_encrypt_data (`*gnutls_pubkey_t* `key`, *unsigned int* `flags`, *const gnutls_datum_t ** `plaintext`, *gnutls_datum_t ** `ciphertext`) [Function]

key: Holds the public key

flags: should be 0 for now

plaintext: The data to be encrypted

ciphertext: contains the encrypted data

This function will encrypt the given data, using the public key. On success the `ciphertext` will be allocated using `gnutls_malloc()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pubkey_export

`int gnutls_pubkey_export` (*gnutls_pubkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *void *** `output_data`, *size_t ** `output_data_size`) [Function]

key: Holds the certificate

format: the format of output params. One of PEM or DER.

output_data: will contain a certificate PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 2.12.0

gnutls_pubkey_export2

`int gnutls_pubkey_export2` (*gnutls_pubkey_t* `key`, *gnutls_x509_crt_fmt_t* `format`, *gnutls_datum_t ** `out`) [Function]

key: Holds the certificate

format: the format of output params. One of PEM or DER.

out: will contain a certificate PEM or DER encoded

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

The output buffer will be allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 3.1.3

gnutls_pubkey_export_dsa_raw

`int gnutls_pubkey_export_dsa_raw` (*gnutls_pubkey_t* `key`, *gnutls_datum_t ** `p`, *gnutls_datum_t ** `q`, *gnutls_datum_t ** `g`, *gnutls_datum_t ** `y`) [Function]

key: Holds the public key

p: will hold the p (may be `NULL`)

q: will hold the q (may be `NULL`)

g: will hold the g (may be `NULL`)

y: will hold the y (may be `NULL`)

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_pubkey_export_dsa_raw2

`int gnutls_pubkey_export_dsa_raw2` (*gnutls_pubkey_t* `key`, *gnutls_datum_t* * `p`, *gnutls_datum_t* * `q`, *gnutls_datum_t* * `g`, *gnutls_datum_t* * `y`, *unsigned* `flags`) [Function]

key: Holds the public key

p: will hold the p (may be `NULL`)

q: will hold the q (may be `NULL`)

g: will hold the g (may be `NULL`)

y: will hold the y (may be `NULL`)

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_pubkey_export_ecc_raw

`int gnutls_pubkey_export_ecc_raw` (*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`) [Function]

key: Holds the public key

curve: will hold the curve (may be `NULL`)

x: will hold x-coordinate (may be `NULL`)

y: will hold y-coordinate (may be `NULL`)

This function will export the ECC public key's parameters found in the given key. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.0

gnutls_pubkey_export_ecc_raw2

`int gnutls_pubkey_export_ecc_raw2 (`*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *unsigned int* `flags``)` [Function]

key: Holds the public key

curve: will hold the curve (may be `NULL`)

x: will hold x-coordinate (may be `NULL`)

y: will hold y-coordinate (may be `NULL`)

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the ECC public key's parameters found in the given key. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

In EdDSA curves the `y` parameter will be `NULL` and the other parameters will be in the native format for the curve.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_pubkey_export_ecc_x962

`int gnutls_pubkey_export_ecc_x962 (`*gnutls_pubkey_t* `key`, *gnutls_datum_t* * `parameters`, *gnutls_datum_t* * `ecpoint``)` [Function]

key: Holds the public key

parameters: DER encoding of an ANSI X9.62 parameters

ecpoint: DER encoding of ANSI X9.62 ECPoint

This function will export the ECC public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_pubkey_export_gost_raw2

`int gnutls_pubkey_export_gost_raw2 (`*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t* * `curve`, *gnutls_digest_algorithm_t* * `digest`, *gnutls_gost_paramset_t* * `paramset`, *gnutls_datum_t* * `x`, *gnutls_datum_t* * `y`, *unsigned int* `flags``)` [Function]

key: Holds the public key

curve: will hold the curve (may be `NULL`)

digest: will hold the curve (may be `NULL`)

paramset: will hold the parameters id (may be `NULL`)

x: will hold the x-coordinate (may be `NULL`)

y: will hold the y-coordinate (may be `NULL`)

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the GOST public key's parameters found in the given key. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

Note: parameters will be stored with least significant byte first. On version 3.6.3 this was incorrectly returned in big-endian format.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.3

gnutls_pubkey_export_rsa_raw

`int gnutls_pubkey_export_rsa_raw (`*gnutls_pubkey_t* `key`, *gnutls_datum_t* * `m`, *gnutls_datum_t* * `e``)` [Function]

key: Holds the certificate

m: will hold the modulus (may be `NULL`)

e: will hold the public exponent (may be `NULL`)

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.3.0

gnutls_pubkey_export_rsa_raw2

`int gnutls_pubkey_export_rsa_raw2 (`*gnutls_pubkey_t* `key`, *gnutls_datum_t* * `m`, *gnutls_datum_t* * `e`, *unsigned* `flags`)` [Function]

key: Holds the certificate

m: will hold the modulus (may be `NULL`)

e: will hold the public exponent (may be `NULL`)

flags: flags from `gnutls_abstract_export_flags_t`

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

This function allows for `NULL` parameters since 3.4.1.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.6.0

gnutls_pubkey_get_key_id

`int gnutls_pubkey_get_key_id (`*gnutls_pubkey_t* `key`, *unsigned int* `flags`, *unsigned char* * `output_data`, *size_t* * `output_data_size`)` [Function]

key: Holds the public key

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given public key.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

Returns: In case of failure a negative error code will be returned, and 0 on success.

Since: 2.12.0

gnutls_pubkey_get_key_usage

`int gnutls_pubkey_get_key_usage` (*gnutls_pubkey_t* `key`, *unsigned int * * `usage`*) [Function]

key: should contain a `gnutls_pubkey_t` type

usage: If set will return the number of bits of the parameters (may be NULL)

This function will return the key usage of the public key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_get_openpgp_key_id

`int gnutls_pubkey_get_openpgp_key_id` (*gnutls_pubkey_t* `key`, *unsigned int* `flags`, *unsigned char * * `output_data`*, *size_t * `output_data_size`*, *unsigned int * * `subkey`*) [Function]

key: Holds the public key

flags: should be one of the flags from `gnutls_keyid_flags_t`

output_data: will contain the key ID

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

subkey: ignored

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

Since: 2.12.0

gnutls_pubkey_get_pk_algorithm

`int gnutls_pubkey_get_pk_algorithm` (*gnutls_pubkey_t* `key`, *unsigned int * * `bits`*) [Function]

key: should contain a `gnutls_pubkey_t` type

bits: If set will return the number of bits of the parameters (may be NULL)

This function will return the public key algorithm of a public key and if possible will return a number of bits that indicates the security parameter of the key.

Returns: a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

Since: 2.12.0

gnutls_pubkey_get_preferred_hash_algorithm

`int gnutls_pubkey_get_preferred_hash_algorithm` [Function]
(*gnutls_pubkey_t* `key`, *gnutls_digest_algorithm_t* * `hash`, *unsigned int* * `mand`)

key: Holds the certificate

hash: The result of the call with the hash algorithm used for signature

mand: If non zero it means that the algorithm MUST use this hash. May be NULL.

This function will read the certificate and return the appropriate digest algorithm to use for signing with this certificate. Some certificates (i.e. DSA might not be able to sign without the preferred algorithm).

To get the signature algorithm instead of just the hash use `gnutls_pk_to_sign()` with the algorithm of the certificate/key and the provided `hash` .

Returns: the 0 if the hash algorithm is found. A negative error code is returned on error.

Since: 2.12.0

gnutls_pubkey_get_spki

`int gnutls_pubkey_get_spki` (*gnutls_pubkey_t* `pubkey`, [Function]
gnutls_x509_spki_t `spki`, *unsigned int* `flags`)

pubkey: a public key of type `gnutls_pubkey_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_pubkey_spki_t`

flags: must be zero

This function will return the public key information if available. The provided `spki` must be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_pubkey_import

`int gnutls_pubkey_import` (*gnutls_pubkey_t* `key`, *const* [Function]
gnutls_datum_t * `data`, *gnutls_x509_crt_fmt_t* `format`)

key: The public key.

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will import the provided public key in a SubjectPublicKeyInfo X.509 structure to a native `gnutls_pubkey_t` type. The output will be stored in `key` . If the public key is PEM encoded it should have a header of "PUBLIC KEY".

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_import_dsa_raw

`int gnutls_pubkey_import_dsa_raw` (*gnutls_pubkey_t* `key`, *const gnutls_datum_t* * `p`, *const gnutls_datum_t* * `q`, *const gnutls_datum_t* * `g`, *const gnutls_datum_t* * `y`) [Function]

key: The structure to store the parsed key

p: holds the p

q: holds the q

g: holds the g

y: holds the y

This function will convert the given DSA raw parameters to the native `gnutls_pubkey_t` format. The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_import_ecc_raw

`int gnutls_pubkey_import_ecc_raw` (*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *const gnutls_datum_t* * `x`, *const gnutls_datum_t* * `y`) [Function]

key: The structure to store the parsed key

curve: holds the curve

x: holds the x-coordinate

y: holds the y-coordinate

This function will convert the given elliptic curve parameters to a `gnutls_pubkey_t` . The output will be stored in `key` .

In EdDSA curves the `y` parameter should be `NULL` and the `x` parameter must be the value in the native format for the curve.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pubkey_import_ecc_x962

`int gnutls_pubkey_import_ecc_x962` (*gnutls_pubkey_t* `key`, *const gnutls_datum_t* * `parameters`, *const gnutls_datum_t* * `ecpoint`) [Function]

key: The structure to store the parsed key

parameters: DER encoding of an ANSI X9.62 parameters

ecpoint: DER encoding of ANSI X9.62 ECPoint

This function will convert the given elliptic curve parameters to a `gnutls_pubkey_t` . The output will be stored in `key` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.0

gnutls_pubkey_import_gost_raw

`int gnutls_pubkey_import_gost_raw` (*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t* `curve`, *gnutls_digest_algorithm_t* `digest`, *gnutls_gost_paramset_t* `paramset`, *const gnutls_datum_t* * `x`, *const gnutls_datum_t* * `y`) [Function]

key: The structure to store the parsed key

curve: holds the curve

digest: holds the digest

paramset: holds the parameters id

x: holds the x-coordinate

y: holds the y-coordinate

This function will convert the given GOST public key's parameters to a `gnutls_pubkey_t` . The output will be stored in `key` . `digest` should be one of GNUTLS_DIG_GOSR_94, GNUTLS_DIG_STREEBOG_256 or GNUTLS_DIG_STREEBOG_512. If `paramset` is set to GNUTLS_GOST_PARAMSET_UNKNOWN default one will be selected depending on `digest` .

Note: parameters should be stored with least significant byte first. On version 3.6.3 big-endian format was used incorrectly.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.3

gnutls_pubkey_import_openpgp

`int gnutls_pubkey_import_openpgp` (*gnutls_pubkey_t* `key`, *gnutls_openpgp_crt_t* `crt`, *unsigned int* `flags`) [Function]

key: The public key

crt: The certificate to be imported

flags: should be zero

This function is no-op.

Returns: GNUTLS_E_UNIMPLEMENTED_FEATURE .

Since: 2.12.0

gnutls_pubkey_import_openpgp_raw

`int gnutls_pubkey_import_openpgp_raw` (*gnutls_pubkey_t* `pkey`, *const gnutls_datum_t ** `data`, *gnutls_openpgp_crt_fmt_t* `format`, *const gnutls_openpgp_keyid_t* `keyid`, *unsigned int* `flags`) [Function]

pkey: The public key

data: The public key data to be imported

format: The format of the public key

keyid: The key id to use (optional)

flags: Should be zero

This function is no-op.

Returns: GNUTLS_E_UNIMPLEMENTED_FEATURE .

Since: 3.1.3

gnutls_pubkey_import_pkcs11

`int gnutls_pubkey_import_pkcs11` (*gnutls_pubkey_t* `key`, *gnutls_pkcs11_obj_t* `obj`, *unsigned int* `flags`) [Function]

key: The public key

obj: The parameters to be imported

flags: should be zero

Imports a public key from a pkcs11 key. This function will import the given public key to the abstract `gnutls_pubkey_t` type.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_import_privkey

`int gnutls_pubkey_import_privkey` (*gnutls_pubkey_t* `key`, *gnutls_privkey_t* `pkey`, *unsigned int* `usage`, *unsigned int* `flags`) [Function]

key: The public key

pkey: The private key

usage: GNUTLS_KEY_* key usage flags.

flags: should be zero

Imports the public key from a private. This function will import the given public key to the abstract `gnutls_pubkey_t` type.

Note that in certain keys this operation may not be possible, e.g., in other than RSA PKCS11 keys.

Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_import_rsa_raw

`int gnutls_pubkey_import_rsa_raw` (*gnutls_pubkey_t* `key`, *const gnutls_datum_t * m*, *const gnutls_datum_t * e*) [Function]

key: The key

m: holds the modulus

e: holds the public exponent

This function will replace the parameters in the given structure. The new parameters should be stored in the appropriate gnutls_datum.

Returns: `GNUTLS_E_SUCCESS` on success, or an negative error code.

Since: 2.12.0

gnutls_pubkey_import_tpm_raw

`int gnutls_pubkey_import_tpm_raw` (*gnutls_pubkey_t* `pkey`, *const gnutls_datum_t * fdata*, *gnutls_tpmkey_fmt_t* `format`, *const char * srk_password*, *unsigned int* `flags`) [Function]

pkey: The public key

fdata: The TPM key to be imported

format: The format of the private key

srk_password: The password for the SRK key (optional)

flags: One of the GNUTLS_PUBKEY_* flags

This function will import the public key from the provided TPM key structure.

With respect to passwords the same as in `gnutls_pubkey_import_tpm_url()` apply.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_pubkey_import_tpm_url

`int gnutls_pubkey_import_tpm_url` (*gnutls_pubkey_t* `pkey`, *const char * url*, *const char * srk_password*, *unsigned int* `flags`) [Function]

pkey: The public key

url: The URL of the TPM key to be imported

srk_password: The password for the SRK key (optional)

flags: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` type.

Note that unless `GNUTLS_PUBKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_pubkey_import_url

`int gnutls_pubkey_import_url` (*gnutls_pubkey_t* `key`, *const char ** `url`, *unsigned int* `flags`) [Function]

key: A key of type `gnutls_pubkey_t`

url: A PKCS 11 url

flags: One of GNUTLS_PKCS11_OBJ_* flags

This function will import a public key from the provided URL.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.0

gnutls_pubkey_import_x509

`int gnutls_pubkey_import_x509` (*gnutls_pubkey_t* `key`, *gnutls_x509_crt_t* `crt`, *unsigned int* `flags`) [Function]

key: The public key

crt: The certificate to be imported

flags: should be zero

This function will import the given public key to the abstract `gnutls_pubkey_t` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_import_x509_crq

`int gnutls_pubkey_import_x509_crq` (*gnutls_pubkey_t* `key`, *gnutls_x509_crq_t* `crq`, *unsigned int* `flags`) [Function]

key: The public key

crq: The certificate to be imported

flags: should be zero

This function will import the given public key to the abstract `gnutls_pubkey_t` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.5

gnutls_pubkey_import_x509_raw

`int gnutls_pubkey_import_x509_raw` (*gnutls_pubkey_t* `pkey`, *const gnutls_datum_t ** `data`, *gnutls_x509_crt_fmt_t* `format`, *unsigned int* `flags`) [Function]

pkey: The public key

data: The public key data to be imported

format: The format of the public key

flags: should be zero

This function will import the given public key to the abstract `gnutls_pubkey_t` type.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.3

gnutls_pubkey_init

`int gnutls_pubkey_init` (*gnutls_pubkey_t * key*) [Function]

key: A pointer to the type to be initialized

This function will initialize a public key.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_print

`int gnutls_pubkey_print` (*gnutls_pubkey_t pubkey*, *gnutls_certificate_print_formats_t format*, *gnutls_datum_t * out*) [Function]

pubkey: The data to be printed

format: Indicate the format to use

out: Newly allocated datum with null terminated string.

This function will pretty print public key information, suitable for display to a human.

Only `GNUTLS_CRT_PRINT_FULL` and `GNUTLS_CRT_PRINT_FULL_NUMBERS` are implemented.

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.1.5

gnutls_pubkey_set_key_usage

`int gnutls_pubkey_set_key_usage` (*gnutls_pubkey_t key*, *unsigned int usage*) [Function]

key: a certificate of type `gnutls_x509_crt_t`

usage: an ORed sequence of the GNUTLS_KEY_* elements.

This function will set the key usage flags of the public key. This is only useful if the key is to be exported to a certificate or certificate request.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_pubkey_set_pin_function

`void gnutls_pubkey_set_pin_function (`*gnutls_pubkey_t* `key`, *gnutls_pin_callback_t* `fn`, *void ** `userdata``)` [Function]

key: A key of type `gnutls_pubkey_t`

fn: the callback

userdata: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

Since: 3.1.0

gnutls_pubkey_set_spki

`int gnutls_pubkey_set_spki (`*gnutls_pubkey_t* `pubkey`, *const gnutls_x509_spki_t* `spki`, *unsigned int* `flags`) [Function]

pubkey: a public key of type `gnutls_pubkey_t`

spki: a SubjectPublicKeyInfo structure of type `gnutls_pubkey_spki_t`

flags: must be zero

This function will set the public key information. The provided `spki` must be initialized.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.6.0

gnutls_pubkey_verify_data2

`int gnutls_pubkey_verify_data2 (`*gnutls_pubkey_t* `pubkey`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t ** `data`, *const gnutls_datum_t ** `signature`) [Function]

pubkey: Holds the public key

algo: The signature algorithm used

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

data: holds the signed data

signature: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success. For known to be insecure signatures this function will return `GNUTLS_E_INSUFFICIENT_SECURITY` unless the flag `GNUTLS_VERIFY_ALLOW_BROKEN` is specified.

Since: 3.0

gnutls_pubkey_verify_hash2

`int gnutls_pubkey_verify_hash2 (`*gnutls_pubkey_t* `key`, *gnutls_sign_algorithm_t* `algo`, *unsigned int* `flags`, *const gnutls_datum_t * * `hash`, *const gnutls_datum_t * * `signature`)` [Function]

key: Holds the public key

algo: The signature algorithm used

flags: Zero or an OR list of `gnutls_certificate_verify_flags`

hash: holds the hash digest to be verified

signature: contains the signature

This function will verify the given signed digest, using the parameters from the public key. Note that unlike `gnutls_privkey_sign_hash()` , this function accepts a signature algorithm instead of a digest algorithm. You can use `gnutls_pk_to_sign()` to get the appropriate value.

Returns: In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success. For known to be insecure signatures this function will return `GNUTLS_E_INSUFFICIENT_SECURITY` unless the flag `GNUTLS_VERIFY_ALLOW_BROKEN` is specified.

Since: 3.0

gnutls_pubkey_verify_params

`int gnutls_pubkey_verify_params (`*gnutls_pubkey_t* `key`) [Function]

key: should contain a `gnutls_pubkey_t` type

This function will verify the public key parameters.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.3.0

gnutls_register_custom_url

`int gnutls_register_custom_url (`*const gnutls_custom_url_st * `st`)` [Function]

st: A `gnutls_custom_url_st` structure

Register a custom URL. This will affect the following functions: `gnutls_url_is_supported()` , `gnutls_privkey_import_url()` , gnutls_pubkey_import_url, `gnutls_x509_crt_import_url()` and all functions that depend on them, e.g., `gnutls_certificate_set_x509_key_file2()` .

The provided structure and callback functions must be valid throughout the lifetime of the process. The registration of an existing URL type will fail with `GNUTLS_E_INVALID_REQUEST` . Since GnuTLS 3.5.0 this function can be used to override the builtin URLs.

This function is not thread safe.

Returns: returns zero if the given structure was imported or a negative value otherwise.

Since: 3.4.0

gnutls_system_key_add_x509

`int gnutls_system_key_add_x509 (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_privkey_t* `privkey`, *const char ** `label`, *char *** `cert_url`, *char *** `key_url``)` [Function]

crt: the certificate to be added

privkey: the key to be added

label: the friendly name to describe the key

cert_url: if non-NULL it will contain an allocated value with the certificate URL

key_url: if non-NULL it will contain an allocated value with the key URL

This function will added the given key and certificate pair, to the system list.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_system_key_delete

`int gnutls_system_key_delete (`*const char ** `cert_url`, *const char ** `key_url``)` [Function]

cert_url: the URL of the certificate

key_url: the URL of the key

This function will delete the key and certificate pair.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_system_key_iter_deinit

`void gnutls_system_key_iter_deinit (`*gnutls_system_key_iter_t* `iter``)` [Function]

iter: an iterator of system keys

This function will deinitialize the iterator.

Since: 3.4.0

gnutls_system_key_iter_get_info

`int gnutls_system_key_iter_get_info (`*gnutls_system_key_iter_t* `* iter`, *unsigned* `cert_type`, *char *** `cert_url`, *char *** `key_url`, *char *** `label`, *gnutls_datum_t ** `der`, *unsigned int* `flags``)` [Function]

iter: an iterator of the system keys (must be set to `NULL` initially)

cert_type: A value of gnutls_certificate_type_t which indicates the type of certificate to look for

cert_url: The certificate URL of the pair (may be `NULL`)

key_url: The key URL of the pair (may be `NULL`)

label: The friendly name (if any) of the pair (may be `NULL`)

der: if non-NULL the DER data of the certificate

flags: should be zero

This function will return on each call a certificate and key pair URLs, as well as a label associated with them, and the DER-encoded certificate. When the iteration is complete it will return `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` .

Typically `cert_type` should be `GNUTLS_CRT_X509` .

All values set are allocated and must be cleared using `gnutls_free()` ,

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 3.4.0

gnutls_x509_crl_privkey_sign

`int gnutls_x509_crl_privkey_sign` (*gnutls_x509_crl_t* `crl`, *gnutls_x509_crt_t* `issuer`, *gnutls_privkey_t* `issuer_key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

crl: should contain a gnutls_x509_crl_t type

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use. GNUTLS_DIG_SHA256 is the safe choice unless you know what you're doing.

flags: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed CRL will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 2.12.0

gnutls_x509_crq_privkey_sign

`int gnutls_x509_crq_privkey_sign` (*gnutls_x509_crq_t* `crq`, *gnutls_privkey_t* `key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a private key

dig: The message digest to use, i.e., `GNUTLS_DIG_SHA1`

flags: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in `gnutls_x509_crt_set_key()` since a certificate request is self signed.

This must be the last step in a certificate request generation since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed request will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code. `GNUTLS_E_ASN1_VALUE_NOT_FOUND` is returned if you didn't set all information in the certificate request (e.g., the version using `gnutls_x509_crq_set_version()`).

Since: 2.12.0

gnutls_x509_crq_set_pubkey

`int gnutls_x509_crq_set_pubkey (`*gnutls_x509_crq_t* `crq`, *gnutls_pubkey_t* `key``)` [Function]

crq: should contain a `gnutls_x509_crq_t` type

key: holds a public key

This function will set the public parameters from the given public key to the request. The `key` can be deallocated after that.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

gnutls_x509_crt_privkey_sign

`int gnutls_x509_crt_privkey_sign (`*gnutls_x509_crt_t* `crt`, *gnutls_x509_crt_t* `issuer`, *gnutls_privkey_t* `issuer_key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`)` [Function]

crt: a certificate of type `gnutls_x509_crt_t`

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

dig: The message digest to use, `GNUTLS_DIG_SHA256` is a safe choice

flags: must be 0

This function will sign the certificate with the issuer's private key, and will copy the issuer's information into the certificate.

This must be the last step in a certificate generation since all the previously set parameters are now signed.

A known limitation of this function is, that a newly-signed certificate will not be fully functional (e.g., for signature verification), until it is exported an re-imported.

After GnuTLS 3.6.1 the value of `dig` may be `GNUTLS_DIG_UNKNOWN` , and in that case, a suitable but reasonable for the key algorithm will be selected.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

gnutls_x509_crt_set_pubkey

`int gnutls_x509_crt_set_pubkey` (*gnutls_x509_crt_t* `crt`, *gnutls_pubkey_t* `key`) [Function]

crt: should contain a `gnutls_x509_crt_t` type

key: holds a public key

This function will set the public parameters from the given public key to the certificate. The `key` can be deallocated after that.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since: 2.12.0

E.10 Socket specific API

The prototypes for the following functions lie in `gnutls/socket.h`.

gnutls_transport_set_fastopen

`void gnutls_transport_set_fastopen` (*gnutls_session_t* `session`, *int* `fd`, *struct sockaddr * * `connect_addr`, *socklen_t* `connect_addrlen`, *unsigned int* `flags`) [Function]

session: is a `gnutls_session_t` type.

fd: is the session's socket descriptor

connect_addr: is the address we want to connect to

connect_addrlen: is the length of `connect_addr`

flags: must be zero

Enables TCP Fast Open (TFO) for the specified TLS client session. That means that TCP connection establishment and the transmission of the first TLS client hello packet are combined. The peer's address must be specified in `connect_addr` and `connect_addrlen` , and the socket specified by `fd` should not be connected.

TFO only works for TCP sockets of type AF_INET and AF_INET6. If the OS doesn't support TCP fast open this function will result to gnutls using `connect()` transparently during the first write.

Note: This function overrides all the transport callback functions. If this is undesirable, TCP Fast Open must be implemented on the user callback functions without calling this function. When using this function, transport callbacks must not be set, and `gnutls_transport_set_ptr()` or `gnutls_transport_set_int()` must not be called.

On GNU/Linux TFO has to be enabled at the system layer, that is in /proc/sys/net/ipv4/tcp_fastopen, bit 0 has to be set.

This function has no effect on server sessions.

Since: 3.5.3

E.11 DANE API

The following functions are to be used for DANE certificate verification. Their prototypes lie in `gnutls/dane.h`. Note that you need to link with the `libgnutls-dane` library to use them.

dane_cert_type_name

`const char * dane_cert_type_name (`*dane_cert_type_t* `type``)` [Function]

type: is a DANE match type

Convert a `dane_cert_type_t` value to a string.

Returns: a string that contains the name of the specified type, or `NULL` .

dane_cert_usage_name

`const char * dane_cert_usage_name (`*dane_cert_usage_t* `usage``)` [Function]

usage: is a DANE certificate usage

Convert a `dane_cert_usage_t` value to a string.

Returns: a string that contains the name of the specified type, or `NULL` .

dane_match_type_name

`const char * dane_match_type_name (`*dane_match_type_t* `type``)` [Function]

type: is a DANE match type

Convert a `dane_match_type_t` value to a string.

Returns: a string that contains the name of the specified type, or `NULL` .

dane_query_data

`int dane_query_data (`*dane_query_t* `q`, *unsigned int* `idx`, *unsigned int * * `usage`, *unsigned int * * `type`, *unsigned int * * `match`, *gnutls_datum_t * * `data`)` [Function]

q: The query result structure

idx: The index of the query response.

usage: The certificate usage (see `dane_cert_usage_t`)

type: The certificate type (see `dane_cert_type_t`)

match: The DANE matching type (see `dane_match_type_t`)

data: The DANE data.

This function will provide the DANE data from the query response.

Returns: On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_query_deinit

`void dane_query_deinit (`*dane_query_t* `q`)` [Function]

q: The structure to be deinitialized

This function will deinitialize a DANE query result structure.

dane_query_entries

`unsigned int dane_query_entries` (*dane_query_t* `q`) [Function]

q: The query result structure

This function will return the number of entries in a query.

Returns: The number of entries.

dane_query_status

`dane_query_status_t dane_query_status` (*dane_query_t* `q`) [Function]

q: The query result structure

This function will return the status of the query response. See `dane_query_status_t` for the possible types.

Returns: The status type.

dane_query_tlsa

`int dane_query_tlsa` (*dane_state_t* `s`, *dane_query_t* * `r`, *const char* * `host`, *const char* * `proto`, *unsigned int* `port`) [Function]

s: The DANE state structure

r: A structure to place the result

host: The host name to resolve.

proto: The protocol type (tcp, udp, etc.)

port: The service port number (eg. 443).

This function will query the DNS server for the TLSA (DANE) data for the given host.

Returns: On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_query_to_raw_tlsa

`int dane_query_to_raw_tlsa` (*dane_query_t* `q`, *unsigned int* * `data_entries`, *char* *** `dane_data`, *int* ** `dane_data_len`, *int* * `secure`, *int* * `bogus`) [Function]

q: The query result structure

data_entries: Pointer set to the number of entries in the query

dane_data: Pointer to contain an array of DNS rdata items, terminated with a NULL pointer; caller must guarantee that the referenced data remains valid until `dane_query_deinit()` is called.

dane_data_len: Pointer to contain the length n bytes of the dane_data items

secure: Pointer set true if the result is validated securely, false if validation failed or the domain queried has no security info

bogus: Pointer set true if the result was not secure due to a security failure

This function will provide the DANE data from the query response.

The pointers dane_data and dane_data_len are allocated with `gnutls_malloc()` to contain the data from the query result structure (individual `dane_data` items simply point to the original data and are not allocated separately). The returned `dane_data` are only valid during the lifetime of `q` .

Returns: On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_raw_tlsa

`int dane_raw_tlsa` (*dane_state_t* `s`, *dane_query_t* * `r`, *char* *const * `dane_data`, *const int* * `dane_data_len`, *int* `secure`, *int* `bogus`) [Function]

s: The DANE state structure

r: A structure to place the result

dane_data: array of DNS rdata items, terminated with a NULL pointer; caller must guarantee that the referenced data remains valid until `dane_query_deinit()` is called.

dane_data_len: the length n bytes of the dane_data items

secure: true if the result is validated securely, false if validation failed or the domain queried has no security info

bogus: if the result was not secure (secure = 0) due to a security failure, and the result is due to a security failure, bogus is true.

This function will fill in the TLSA (DANE) structure from the given raw DNS record data. The `dane_data` must be valid during the lifetime of the query.

Returns: On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_state_deinit

`void dane_state_deinit` (*dane_state_t* `s`) [Function]

s: The structure to be deinitialized

This function will deinitialize a DANE query structure.

dane_state_init

`int dane_state_init` (*dane_state_t* * `s`, *unsigned int* `flags`) [Function]

s: The structure to be initialized

flags: flags from the `dane_state_flags` enumeration

This function will initialize the backend resolver. It is intended to be used in scenarios where multiple resolvings occur, to optimize against multiple re-initializations.

Returns: On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_state_set_dlv_file

`int dane_state_set_dlv_file` (*dane_state_t* `s`, *const char* * `file`) [Function]

s: The structure to be deinitialized

file: The file holding the DLV keys.

This function will set a file with trusted keys for DLV (DNSSEC Lookaside Validation).

dane_strerror

`const char * dane_strerror` (*int* `error`) [Function]

error: is a DANE error code, a negative error code

This function is similar to strerror. The difference is that it accepts an error number returned by a gnutls function; In case of an unknown error a descriptive string is sent instead of `NULL` .

Error codes are always a negative error code.

Returns: A string explaining the DANE error message.

dane_verification_status_print

`int dane_verification_status_print` (*unsigned int* `status`, *gnutls_datum_t* * `out`, *unsigned int* `flags`) [Function]

status: The status flags to be printed

out: Newly allocated datum with (0) terminated string.

flags: should be zero

This function will pretty print the status of a verification process – eg. the one obtained by `dane_verify_crt()` .

The output `out` needs to be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

dane_verify_crt

`int dane_verify_crt` (*dane_state_t* `s`, *const gnutls_datum_t* * `chain`, *unsigned* `chain_size`, *gnutls_certificate_type_t* `chain_type`, *const char* * `hostname`, *const char* * `proto`, *unsigned int* `port`, *unsigned int* `sflags`, *unsigned int* `vflags`, *unsigned int* * `verify`) [Function]

s: A DANE state structure (may be NULL)

chain: A certificate chain

chain_size: The size of the chain

chain_type: The type of the certificate chain

hostname: The hostname associated with the chain

proto: The protocol of the service connecting (e.g. tcp)

port: The port of the service connecting (e.g. 443)

sflags: Flags for the initialization of `s` (if NULL)

vflags: Verification flags; an OR'ed list of `dane_verify_flags_t` .

verify: An OR'ed list of `dane_verify_status_t` .

This function will verify the given certificate chain against the CA constrains and/or the certificate available via DANE. If no information via DANE can be obtained the flag `DANE_VERIFY_NO_DANE_INFO` is set. If a DNSSEC signature is not available for the DANE record then the verify flag `DANE_VERIFY_NO_DNSSEC_DATA` is set.

Due to the many possible options of DANE, there is no single threat model countered. When notifying the user about DANE verification results it may be better to mention: DANE verification did not reject the certificate, rather than mentioning a successful DANE verication.

Note that this function is designed to be run in addition to PKIX - certificate chain - verification. To be run independently the `DANE_VFLAG_ONLY_CHECK_EE_USAGE` flag should be specified; then the function will check whether the key of the peer matches the key advertized in the DANE entry.

Returns: a negative error code on error and `DANE_E_SUCCESS` (0) when the DANE entries were successfully parsed, irrespective of whether they were verified (see `verify` for that information). If no usable entries were encountered `DANE_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

dane_verify_crt_raw

`int dane_verify_crt_raw` (*dane_state_t* `s`, *const gnutls_datum_t ** `chain`, *unsigned* `chain_size`, *gnutls_certificate_type_t* `chain_type`, *dane_query_t* `r`, *unsigned int* `sflags`, *unsigned int* `vflags`, *unsigned int ** `verify`) [Function]

s: A DANE state structure (may be NULL)

chain: A certificate chain

chain_size: The size of the chain

chain_type: The type of the certificate chain

r: DANE data to check against

sflags: Flags for the initialization of `s` (if NULL)

vflags: Verification flags; an OR'ed list of `dane_verify_flags_t` .

verify: An OR'ed list of `dane_verify_status_t` .

This is the low-level function of `dane_verify_crt()` . See the high level function for documentation.

This function does not perform any resolving, it utilizes cached entries from `r` .

Returns: a negative error code on error and `DANE_E_SUCCESS` (0) when the DANE entries were successfully parsed, irrespective of whether they were verified (see `verify` for that information). If no usable entries were encountered `DANE_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

dane_verify_session_crt

`int dane_verify_session_crt` (*dane_state_t* `s`, *gnutls_session_t* `session`, *const char * hostname*, *const char * proto*, *unsigned int port*, *unsigned int sflags*, *unsigned int vflags*, *unsigned int * verify*) [Function]

s: A DANE state structure (may be NULL)

session: A gnutls session

hostname: The hostname associated with the chain

proto: The protocol of the service connecting (e.g. tcp)

port: The port of the service connecting (e.g. 443)

sflags: Flags for the initialization of `s` (if NULL)

vflags: Verification flags; an OR'ed list of `dane_verify_flags_t` .

verify: An OR'ed list of `dane_verify_status_t` .

This function will verify session's certificate chain against the CA constrains and/or the certificate available via DANE. See `dane_verify_crt()` for more information.

This will not verify the chain for validity; unless the DANE verification is restricted to end certificates, this must be be performed separately using `gnutls_certificate_verify_peers3()` .

Returns: a negative error code on error and `DANE_E_SUCCESS` (0) when the DANE entries were successfully parsed, irrespective of whether they were verified (see `verify` for that information). If no usable entries were encountered `DANE_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

E.12 Cryptographic API

The following functions are to be used for low-level cryptographic operations. Their prototypes lie in `gnutls/crypto.h`.

Note that due to historic reasons several functions, (e.g. [gnutls_mac_list], page 424, [gnutls_mac_get_name], page 424) of this API are part of the Section E.1 [Core TLS API], page 360.

gnutls_aead_cipher_decrypt

`int gnutls_aead_cipher_decrypt` (*gnutls_aead_cipher_hd_t* `handle`, *const void * nonce*, *size_t nonce_len*, *const void * auth*, *size_t auth_len*, *size_t tag_size*, *const void * ctext*, *size_t ctext_len*, *void * ptext*, *size_t * ptext_len*) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth: additional data to be authenticated

auth_len: The length of the data

tag_size: The size of the tag to use (use zero for the default)

ctext: the data to decrypt (including the authentication tag)

ctext_len: the length of data to decrypt (includes tag size)

ptext: the decrypted data

ptext_len: the length of decrypted data (initially must hold the maximum available size)

This function will decrypt the given data using the algorithm specified by the context. This function must be provided the complete data to be decrypted, including the authentication tag. On several AEAD ciphers, the authentication tag is appended to the ciphertext, though this is not a general rule. This function will fail if the tag verification fails.

Returns: Zero or a negative error code on verification failure or other error.

Since: 3.4.0

gnutls_aead_cipher_decryptv2

`int gnutls_aead_cipher_decryptv2 (`*gnutls_aead_cipher_hd_t* `handle`, *const void ** `nonce`, *size_t* `nonce_len`, *const giovec_t ** `auth_iov`, *int* `auth_iovcnt`, *const giovec_t ** `iov`, *int* `iovcnt`, *void ** `tag`, *size_t* `tag_size`) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth_iov: additional data to be authenticated

auth_iovcnt: The number of buffers in `auth_iov`

iov: the data to decrypt

iovcnt: The number of buffers in `iov`

tag: The authentication tag

tag_size: The size of the tag to use (use zero for the default)

This is similar to `gnutls_aead_cipher_decrypt()` , but it performs in-place encryption on the provided data buffers.

Returns: Zero or a negative error code on error.

Since: 3.6.10

gnutls_aead_cipher_deinit

`void gnutls_aead_cipher_deinit (`*gnutls_aead_cipher_hd_t* `handle`) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

This function will deinitialize all resources occupied by the given authenticated-encryption context.

Since: 3.4.0

gnutls_aead_cipher_encrypt

`int gnutls_aead_cipher_encrypt` (*gnutls_aead_cipher_hd_t* `handle`, *const void ** `nonce`, *size_t* `nonce_len`, *const void ** `auth`, *size_t* `auth_len`, *size_t* `tag_size`, *const void ** `ptext`, *size_t* `ptext_len`, *void ** `ctext`, *size_t ** `ctext_len`) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth: additional data to be authenticated

auth_len: The length of the data

tag_size: The size of the tag to use (use zero for the default)

ptext: the data to encrypt

ptext_len: The length of data to encrypt

ctext: the encrypted data including authentication tag

ctext_len: the length of encrypted data (initially must hold the maximum available size, including space for tag)

This function will encrypt the given data using the algorithm specified by the context. The output data will contain the authentication tag.

Returns: Zero or a negative error code on error.

Since: 3.4.0

gnutls_aead_cipher_encryptv

`int gnutls_aead_cipher_encryptv` (*gnutls_aead_cipher_hd_t* `handle`, *const void ** `nonce`, *size_t* `nonce_len`, *const giovec_t ** `auth_iov`, *int* `auth_iovcnt`, *size_t* `tag_size`, *const giovec_t ** `iov`, *int* `iovcnt`, *void ** `ctext`, *size_t ** `ctext_len`) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth_iov: additional data to be authenticated

auth_iovcnt: The number of buffers in `auth_iov`

tag_size: The size of the tag to use (use zero for the default)

iov: the data to be encrypted

iovcnt: The number of buffers in `iov`

ctext: the encrypted data including authentication tag

ctext_len: the length of encrypted data (initially must hold the maximum available size, including space for tag)

This function will encrypt the provided data buffers using the algorithm specified by the context. The output data will contain the authentication tag.

Returns: Zero or a negative error code on error.

Since: 3.6.3

gnutls_aead_cipher_encryptv2

`int gnutls_aead_cipher_encryptv2` (*gnutls_aead_cipher_hd_t* `handle`, *const void * nonce*, *size_t* `nonce_len`, *const giovec_t * auth_iov*, *int* `auth_iovcnt`, *const giovec_t * iov*, *int* `iovcnt`, *void * tag*, *size_t * tag_size*) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

nonce: the nonce to set

nonce_len: The length of the nonce

auth_iov: additional data to be authenticated

auth_iovcnt: The number of buffers in `auth_iov`

iov: the data to be encrypted

iovcnt: The number of buffers in `iov`

tag: The authentication tag

tag_size: The size of the tag to use (use zero for the default)

This is similar to `gnutls_aead_cipher_encrypt()` , but it performs in-place encryption on the provided data buffers.

Returns: Zero or a negative error code on error.

Since: 3.6.10

gnutls_aead_cipher_init

`int gnutls_aead_cipher_init` (*gnutls_aead_cipher_hd_t * handle*, *gnutls_cipher_algorithm_t cipher*, *const gnutls_datum_t * key*) [Function]

handle: is a `gnutls_aead_cipher_hd_t` type.

cipher: the authenticated-encryption algorithm to use

key: The key to be used for encryption

This function will initialize an context that can be used for encryption/decryption of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

Returns: Zero or a negative error code on error.

Since: 3.4.0

gnutls_cipher_add_auth

`int gnutls_cipher_add_auth` (*gnutls_cipher_hd_t* `handle`, *const void * ptext*, *size_t ptext_size*) [Function]

handle: is a `gnutls_cipher_hd_t` type

ptext: the data to be authenticated

ptext_size: the length of the data

This function operates on authenticated encryption with associated data (AEAD) ciphers and authenticate the input data. This function can only be called once and before any encryption operations.

Returns: Zero or a negative error code on error.

Since: 3.0

gnutls_cipher_decrypt

`int gnutls_cipher_decrypt (`*gnutls_cipher_hd_t* **`handle`**, *void ** **`ctext`**, *size_t* **`ctext_len`**`)` [Function]

handle: is a `gnutls_cipher_hd_t` type

ctext: the data to decrypt

ctext_len: the length of data to decrypt

This function will decrypt the given data using the algorithm specified by the context.

Note that in AEAD ciphers, this will not check the tag. You will need to compare the tag sent with the value returned from `gnutls_cipher_tag()` .

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_cipher_decrypt2

`int gnutls_cipher_decrypt2 (`*gnutls_cipher_hd_t* **`handle`**, *const void ** **`ctext`**, *size_t* **`ctext_len`**, *void ** **`ptext`**, *size_t* **`ptext_len`**`)` [Function]

handle: is a `gnutls_cipher_hd_t` type

ctext: the data to decrypt

ctext_len: the length of data to decrypt

ptext: the decrypted data

ptext_len: the available length for decrypted data

This function will decrypt the given data using the algorithm specified by the context. For block ciphers the `ctext_len` must be a multiple of the block size. For the supported ciphers the plaintext data length will equal the ciphertext size.

Note that in AEAD ciphers, this will not check the tag. You will need to compare the tag sent with the value returned from `gnutls_cipher_tag()` .

Returns: Zero or a negative error code on error.

Since: 2.12.0

gnutls_cipher_deinit

`void gnutls_cipher_deinit (`*gnutls_cipher_hd_t* **`handle`**`)` [Function]

handle: is a `gnutls_cipher_hd_t` type

This function will deinitialize all resources occupied by the given encryption context.

Since: 2.10.0

gnutls_cipher_encrypt

`int gnutls_cipher_encrypt (`*gnutls_cipher_hd_t* **`handle`**, *void ** **`ptext`**, *size_t* **`ptext_len`**`)` [Function]

handle: is a `gnutls_cipher_hd_t` type

ptext: the data to encrypt

ptext_len: the length of data to encrypt

This function will encrypt the given data using the algorithm specified by the context.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_cipher_encrypt2

`int gnutls_cipher_encrypt2` (*gnutls_cipher_hd_t* `handle`, *const void ** `ptext`, *size_t* `ptext_len`, *void ** `ctext`, *size_t* `ctext_len`) [Function]

handle: is a `gnutls_cipher_hd_t` type

ptext: the data to encrypt

ptext_len: the length of data to encrypt

ctext: the encrypted data

ctext_len: the available length for encrypted data

This function will encrypt the given data using the algorithm specified by the context. For block ciphers the `ptext_len` must be a multiple of the block size. For the supported ciphers the encrypted data length will equal the plaintext size.

Returns: Zero or a negative error code on error.

Since: 2.12.0

gnutls_cipher_get_block_size

`unsigned gnutls_cipher_get_block_size` (*gnutls_cipher_algorithm_t* `algorithm`) [Function]

algorithm: is an encryption algorithm

Returns: the block size of the encryption algorithm.

Since: 2.10.0

gnutls_cipher_get_iv_size

`unsigned gnutls_cipher_get_iv_size` (*gnutls_cipher_algorithm_t* `algorithm`) [Function]

algorithm: is an encryption algorithm

This function returns the size of the initialization vector (IV) for the provided algorithm. For algorithms with variable size IV (e.g., AES-CCM), the returned size will be the one used by TLS.

Returns: block size for encryption algorithm.

Since: 3.2.0

gnutls_cipher_get_tag_size

`unsigned gnutls_cipher_get_tag_size` (*gnutls_cipher_algorithm_t* `algorithm`) [Function]

algorithm: is an encryption algorithm

This function returns the tag size of an authenticated encryption algorithm. For non-AEAD algorithms, it returns zero.

Returns: the tag size of the authenticated encryption algorithm.

Since: 3.2.2

gnutls_cipher_init

`int gnutls_cipher_init` (*gnutls_cipher_hd_t* * `handle`, *gnutls_cipher_algorithm_t* `cipher`, *const gnutls_datum_t* * `key`, *const gnutls_datum_t* * `iv`) [Function]

handle: is a `gnutls_cipher_hd_t` type

cipher: the encryption algorithm to use

key: the key to be used for encryption/decryption

iv: the IV to use (if not applicable set NULL)

This function will initialize the `handle` context to be usable for encryption/decryption of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_cipher_set_iv

`void gnutls_cipher_set_iv` (*gnutls_cipher_hd_t* `handle`, *void* * `iv`, *size_t* `ivlen`) [Function]

handle: is a `gnutls_cipher_hd_t` type

iv: the IV to set

ivlen: the length of the IV

This function will set the IV to be used for the next encryption block.

Since: 3.0

gnutls_cipher_tag

`int gnutls_cipher_tag` (*gnutls_cipher_hd_t* `handle`, *void* * `tag`, *size_t* `tag_size`) [Function]

handle: is a `gnutls_cipher_hd_t` type

tag: will hold the tag

tag_size: the length of the tag to return

This function operates on authenticated encryption with associated data (AEAD) ciphers and will return the output tag.

Returns: Zero or a negative error code on error.

Since: 3.0

gnutls_crypto_register_aead_cipher

`int gnutls_crypto_register_aead_cipher` (*gnutls_cipher_algorithm_t* `algorithm`, *int* `priority`, *gnutls_cipher_init_func* `init`, *gnutls_cipher_setkey_func* `setkey`, *gnutls_cipher_aead_encrypt_func* `aead_encrypt`, *gnutls_cipher_aead_decrypt_func* `aead_decrypt`, *gnutls_cipher_deinit_func* `deinit`) [Function]

algorithm: is the gnutls AEAD cipher identifier

priority: is the priority of the algorithm

init: A function which initializes the cipher

setkey: A function which sets the key of the cipher

aead_encrypt: Perform the AEAD encryption

aead_decrypt: Perform the AEAD decryption

deinit: A function which deinitializes the cipher

This function will register a cipher algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

In the case the registered init or setkey functions return `GNUTLS_E_NEED_FALLBACK` , GnuTLS will attempt to use the next in priority registered cipher.

The functions registered will be used with the new AEAD API introduced in GnuTLS 3.4.0. Internally GnuTLS uses the new AEAD API.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_crypto_register_cipher

`int gnutls_crypto_register_cipher` (*gnutls_cipher_algorithm_t* `algorithm`, *int* `priority`, *gnutls_cipher_init_func* `init`, *gnutls_cipher_setkey_func* `setkey`, *gnutls_cipher_setiv_func* `setiv`, *gnutls_cipher_encrypt_func* `encrypt`, *gnutls_cipher_decrypt_func* `decrypt`, *gnutls_cipher_deinit_func* `deinit`) [Function]

algorithm: is the gnutls algorithm identifier

priority: is the priority of the algorithm

init: A function which initializes the cipher

setkey: A function which sets the key of the cipher

setiv: A function which sets the nonce/IV of the cipher (non-AEAD)

encrypt: A function which performs encryption (non-AEAD)

decrypt: A function which performs decryption (non-AEAD)

deinit: A function which deinitializes the cipher

This function will register a cipher algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

In the case the registered init or setkey functions return `GNUTLS_E_NEED_FALLBACK` , GnuTLS will attempt to use the next in priority registered cipher.

The functions which are marked as non-AEAD they are not required when registering a cipher to be used with the new AEAD API introduced in GnuTLS 3.4.0. Internally GnuTLS uses the new AEAD API.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_crypto_register_digest

`int gnutls_crypto_register_digest` (*gnutls_digest_algorithm_t* `algorithm`, *int* `priority`, *gnutls_digest_init_func* `init`, *gnutls_digest_hash_func* `hash`, *gnutls_digest_output_func* `output`, *gnutls_digest_deinit_func* `deinit`, *gnutls_digest_fast_func* `hash_fast`) [Function]

algorithm: is the gnutls digest identifier

priority: is the priority of the algorithm

init: A function which initializes the digest

hash: Perform the hash operation

output: Provide the output of the digest

deinit: A function which deinitializes the digest

hash_fast: Perform the digest operation in one go

This function will register a digest algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_crypto_register_mac

`int gnutls_crypto_register_mac` (*gnutls_mac_algorithm_t* `algorithm`, *int* `priority`, *gnutls_mac_init_func* `init`, *gnutls_mac_setkey_func* `setkey`, *gnutls_mac_setnonce_func* `setnonce`, *gnutls_mac_hash_func* `hash`, *gnutls_mac_output_func* `output`, *gnutls_mac_deinit_func* `deinit`, *gnutls_mac_fast_func* `hash_fast`) [Function]

algorithm: is the gnutls MAC identifier

priority: is the priority of the algorithm

init: A function which initializes the MAC

setkey: A function which sets the key of the MAC

setnonce: A function which sets the nonce for the mac (may be `NULL` for common MAC algorithms)

hash: Perform the hash operation

output: Provide the output of the MAC

deinit: A function which deinitializes the MAC

hash_fast: Perform the MAC operation in one go

This function will register a MAC algorithm to be used by gnutls. Any algorithm registered will override the included algorithms and by convention kernel implemented algorithms have priority of 90 and CPU-assisted of 80. The algorithm with the lowest priority will be used by gnutls.

Returns: `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

Since: 3.4.0

gnutls_decode_ber_digest_info

`int gnutls_decode_ber_digest_info` (*const gnutls_datum_t * `info`, gnutls_digest_algorithm_t * `hash`, unsigned char * `digest`, unsigned int * `digest_size`*) [Function]

info: an RSA BER encoded DigestInfo structure

hash: will contain the hash algorithm of the structure

digest: will contain the hash output of the structure

digest_size: will contain the hash size of the structure; initially must hold the maximum size of `digest`

This function will parse an RSA PKCS1 1.5 DigestInfo structure and report the hash algorithm used as well as the digest data.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.5.0

gnutls_decode_gost_rs_value

`int gnutls_decode_gost_rs_value` (*const gnutls_datum_t * `sig_value`, gnutls_datum_t * `r`, gnutls_datum_t * `s`*) [Function]

sig_value: will holds a GOST signature according to RFC 4491 section 2.2.2

r: will contain the r value

s: will contain the s value

This function will decode the provided `sig_value` , into `r` and `s` elements. See RFC 4491 section 2.2.2 for the format of signature value.

The output values may be padded with a zero byte to prevent them from being interpreted as negative values. The value should be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_decode_rs_value

`int gnutls_decode_rs_value` (*const gnutls_datum_t * `sig_value`, gnutls_datum_t * `r`, gnutls_datum_t * `s`*) [Function]

sig_value: holds a Dss-Sig-Value DER or BER encoded structure

r: will contain the r value

s: will contain the s value

This function will decode the provided `sig_value` , into `r` and `s` elements. The Dss-Sig-Value is used for DSA and ECDSA signatures.

The output values may be padded with a zero byte to prevent them from being interpreted as negative values. The value should be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_encode_ber_digest_info

`int gnutls_encode_ber_digest_info` (*gnutls_digest_algorithm_t* `hash`, *const gnutls_datum_t ** `digest`, *gnutls_datum_t ** `output`) [Function]

hash: the hash algorithm that was used to get the digest

digest: must contain the digest data

output: will contain the allocated DigestInfo BER encoded data

This function will encode the provided digest data, and its algorithm into an RSA PKCS1 1.5 DigestInfo structure.

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.5.0

gnutls_encode_gost_rs_value

`int gnutls_encode_gost_rs_value` (*gnutls_datum_t ** `sig_value`, *const gnutls_datum_t ** `r`, *const gnutls_datum_t ** `s`) [Function]

sig_value: will hold a GOST signature according to RFC 4491 section 2.2.2

r: must contain the r value

s: must contain the s value

This function will encode the provided r and s values, into binary representation according to RFC 4491 section 2.2.2, used for GOST R 34.10-2001 (and thus also for GOST R 34.10-2012) signatures.

The output value should be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_encode_rs_value

`int gnutls_encode_rs_value` (*gnutls_datum_t ** `sig_value`, *const gnutls_datum_t ** `r`, *const gnutls_datum_t ** `s`) [Function]

sig_value: will hold a Dss-Sig-Value DER encoded structure

r: must contain the r value

s: must contain the s value

This function will encode the provided r and s values, into a Dss-Sig-Value structure, used for DSA and ECDSA signatures.

The output value should be deallocated using `gnutls_free()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since: 3.6.0

gnutls_hash

`int gnutls_hash (`*gnutls_hash_hd_t* **`handle`**, *const void ** **`ptext`**, *size_t* **`ptext_len`**`)` [Function]

handle: is a `gnutls_hash_hd_t` type

ptext: the data to hash

ptext_len: the length of data to hash

This function will hash the given data using the algorithm specified by the context.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hash_copy

`gnutls_hash_hd_t gnutls_hash_copy (`*gnutls_hash_hd_t* **`handle`**`)` [Function]

handle: is a `gnutls_hash_hd_t` type

This function will create a copy of Message Digest context, containing all its current state. Copying contexts for Message Digests registered using `gnutls_crypto_register_digest()` is not supported and will always result in an error.

Returns: new Message Digest context or NULL in case of an error.

Since: 3.6.9

gnutls_hash_deinit

`void gnutls_hash_deinit (`*gnutls_hash_hd_t* **`handle`**, *void ** **`digest`**`)` [Function]

handle: is a `gnutls_hash_hd_t` type

digest: is the output value of the hash

This function will deinitialize all resources occupied by the given hash context.

Since: 2.10.0

gnutls_hash_fast

`int gnutls_hash_fast (`*gnutls_digest_algorithm_t* **`algorithm`**, *const void ** **`ptext`**, *size_t* **`ptext_len`**, *void ** **`digest`**`)` [Function]

algorithm: the hash algorithm to use

ptext: the data to hash

ptext_len: the length of data to hash

digest: is the output value of the hash

This convenience function will hash the given data and return output on a single call.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hash_get_len

`unsigned gnutls_hash_get_len (`*gnutls_digest_algorithm_t* `algorithm`) [Function]

algorithm: the hash algorithm to use

This function will return the length of the output data of the given hash algorithm.

Returns: The length or zero on error.

Since: 2.10.0

gnutls_hash_init

`int gnutls_hash_init (`*gnutls_hash_hd_t* * `dig`, *gnutls_digest_algorithm_t* `algorithm`) [Function]

dig: is a `gnutls_hash_hd_t` type

algorithm: the hash algorithm to use

This function will initialize an context that can be used to produce a Message Digest of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hash_output

`void gnutls_hash_output (`*gnutls_hash_hd_t* `handle`, *void* * `digest`) [Function]

handle: is a `gnutls_hash_hd_t` type

digest: is the output value of the hash

This function will output the current hash value and reset the state of the hash.

Since: 2.10.0

gnutls_hkdf_expand

`int gnutls_hkdf_expand (`*gnutls_mac_algorithm_t* `mac`, *const gnutls_datum_t* * `key`, *const gnutls_datum_t* * `info`, *void* * `output`, *size_t* `length`) [Function]

mac: the mac algorithm used internally

key: the pseudorandom key created with HKDF-Extract

info: the optional informational data

output: the output value of the expand operation

length: the desired length of the output key

This function will derive a variable length keying material from the pseudorandom key using the HKDF-Expand function as defined in RFC 5869.

Returns: Zero or a negative error code on error.

Since: 3.6.13

gnutls_hkdf_extract

`int gnutls_hkdf_extract` (*gnutls_mac_algorithm_t* `mac`, *const gnutls_datum_t * key*, *const gnutls_datum_t * salt*, *void * output*) [Function]

mac: the mac algorithm used internally

key: the initial keying material

salt: the optional salt

output: the output value of the extract operation

This function will derive a fixed-size key using the HKDF-Extract function as defined in RFC 5869.

Returns: Zero or a negative error code on error.

Since: 3.6.13

gnutls_hmac

`int gnutls_hmac` (*gnutls_hmac_hd_t* `handle`, *const void * ptext*, *size_t ptext_len*) [Function]

handle: is a `gnutls_hmac_hd_t` type

ptext: the data to hash

ptext_len: the length of data to hash

This function will hash the given data using the algorithm specified by the context.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hmac_copy

`gnutls_hmac_hd_t gnutls_hmac_copy` (*gnutls_hmac_hd_t* `handle`) [Function]

handle: is a `gnutls_hmac_hd_t` type

This function will create a copy of MAC context, containing all its current state. Copying contexts for MACs registered using `gnutls_crypto_register_mac()` is not supported and will always result in an error.

Returns: new MAC context or NULL in case of an error.

Since: 3.6.9

gnutls_hmac_deinit

`void gnutls_hmac_deinit` (*gnutls_hmac_hd_t* `handle`, *void * digest*) [Function]

handle: is a `gnutls_hmac_hd_t` type

digest: is the output value of the MAC

This function will deinitialize all resources occupied by the given hmac context.

Since: 2.10.0

gnutls_hmac_fast

`int gnutls_hmac_fast (`*gnutls_mac_algorithm_t* `algorithm`, *const void ** `key`, *size_t* `keylen`, *const void ** `ptext`, *size_t* `ptext_len`, *void ** `digest``)` [Function]

algorithm: the hash algorithm to use

key: the key to use

keylen: the length of the key

ptext: the data to hash

ptext_len: the length of data to hash

digest: is the output value of the hash

This convenience function will hash the given data and return output on a single call. Note, this call will not work for MAC algorithms that require nonce (like UMAC or GMAC).

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hmac_get_key_size

`unsigned gnutls_hmac_get_key_size (`*gnutls_mac_algorithm_t* `algorithm``)` [Function]

algorithm: the mac algorithm to use

This function will return the size of the key to be used with this algorithm. On the algorithms which may accept arbitrary key sizes, the returned size is the MAC key size used in the TLS protocol.

Returns: The key size or zero on error.

Since: 3.6.12

gnutls_hmac_get_len

`unsigned gnutls_hmac_get_len (`*gnutls_mac_algorithm_t* `algorithm``)` [Function]

algorithm: the hmac algorithm to use

This function will return the length of the output data of the given hmac algorithm.

Returns: The length or zero on error.

Since: 2.10.0

gnutls_hmac_init

`int gnutls_hmac_init (`*gnutls_hmac_hd_t ** `dig`, *gnutls_mac_algorithm_t* `algorithm`, *const void ** `key`, *size_t* `keylen``)` [Function]

dig: is a `gnutls_hmac_hd_t` type

algorithm: the HMAC algorithm to use

key: the key to be used for encryption

keylen: the length of the key

This function will initialize an context that can be used to produce a Message Authentication Code (MAC) of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

Note that despite the name of this function, it can be used for other MAC algorithms than HMAC.

Returns: Zero or a negative error code on error.

Since: 2.10.0

gnutls_hmac_output

`void gnutls_hmac_output (`*gnutls_hmac_hd_t* `handle`, *void ** `digest``)` [Function]

handle: is a `gnutls_hmac_hd_t` type

digest: is the output value of the MAC

This function will output the current MAC value and reset the state of the MAC.

Since: 2.10.0

gnutls_hmac_set_nonce

`void gnutls_hmac_set_nonce (`*gnutls_hmac_hd_t* `handle`, *const void ** `nonce`, *size_t* `nonce_len``)` [Function]

handle: is a `gnutls_hmac_hd_t` type

nonce: the data to set as nonce

nonce_len: the length of data

This function will set the nonce in the MAC algorithm.

Since: 3.2.0

gnutls_mac_get_nonce_size

`size_t gnutls_mac_get_nonce_size (`*gnutls_mac_algorithm_t* `algorithm`)` [Function]

algorithm: is an encryption algorithm

Returns the size of the nonce used by the MAC in TLS.

Returns: length (in bytes) of the given MAC nonce size, or 0.

Since: 3.2.0

gnutls_pbkdf2

`int gnutls_pbkdf2 (`*gnutls_mac_algorithm_t* `mac`, *const gnutls_datum_t ** `key`, *const gnutls_datum_t ** `salt`, *unsigned* `iter_count`, *void ** `output`, *size_t* `length`)` [Function]

mac: the mac algorithm used internally

key: the initial keying material

salt: the salt

iter_count: the iteration count

output: the output value

length: the desired length of the output key

This function will derive a variable length keying material from a password according to PKCS 5 PBKDF2.

Returns: Zero or a negative error code on error.

Since: 3.6.13

gnutls_rnd

`int gnutls_rnd` (*gnutls_rnd_level_t* `level`, *void* * `data`, *size_t* `len`) [Function]

level: a security level

data: place to store random bytes

len: The requested size

This function will generate random data and store it to output buffer. The value of `level` should be one of `GNUTLS_RND_NONCE` , `GNUTLS_RND_RANDOM` and `GNUTLS_RND_KEY` . See the manual and `gnutls_rnd_level_t` for detailed information.

This function is thread-safe and also fork-safe.

Returns: Zero on success, or a negative error code on error.

Since: 2.12.0

gnutls_rnd_refresh

`void gnutls_rnd_refresh` (*void*) [Function]

This function refreshes the random generator state. That is the current precise time, CPU usage, and other values are input into its state.

On a slower rate input from /dev/urandom is mixed too.

Since: 3.1.7

E.13 Compatibility API

The following functions are carried over from old GnuTLS released. They might be removed at a later version. Their prototypes lie in `gnutls/compat.h`.

gnutls_compression_get

`gnutls_compression_method_t gnutls_compression_get` (*gnutls_session_t* `session`) [Function]

session: is a `gnutls_session_t` type.

Get the currently used compression algorithm.

Returns: the currently used compression method, a `gnutls_compression_method_t` value.

gnutls_compression_get_id

`gnutls_compression_method_t gnutls_compression_get_id` [Function]
(*const char * name*)

name: is a compression method name

The names are compared in a case insensitive way.

Returns: an id of the specified in a string compression method, or `GNUTLS_COMP_UNKNOWN` on error.

gnutls_compression_get_name

`const char * gnutls_compression_get_name` [Function]
(*gnutls_compression_method_t algorithm*)

algorithm: is a Compression algorithm

Convert a `gnutls_compression_method_t` value to a string.

Returns: a pointer to a string that contains the name of the specified compression algorithm, or `NULL` .

gnutls_compression_list

`const gnutls_compression_method_t *` [Function]
`gnutls_compression_list ( void)`

Get a list of compression methods.

Returns: a zero-terminated list of `gnutls_compression_method_t` integers indicating the available compression methods.

gnutls_global_set_mem_functions

`void gnutls_global_set_mem_functions (gnutls_alloc_function` [Function]
alloc_func, *gnutls_alloc_function secure_alloc_func*, *gnutls_is_secure_function is_secure_func*, *gnutls_realloc_function realloc_func*, *gnutls_free_function free_func*)

alloc_func: it's the default memory allocation function. Like `malloc()` .

secure_alloc_func: This is the memory allocation function that will be used for sensitive data.

is_secure_func: a function that returns 0 if the memory given is not secure. May be NULL.

realloc_func: A realloc function

free_func: The function that frees allocated data. Must accept a NULL pointer.

Deprecated: since 3.3.0 it is no longer possible to replace the internally used memory allocation functions

This is the function where you set the memory allocation functions gnutls is going to use. By default the libc's allocation functions (`malloc()` , `free()`), are used by gnutls, to allocate both sensitive and not sensitive data. This function is provided to set the memory allocation functions to something other than the defaults

This function must be called before `gnutls_global_init()` is called. This function is not thread safe.

gnutls_openpgp_privkey_sign_hash

`int gnutls_openpgp_privkey_sign_hash` [Function]
(*gnutls_openpgp_privkey_t* `key`, *const gnutls_datum_t ** `hash`, *gnutls_datum_t ** `signature`)

key: Holds the key

hash: holds the data to be signed

signature: will contain newly allocated signature

This function is no-op.

Returns: `GNUTLS_E_UNIMPLEMENTED_FEATURE` .

gnutls_priority_compression_list

`int gnutls_priority_compression_list` (*gnutls_priority_t* `pcache`, *const unsigned int *** `list`) [Function]

pcache: is a `gnutls_prioritity_t` type.

list: will point to an integer list

Get a list of available compression method in the priority structure.

Returns: the number of methods, or an error code.

Since: 3.0

gnutls_x509_crt_get_preferred_hash_algorithm

`int gnutls_x509_crt_get_preferred_hash_algorithm` [Function]
(*gnutls_x509_crt_t* `crt`, *gnutls_digest_algorithm_t ** `hash`, *unsigned int ** `mand`)

crt: Holds the certificate

hash: The result of the call with the hash algorithm used for signature

mand: If non-zero it means that the algorithm MUST use this hash. May be `NULL` .

This function will read the certificate and return the appropriate digest algorithm to use for signing with this certificate. Some certificates (i.e. DSA might not be able to sign without the preferred algorithm).

Deprecated: Please use `gnutls_pubkey_get_preferred_hash_algorithm()` .

Returns: the 0 if the hash algorithm is found. A negative error code is returned on error.

Since: 2.12.0

gnutls_x509_privkey_sign_hash

`int gnutls_x509_privkey_sign_hash` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t ** `hash`, *gnutls_datum_t ** `signature`) [Function]

key: a key

hash: holds the data to be signed

signature: will contain newly allocated signature

This function will sign the given hash using the private key. Do not use this function directly unless you know what it is. Typical signing requires the data to be hashed and stored in special formats (e.g. BER Digest-Info for RSA).

This API is provided only for backwards compatibility, and thus restricted to RSA, DSA and ECDSA key types. For other key types please use `gnutls_privkey_sign_hash()` and `gnutls_privkey_sign_data()` .

Returns: On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Deprecated in: 2.12.0

Appendix F Copying Information

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
`https://fsf.org/`

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at

your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[CBCATT]
Bodo Moeller, "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", 2002, available from https://www.openssl.org/~bodo/tls-cbc.txt.

[GPGH] Mike Ashley, "The GNU Privacy Handbook", 2002, available from https://www.gnupg.org/gph/en/manual.pdf.

[GUTPKI]
Peter Gutmann, "Everything you never wanted to know about PKI but were forced to find out", Available from https://www.cs.auckland.ac.nz/~pgut001/.

[PRNGATTACKS]
John Kelsey and Bruce Schneier, "Cryptanalytic Attacks on Pseudorandom Number Generators", Available from https://www.schneier.com/academic/paperfiles/paper-prngs.pdf.

[KEYPIN] Chris Evans and Chris Palmer, "Public Key Pinning Extension for HTTP", Available from https://tools.ietf.org/html/draft-ietf-websec-key-pinning-01.

[NISTSP80057]
NIST Special Publication 800-57, "Recommendation for Key Management - Part 1: General (Revised)", March 2007, available from https://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf.

[RFC7413]
Y. Cheng and J. Chu and S. Radhakrishnan and A. Jain, "TCP Fast Open", December 2014, Available from https://www.ietf.org/rfc/rfc7413.txt.

[RFC7918]
A. Langley, N. Modadugu, B. Moeller, "Transport Layer Security (TLS) False Start", August 2016, Available from https://www.ietf.org/rfc/rfc7918.txt.

[RFC6125]
Peter Saint-Andre and Jeff Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", March 2011, Available from https://www.ietf.org/rfc/rfc6125.txt.

[RFC7685]
Adam Langley, "A Transport Layer Security (TLS) ClientHello Padding Extension", October 2015, Available from https://www.ietf.org/rfc/rfc7685.txt.

[RFC7613]
Peter Saint-Andre and Alexey Melnikov, "Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords", August 2015, Available from https://www.ietf.org/rfc/rfc7613.txt.

[RFC2246]
Tim Dierks and Christopher Allen, "The TLS Protocol Version 1.0", January 1999, Available from https://www.ietf.org/rfc/rfc2246.txt.

[RFC6083]
M. Tuexen and R. Seggelmann and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", January 2011, Available from https://www.ietf.org/rfc/rfc6083.txt.

[RFC4418]
Ted Krovetz, "UMAC: Message Authentication Code using Universal Hashing", March 2006, Available from https://www.ietf.org/rfc/rfc4418.txt.

[RFC4680]
S. Santesson, "TLS Handshake Message for Supplemental Data", September 2006, Available from https://www.ietf.org/rfc/rfc4680.txt.

[RFC7633]
P. Hallam-Baker, "X.509v3 Transport Layer Security (TLS) Feature Extension", October 2015, Available from https://www.ietf.org/rfc/rfc7633.txt.

[RFC7919]
D. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)", August 2016, Available from https://www.ietf.org/rfc/rfc7919.txt.

[RFC4514]
Kurt D. Zeilenga, "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", June 2006, Available from https://www.ietf.org/rfc/rfc4513.txt.

[RFC4346]
Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.1", Match 2006, Available from https://www.ietf.org/rfc/rfc4346.txt.

[RFC4347]
Eric Rescorla and Nagendra Modadugu, "Datagram Transport Layer Security", April 2006, Available from https://www.ietf.org/rfc/rfc4347.txt.

[RFC5246]
Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.2", August 2008, Available from https://www.ietf.org/rfc/rfc5246.txt.

[RFC2440]
Jon Callas, Lutz Donnerhacke, Hal Finney and Rodney Thayer, "OpenPGP Message Format", November 1998, Available from https://www.ietf.org/rfc/rfc2440.txt.

[RFC4880]
Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw and Rodney Thayer, "OpenPGP Message Format", November 2007, Available from https://www.ietf.org/rfc/rfc4880.txt.

[RFC4211]
J. Schaad, "Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)", September 2005, Available from https://www.ietf.org/rfc/rfc4211.txt.

[RFC2817]
Rohit Khare and Scott Lawrence, "Upgrading to TLS Within HTTP/1.1", May 2000, Available from https://www.ietf.org/rfc/rfc2817.txt

[RFC2818]
Eric Rescorla, "HTTP Over TLS", May 2000, Available from https://www.ietf/rfc/rfc2818.txt.

[RFC2945]
Tom Wu, "The SRP Authentication and Key Exchange System", September 2000, Available from https://www.ietf.org/rfc/rfc2945.txt.

[RFC7301]
S. Friedl, A. Popov, A. Langley, E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", July 2014, Available from https://www.ietf.org/rfc/rfc7301.txt.

[RFC2986]
Magnus Nystrom and Burt Kaliski, "PKCS 10 v1.7: Certification Request Syntax Specification", November 2000, Available from https://www.ietf.org/rfc/rfc2986.txt.

[PKIX] D. Cooper, S. Santesson, S. Farrel, S. Boeyen, R. Housley, W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", May 2008, available from https://www.ietf.org/rfc/rfc5280.txt.

[RFC3749]
Scott Hollenbeck, "Transport Layer Security Protocol Compression Methods", May 2004, available from https://www.ietf.org/rfc/rfc3749.txt.

[RFC3820]
Steven Tuecke, Von Welch, Doug Engert, Laura Pearlman, and Mary Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", June 2004, available from https://www.ietf.org/rfc/rfc3820.

[RFC6520]
R. Seggelmann, M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", February 2012, available from https://www.ietf.org/rfc/rfc6520.

[RFC5746]
E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", February 2010, available from `https://www.ietf.org/rfc/rfc5746`.

[RFC5280]
D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", May 2008, available from `https://www.ietf.org/rfc/rfc5280`.

[TLSTKT]
Joseph Salowey, Hao Zhou, Pasi Eronen, Hannes Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", January 2008, available from `https://www.ietf.org/rfc/rfc5077`.

[PKCS12] RSA Laboratories, "PKCS 12 v1.0: Personal Information Exchange Syntax", June 1999, Available from `https://www.rsa.com`.

[PKCS11] RSA Laboratories, "PKCS #11 Base Functionality v2.30: Cryptoki – Draft 4", July 2009, Available from `https://www.rsa.com`.

[RESCORLA]
Eric Rescorla, "SSL and TLS: Designing and Building Secure Systems", 2001

[SELKEY]
Arjen Lenstra and Eric Verheul, "Selecting Cryptographic Key Sizes", 2003, available from `https://www.win.tue.nl/~klenstra/key.pdf`.

[SSL3] Alan Freier, Philip Karlton and Paul Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", August 2011, Available from `https://www.ietf.org/rfc/rfc6101.txt`.

[STEVENS]
Richard Stevens, "UNIX Network Programming, Volume 1", Prentice Hall PTR, January 1998

[TLSEXT] Simon Blake-Wilson, Magnus Nystrom, David Hopwood, Jan Mikkelsen and Tim Wright, "Transport Layer Security (TLS) Extensions", June 2003, Available from `https://www.ietf.org/rfc/rfc3546.txt`.

[TLSPGP] Nikos Mavrogiannopoulos, "Using OpenPGP keys for TLS authentication", January 2011. Available from `https://www.ietf.org/rfc/rfc6091.txt`.

[TLSSRP] David Taylor, Trevor Perrin, Tom Wu and Nikos Mavrogiannopoulos, "Using SRP for TLS Authentication", November 2007. Available from `https://www.ietf.org/rfc/rfc5054.txt`.

[TLSPSK] Pasi Eronen and Hannes Tschofenig, "Pre-shared key Ciphersuites for TLS", December 2005, Available from `https://www.ietf.org/rfc/rfc4279.txt`.

[TOMSRP]
Tom Wu, "The Stanford SRP Authentication Project", Available at `https://srp.stanford.edu/`.

[WEGER] Arjen Lenstra and Xiaoyun Wang and Benne de Weger, "Colliding X.509 Certificates", Cryptology ePrint Archive, Report 2005/067, Available at https://eprint.iacr.org/.

[ECRYPT] European Network of Excellence in Cryptology II, "ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010)", Available at https://www.ecrypt.eu.org/documents/D.SPA.13.pdf.

[RFC5056] N. Williams, "On the Use of Channel Bindings to Secure Channels", November 2007, available from https://www.ietf.org/rfc/rfc5056.

[RFC5764] D. McGrew, E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)On the Use of Channel Bindings to Secure Channels", May 2010, available from https://www.ietf.org/rfc/rfc5764.

[RFC5929] J. Altman, N. Williams, L. Zhu, "Channel Bindings for TLS", July 2010, available from https://www.ietf.org/rfc/rfc5929.

[PKCS11URI] J. Pechanec, D. Moffat, "The PKCS#11 URI Scheme", April 2015, available from https://www.ietf.org/rfc/rfc7512.

[TPMURI] C. Latze, N. Mavrogiannopoulos, "The TPMKEY URI Scheme", January 2013, Work in progress, available from https://tools.ietf.org/html/draft-mavrogiannopoulos-tpmuri-01.

[ANDERSON] R. J. Anderson, "Security Engineering: A Guide to Building Dependable Distributed Systems", John Wiley \& Sons, Inc., 2001.

[RFC4821] M. Mathis, J. Heffner, "Packetization Layer Path MTU Discovery", March 2007, available from https://www.ietf.org/rfc/rfc4821.txt.

[RFC2560] M. Myers et al, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", June 1999, Available from https://www.ietf.org/rfc/rfc2560.txt.

[RIVESTCRL] R. L. Rivest, "Can We Eliminate Certificate Revocation Lists?", Proceedings of Financial Cryptography '98; Springer Lecture Notes in Computer Science No. 1465 (Rafael Hirschfeld, ed.), February 1998), pages 178–183, available from https://people.csail.mit.edu/rivest/Rivest-CanWeEliminateCertificateRevocationLists.pdf.

Function and Data Index

D

dane_cert_type_name 716
dane_cert_usage_name 716
dane_match_type_name 716
dane_query_data 716
dane_query_deinit 716
dane_query_entries 717
dane_query_status 717
dane_query_tlsa 717
dane_query_to_raw_tlsa 717
dane_raw_tlsa 718
dane_state_deinit 718
dane_state_init 718
dane_state_set_dlv_file 719
dane_strerror 719
dane_verification_status_print 719
dane_verify_crt 189, 719
dane_verify_crt_raw 720
dane_verify_session_crt 721

G

gnutls_aead_cipher_decrypt 721
gnutls_aead_cipher_decryptv2 722
gnutls_aead_cipher_deinit 722
gnutls_aead_cipher_encrypt 723
gnutls_aead_cipher_encryptv 285, 723
gnutls_aead_cipher_encryptv2 724
gnutls_aead_cipher_init 724
gnutls_alert_get 171, 360
gnutls_alert_get_name 171, 360
gnutls_alert_get_strname 360
gnutls_alert_send 171, 360
gnutls_alert_send_appropriate 361
gnutls_alpn_get_selected_protocol 361
gnutls_alpn_set_protocols 361
gnutls_anon_allocate_client_credentials 362
gnutls_anon_allocate_server_credentials 362
gnutls_anon_free_client_credentials 362
gnutls_anon_free_server_credentials 362
gnutls_anon_set_params_function 362
gnutls_anon_set_server_dh_params 363
gnutls_anon_set_server_known_dh_params 363
gnutls_anon_set_server_params_function 363
gnutls_anti_replay_deinit 363
gnutls_anti_replay_enable 364
gnutls_anti_replay_init 364
gnutls_anti_replay_set_add_function 364
gnutls_anti_replay_set_ptr 364
gnutls_anti_replay_set_window 365
gnutls_auth_client_get_type 365
gnutls_auth_get_type 365
gnutls_auth_server_get_type 366
gnutls_base64_decode2 366
gnutls_base64_encode2 366
gnutls_buffer_append_data 366
gnutls_bye 169, 367
gnutls_certificate_activation_time_peers 367
gnutls_certificate_allocate_credentials 367
gnutls_certificate_client_get_request_status 368
gnutls_certificate_expiration_time_peers 368
gnutls_certificate_free_ca_names 368
gnutls_certificate_free_cas 368
gnutls_certificate_free_credentials 368
gnutls_certificate_free_crls 369
gnutls_certificate_free_keys 369
gnutls_certificate_get_crt_raw 369
gnutls_certificate_get_issuer 369
gnutls_certificate_get_ocsp_expiration 55, 370
gnutls_certificate_get_ours 370
gnutls_certificate_get_peers 371
gnutls_certificate_get_peers_subkey_id 371
gnutls_certificate_get_trust_list 493
gnutls_certificate_get_verify_flags 371
gnutls_certificate_get_x509_crt 371
gnutls_certificate_get_x509_key 372
gnutls_certificate_send_x509_rdn_sequence 150, 372
gnutls_certificate_server_set_request 150, 373
gnutls_certificate_set_dh_params 373
gnutls_certificate_set_flags 373
gnutls_certificate_set_key 148, 671
gnutls_certificate_set_known_dh_params 374
gnutls_certificate_set_ocsp_status_request_file 374
gnutls_certificate_set_ocsp_status_request_file2 375
gnutls_certificate_set_ocsp_status_request_function 375
gnutls_certificate_set_ocsp_status_request_function2 376
gnutls_certificate_set_ocsp_status_request_mem 376
gnutls_certificate_set_params_function 377
gnutls_certificate_set_pin_function 148, 377
gnutls_certificate_set_rawpk_key_file 377
gnutls_certificate_set_rawpk_key_mem 379
gnutls_certificate_set_retrieve_function 380
gnutls_certificate_set_retrieve_function2 672

gnutls_certificate_set_retrieve_function3 673
gnutls_certificate_set_trust_list 493
gnutls_certificate_set_verify_flags 380
gnutls_certificate_set_verify_function 381
gnutls_certificate_set_verify_limits 381
gnutls_certificate_set_x509_crl 381
gnutls_certificate_set_x509_crl_file 382
gnutls_certificate_set_x509_crl_mem 382
gnutls_certificate_set_x509_key 382
gnutls_certificate_set_x509_key_file 383
gnutls_certificate_set_x509_key_file2 384
gnutls_certificate_set_x509_key_mem 384
gnutls_certificate_set_x509_key_mem2 385
gnutls_certificate_set_x509_simple_pkcs12_file 386
gnutls_certificate_set_x509_simple_pkcs12_mem 386
gnutls_certificate_set_x509_system_trust 151, 387
gnutls_certificate_set_x509_trust 387
gnutls_certificate_set_x509_trust_dir 388
gnutls_certificate_set_x509_trust_file 388
gnutls_certificate_set_x509_trust_mem 389
gnutls_certificate_type_get 389
gnutls_certificate_type_get_id 390
gnutls_certificate_type_get_name 390
gnutls_certificate_type_get2 389
gnutls_certificate_type_list 390
gnutls_certificate_verification_profile_get_id 493
gnutls_certificate_verification_profile_get_name 494
gnutls_certificate_verification_status_print 390
gnutls_certificate_verify_flags 36, 186
gnutls_certificate_verify_peers 391
gnutls_certificate_verify_peers2 392
gnutls_certificate_verify_peers3 392
gnutls_check_version 393
gnutls_cipher_add_auth 724
gnutls_cipher_decrypt 725
gnutls_cipher_decrypt2 725
gnutls_cipher_deinit 725
gnutls_cipher_encrypt 725
gnutls_cipher_encrypt2 726
gnutls_cipher_get 393
gnutls_cipher_get_block_size 726
gnutls_cipher_get_id 393
gnutls_cipher_get_iv_size 726
gnutls_cipher_get_key_size 394
gnutls_cipher_get_name 394
gnutls_cipher_get_tag_size 726
gnutls_cipher_init 727
gnutls_cipher_list 394
gnutls_cipher_set_iv 727
gnutls_cipher_suite_get_name 394
gnutls_cipher_suite_info 395
gnutls_cipher_tag 727
gnutls_compression_get 737
gnutls_compression_get_id 738
gnutls_compression_get_name 738
gnutls_compression_list 738
gnutls_credentials_clear 395
gnutls_credentials_get 395
gnutls_credentials_set 146, 396
gnutls_crypto_register_aead_cipher 295, 727
gnutls_crypto_register_cipher 294, 728
gnutls_crypto_register_digest 296, 729
gnutls_crypto_register_mac 296, 729
gnutls_db_check_entry 396
gnutls_db_check_entry_expire_time 396
gnutls_db_check_entry_time 397
gnutls_db_get_default_cache_expiration 397
gnutls_db_get_ptr 397
gnutls_db_remove_session 397
gnutls_db_set_cache_expiration 397
gnutls_db_set_ptr 397
gnutls_db_set_remove_function 398
gnutls_db_set_retrieve_function 398
gnutls_db_set_store_function 398
gnutls_decode_ber_digest_info 730
gnutls_decode_gost_rs_value 730
gnutls_decode_rs_value 730
gnutls_deinit 170, 398
gnutls_dh_get_group 399
gnutls_dh_get_peers_public_bits 399
gnutls_dh_get_prime_bits 399
gnutls_dh_get_pubkey 399
gnutls_dh_get_secret_bits 400
gnutls_dh_params_cpy 400
gnutls_dh_params_deinit 400
gnutls_dh_params_export_pkcs3 401
gnutls_dh_params_export_raw 401
gnutls_dh_params_export2_pkcs3 400
gnutls_dh_params_generate2 401
gnutls_dh_params_import_dsa 402
gnutls_dh_params_import_pkcs3 402
gnutls_dh_params_import_raw 402
gnutls_dh_params_import_raw2 403
gnutls_dh_params_import_raw3 403
gnutls_dh_params_init 403
gnutls_dh_set_prime_bits 403
gnutls_digest_get_id 404
gnutls_digest_get_name 404
gnutls_digest_get_oid 404
gnutls_digest_list 405
gnutls_dtls_cookie_send 490
gnutls_dtls_cookie_verify 490
gnutls_dtls_get_data_mtu 491
gnutls_dtls_get_mtu 491
gnutls_dtls_get_timeout 159, 491
gnutls_dtls_prestate_set 491
gnutls_dtls_set_data_mtu 492
gnutls_dtls_set_mtu 492
gnutls_dtls_set_timeouts 492

gnutls_ecc_curve_get 405
gnutls_ecc_curve_get_id 405
gnutls_ecc_curve_get_name 405
gnutls_ecc_curve_get_oid 405
gnutls_ecc_curve_get_pk 406
gnutls_ecc_curve_get_size 406
gnutls_ecc_curve_list 406
gnutls_encode_ber_digest_info 731
gnutls_encode_gost_rs_value 731
gnutls_encode_rs_value 731
gnutls_error_is_fatal 168, 406
gnutls_error_to_alert 171, 406
gnutls_est_record_overhead_size 407
gnutls_ext_get_current_msg 407
gnutls_ext_get_data 407
gnutls_ext_get_name 408
gnutls_ext_get_name2 408
gnutls_ext_raw_parse 408
gnutls_ext_register 409
gnutls_ext_set_data 409
gnutls_fingerprint 410
gnutls_fips140_mode_enabled 410
gnutls_fips140_set_mode 410
gnutls_get_system_config_file 280, 411
gnutls_global_deinit 411
gnutls_global_init 411
gnutls_global_set_audit_
log_function 140, 411
gnutls_global_set_log_function 412
gnutls_global_set_log_level 412
gnutls_global_set_mem_functions 738
gnutls_global_set_mutex 412
gnutls_global_set_time_function 413
gnutls_gost_paramset_get_name 413
gnutls_gost_paramset_get_oid 413
gnutls_group_get 413
gnutls_group_get_id 413
gnutls_group_get_name 414
gnutls_group_list 414
gnutls_handshake 166, 414
gnutls_handshake_description_get_name 415
gnutls_handshake_get_last_in 415
gnutls_handshake_get_last_out 415
gnutls_handshake_set_hook_function .. 183, 415
gnutls_handshake_set_max_packet_length ... 416
gnutls_handshake_set_post_
client_hello_function 416
gnutls_handshake_set_
private_extensions 417
gnutls_handshake_set_random 417
gnutls_handshake_set_timeout 166, 417
gnutls_hash 732
gnutls_hash_copy 732
gnutls_hash_deinit 732
gnutls_hash_fast 732
gnutls_hash_get_len 733
gnutls_hash_init 733
gnutls_hash_output 733
gnutls_heartbeat_allowed 418
gnutls_heartbeat_enable 418
gnutls_heartbeat_get_timeout 418
gnutls_heartbeat_ping 418
gnutls_heartbeat_pong 419
gnutls_heartbeat_set_timeouts 419
gnutls_hex_decode 420
gnutls_hex_decode2 420
gnutls_hex_encode 420
gnutls_hex_encode2 421
gnutls_hex2bin 419
gnutls_hkdf_expand 733
gnutls_hkdf_extract 734
gnutls_hmac 734
gnutls_hmac_copy 734
gnutls_hmac_deinit 734
gnutls_hmac_fast 735
gnutls_hmac_get_key_size 735
gnutls_hmac_get_len 735
gnutls_hmac_init 735
gnutls_hmac_output 736
gnutls_hmac_set_nonce 736
gnutls_idna_map 421
gnutls_idna_reverse_map 421
gnutls_init 144, 422
gnutls_key_generate 422
gnutls_kx_get 422
gnutls_kx_get_id 422
gnutls_kx_get_name 423
gnutls_kx_list 423
gnutls_load_file 423
gnutls_mac_get 423
gnutls_mac_get_id 423
gnutls_mac_get_key_size 424
gnutls_mac_get_name 424
gnutls_mac_get_nonce_size 736
gnutls_mac_list 424
gnutls_memcmp 424
gnutls_memset 424
gnutls_ocsp_req_add_cert 629
gnutls_ocsp_req_add_cert_id 630
gnutls_ocsp_req_deinit 630
gnutls_ocsp_req_export 630
gnutls_ocsp_req_get_cert_id 631
gnutls_ocsp_req_get_extension 631
gnutls_ocsp_req_get_nonce 632
gnutls_ocsp_req_get_version 632
gnutls_ocsp_req_import 632
gnutls_ocsp_req_init 632
gnutls_ocsp_req_print 632
gnutls_ocsp_req_randomize_nonce 633
gnutls_ocsp_req_set_extension 633
gnutls_ocsp_req_set_nonce 633
gnutls_ocsp_resp_check_crt 634
gnutls_ocsp_resp_deinit 634
gnutls_ocsp_resp_export 634
gnutls_ocsp_resp_export2 634
gnutls_ocsp_resp_get_certs 634

gnutls_ocsp_resp_get_extension 635
gnutls_ocsp_resp_get_nonce 635
gnutls_ocsp_resp_get_produced 636
gnutls_ocsp_resp_get_responder 636
gnutls_ocsp_resp_get_responder_raw_id 637
gnutls_ocsp_resp_get_responder2 636
gnutls_ocsp_resp_get_response 637
gnutls_ocsp_resp_get_signature 637
gnutls_ocsp_resp_get_
signature_algorithm 638
gnutls_ocsp_resp_get_single 52, 638
gnutls_ocsp_resp_get_status 638
gnutls_ocsp_resp_get_version 639
gnutls_ocsp_resp_import 639
gnutls_ocsp_resp_import2 639
gnutls_ocsp_resp_init 639
gnutls_ocsp_resp_list_import2 640
gnutls_ocsp_resp_print 640
gnutls_ocsp_resp_verify 640
gnutls_ocsp_resp_verify_direct 641
gnutls_ocsp_status_request_
enable_client 425
gnutls_ocsp_status_request_get 425
gnutls_ocsp_status_request_get2 425
gnutls_ocsp_status_request_is_checked 426
gnutls_oid_to_digest 426
gnutls_oid_to_ecc_curve 427
gnutls_oid_to_gost_paramset 427
gnutls_oid_to_mac 427
gnutls_oid_to_pk 427
gnutls_oid_to_sign 427
gnutls_openpgp_privkey_sign_hash 739
gnutls_openpgp_send_cert 428
gnutls_packet_deinit 428
gnutls_packet_get 428
gnutls_pbkdf2 736
gnutls_pcert_deinit 674
gnutls_pcert_export_openpgp 674
gnutls_pcert_export_x509 674
gnutls_pcert_import_openpgp 674
gnutls_pcert_import_openpgp_raw 675
gnutls_pcert_import_rawpk 675
gnutls_pcert_import_rawpk_raw 675
gnutls_pcert_import_x509 676
gnutls_pcert_import_x509_list 676
gnutls_pcert_import_x509_raw 677
gnutls_pcert_list_import_x509_file 677
gnutls_pcert_list_import_x509_raw 678
gnutls_pem_base64_decode 428
gnutls_pem_base64_decode2 429
gnutls_pem_base64_encode 429
gnutls_pem_base64_encode2 429
gnutls_perror 430
gnutls_pk_algorithm_get_name 430
gnutls_pk_bits_to_sec_param 181, 430
gnutls_pk_get_id 430
gnutls_pk_get_name 431
gnutls_pk_get_oid 431
gnutls_pk_list 431
gnutls_pk_to_sign 431
gnutls_pkcs_schema_get_name 494
gnutls_pkcs_schema_get_oid 495
gnutls_pkcs11_add_provider 111, 649
gnutls_pkcs11_copy_attached_extension 650
gnutls_pkcs11_copy_pubkey 650
gnutls_pkcs11_copy_secret_key 651
gnutls_pkcs11_copy_x509_crt 651
gnutls_pkcs11_copy_x509_crt2 ..... 117, 651
gnutls_pkcs11_copy_x509_privkey 652
gnutls_pkcs11_copy_x509_privkey2 ..... 116, 652
gnutls_pkcs11_crt_is_known 653
gnutls_pkcs11_deinit 653
gnutls_pkcs11_delete_url 117, 653
gnutls_pkcs11_get_pin_function 653
gnutls_pkcs11_get_raw_issuer 654
gnutls_pkcs11_get_raw_issuer_by_dn 654
gnutls_pkcs11_get_raw_issuer_by_
subject_key_id 654
gnutls_pkcs11_init 112, 655
gnutls_pkcs11_obj_deinit 655
gnutls_pkcs11_obj_export 655
gnutls_pkcs11_obj_export_url 656
gnutls_pkcs11_obj_export2 656
gnutls_pkcs11_obj_export3 656
gnutls_pkcs11_obj_flags_get_str 657
gnutls_pkcs11_obj_get_exts 657
gnutls_pkcs11_obj_get_flags 657
gnutls_pkcs11_obj_get_info 114, 658
gnutls_pkcs11_obj_get_ptr 118, 658
gnutls_pkcs11_obj_get_type 658
gnutls_pkcs11_obj_import_url 659
gnutls_pkcs11_obj_init 659
gnutls_pkcs11_obj_list_import_url3 659
gnutls_pkcs11_obj_list_import_url4 660
gnutls_pkcs11_obj_set_info 660
gnutls_pkcs11_obj_set_pin_function 661
gnutls_pkcs11_privkey_cpy 661
gnutls_pkcs11_privkey_deinit 661
gnutls_pkcs11_privkey_export_pubkey 661
gnutls_pkcs11_privkey_export_url 662
gnutls_pkcs11_privkey_generate 662
gnutls_pkcs11_privkey_generate2 662
gnutls_pkcs11_privkey_generate3 663
gnutls_pkcs11_privkey_get_info 664
gnutls_pkcs11_privkey_get_pk_algorithm ... 664
gnutls_pkcs11_privkey_import_url 664
gnutls_pkcs11_privkey_init 664
gnutls_pkcs11_privkey_set_pin_function ... 665
gnutls_pkcs11_privkey_status 665
gnutls_pkcs11_reinit 665
gnutls_pkcs11_set_pin_function 665
gnutls_pkcs11_set_token_function 666
gnutls_pkcs11_token_check_mechanism 666
gnutls_pkcs11_token_get_flags 666
gnutls_pkcs11_token_get_info 667
gnutls_pkcs11_token_get_mechanism 667

gnutls_pkcs11_token_get_ptr 118, 667
gnutls_pkcs11_token_get_random 668
gnutls_pkcs11_token_get_url 668
gnutls_pkcs11_token_init 668
gnutls_pkcs11_token_set_pin 668
gnutls_pkcs11_type_get_name 669
gnutls_pkcs12_bag_decrypt 641
gnutls_pkcs12_bag_deinit 642
gnutls_pkcs12_bag_enc_info 642
gnutls_pkcs12_bag_encrypt 642
gnutls_pkcs12_bag_get_count 643
gnutls_pkcs12_bag_get_data 643
gnutls_pkcs12_bag_get_friendly_name 643
gnutls_pkcs12_bag_get_key_id 643
gnutls_pkcs12_bag_get_type 644
gnutls_pkcs12_bag_init 644
gnutls_pkcs12_bag_set_crl 644
gnutls_pkcs12_bag_set_crt 644
gnutls_pkcs12_bag_set_data 644
gnutls_pkcs12_bag_set_friendly_name 645
gnutls_pkcs12_bag_set_key_id 645
gnutls_pkcs12_bag_set_privkey 645
gnutls_pkcs12_deinit 645
gnutls_pkcs12_export 646
gnutls_pkcs12_export2 646
gnutls_pkcs12_generate_mac 646
gnutls_pkcs12_generate_mac2 646
gnutls_pkcs12_get_bag 647
gnutls_pkcs12_import 647
gnutls_pkcs12_init 647
gnutls_pkcs12_mac_info 648
gnutls_pkcs12_set_bag 648
gnutls_pkcs12_simple_parse 59, 648
gnutls_pkcs12_verify_mac 649
gnutls_pkcs7_add_attr 620
gnutls_pkcs7_attrs_deinit 621
gnutls_pkcs7_deinit 621
gnutls_pkcs7_delete_crl 621
gnutls_pkcs7_delete_crt 621
gnutls_pkcs7_export 621
gnutls_pkcs7_export2 622
gnutls_pkcs7_get_attr 622
gnutls_pkcs7_get_crl_count 623
gnutls_pkcs7_get_crl_raw 623
gnutls_pkcs7_get_crl_raw2 623
gnutls_pkcs7_get_crt_count 623
gnutls_pkcs7_get_crt_raw 624
gnutls_pkcs7_get_crt_raw2 624
gnutls_pkcs7_get_embedded_data 624
gnutls_pkcs7_get_embedded_data_oid 625
gnutls_pkcs7_get_signature_count 625
gnutls_pkcs7_get_signature_info 625
gnutls_pkcs7_import 625
gnutls_pkcs7_init 626
gnutls_pkcs7_print 626
gnutls_pkcs7_print_signature_info 626
gnutls_pkcs7_set_crl 627
gnutls_pkcs7_set_crl_raw 627
gnutls_pkcs7_set_crt 627
gnutls_pkcs7_set_crt_raw 627
gnutls_pkcs7_sign 289, 627
gnutls_pkcs7_signature_info_deinit 628
gnutls_pkcs7_verify 628
gnutls_pkcs7_verify_direct 629
gnutls_pkcs8_info 494
gnutls_prf 431
gnutls_prf_early 432
gnutls_prf_hash_get 433
gnutls_prf_raw 433
gnutls_prf_rfc5705 ..... 193, 434
gnutls_priority_certificate_type_list 434
gnutls_priority_certificate_type_list2 435
gnutls_priority_cipher_list 435
gnutls_priority_compression_list 739
gnutls_priority_deinit 435
gnutls_priority_ecc_curve_list 435
gnutls_priority_get_cipher_suite_index 436
gnutls_priority_group_list 436
gnutls_priority_init 436
gnutls_priority_init2 437
gnutls_priority_kx_list 438
gnutls_priority_mac_list 438
gnutls_priority_protocol_list 439
gnutls_priority_set 439
gnutls_priority_set_direct 439
gnutls_priority_sign_list 439
gnutls_priority_string_list 440
gnutls_privkey_decrypt_data 107, 678
gnutls_privkey_decrypt_data2 678
gnutls_privkey_deinit 679
gnutls_privkey_export_dsa_raw 679
gnutls_privkey_export_dsa_raw2 679
gnutls_privkey_export_ecc_raw 680
gnutls_privkey_export_ecc_raw2 680
gnutls_privkey_export_gost_raw2 681
gnutls_privkey_export_openpgp 681
gnutls_privkey_export_pkcs11 681
gnutls_privkey_export_rsa_raw 682
gnutls_privkey_export_rsa_raw2 682
gnutls_privkey_export_x509 683
gnutls_privkey_generate 683
gnutls_privkey_generate2 ..... 287, 683
gnutls_privkey_get_pk_algorithm 684
gnutls_privkey_get_seed 684
gnutls_privkey_get_spki 685
gnutls_privkey_get_type 685
gnutls_privkey_import_dsa_raw 685
gnutls_privkey_import_ecc_raw 686
gnutls_privkey_import_ext 686
gnutls_privkey_import_ext2 687
gnutls_privkey_import_ext3 687
gnutls_privkey_import_ext4 ..... 103, 688
gnutls_privkey_import_gost_raw 689
gnutls_privkey_import_openpgp 689
gnutls_privkey_import_openpgp_raw 690
gnutls_privkey_import_pkcs11 690

gnutls_privkey_import_pkcs11_url 690
gnutls_privkey_import_rsa_raw 691
gnutls_privkey_import_tpm_raw 691
gnutls_privkey_import_tpm_url 132, 692
gnutls_privkey_import_url 103, 692
gnutls_privkey_import_x509 692
gnutls_privkey_import_x509_raw 56, 693
gnutls_privkey_init 693
gnutls_privkey_set_flags 694
gnutls_privkey_set_pin_function 694
gnutls_privkey_set_spki 694
gnutls_privkey_sign_data 106, 694
gnutls_privkey_sign_data2 695
gnutls_privkey_sign_hash 106, 695
gnutls_privkey_sign_hash2 696
gnutls_privkey_status 696
gnutls_privkey_verify_params 697
gnutls_privkey_verify_seed 697
gnutls_protocol_get_id 440
gnutls_protocol_get_name 440
gnutls_protocol_get_version 440
gnutls_protocol_list 440
gnutls_psk_allocate_client_credentials ... 441
gnutls_psk_allocate_server_credentials ... 441
gnutls_psk_client_get_hint 441
gnutls_psk_free_client_credentials 441
gnutls_psk_free_server_credentials 441
gnutls_psk_server_get_username 442
gnutls_psk_server_get_username2 442
gnutls_psk_set_client_credentials 442
gnutls_psk_set_client_
credentials_function 155, 443
gnutls_psk_set_client_
credentials_function2 443
gnutls_psk_set_client_credentials2 443
gnutls_psk_set_params_function 444
gnutls_psk_set_server_
credentials_file 156, 444
gnutls_psk_set_server_
credentials_function 444
gnutls_psk_set_server_
credentials_function2 445
gnutls_psk_set_server_credentials_hint ... 445
gnutls_psk_set_server_dh_params 445
gnutls_psk_set_server_known_dh_params 446
gnutls_psk_set_server_params_function 446
gnutls_pubkey_deinit 697
gnutls_pubkey_encrypt_data 105, 697
gnutls_pubkey_export 698
gnutls_pubkey_export_dsa_raw 698
gnutls_pubkey_export_dsa_raw2 699
gnutls_pubkey_export_ecc_raw 699
gnutls_pubkey_export_ecc_raw2 700
gnutls_pubkey_export_ecc_x962 700
gnutls_pubkey_export_gost_raw2 700
gnutls_pubkey_export_rsa_raw 701
gnutls_pubkey_export_rsa_raw2 701
gnutls_pubkey_export2 ..... 101, 698
gnutls_pubkey_get_key_id 701
gnutls_pubkey_get_key_usage 702
gnutls_pubkey_get_openpgp_key_id 702
gnutls_pubkey_get_pk_algorithm 702
gnutls_pubkey_get_preferred_
hash_algorithm 703
gnutls_pubkey_get_spki 703
gnutls_pubkey_import 703
gnutls_pubkey_import_dsa_raw 704
gnutls_pubkey_import_ecc_raw 704
gnutls_pubkey_import_ecc_x962 705
gnutls_pubkey_import_gost_raw 705
gnutls_pubkey_import_openpgp 705
gnutls_pubkey_import_openpgp_raw 706
gnutls_pubkey_import_pkcs11 706
gnutls_pubkey_import_privkey 706
gnutls_pubkey_import_rsa_raw 707
gnutls_pubkey_import_tpm_raw 707
gnutls_pubkey_import_tpm_url 132, 707
gnutls_pubkey_import_url 708
gnutls_pubkey_import_x509 708
gnutls_pubkey_import_x509_crq 708
gnutls_pubkey_import_x509_raw 708
gnutls_pubkey_init 709
gnutls_pubkey_print 709
gnutls_pubkey_set_key_usage 709
gnutls_pubkey_set_pin_function 710
gnutls_pubkey_set_spki 710
gnutls_pubkey_verify_data2 ..... 105, 710
gnutls_pubkey_verify_hash2 ..... 105, 711
gnutls_pubkey_verify_params 711
gnutls_random_art 446
gnutls_range_split 447
gnutls_reauth 447
gnutls_record_can_use_length_hiding 448
gnutls_record_check_corked 448
gnutls_record_check_pending 169, 448
gnutls_record_cork 170, 448
gnutls_record_disable_padding 448
gnutls_record_discard_queued 449
gnutls_record_get_direction 161, 449
gnutls_record_get_discarded 493
gnutls_record_get_max_early_data_size 449
gnutls_record_get_max_size 449
gnutls_record_get_state 450
gnutls_record_overhead_size 450
gnutls_record_recv 168, 450
gnutls_record_recv_early_data 451
gnutls_record_recv_packet 451
gnutls_record_recv_seq 168, 451
gnutls_record_send 167, 452
gnutls_record_send_early_data 453
gnutls_record_send_range 453
gnutls_record_send2 ..... 8, 452
gnutls_record_set_max_early_data_size 454
gnutls_record_set_max_recv_size 454
gnutls_record_set_max_size 454
gnutls_record_set_state 455

gnutls_record_set_timeout 455
gnutls_record_uncork 170, 455
gnutls_register_custom_url 109, 711
gnutls_rehandshake 191, 456
gnutls_rnd 294, 737
gnutls_rnd_refresh 737
gnutls_safe_renegotiation_status 190, 456
gnutls_sec_param_get_name 457
gnutls_sec_param_to_pk_bits 181, 457
gnutls_sec_param_to_symmetric_bits 457
gnutls_server_name_get 457
gnutls_server_name_set 458
gnutls_session_channel_binding 458
gnutls_session_enable_
compatibility_mode 459
gnutls_session_etm_status 459
gnutls_session_ext_master_
secret_status 459
gnutls_session_ext_register 459
gnutls_session_force_valid 460
gnutls_session_get_data 460
gnutls_session_get_data2 460
gnutls_session_get_desc 461
gnutls_session_get_flags 461
gnutls_session_get_id 462
gnutls_session_get_id2 ..... 184, 462
gnutls_session_get_keylog_function 463
gnutls_session_get_master_secret 463
gnutls_session_get_ptr 463
gnutls_session_get_random 463
gnutls_session_get_verify_cert_status 464
gnutls_session_is_resumed 184, 464
gnutls_session_key_update 464
gnutls_session_resumption_requested .. 185, 464
gnutls_session_set_data 465
gnutls_session_set_id 465
gnutls_session_set_keylog_function 465
gnutls_session_set_premaster 465
gnutls_session_set_ptr 466
gnutls_session_set_verify_cert 152, 466
gnutls_session_set_verify_cert2 467
gnutls_session_set_verify_function 467
gnutls_session_supplemental_register 467
gnutls_session_ticket_enable_client 468
gnutls_session_ticket_enable_server .. 185, 468
gnutls_session_ticket_key_generate .. 185, 469
gnutls_session_ticket_send 186, 469
gnutls_set_default_priority 469
gnutls_set_default_priority_append 469
gnutls_sign_algorithm_get 470
gnutls_sign_algorithm_get_client 470
gnutls_sign_algorithm_get_requested 470
gnutls_sign_get_hash_algorithm 471
gnutls_sign_get_id 471
gnutls_sign_get_name 471
gnutls_sign_get_oid 471
gnutls_sign_get_pk_algorithm 472
gnutls_sign_is_secure 472
gnutls_sign_is_secure2 472
gnutls_sign_list 472
gnutls_sign_supports_pk_algorithm 472
gnutls_srp_allocate_client_credentials ... 473
gnutls_srp_allocate_server_credentials ... 473
gnutls_srp_base64_decode 473
gnutls_srp_base64_decode2 473
gnutls_srp_base64_encode 474
gnutls_srp_base64_encode2 474
gnutls_srp_free_client_credentials 474
gnutls_srp_free_server_credentials 474
gnutls_srp_server_get_username 475
gnutls_srp_set_client_credentials 475
gnutls_srp_set_client_
credentials_function 153, 475
gnutls_srp_set_prime_bits 476
gnutls_srp_set_server_
credentials_file 154, 476
gnutls_srp_set_server_
credentials_function 154, 476
gnutls_srp_set_server_fake_salt_seed 477
gnutls_srp_verifier 94, 477
gnutls_srtp_get_keys 15, 478
gnutls_srtp_get_mki 478
gnutls_srtp_get_profile_id 479
gnutls_srtp_get_profile_name 479
gnutls_srtp_get_selected_profile 479
gnutls_srtp_set_mki 479
gnutls_srtp_set_profile 480
gnutls_srtp_set_profile_direct 480
gnutls_store_commitment 188, 480
gnutls_store_pubkey 187, 481
gnutls_strerror 481
gnutls_strerror_name 482
gnutls_subject_alt_names_deinit 495
gnutls_subject_alt_names_get 495
gnutls_subject_alt_names_init 495
gnutls_subject_alt_names_set 496
gnutls_supplemental_get_name 482
gnutls_supplemental_recv 482
gnutls_supplemental_register 482
gnutls_supplemental_send 483
gnutls_system_key_add_x509 712
gnutls_system_key_delete 712
gnutls_system_key_iter_deinit 712
gnutls_system_key_iter_get_info 108, 712
gnutls_system_recv_timeout 483
gnutls_tdb_deinit 483
gnutls_tdb_init 483
gnutls_tdb_set_store_commitment_func 484
gnutls_tdb_set_store_func 484
gnutls_tdb_set_verify_func 484
gnutls_tpm_get_registered 670
gnutls_tpm_key_list_deinit 670
gnutls_tpm_key_list_get_url 670
gnutls_tpm_privkey_delete 131, 133, 670
gnutls_tpm_privkey_generate 131, 671
gnutls_transport_get_int 484

gnutls_transport_get_int2 485
gnutls_transport_get_ptr 485
gnutls_transport_get_ptr2 485
gnutls_transport_set_errno 158, 485
gnutls_transport_set_errno_function 486
gnutls_transport_set_fastopen 161, 715
gnutls_transport_set_int 486
gnutls_transport_set_int2 486
gnutls_transport_set_ptr 486
gnutls_transport_set_ptr2 487
gnutls_transport_set_pull_function .. 157, 487
gnutls_transport_set_pull_
timeout_function 157, 159, 487
gnutls_transport_set_push_function .. 157, 488
gnutls_transport_set_vec_
push_function 157, 488
gnutls_url_is_supported 102, 488
gnutls_utf8_password_normalize 489
gnutls_verify_stored_pubkey 186, 489
gnutls_x509_aia_deinit 496
gnutls_x509_aia_get 496
gnutls_x509_aia_init 497
gnutls_x509_aia_set 497
gnutls_x509_aki_deinit 497
gnutls_x509_aki_get_cert_issuer 497
gnutls_x509_aki_get_id 498
gnutls_x509_aki_init 498
gnutls_x509_aki_set_cert_issuer 498
gnutls_x509_aki_set_id 499
gnutls_x509_cidr_to_rfc5280 499
gnutls_x509_crl_check_issuer 499
gnutls_x509_crl_deinit 499
gnutls_x509_crl_dist_points_deinit 500
gnutls_x509_crl_dist_points_get 500
gnutls_x509_crl_dist_points_init 500
gnutls_x509_crl_dist_points_set 500
gnutls_x509_crl_export 501
gnutls_x509_crl_export2 501
gnutls_x509_crl_get_authority_
key_gn_serial 501
gnutls_x509_crl_get_authority_key_id 502
gnutls_x509_crl_get_crt_count 502
gnutls_x509_crl_get_crt_serial 47, 502
gnutls_x509_crl_get_dn_oid 503
gnutls_x509_crl_get_extension_data 503
gnutls_x509_crl_get_extension_data2 504
gnutls_x509_crl_get_extension_info 504
gnutls_x509_crl_get_extension_oid 504
gnutls_x509_crl_get_issuer_dn 505
gnutls_x509_crl_get_issuer_dn_by_oid 506
gnutls_x509_crl_get_issuer_dn2 505
gnutls_x509_crl_get_issuer_dn3 505
gnutls_x509_crl_get_next_update 506
gnutls_x509_crl_get_number 507
gnutls_x509_crl_get_raw_issuer_dn 507
gnutls_x509_crl_get_signature 507
gnutls_x509_crl_get_
signature_algorithm 507
gnutls_x509_crl_get_signature_oid 508
gnutls_x509_crl_get_this_update 508
gnutls_x509_crl_get_version 508
gnutls_x509_crl_import 508
gnutls_x509_crl_init 508
gnutls_x509_crl_iter_crt_serial 509
gnutls_x509_crl_iter_deinit 509
gnutls_x509_crl_list_import 509
gnutls_x509_crl_list_import2 510
gnutls_x509_crl_print 510
gnutls_x509_crl_privkey_sign 48, 713
gnutls_x509_crl_set_authority_key_id 510
gnutls_x509_crl_set_crt 511
gnutls_x509_crl_set_crt_serial 511
gnutls_x509_crl_set_next_update 511
gnutls_x509_crl_set_number 511
gnutls_x509_crl_set_this_update 512
gnutls_x509_crl_set_version 512
gnutls_x509_crl_sign 512
gnutls_x509_crl_sign2 ..... 48, 513
gnutls_x509_crl_verify 513
gnutls_x509_crq_deinit 514
gnutls_x509_crq_export 514
gnutls_x509_crq_export2 514
gnutls_x509_crq_get_attribute_by_oid 515
gnutls_x509_crq_get_attribute_data 515
gnutls_x509_crq_get_attribute_info 515
gnutls_x509_crq_get_basic_constraints 516
gnutls_x509_crq_get_challenge_password ... 516
gnutls_x509_crq_get_dn 516
gnutls_x509_crq_get_dn_by_oid 518
gnutls_x509_crq_get_dn_oid 518
gnutls_x509_crq_get_dn2 517
gnutls_x509_crq_get_dn3 517
gnutls_x509_crq_get_extension_by_oid 518
gnutls_x509_crq_get_extension_by_oid2 519
gnutls_x509_crq_get_extension_data 519
gnutls_x509_crq_get_extension_data2 520
gnutls_x509_crq_get_extension_info 520
gnutls_x509_crq_get_key_id 521
gnutls_x509_crq_get_key_purpose_oid 521
gnutls_x509_crq_get_key_rsa_raw 521
gnutls_x509_crq_get_key_usage 522
gnutls_x509_crq_get_pk_algorithm 522
gnutls_x509_crq_get_pk_oid 522
gnutls_x509_crq_get_private_
key_usage_period 523
gnutls_x509_crq_get_
signature_algorithm 523
gnutls_x509_crq_get_signature_oid 523
gnutls_x509_crq_get_spki 524
gnutls_x509_crq_get_subject_alt_name 524
gnutls_x509_crq_get_subject_
alt_othername_oid 524
gnutls_x509_crq_get_tlsfeatures 525
gnutls_x509_crq_get_version 525
gnutls_x509_crq_import 525
gnutls_x509_crq_init 526

gnutls_x509_crq_print 526
gnutls_x509_crq_privkey_sign 713
gnutls_x509_crq_set_attribute_by_oid 526
gnutls_x509_crq_set_basic_constraints 527
gnutls_x509_crq_set_challenge_password ... 527
gnutls_x509_crq_set_dn 527
gnutls_x509_crq_set_dn_by_oid 527
gnutls_x509_crq_set_extension_by_oid 528
gnutls_x509_crq_set_key 43, 528
gnutls_x509_crq_set_key_purpose_oid 528
gnutls_x509_crq_set_key_rsa_raw 529
gnutls_x509_crq_set_key_usage 529
gnutls_x509_crq_set_private_
key_usage_period 529
gnutls_x509_crq_set_pubkey 107, 714
gnutls_x509_crq_set_spki 529
gnutls_x509_crq_set_subject_alt_name 530
gnutls_x509_crq_set_subject_
alt_othername 530
gnutls_x509_crq_set_tlsfeatures 531
gnutls_x509_crq_set_version 531
gnutls_x509_crq_sign 531
gnutls_x509_crq_sign2 ..... 43, 531
gnutls_x509_crq_verify 532
gnutls_x509_crt_check_email 532
gnutls_x509_crt_check_hostname 532
gnutls_x509_crt_check_hostname2 533
gnutls_x509_crt_check_ip 533
gnutls_x509_crt_check_issuer 534
gnutls_x509_crt_check_key_purpose 534
gnutls_x509_crt_check_revocation 534
gnutls_x509_crt_cpy_crl_dist_points 534
gnutls_x509_crt_deinit 535
gnutls_x509_crt_equals 535
gnutls_x509_crt_equals2 535
gnutls_x509_crt_export 535
gnutls_x509_crt_export2 536
gnutls_x509_crt_get_activation_time 536
gnutls_x509_crt_get_
authority_info_access 536
gnutls_x509_crt_get_authority_
key_gn_serial 537
gnutls_x509_crt_get_authority_key_id 538
gnutls_x509_crt_get_basic_constraints 538
gnutls_x509_crt_get_ca_status 539
gnutls_x509_crt_get_crl_dist_points 539
gnutls_x509_crt_get_dn 539
gnutls_x509_crt_get_dn_by_oid 541
gnutls_x509_crt_get_dn_oid 541
gnutls_x509_crt_get_dn2 ..... 25, 540
gnutls_x509_crt_get_dn3 540
gnutls_x509_crt_get_expiration_time 542
gnutls_x509_crt_get_extension_by_oid 542
gnutls_x509_crt_get_extension_by_oid2 542
gnutls_x509_crt_get_extension_data 543
gnutls_x509_crt_get_extension_data2 543
gnutls_x509_crt_get_extension_info 543
gnutls_x509_crt_get_extension_oid 544
gnutls_x509_crt_get_fingerprint 544
gnutls_x509_crt_get_inhibit_anypolicy 544
gnutls_x509_crt_get_issuer 545
gnutls_x509_crt_get_issuer_alt_name 545
gnutls_x509_crt_get_issuer_alt_name2 546
gnutls_x509_crt_get_issuer_
alt_othername_oid 546
gnutls_x509_crt_get_issuer_dn 547
gnutls_x509_crt_get_issuer_dn_by_oid 548
gnutls_x509_crt_get_issuer_dn_oid 549
gnutls_x509_crt_get_issuer_dn2 547
gnutls_x509_crt_get_issuer_dn3 548
gnutls_x509_crt_get_issuer_unique_id 549
gnutls_x509_crt_get_key_id 29, 549
gnutls_x509_crt_get_key_purpose_oid 550
gnutls_x509_crt_get_key_usage 550
gnutls_x509_crt_get_name_constraints 551
gnutls_x509_crt_get_pk_algorithm 551
gnutls_x509_crt_get_pk_dsa_raw 551
gnutls_x509_crt_get_pk_ecc_raw 552
gnutls_x509_crt_get_pk_gost_raw 552
gnutls_x509_crt_get_pk_oid 553
gnutls_x509_crt_get_pk_rsa_raw 553
gnutls_x509_crt_get_policy 553
gnutls_x509_crt_get_preferred_
hash_algorithm 739
gnutls_x509_crt_get_private_
key_usage_period 554
gnutls_x509_crt_get_proxy 554
gnutls_x509_crt_get_raw_dn 554
gnutls_x509_crt_get_raw_issuer_dn 555
gnutls_x509_crt_get_serial 555
gnutls_x509_crt_get_signature 555
gnutls_x509_crt_get_
signature_algorithm 555
gnutls_x509_crt_get_signature_oid 556
gnutls_x509_crt_get_spki 556
gnutls_x509_crt_get_subject 556
gnutls_x509_crt_get_subject_alt_name 556
gnutls_x509_crt_get_subject_alt_name2 557
gnutls_x509_crt_get_subject_
alt_othername_oid 557
gnutls_x509_crt_get_subject_key_id 558
gnutls_x509_crt_get_subject_unique_id 558
gnutls_x509_crt_get_tlsfeatures 559
gnutls_x509_crt_get_version 559
gnutls_x509_crt_import 559
gnutls_x509_crt_import_pkcs11 669
gnutls_x509_crt_import_url 560
gnutls_x509_crt_init 560
gnutls_x509_crt_list_import 560
gnutls_x509_crt_list_import_pkcs11 669
gnutls_x509_crt_list_import_url 561
gnutls_x509_crt_list_import2 561
gnutls_x509_crt_list_verify 562
gnutls_x509_crt_print 562
gnutls_x509_crt_privkey_sign 714
gnutls_x509_crt_set_activation_time 563

gnutls_x509_crt_set_
authority_info_access 563
gnutls_x509_crt_set_authority_key_id 563
gnutls_x509_crt_set_basic_constraints 563
gnutls_x509_crt_set_ca_status 564
gnutls_x509_crt_set_crl_dist_points 564
gnutls_x509_crt_set_crl_dist_points2 564
gnutls_x509_crt_set_crq 565
gnutls_x509_crt_set_crq_
extension_by_oid 565
gnutls_x509_crt_set_crq_extensions 565
gnutls_x509_crt_set_dn 565
gnutls_x509_crt_set_dn_by_oid 566
gnutls_x509_crt_set_expiration_time 566
gnutls_x509_crt_set_extension_by_oid 566
gnutls_x509_crt_set_flags 567
gnutls_x509_crt_set_inhibit_anypolicy 567
gnutls_x509_crt_set_issuer_alt_name 567
gnutls_x509_crt_set_issuer_
alt_othername 568
gnutls_x509_crt_set_issuer_dn 568
gnutls_x509_crt_set_issuer_dn_by_oid 568
gnutls_x509_crt_set_issuer_unique_id 569
gnutls_x509_crt_set_key 569
gnutls_x509_crt_set_key_purpose_oid 569
gnutls_x509_crt_set_key_usage 570
gnutls_x509_crt_set_name_constraints 570
gnutls_x509_crt_set_pin_function 570
gnutls_x509_crt_set_policy 571
gnutls_x509_crt_set_private_
key_usage_period 571
gnutls_x509_crt_set_proxy 571
gnutls_x509_crt_set_proxy_dn 572
gnutls_x509_crt_set_pubkey 107, 715
gnutls_x509_crt_set_serial 572
gnutls_x509_crt_set_spki 572
gnutls_x509_crt_set_subject_alt_name 573
gnutls_x509_crt_set_subject_
alt_othername 573
gnutls_x509_crt_set_subject_
alternative_name 574
gnutls_x509_crt_set_subject_key_id 574
gnutls_x509_crt_set_subject_unique_id 574
gnutls_x509_crt_set_tlsfeatures 575
gnutls_x509_crt_set_version 575
gnutls_x509_crt_sign 575
gnutls_x509_crt_sign2 575
gnutls_x509_crt_verify 576
gnutls_x509_crt_verify_data2 576
gnutls_x509_dn_deinit 577
gnutls_x509_dn_export 577
gnutls_x509_dn_export2 577
gnutls_x509_dn_get_rdn_ava 25, 578
gnutls_x509_dn_get_str 578
gnutls_x509_dn_get_str2 578
gnutls_x509_dn_import 579
gnutls_x509_dn_init 579
gnutls_x509_dn_oid_known 579
gnutls_x509_dn_oid_name 579
gnutls_x509_dn_set_str 580
gnutls_x509_ext_deinit 580
gnutls_x509_ext_export_aia 580
gnutls_x509_ext_export_
authority_key_id 580
gnutls_x509_ext_export_
basic_constraints 581
gnutls_x509_ext_export_crl_dist_points ... 581
gnutls_x509_ext_export_
inhibit_anypolicy 581
gnutls_x509_ext_export_key_purposes 582
gnutls_x509_ext_export_key_usage 582
gnutls_x509_ext_export_
name_constraints 582
gnutls_x509_ext_export_policies 582
gnutls_x509_ext_export_private_
key_usage_period 583
gnutls_x509_ext_export_proxy 583
gnutls_x509_ext_export_
subject_alt_names 583
gnutls_x509_ext_export_subject_key_id 584
gnutls_x509_ext_export_tlsfeatures 584
gnutls_x509_ext_import_aia 584
gnutls_x509_ext_import_
authority_key_id 585
gnutls_x509_ext_import_
basic_constraints 585
gnutls_x509_ext_import_crl_dist_points ... 585
gnutls_x509_ext_import_
inhibit_anypolicy 585
gnutls_x509_ext_import_key_purposes 586
gnutls_x509_ext_import_key_usage 586
gnutls_x509_ext_import_
name_constraints 587
gnutls_x509_ext_import_policies 587
gnutls_x509_ext_import_private_
key_usage_period 587
gnutls_x509_ext_import_proxy 588
gnutls_x509_ext_import_
subject_alt_names 588
gnutls_x509_ext_import_subject_key_id 588
gnutls_x509_ext_import_tlsfeatures 589
gnutls_x509_ext_print 589
gnutls_x509_key_purpose_deinit 589
gnutls_x509_key_purpose_get 590
gnutls_x509_key_purpose_init 590
gnutls_x509_key_purpose_set 590
gnutls_x509_name_
constraints_add_excluded 590
gnutls_x509_name_constraints_
add_permitted 591
gnutls_x509_name_constraints_check 591
gnutls_x509_name_constraints_check_crt ... 591
gnutls_x509_name_constraints_deinit 592
gnutls_x509_name_
constraints_get_excluded 592

gnutls_x509_name_constraints_
get_permitted 592
gnutls_x509_name_constraints_init 593
gnutls_x509_othername_to_virtual 593
gnutls_x509_policies_deinit 593
gnutls_x509_policies_get 594
gnutls_x509_policies_init 594
gnutls_x509_policies_set 594
gnutls_x509_policy_release 594
gnutls_x509_privkey_cpy 595
gnutls_x509_privkey_deinit 595
gnutls_x509_privkey_export 595
gnutls_x509_privkey_export_dsa_raw 596
gnutls_x509_privkey_export_ecc_raw 597
gnutls_x509_privkey_export_gost_raw 597
gnutls_x509_privkey_export_pkcs8 598
gnutls_x509_privkey_export_rsa_raw 598
gnutls_x509_privkey_export_rsa_raw2 599
gnutls_x509_privkey_export2 595
gnutls_x509_privkey_export2_pkcs8 596
gnutls_x509_privkey_fix 599
gnutls_x509_privkey_generate 599
gnutls_x509_privkey_generate2 600
gnutls_x509_privkey_get_key_id 601
gnutls_x509_privkey_get_pk_algorithm 601
gnutls_x509_privkey_get_pk_algorithm2 601
gnutls_x509_privkey_get_seed 601
gnutls_x509_privkey_get_spki 602
gnutls_x509_privkey_import 602
gnutls_x509_privkey_import_dsa_raw 603
gnutls_x509_privkey_import_ecc_raw 603
gnutls_x509_privkey_import_gost_raw 604
gnutls_x509_privkey_import_openssl 61, 604
gnutls_x509_privkey_import_pkcs8 605
gnutls_x509_privkey_import_rsa_raw 605
gnutls_x509_privkey_import_rsa_raw2 606
gnutls_x509_privkey_import2 ..... 57, 602
gnutls_x509_privkey_init 606
gnutls_x509_privkey_sec_param 606
gnutls_x509_privkey_set_flags 606
gnutls_x509_privkey_set_pin_function 607
gnutls_x509_privkey_set_spki 607
gnutls_x509_privkey_sign_data 607
gnutls_x509_privkey_sign_hash 739
gnutls_x509_privkey_verify_params 608
gnutls_x509_privkey_verify_seed 608
gnutls_x509_rdn_get 608
gnutls_x509_rdn_get_by_oid 609
gnutls_x509_rdn_get_oid 609
gnutls_x509_rdn_get2 609
gnutls_x509_spki_deinit 610
gnutls_x509_spki_get_rsa_pss_params 610
gnutls_x509_spki_init 610
gnutls_x509_spki_set_rsa_pss_params 610
gnutls_x509_tlsfeatures_add 611
gnutls_x509_tlsfeatures_check_crt 611
gnutls_x509_tlsfeatures_deinit 611
gnutls_x509_tlsfeatures_get 611
gnutls_x509_tlsfeatures_init 612
gnutls_x509_trust_list_add_cas 30, 612
gnutls_x509_trust_list_add_crls 31, 612
gnutls_x509_trust_list_
add_named_crt 30, 613
gnutls_x509_trust_list_add_
system_trust 34, 613
gnutls_x509_trust_list_add_trust_dir 614
gnutls_x509_trust_list_
add_trust_file 33, 614
gnutls_x509_trust_list_
add_trust_mem 34, 614
gnutls_x509_trust_list_deinit 615
gnutls_x509_trust_list_get_issuer 615
gnutls_x509_trust_list_get_
issuer_by_dn 616
gnutls_x509_trust_list_get_issuer_
by_subject_key_id 616
gnutls_x509_trust_list_init 616
gnutls_x509_trust_list_iter_deinit 617
gnutls_x509_trust_list_iter_get_ca 617
gnutls_x509_trust_list_remove_cas 617
gnutls_x509_trust_list_
remove_trust_file 618
gnutls_x509_trust_list_
remove_trust_mem 618
gnutls_x509_trust_list_verify_crt 31, 618
gnutls_x509_trust_list_verify_crt2 ..... 32, 619
gnutls_x509_trust_list_
verify_named_crt 33, 620

Concept Index

A

abstract types 100
alert protocol 9
ALPN 16
anonymous authentication 97
API reference 360
Application Layer Protocol Negotiation 16
Application-specific keys 108
authentication methods 20

B

bad_record_mac 8

C

callback functions 142
certificate authentication 20, 43
certificate requests 43
certificate revocation lists 46
certificate status 49, 54
Certificate status request 14
Certificate verification 40
certification 340
`certtool` 61
certtool help 61
channel bindings 194
ciphersuites 350
client certificate authentication 11
CMS 288
compression algorithms 8
contributing 339
credentials 181
CRL 46
cryptographic message syntax 288

D

`danetool` 87
danetool help 87
DANE 41, 186
deriving keys 193
digital signatures 41
DNSSEC 41, 186
download 2

E

Encrypted keys 56
error codes 341
example programs 196
examples 196
exporting keying material 193

F

False Start 16
FDL, GNU Free Documentation License 741
file signing 288
fork 142

G

generating parameters 192
giovec_t 138
`gnutls-cli` 297
gnutls-cli help 297
`gnutls-cli-debug` 314
gnutls-cli-debug help 314
`gnutls-serv` 306
gnutls-serv help 307
gnutls_datum_t 138

H

hacking 339
handshake protocol 10
hardware security modules 110
hardware tokens 110
hash functions 290
heartbeat 12
HMAC functions 290

I

installation 2, 3
internal architecture 317
isolated mode 141

K

key extraction 193
Key pinning 40, 186
key sizes 179
keying material exporters 193

M

MAC functions 290
maximum fragment length 12

O

OCSP 49
OCSP Functions 629
OCSP stapling 54
OCSP status request 14
`ocsptool` 82
ocsptool help 82
Online Certificate Status Protocol 49, 54
OpenPGP certificates 39
OpenSSL 195
OpenSSL encrypted keys 60
overriding algorithms 294

P

`p11tool` 119
p11tool help 119
parameter generation 192
PCT 19
PKCS #10 43
PKCS #11 tokens 110
PKCS #12 59
PKCS #7 288
PKCS #8 57
post-handshake authentication 191
Priority strings 172
PSK authentication 92
`psktool` 92
psktool help 92
public key algorithms 286, 288

R

random numbers 294
Raw public-keys 40
re-authentication 190, 191
re-key 191
re-negotiation 190, 191
record padding 8
record protocol 6
renegotiation 13
reporting bugs 339
resuming sessions 11, 183

S

safe renegotiation 13
seccomp 141
Secure RTP 15
server name indication 12
session resumption 11, 183
session tickets 12
Smart card example 202
smart cards 110
SRP authentication 94
`srptool` 95
srptool help 95
SRTP 15
SSH-style authentication 40, 186
SSL 2 19
Supplemental data 17
symmetric algorithms 283
symmetric cryptography 283
symmetric encryption algorithms 6
System-specific keys 108
System-wide configuration 280

T

thread safety 141
tickets 12
TLS extensions 12
TLS False Start 16
TLS layers 5
`tpmtool` 133
tpmtool help 133
TPM 130
transport layer 6
transport protocol 6
Trust on first use 40, 186
trusted platform module 130

U

upgrading 332

V

verifying certificate paths 30, 36, 40, 41
verifying certificate with pkcs11 38
virtual hosts 181

X

X.509 certificate name 24
X.509 certificates 21
X.509 distinguished name 24
X.509 extensions 26
X.509 Functions 493