This document includes text contributed by Nikos Mavrogiannopoulos, Simon Josefsson, Daiki Ueno, Carolin Latze, Alfredo Pironti, Ted Zlatanov and Andrew McDonald. Several corrections are due to Patrick Pelletier and Andreas Metzler.

ISBN 978-1-326-00266-4

Contents

List of Tables

List of Figures

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 [38], to public key infrastructure in [14] and to security engineering in [5].

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

1

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 2, 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.

1.1. 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.

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`.

1.2. 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.

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

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].

1.3. Overview

In this document we present an overview of the supported security protocols in chapter 2, and continue by providing more information on the certificate authentication in section 3.1, and shared-key as well anonymous authentication in section 3.3. We elaborate on certificate authentication by demonstrating advanced usage of the API in section 3.2. The core of the TLS library is presented in chapter 5 and example applications are listed in chapter 6. In chapter 8 the usage of few included programs that may assist debugging is presented. The last chapter is chapter 9 that provides a short introduction to GnuTLS' internal architecture.

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

2

Introduction to TLS and DTLS

TLS stands for "Transport Layer Security" and is the successor of SSL, the Secure Sockets Layer protocol [12] designed by Netscape. TLS is an Internet protocol, defined by IETF[1], described in [9]. 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" [31] 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.

2.1. 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. section 2.4, 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. section 2.5, for more information about the handshake protocol. The protocol layering in TLS is shown in Figure 2.1.

2.2. 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

[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.

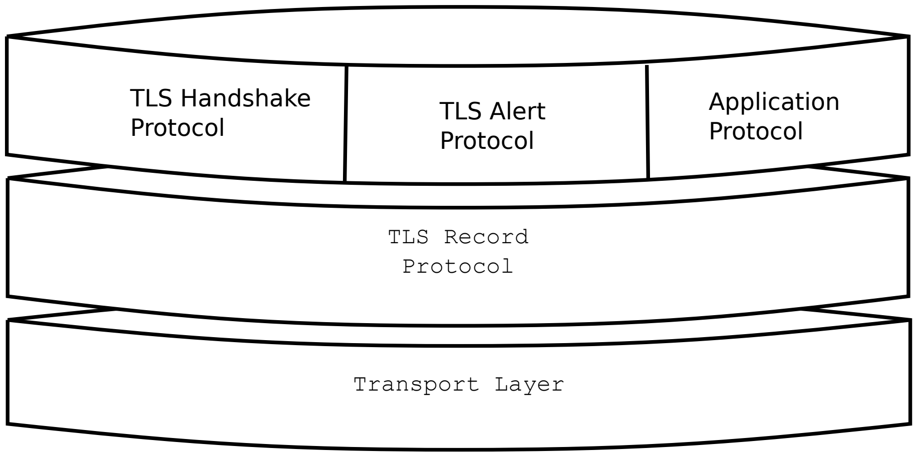

Figure 2.1.: The TLS protocol layers.

transport layer can be used by providing callbacks for GnuTLS to access the transport layer (for details see section 5.5).

2.3. 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.

2.3.1. 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 2.1 and Table 2.2.

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.
GOST28147-TC26Z-CNT	Legacy (stream)	TLS 1.2	This is a 64-bit block cipher GOST 28147-89 with TC26Z S-Box working in CNT mode. It is one of the approved ciphers in Russia. It is not enabled by default.
NULL	Legacy (stream)	TLS 1.0, TLS 1.1, TLS 1.2	NULL is the empty/identity cipher which doesn't encrypt any data. It can be combined with data authentication under TLS 1.2 or earlier, but is only used transiently under TLS 1.3 until encryption starts. This cipher cannot be negotiated by default (need to be explicitly enabled) under TLS 1.2, and cannot be negotiated at all under TLS 1.3. When

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 2.2.: Supported MAC algorithms in TLS.

2.3.2. 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.

2.3.3. 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`, and is made available when under TLS1.3; alternatively `gnutls_record_can_use_length_hiding` 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` function must be called with the flag `GNUTLS_SAFE_PADDING_CHECK`.

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.

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

Description: 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.

unsigned **gnutls_record_can_use_length_hiding** (*gnutls_session_t* **session**)

ssize_t **gnutls_record_send_range** (*gnutls_session_t* **session**, *const void ** **data**, *size_t* **data_size**, *const gnutls_range_st ** **range**)

Note: *This function currently is limited to blocking sockets.*

2.4. 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

Table 2.3.: The TLS alert table

2.5. 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`. In the next paragraphs we elaborate on the handshake protocol, i.e., the ciphersuite negotiation.

2.5.1. 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 Appendix C.

2.5.2. 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.

2.5.3. 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` function. We elaborate in subsection 5.4.1.

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

2.5.4. 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 – subsection 2.6.3).

Session resumption is an integral part of GnuTLS, and subsection 5.12.2, subsection 6.1.4 illustrate typical uses of it.

2.6. TLS extensions

A number of extensions to the TLS protocol have been proposed mainly in [6]. The extensions supported in GnuTLS are discussed in the subsections that follow.

2.6.1. 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** (*gnutls_session_t* **session**)

ssize_t **gnutls_record_set_max_size** (*gnutls_session_t* **session**, *size_t* **size**)

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.`

2.6.2. 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` and `gnutls_server_name_get` can be used to enable this extension, or to retrieve the name sent by a client.

int **gnutls_server_name_set** (*gnutls_session_t* **session**, *gnutls_server_name_type_t* **type**, *const void ** **name**, *size_t* **name_length**)

int **gnutls_server_name_get** (*gnutls_session_t* **session**, *void ** **data**, *size_t ** **data_length**, *unsigned int ** **type**, *unsigned int* **indx**)

2.6.3. 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 [36].

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.

2.6.4. HeartBeat

This is a TLS extension that allows to ping and receive confirmation from the peer, and is described in [29]. The extension is disabled by default and `gnutls_heartbeat_enable` 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`. 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`.

unsigned **gnutls_heartbeat_allowed** (*gnutls_session_t* **session**, *unsigned int* **type**)

void **gnutls_heartbeat_enable** (*gnutls_session_t* **session**, *unsigned int* **type**) *int* **gnutls_heartbeat_ping** (*gnutls_session_t* **session**, *size_t* **data_size**, *unsigned int* **max_tries**, *unsigned int* **flags**)

int **gnutls_heartbeat_pong** (*gnutls_session_t* **session**, *unsigned int* **flags**)

void **gnutls_heartbeat_set_timeouts** (*gnutls_session_t* **session**, *unsigned int* **retrans_timeout**, *unsigned int* **total_timeout**)

unsigned int **gnutls_heartbeat_get_timeout** (*gnutls_session_t* **session**)

2.6.5. 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 renegotiated 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 [32].

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 5.10). 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` function is used to check if the extension has been negotiated on a session, and can be used both by clients and servers.

2.6.6. 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 subsection 3.2.7) 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 subsection 3.2.4 for further information.

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

2.6.7. SRTP

The TLS protocol was extended in [25] 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.

enum **gnutls_srtp_profile_t**:

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`
GNUTLS_SRTP_AEAD_AES_128_GCM	`128 bit AES with GCM`
GNUTLS_SRTP_AEAD_AES_256_GCM	`256 bit AES with GCM`

Table 2.4.: Supported SRTP profiles

To enable use the following functions.

int **gnutls_srtp_set_profile** (*gnutls_session_t* **session**, *gnutls_srtp_profile_t* **profile**)

int **gnutls_srtp_set_profile_direct** (*gnutls_session_t* **session**, *const char ** **profiles**, *const char *** **err_pos**)

To obtain the negotiated keys use the function below.

Other helper functions are listed below.

int **gnutls_srtp_get_selected_profile** (*gnutls_session_t* **session**, *gnutls_srtp_profile_t ** **profile**)

*const char ** **gnutls_srtp_get_profile_name** (*gnutls_srtp_profile_t* **profile**)

int **gnutls_srtp_get_profile_id** (*const char ** **name**, *gnutls_srtp_profile_t ** **profile**) *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**)

Description: 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

2.6.8. False Start

The TLS protocol was extended in [21] 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` to request an early return of the `gnutls_handshake` 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`, and call `gnutls_record_recv` 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`.

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 enabled 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` and check for the GNUTLS_SFLAGS_FALSE_START flag. For GnuTLS the false start is enabled for the following key exchange methods (see [21] for rationale)

- DHE
- ECDHE

but only when the negotiated parameters exceed `GNUTLS_SEC_PARAM_HIGH` –see Table 5.8, and when under (D)TLS 1.2 or later.

2.6.9. 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** (*gnutls_session_t* **session**, *const gnutls_datum_t ** **protocols**, *unsigned* **protocols_size**, *unsigned int* **flags**)

int **gnutls_alpn_get_selected_protocol** (*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.

2.6.10. Extensions and Supplemental Data

It is possible to transfer supplemental data during the TLS handshake, following [37]. 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 9.4.

2.7. 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.

2.7.1. 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 [30].

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.

[4] See also the Server Name Indication extension on subsection 2.6.2.

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.

2.7.2. 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" 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 [18].

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".

[5]See LDAP, IMAP etc.

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!

2.8. 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.

3 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.

3.1. Certificate authentication

The most known authentication method of TLS are certificates. The PKIX [16] 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 [16].

The key exchange algorithms supported by certificate authentication are shown in Table 3.1.

3.1.1. 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.

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 3.1.

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 [16] as shown in Table 3.2.

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

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 3.1.: Supported key exchange algorithms.

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 3.2.: X.509 certificate fields.

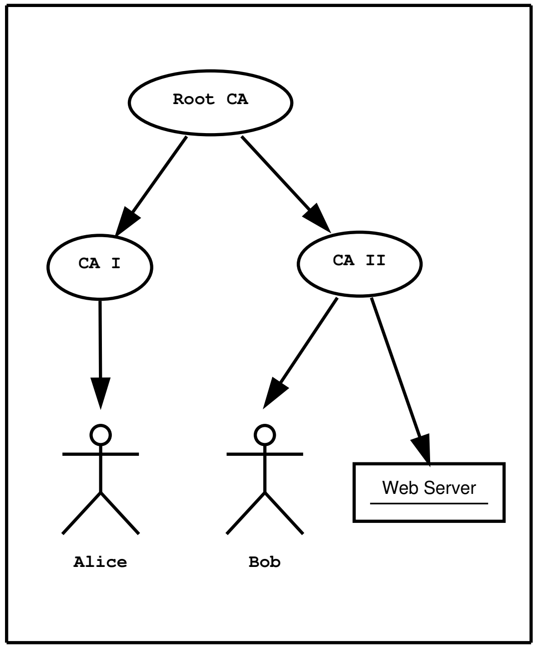

Figure 3.1.: An example of the X.509 hierarchical trust model.

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 subsection 6.5.2.

Importing an X.509 certificate

The certificate structure should be initialized using `gnutls_x509_crt_init`, and a certificate structure can be imported using `gnutls_x509_crt_import`.

int **gnutls_x509_crt_init** (*gnutls_x509_crt_t * * **cert**)

int **gnutls_x509_crt_import** (*gnutls_x509_crt_t* **cert**, *const gnutls_datum_t * * **data**, *gnutls_x509_crt_fmt_t* **format**)

void **gnutls_x509_crt_deinit** (*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** (*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** (*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`. Note that although the functions above apply to `gnutls_x509_crt_t` structure, similar functions exist for the CRL structure `gnutls_x509_crl_t`.

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 subsubsection 3.1.1) 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** (*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** (*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** (*gnutls_subject_alt_names_t* * **sans**)

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**)

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**)

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 [34]), because only a single address can be specified, and the CN field is free-form making matching ambiguous.

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` which follows the rules in [46] and returns a single string. Access to each string by individual object identifiers can be accessed using `gnutls_x509_crt_get_dn_by_oid`.

int **gnutls_x509_crt_get_dn2** (*gnutls_x509_crt_t* **cert**, *gnutls_datum_t* * **dn**)

Description: 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.

int **gnutls_x509_crt_get_dn** (*gnutls_x509_crt_t* **cert**, *char* * **buf**, *size_t* * **buf_size**)

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**)

int **gnutls_x509_crt_get_dn_oid** (*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** (*gnutls_x509_crt_t* **cert**, *char* * **buf**, *size_t* * **buf_size**)

int **gnutls_x509_crt_get_issuer_dn2** (*gnutls_x509_crt_t* **cert**, *gnutls_datum_t* * **dn**)

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**)

int **gnutls_x509_crt_get_issuer_dn_oid** (*gnutls_x509_crt_t* **cert**, *unsigned* **indx**, *void* * **oid**, *size_t* * **oid_size**)

int **gnutls_x509_crt_get_issuer** (*gnutls_x509_crt_t* **cert**, *gnutls_x509_dn_t* * **dn**)

The more powerful `gnutls_x509_crt_get_subject` and `gnutls_x509_dn_get_rdn_ava` provide efficient but low-level access to the contents of the distinguished name structure.

int **gnutls_x509_crt_get_subject** (*gnutls_x509_crt_t* **cert**, *gnutls_x509_dn_t* * **dn**)

int **gnutls_x509_crt_get_issuer** (*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**)

Description: 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.

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 3.3.

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** (*gnutls_x509_crt_t* **cert**, *unsigned* **indx**, *void ** **oid**, *size_t ** **oid_size**, *unsigned int ** **critical**)

int **gnutls_x509_crt_get_extension_data2** (*gnutls_x509_crt_t* **cert**, *unsigned* **indx**, *gnutls_datum_t ** **data**)

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**)

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** (*const gnutls_datum_t ** **ext**, *unsigned int ** **ca**, *int ** **pathlen**)

int **gnutls_x509_ext_export_basic_constraints** (*unsigned int* **ca**, *int* **pathlen**, *gnutls_datum_t ** **ext**)

int **gnutls_x509_ext_import_key_usage** (*const gnutls_datum_t ** **ext**, *unsigned int ** **key_usage**)

int **gnutls_x509_ext_export_key_usage** (*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** (*const gnutls_datum_t * * **ext**, *gnutls_x509_name_constraints_t* **nc**, *unsigned int* **flags**)

int **gnutls_x509_ext_export_name_constraints** (*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** (*gnutls_x509_name_constraints_t* **nc**, *unsigned* **idx**, *unsigned * * **type**, *gnutls_datum_t * * **name**)

int **gnutls_x509_name_constraints_get_excluded** (*gnutls_x509_name_constraints_t* **nc**, *unsigned* **idx**, *unsigned * * **type**, *gnutls_datum_t * * **name**)

int **gnutls_x509_name_constraints_add_permitted** (*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** (*gnutls_x509_name_constraints_t* **nc**, *gnutls_x509_subject_alt_name_t* **type**, *const gnutls_datum_t * * **name**)

unsigned **gnutls_x509_name_constraints_check** (*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** (*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** (*gnutls_x509_name_constraints_t * * **nc**)

void **gnutls_x509_name_constraints_deinit** (*gnutls_x509_name_constraints_t* **nc**)

Similar functions exist for all of the other supported extensions, listed in Table 3.3.

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.

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.
Proxy Certification Information	1.3.6.1.5.5.7.1.14	Proxy Certificates includes this extension that contains the OID of the proxy policy language used, and can specify limits on the maximum lengths of proxy chains. Proxy Certificates are specified in [40].

Table 3.3.: Supported X.509 certificate extensions.

int **gnutls_x509_crt_get_basic_constraints** (*gnutls_x509_crt_t* **cert**, *unsigned int ** **critical**, *unsigned int ** **ca**, *int ** **pathlen**)

int **gnutls_x509_crt_set_basic_constraints** (*gnutls_x509_crt_t* **crt**, *unsigned int* **ca**, *int* **pathLenConstraint**)

int **gnutls_x509_crt_get_key_usage** (*gnutls_x509_crt_t* **cert**, *unsigned int ** **key_usage**, *unsigned int ** **critical**)

int **gnutls_x509_crt_set_key_usage** (*gnutls_x509_crt_t* **crt**, *unsigned int* **usage**)

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` 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 subsection 4.1.1 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**)

Description: `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** (*gnutls_x509_privkey_t* **key**, *unsigned int* * **bits**)

int **gnutls_x509_privkey_export_rsa_raw2** (*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** (*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** (*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** (*gnutls_x509_privkey_t* **key**, *unsigned int* **flags**, *unsigned char* * **output_data**, *size_t* * **output_data_size**)

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**)

Description: 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.

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 Table 3.4. The `GNUTLS_CERT_INVALID` flag is always set on a verification error and more detailed flags will also be set when appropriate.

An example of certificate verification is shown in subsection 6.3.4. It is also possible to have a set of certificates that are trusted for a particular server but not to authorize other certificates.

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**)

Description: 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.

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**)

Description: 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.

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**)

Description: 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.

int **gnutls_x509_trust_list_verify_crt2** (*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**)

Description: 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.

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**)

Description: 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.

int **gnutls_x509_trust_list_add_trust_file** (*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**)

Description: 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.

int **gnutls_x509_trust_list_add_trust_mem** (*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**)

Description: 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.

This purpose is served by the functions `gnutls_x509_trust_list_add_named_crt` and `gnutls_x509_trust_list_verify_named_crt`.

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_x509_trust_list_add_system_trust** (*gnutls_x509_trust_list_t* **list**, *unsigned int* **tl_flags**, *unsigned int* **tl_vflags**)

Description: 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.

int **gnutls_certificate_set_x509_trust_file** (*gnutls_certificate_credentials_t* **cred**, *const char ** **cafile**, *gnutls_x509_crt_fmt_t* **type**)

int **gnutls_certificate_set_x509_trust_dir** (*gnutls_certificate_credentials_t* **cred**, *const char ** **ca_dir**, *gnutls_x509_crt_fmt_t* **type**)

int **gnutls_certificate_set_x509_crl_file** (*gnutls_certificate_credentials_t* **res**, *const char ** **crlfile**, *gnutls_x509_crt_fmt_t* **type**)

int **gnutls_certificate_set_x509_system_trust** (*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`).

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`, but for older GnuTLS versions you may use an explicit callback set via `gnutls_certificate_set_verify_function` and then utilize `gnutls_certificate_verify_peers3` 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` and `gnutls_certificate_verify_peers` 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` the flags are passed directly, but for `gnutls_certificate_verify_peers3`, the flags are set using `gnutls_certificate_set_verify_flags`. All the available flags are part of the enumeration `gnutls_certificate_verify_flags` shown in Table 3.5.

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_`

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

x509_trust_file), 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 3.6. Note that typically according to [8] the extended key usage object identifiers apply to end certificates. Their application to CA certificates is an extension used by the trust modules.

With such modules, it is recommended to use the verification functions gnutls_x509_trust_list_verify_crt2, or gnutls_certificate_verify_peers, 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.

3.1.2. 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.

3.1.3. 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.

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 4.1.

3.1.4. 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`), 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.

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 subsection 6.1.5.

See subsection 5.12.3 on how to use the available functionality.

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 subsection 5.12.3 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.

3.1.5. 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 *textasciicircum*(hash size)/2 operations. Today colliding certificates using the MD5 hash algorithm have been generated as shown in [23].

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 *textasciicircum*64, but it has been showed to have a collision strength well under 2 *textasciicircum*50. As of November 2005, it is believed that SHA-1's collision strength is around 2 *textasciicircum*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 *textasciicircum*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.

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 subsubsection 3.1.1), 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`** to verify the certificate chain, you can call **`gnutls_certificate_set_verify_flags`** 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`** or **`gnutls_x509_crt_list_verify`**, 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`** to extract the raw server's certificate chain, **`gnutls_x509_crt_list_import`** to parse each of the certificates, and then **`gnutls_x509_crt_get_signature_algorithm`** 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.

3.2. 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.

3.2.1. 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 [27]. 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** (*gnutls_x509_crq_t* **crq**, *unsigned int* **version**)

int **gnutls_x509_crq_set_dn** (*gnutls_x509_crq_t* **crq**, *const char ** **dn**, *const char *** **err**)

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**)

int **gnutls_x509_crq_set_key_usage** (*gnutls_x509_crq_t* **crq**, *unsigned int* **usage**)

int **gnutls_x509_crq_set_key_purpose_oid** (*gnutls_x509_crq_t* **crq**, *const void ** **oid**, *unsigned int* **critical**)

int **gnutls_x509_crq_set_basic_constraints** (*gnutls_x509_crq_t* **crq**, *unsigned int* **ca**, *int* **pathLenConstraint**)

The `gnutls_x509_crq_set_key` and `gnutls_x509_crq_sign2` 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 4.1.

int **gnutls_x509_crq_set_key** (*gnutls_x509_crq_t* **crq**, *gnutls_x509_privkey_t* **key**)

Description: 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.

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>
```

int **gnutls_x509_crq_sign2** (*gnutls_x509_crq_t* **crq**, *gnutls_x509_privkey_t* **key**, *gnutls_digest_algorithm_t* **dig**, *unsigned int* **flags**)

Description: 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()).

```
#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;
}
```

3.2.2. 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 3.7. 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.

The basic CRL structure functions follow.

int **gnutls_x509_crl_init** (*gnutls_x509_crl_t* * **crl**)

int **gnutls_x509_crl_import** (*gnutls_x509_crl_t* **crl**, *const gnutls_datum_t* * **data**, *gnutls_x509_crt_fmt_t* **format**)

int **gnutls_x509_crl_export** (*gnutls_x509_crl_t* **crl**, *gnutls_x509_crt_fmt_t* **format**, *void* * **output_data**, *size_t* * **output_data_size**)

int **gnutls_x509_crl_export** (*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`. 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**)

Description: 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** (*gnutls_x509_crl_t* **crl**)

int **gnutls_x509_crl_get_issuer_dn** (*gnutls_x509_crl_t* **crl**, *char* * **buf**, *size_t* * **sizeof_buf**)

int **gnutls_x509_crl_get_issuer_dn2** (*gnutls_x509_crl_t* **crl**, *gnutls_datum_t* * **dn**)

time_t **gnutls_x509_crl_get_this_update** (*gnutls_x509_crl_t* **crl**)

time_t **gnutls_x509_crl_get_next_update** (*gnutls_x509_crl_t* **crl**)

int **gnutls_x509_crl_get_crt_count** (*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** (*gnutls_x509_crl_t* **crl**, *unsigned int* **version**)

int **gnutls_x509_crl_set_crt_serial** (*gnutls_x509_crl_t* **crl**, *const void* * **serial**, *size_t* **serial_size**, *time_t* **revocation_time**)

int **gnutls_x509_crl_set_crt** (*gnutls_x509_crl_t* **crl**, *gnutls_x509_crt_t* **crt**, *time_t* **revocation_time**)

int **gnutls_x509_crl_set_next_update** (*gnutls_x509_crl_t* **crl**, *time_t* **exp_time**)

int **gnutls_x509_crl_set_this_update** (*gnutls_x509_crl_t* **crl**, *time_t* **act_time**)

The `gnutls_x509_crl_sign2` and `gnutls_x509_crl_privkey_sign` 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.

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

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**)

Description: 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**)

Description: 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

int **gnutls_x509_crl_set_number** (*gnutls_x509_crl_t* **crl**, *const void ** **nr**, *size_t* **nr_size**)

int **gnutls_x509_crl_set_authority_key_id** (*gnutls_x509_crl_t* **crl**, *const void ** **id**, *size_t* **id_size**)

3.2.3. 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 [33].

The Online Certificate Status Protocol, or OCSP [26], 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 subsection 3.2.4).

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** (*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** (*gnutls_ocsp_req_t* * **req**)

void **gnutls_ocsp_req_deinit** (*gnutls_ocsp_req_t* **req**)

int **gnutls_ocsp_req_import** (*gnutls_ocsp_req_t* **req**, *const gnutls_datum_t* * **data**)

int **gnutls_ocsp_req_export** (*gnutls_ocsp_req_const_t* **req**, *gnutls_datum_t* * **data**)

int **gnutls_ocsp_req_print** (*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** (*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** (*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** (*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** (*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** (*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** (*gnutls_ocsp_req_const_t* **req**, *unsigned int ** **critical**, *gnutls_datum_t ** **nonce**)

int **gnutls_ocsp_req_set_nonce** (*gnutls_ocsp_req_t* **req**, *unsigned int* **critical**, *const gnutls_datum_t ** **nonce**)

int **gnutls_ocsp_req_randomize_nonce** (*gnutls_ocsp_req_t* **req**)

The OCSP response structures is a complex structure. A simplified overview of it is in Table 3.8. Note that a response may contain information on multiple certificates.

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

int **gnutls_ocsp_resp_init** (*gnutls_ocsp_resp_t ** **resp**)

void **gnutls_ocsp_resp_deinit** (*gnutls_ocsp_resp_t* **resp**)

int **gnutls_ocsp_resp_import** (*gnutls_ocsp_resp_t* **resp**, *const gnutls_datum_t ** **data**)

int **gnutls_ocsp_resp_export** (*gnutls_ocsp_resp_const_t* **resp**, *gnutls_datum_t ** **data**)

int **gnutls_ocsp_resp_print** (*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**)

Description: 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.

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** (*gnutls_ocsp_resp_const_t* **resp**, *gnutls_x509_trust_list_t* **trustlist**, *unsigned int* * **verify**, *unsigned int* **flags**)

int **gnutls_ocsp_resp_verify_direct** (*gnutls_ocsp_resp_const_t* **resp**, *gnutls_x509_crt_t* **issuer**, *unsigned int* * **verify**, *unsigned int* **flags**)

int **gnutls_ocsp_resp_check_crt** (*gnutls_ocsp_resp_const_t* **resp**, *unsigned int* **indx**, *gnutls_x509_crt_t* **crt**)

3.2.4. 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`). To disable the automatic OCSP verification the flag `GNUTLS_VERIFY_DISABLE_CRL_CHECKS` should be specified to `gnutls_certificate_set_verify_flags`.

Since GnuTLS 3.5.1 the client certificate verification will consider the [15] 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** (*gnutls_session_t* **session**, *gnutls_datum_t * **response**)

int **gnutls_ocsp_status_request_get2** (*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** (*gnutls_certificate_credentials_t* **cred**, *gnutls_certificate_retrieve_function3 * **func**)

int **gnutls_certificate_set_ocsp_status_request_file2** (*gnutls_certificate_credentials_t* **sc**, *const char * **response_file**, *unsigned* **idx**, *gnutls_x509_crt_fmt_t* **fmt**)

unsigned **gnutls_ocsp_status_request_is_checked** (*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`, 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` function.

time_t **gnutls_certificate_get_ocsp_expiration** (*gnutls_certificate_credentials_t* **sc**, *unsigned* **idx**, *int* **oidx**, *unsigned* **flags**)

Description: 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.

Prior to GnuTLS 3.6.4, the functions `gnutls_certificate_set_ocsp_status_request_function2` `gnutls_certificate_set_ocsp_status_request_file` 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 subsection 3.2.7).

```
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.

3.2.5. 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 4.3) 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**)

Description: This function will import the given private key to the abstract *gnutls_privkey_t* type. The supported formats are basic unencrypted key, PKCS8, PKCS12, TSS2, and the openssl format.

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`, or alternatively they can be directly imported using `gnutls_certificate_set_x509_key_file2`.

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**)

Description: This function will import the given DER or PEM encoded key, to the native *gnutls_x*509_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.

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** (*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** (*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** (*gnutls_x509_privkey_t* **key**, *gnutls_x509_crt_fmt_t* **format**, *const char ** **password**, *unsigned int* **flags**, *gnutls_datum_t ** **out**)

PKCS #12 structures

A PKCS #12 structure [19] 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`.

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` is provided. For more advanced uses, manual parsing of the structure is required using the functions below.

int **gnutls_pkcs12_get_bag** (*gnutls_pkcs12_t* **pkcs12**, *int* **indx**, *gnutls_pkcs12_bag_t* **bag**)

int **gnutls_pkcs12_verify_mac** (*gnutls_pkcs12_t* **pkcs12**, *const char ** **pass**)

int **gnutls_pkcs12_bag_decrypt** (*gnutls_pkcs12_bag_t* **bag**, *const char ** **pass**)

int **gnutls_pkcs12_bag_get_count** (*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**)

Description: 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_PKCS**12_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.

int **gnutls_pkcs12_bag_get_data** (*gnutls_pkcs12_bag_t* **bag**, *unsigned* **indx**, *gnutls_datum_t ** **data**)

int **gnutls_pkcs12_bag_get_key_id** (*gnutls_pkcs12_bag_t* **bag**, *unsigned* **indx**, *gnutls_datum_t ** **id**)

int **gnutls_pkcs12_bag_get_friendly_name** (*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 subsection 6.5.4.

int **gnutls_pkcs12_set_bag** (*gnutls_pkcs12_t* **pkcs12**, *gnutls_pkcs12_bag_t* **bag**)

int **gnutls_pkcs12_bag_encrypt** (*gnutls_pkcs12_bag_t* **bag**, *const char ** **pass**, *unsigned int* **flags**)

int **gnutls_pkcs12_generate_mac** (*gnutls_pkcs12_t* **pkcs12**, *const char ** **pass**)

int **gnutls_pkcs12_bag_set_data** (*gnutls_pkcs12_bag_t* **bag**, *gnutls_pkcs12_bag_type_t* **type**, *const gnutls_datum_t ** **data**)

int **gnutls_pkcs12_bag_set_crl** (*gnutls_pkcs12_bag_t* **bag**, *gnutls_x509_crl_t* **crl**)

int **gnutls_pkcs12_bag_set_crt** (*gnutls_pkcs12_bag_t* **bag**, *gnutls_x509_crt_t* **crt**)

int **gnutls_pkcs12_bag_set_key_id** (*gnutls_pkcs12_bag_t* **bag**, *unsigned* **indx**, *const gnutls_datum_t ** **id**)

int **gnutls_pkcs12_bag_set_friendly_name** (*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`.

int **gnutls_x509_privkey_import_openssl** (*gnutls_x509_privkey_t* **key**, *const gnutls_datum_t ** **data**, *const char ** **password**)

Description: 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.

3.2.6. 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.

certtool help/usage (@option-?)

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>] ]...

None:

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --attime=str           Perform validation at the timestamp instead of the system time

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 provable method
       --verify-provable-privkey  Verify a private key generated from a seed using a provable method
       --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 set
       --verify-hostname=str  Specify a hostname to be used for certificate chain verification
       --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 verification
       --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
                               - enabled by default
                               - disabled as '--no-p7-include-cert'
      --p7-time              Will include a timestamp in the PKCS #7 structure
      --p7-show-data         Will show the embedded data in the PKCS #7 structure
      --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 format
      --hash=str             Hash algorithm to use for signing
      --salt-size=num        Specify the RSA-PSS key default salt size
      --label=str            Specify the RSA-OAEP label, encoded in hexadecimal
      --inder                Use DER format for input certificates, private keys, and DH parameters
      --inraw                an alias for the 'inder' option
      --outder               Use DER format for output certificates, private keys, and DH parameters
      --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, private keys, etc
                               - enabled by default
                               - disabled as '--no-text'

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.
```

```


Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.NUMBER argument. Specifies the debug level.

attime option.

This is the “perform validation at the timestamp instead of the system time” option. This option takes a ArgumentType.STRING argument “`timestamp`”. timestamp is an instance in time encoded as Unix time or in a human readable timestring such as ”29 Feb 2004”, ”2004-02-29”. Full documentation available at <https://www.gnu.org/software/coreutils/manual/html_node/Date-input-formats.html> or locally via info ’(coreutils) date invocation’.

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 ArgumentType.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 ArgumentType.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 or RSA-OAEP private keys, the –hash option will restrict the allowed hash for the key; For RSA-PSS 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 ArgumentType.STRING argument. This option can be combined with –generate-privkey, to specify the key type to be generated. Valid options are, ’rsa’, ’rsa-pss’, ’rsa-oaep’, ’dsa’, ’ecdsa’, ’ed25519, ’ed448’, ’x25519’, and ’x448’.’. 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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 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 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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.STRING argument. This option can be used with a file

load-data option.

This is the "loads auxiliary data" option. This option takes a ArgumentType.STRING argument. This option can be used with a file

password option.

This is the "password to use" option. This option takes a ArgumentType.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, section 3.2.6.

hash option.

This is the "hash algorithm to use for signing" option. This option takes a ArgumentType.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 ArgumentType.NUMBER argument. Typical keys shouldn't set or restrict this option.

label option.

This is the "specify the rsa-oaep label, encoded in hexadecimal" option. This option takes a ArgumentType.STRING 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. 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, section 3.2.6.

outder option.

This is the "use der format for output certificates, private keys, and dh parameters" option. The output will be in DER or RAW format.

outraw option.

This is an alias for the `outder` option, section 3.2.6.

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 ArgumentType.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 ArgumentType.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

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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=Arkadias"

# 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 302ca00d1b0b56414e5245494e2e4f5247a11b3019a006020400000002a10f300d1b047269636b1b05
#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. Its purpose is loosely
# defined. Not use it unless required by a protocol.
#key_agreement

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

# The nonRepudiation flag of RFC5280. Its 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"

```

3.2.7. 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.

ocsptool help/usage (@option-?)

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>] ]...

None:

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --ask[=str]            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
       --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
       --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
       --attime=str           Perform validation at the timestamp instead of the system time

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.

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.

Please send bug reports to:  <bugs@gnutls.org>
```

debug option (-d).

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

ask option.

This is the "ask an ocsp/http server on a certificate validity" option. This option takes a ArgumentType.STRING argument @fileserver 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 ArgumentType.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.

attime option.

This is the “perform validation at the timestamp instead of the system time” option. This option takes a ArgumentType.STRING argument “`timestamp`”. timestamp is an instance in time encoded as Unix time or in a human readable timestring such as ”29 Feb 2004”, ”2004-02-29”. Full documentation available at <https://www.gnu.org/software/coreutils/manual/html_node/Date-input-formats.html> or locally via info ’(coreutils) date invocation’.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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
```

3.2.8. Invoking danetool

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

danetool help/usage (@option-?)

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>] ]...

None:

```

```
   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
       --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            an alias for the 'starttls-proto' option
       --starttls-proto=str   The application protocol to be used to obtain the server's certificate (https, ft
       --ca                   Whether the provided certificate or public key is a Certificate Authority
       --x509                 Use the hash of the X.509 certificate, rather than the public key
       --local                an alias for the 'domain' option
       --domain               The provided certificate or public key is issued by the local domain
                                - enabled by default
                                - disabled as '--no-domain'
       --local-dns            Use the local DNS server for DNSSEC resolving
       --insecure             Do not verify any DNSSEC signature
       --inder                Use DER format for input certificates and private keys
       --inraw                an alias for the 'inder' option
       --print-raw            Print the received DANE data in raw format
       --quiet                Suppress several informational messages

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 generate and check DNS resource records for the DANE protocol.

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.NUMBER argument. Specifies the debug level.

load-pubkey option.

This is the "loads a public key file" option. This option takes a ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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, section 3.2.8.

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 ArgumentType.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, section 3.2.8.

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 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. 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, section 3.2.8.

print-raw option.

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

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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 locally 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.pem
```

To verify an SMTP server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --starttls-proto=smtp --load-certificate chain.pem
```

3.3. 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.

3.3.1. PSK authentication

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 [10]. 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** (*gnutls_datum_t* * **key**, *unsigned int* **key_size**)

int **gnutls_hex_encode** (*const gnutls_datum_t* * **data**, *char* * **result**, *size_t* * **result_size**)

int **gnutls_hex_decode** (*const gnutls_datum_t* * **hex_data**, *void* * **result**, *size_t* * **result_size**)

Invoking psktool

Program that generates random keys for use with TLS-PSK. The keys are stored in hexadecimal format in a key file.

psktool help/usage (@option-?)

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>] ]...

None:

   -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-bits)
                                - 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

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.

Program  that generates random keys for use with TLS-PSK. The
keys are stored in hexadecimal format in a key file.

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

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

pskfile option (-p).

This is the "specify a pre-shared key file" option. This option takes a ArgumentType.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, section 3.3.1.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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'.

3.3.2. SRP authentication

Authentication using SRP

GnuTLS supports authentication via the Secure Remote Password or SRP protocol (see [45, 44] 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 [39]. 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**)

Description: 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** (*const gnutls_datum_t* * **data**, *gnutls_datum_t* * **result**)

int **gnutls_srp_base64_decode2** (*const gnutls_datum_t* * **b64_data**, *gnutls_datum_t* * **result**)

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).

srptool help/usage (@option-?)

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>] ]...

None:

   -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 use
   -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

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.

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).

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the "enable debugging" option. This option takes a ArgumentType.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 ArgumentType.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 ArgumentType.STRING argument. This generates a password configuration file (tpasswd.conf) containing the required for TLS parameters.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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
```

3.3.3. 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 identities 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 subsubsection 3.1.4).

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.

3.4. Selecting an appropriate authentication method

This section provides some guidance on how to use the available authentication methods in GnuTLS in various scenarios.

3.4.1. 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 subsection 3.3.1). 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 subsection 3.3.2). 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 3.1). 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 subsubsection 3.1.4).

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.

3.4.2. 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 3.1). 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 subsubsection 3.1.4).

To mitigate the uncertainty of the information exchanged in the first connection other channels over the Internet may be used, e.g., DNSSEC (see subsubsection 3.1.4).

3.4.3. 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 3.1). 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 subsection 3.3.2). 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 3.1). 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 subsubsection 3.1.4).

enum **gnutls_certificate_status_t**:

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 certificate 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.`

Table 3.4.: The `gnutls_certificate_status_t` enumeration.

enum gnutls_certificate_verify_flags:

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.
GNUTLS_VERIFY_RSA_PSS_FIXED_SALT_LENGTH	Disallow RSA-PSS signatures made with mismatching salt length with digest length, as mandated in RFC 8446 4.2.3.

Table 3.5.: The `gnutls_certificate_verify_flags` enumeration.

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 3.6.: Key purpose object identifiers.

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 3.7.: Certificate revocation list fields.

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 3.8.: The most important OCSP response fields.

`enum` **gnutls_x509_crl_reason_t:**

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.`

Table 3.9.: The revocation reasons

enum gnutls_pkcs_encrypt_flags_t:

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.

Table 3.10.: Encryption flags

4 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 [28].

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 subsection 4.3.9).

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315; \
manufacturer=EnterSafe;object=test1;type=cert

```

4.1. 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);
}
```

4.1.1. Public keys

An abstract `gnutls_pubkey_t` can be initialized and freed by using the functions below.

int **gnutls_pubkey_init** (*gnutls_pubkey_t ** **key**)

void **gnutls_pubkey_deinit** (*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** (*gnutls_pubkey_t* **key**, *gnutls_x509_crt_t* **crt**, *unsigned int* **flags**)

int **gnutls_pubkey_import_pkcs11** (*gnutls_pubkey_t* **key**, *gnutls_pkcs11_obj_t* **obj**, *unsigned int* **flags**) *int* **gnutls_pubkey_import_url** (*gnutls_pubkey_t* **key**, *const char ** **url**, *unsigned int* **flags**)

int **gnutls_pubkey_import_privkey** (*gnutls_pubkey_t* **key**, *gnutls_privkey_t* **pkey**, *unsigned int* **usage**, *unsigned int* **flags**)

int **gnutls_pubkey_import** (*gnutls_pubkey_t* **key**, *const gnutls_datum_t ** **data**, *gnutls_x509_crt_fmt_t* **format**)

int **gnutls_pubkey_export** (*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**)

Description: `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.`

Other helper functions that allow directly importing from raw X.509 structures are shown below.

int **gnutls_pubkey_import_x509_raw** (*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` which will import public keys from URLs that identify objects stored in tokens (see section 4.3 and section 4.4). A function to check for a supported by GnuTLS URL is `gnutls_url_is_supported`.

unsigned **gnutls_url_is_supported** (*const char ** **url**)

Description: `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.`

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` 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 [8] which calculates a digest on a part of the subjectPublicKeyInfo object.

int **gnutls_pubkey_get_pk_algorithm** (*gnutls_pubkey_t* **key**, *unsigned int ** **bits**)

int **gnutls_pubkey_get_preferred_hash_algorithm** (*gnutls_pubkey_t* **key**, *gnutls_digest_algorithm_t ** **hash**, *unsigned int ** **mand**)

int **gnutls_pubkey_get_key_id** (*gnutls_pubkey_t* **key**, *unsigned int* **flags**, *unsigned char ** **output_data**, *size_t ** **output_data_size**)

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**)

To export the key-specific parameters, or obtain a unique key ID the following functions are provided.

int **gnutls_pubkey_export_rsa_raw2** (*gnutls_pubkey_t* **key**, *gnutls_datum_t ** **m**, *gnutls_datum_t ** **e**, *unsigned* **flags**)

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**)

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**)

int **gnutls_pubkey_export_ecc_x962** (*gnutls_pubkey_t* **key**, *gnutls_datum_t ** **parameters**, *gnutls_datum_t ** **ecpoint**)

4.1.2. Private keys

An abstract `gnutls_privkey_t` can be initialized and freed by using the functions below.

int **gnutls_privkey_init** (*gnutls_privkey_t ** **key**)

void **gnutls_privkey_deinit** (*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** (*gnutls_privkey_t* **pkey**, *gnutls_x509_privkey_t* **key**, *unsigned int* **flags**)

int **gnutls_privkey_import_pkcs11** (*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**)

Description: `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.`

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**)

int **gnutls_privkey_get_pk_algorithm** (*gnutls_privkey_t* **key**, *unsigned int ** **bits**)

gnutls_privkey_type_t **gnutls_privkey_get_type** (*gnutls_privkey_t* **key**)

int **gnutls_privkey_status** (*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**)

Description: 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 PKCS#1 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 PKCS#1 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.

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** (*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** (*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** (*gnutls_privkey_t* **key**, *gnutls_ecc_curve_t* * **curve**, *gnutls_datum_t* * **x**, *gnutls_datum_t* * **y**, *gnutls_datum_t* * **k**, *unsigned int* **flags**)

4.1.3. 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**)

Description: 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.

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_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**)

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**)

int **gnutls_x509_crq_privkey_sign** (*gnutls_x509_crq_t* **crq**, *gnutls_privkey_t* **key**, *gnutls_digest_algorithm_t* **dig**, *unsigned int* **flags**) *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**)

Description: 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.

int **gnutls_pubkey_encrypt_data** (*gnutls_pubkey_t* **key**, *unsigned int* **flags**, *const gnutls_datum_t* * **plaintext**, *gnutls_datum_t* * **ciphertext**)

Description: 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.

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**)

Description: 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.

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**)

Description: 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_TLS**1_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.

int **gnutls_privkey_decrypt_data** (*gnutls_privkey_t* **key**, *unsigned int* **flags**, *const gnutls_datum_t ** **ciphertext**, *gnutls_datum_t ** **plaintext**)

Description: 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.

int **gnutls_x509_crq_set_pubkey** (*gnutls_x509_crq_t* **crq**, *gnutls_pubkey_t* **key**)

Description: 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.

int **gnutls_x509_crt_set_pubkey** (*gnutls_x509_crt_t* **crt**, *gnutls_pubkey_t* **key**)

Description: 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.

4.2. System and application-specific keys

4.2.1. 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**)

Description: 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_X**509. 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.

void **gnutls_system_key_iter_deinit** (*gnutls_system_key_iter_t* **iter**)

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**)

int **gnutls_system_key_delete** (*const char ** **cert_url**, *const char ** **key_url**)

4.2.2. 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`. Such keys can be registered using the API in `gnutls/urls.h`. The function which registers such keys is `gnutls_register_custom_url`.

int **gnutls_register_custom_url** (*const gnutls_custom_url_st ** **st**)

Description: 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.

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`.

```
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;
```

4.3. Smart cards and HSMs

In this section we present the smart-card and hardware security module (HSM) support in GnuTLS using PKCS #11 [2]. 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`.

4.3.1. 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 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:

[1] For example, OpenSC-supported cards.
[2] https://p11-glue.github.io/p11-glue/p11-kit.html

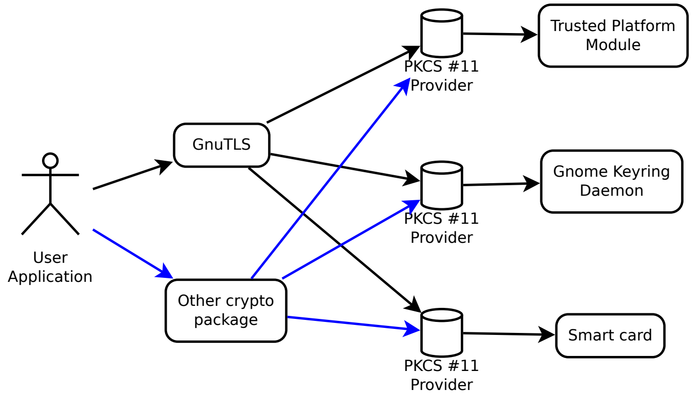

Figure 4.1.: PKCS #11 module usage.

```
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 subsection 4.3.2 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` after a fork() call; since 3.3.0 this is no longer required.

4.3.2. 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`, after having called `gnutls_pkcs11_init` specifying the `GNUTLS_PKCS11_FLAG_MANUAL` flag.

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`

[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).

int **gnutls_pkcs11_add_provider** (*const char ** **name**, *const char ** **params**)

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.

are made, then the PKCS #11 functionality is effectively disabled.

int **gnutls_pkcs11_init** (*unsigned int* **flags**, *const char ** **deprecated_config_file**)

Description: This function will initialize the PKCS 11 subsystem in gnutls. It will read configuration files if **GNUTLS_PKCS**11_FLAG_AUTO is used or allow you to independently load PKCS 11 modules using gnutls_pkcs11_add_provider() if **GNUTLS_PKCS**11_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_PKCS**11_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.

4.3.3. 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** (*gnutls_pkcs11_token_callback_t* **fn**, *void* * **userdata**)

void **gnutls_pkcs11_set_pin_function** (*gnutls_pin_callback_t* **fn**, *void* * **userdata**)

int **gnutls_pkcs11_add_provider** (*const char* * **name**, *const char* * **params**)

gnutls_pin_callback_t **gnutls_pkcs11_get_pin_function** (*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.

enum **gnutls_pin_flag_t**:

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.

Table 4.1.: 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` 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** (*gnutls_certificate_credentials_t* **cred**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

void **gnutls_pubkey_set_pin_function** (*gnutls_pubkey_t* **key**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

void **gnutls_privkey_set_pin_function** (*gnutls_privkey_t* **key**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

void **gnutls_pkcs11_obj_set_pin_function** (*gnutls_pkcs11_obj_t* **obj**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

void **gnutls_x509_crt_set_pin_function** (*gnutls_x509_crt_t* **crt**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

4.3.4. Reading objects

All PKCS #11 objects are referenced by GnuTLS functions by URLs as described in [28]. 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** (*gnutls_pkcs11_obj_t *** **p_list**, *unsigned int ** **n_list**, *const char ** **url**, *unsigned int* **flags**)

int **gnutls_pkcs11_obj_import_url** (*gnutls_pkcs11_obj_t* **obj**, *const char ** **url**, *unsigned int* **flags**)

int **gnutls_pkcs11_obj_export_url** (*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**)

Description: 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.

int **gnutls_x509_crt_import_pkcs11** (*gnutls_x509_crt_t* **crt**, *gnutls_pkcs11_obj_t* **pkcs11_crt**)

int **gnutls_x509_crt_import_url** (*gnutls_x509_crt_t* **crt**, *const char ** **url**, *unsigned int* **flags**)

int **gnutls_x509_crt_list_import_pkcs11** (*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** (*const char ** **token_url**, *const char ** **so_pin**, *const char ** **label**)

int **gnutls_pkcs11_token_get_url** (*unsigned int* **seq**, *gnutls_pkcs11_url_type_t* **detailed**, *char *** **url**)

int **gnutls_pkcs11_token_get_info** (*const char ** **url**, *gnutls_pkcs11_token_info_t* **ttype**, *void ** **output**, *size_t ** **output_size**)

int **gnutls_pkcs11_token_get_flags** (*const char ** **url**, *unsigned int ** **flags**)

int **gnutls_pkcs11_token_set_pin** (*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;
}
```

4.3.5. 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**)

Description: 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_PKCS**11_OBJ_FLAG_MARK_NOT_SENSITIVE is specified.

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

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**)

Description: This function will copy a certificate into a PKCS #11 token specified by a URL. Valid flags to mark the certificate: **GNUTLS_PKCS**11_OBJ_FLAG_MARK_TRUSTED, **GNUTLS_PKCS**11_OBJ_FLAG_MARK_PRIVATE, **GNUTLS_PKCS**11_OBJ_FLAG_MARK_CA, **GNUTLS_PKCS**11_OBJ_FLAG_MARK_ALWAYS_AUTH.

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

int **gnutls_pkcs11_delete_url** (*const char ** **object_url**, *unsigned int* **flags**)

Description: 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.

4.3.6. 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**)

Description: 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.

int **gnutls_pkcs11_obj_get_ptr** (*gnutls_pkcs11_obj_t* **obj**, *void *** **ptr**, *void *** **session**, *void *** **ohandle**, *unsigned long ** **slot_id**, *unsigned int* **flags**)

Description: Obtains the PKCS#11 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.

4.3.7. Using a PKCS #11 token with TLS

It is possible to use a PKCS #11 token to a TLS session, as shown in subsection 6.1.3. 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** (*gnutls_certificate_credentials_t* **cred**, *const char ** **cafile**, *gnutls_x509_crt_fmt_t* **type**)

int **gnutls_certificate_set_x509_key_file2** (*gnutls_certificate_credentials_t* **res**, *const char ** **certfile**, *const char ** **keyfile**, *gnutls_x509_crt_fmt_t* **type**, *const char ** **pass**, *unsigned int* **flags**)

4.3.8. 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 4.1. 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 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 distrusting 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.

4.3.9. 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.

p11tool help/usage (@option-?)

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]
```

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

[5] See the 'Restricting the scope of CA certificates' post at https://nmav.gnutls.org/2016/06/restricting-scope-of-ca-certificates.html

```

None:


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 initialization

Object listing:

       --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 trust
                                - 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
       --mark-trusted         Marks the object to be written as trusted
                                - prohibits the option 'mark-distrusted'
       --mark-distrusted      When retrieving objects, it requires the objects to be distrusted
                                - prohibits the option 'mark-trusted'
       --mark-decrypt         Marks the object to be written for decryption
       --mark-sign            Marks the object to be written for signature generation
       --mark-ca              Marks the object to be written as a CA
       --mark-private         Marks the object to be written as private
       --ca                   an alias for the 'mark-ca' option
       --private              an alias for the 'mark-private' option
       --mark-always-authenticate  Marks the object to be written as always authenticate
       --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
       --so-login             Force security officer login to token
       --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
       --inraw                an alias for the 'inder' option
       --outder               Use DER format for output certificates, private keys, and DH parameters
       --outraw               an alias for the 'outder' option
       --provider=file        Specify the PKCS #11 provider library
       --detailed-url         Print detailed URLs
       --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.


Please send bug reports to:  <bugs@gnutls.org>

```

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 ArgumentType.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 ArgumentType.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, section 4.3.9.

list-keys option.

This is an alias for the `list-all-privkeys` option, section 4.3.9.

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, section 4.3.9.

distrusted option.

This is an alias for the `mark-distrusted` option, section 4.3.9.

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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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.

When writing a certificate object, its CKA_ID is set to the same CKA_ID of the corresponding public key, if it exists on the token; otherwise it will be derived from the X.509 Subject Key Identifier of the certificate. If this behavior is undesired, write the public key to the token beforehand.

id option.

This is the "sets an id for the write operation" option. This option takes a ArgumentType.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. 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:

- must not appear in combination with any of the following options: mark-distrusted.
- can be disabled with –no-mark-trusted.

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" 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. 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. 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. 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. 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, section 4.3.9.

private option.

This is an alias for the `mark-private` option, section 4.3.9.

mark-always-authenticate option.

This is the "marks the object to be written as always authenticate" option. Marks the object to be generated/written with the CKA_ALWAYS_AUTHENTICATE flag. The written object will Mark the object as requiring authentication (pin entry) before every operation.

secret-key option.

This is the "provide a hex encoded secret key" option. This option takes a ArgumentType.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 ArgumentType.NUMBER argument. Specifies the debug level.

so-login option.

This is the "force security officer login to token" option. Forces login to the token as security officer (admin).

admin-login option.

This is an alias for the `so-login` option, section 4.3.9.

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 ArgumentType.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 ArgumentType.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 ArgumentType.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. Use DER/RAW format for input certificates and private keys.

inraw option.

This is an alias for the `inder` option, section 4.3.9.

outder option.

This is the “use der format for output certificates, private keys, and dh parameters” option. The output will be in DER or RAW format.

outraw option.

This is an alias for the `outder` option, section 4.3.9.

provider option.

This is the “specify the pkcs #11 provider library” option. This option takes a ArgumentType.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 ArgumentType.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.

version option (-v).

This is the “output version information and exit” option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is ‘v’, a simple version. The ‘c’ mode will print copyright information and ‘n’ will print the full copyright notice.

help option (-h).

This is the “display extended usage information and exit” option. Display usage information and exit.

more-help option (-!).

This is the “extended usage information passed thru pager” option. Pass the extended usage information through a pager.

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
```

4.4. 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 4.3) 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`.

4.4.1. 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 [22] 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`

[6] https://github.com/google/chaps-linux
[7] https://sourceforge.net/projects/opencryptoki/

When objects require a PIN to be accessed the same callbacks as with PKCS #11 objects are expected (see subsection 4.3.3). 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.

4.4.2. Key generation

All keys used by the TPM must be generated by the TPM. This can be done using `gnutls_tpm_privkey_generate`.

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**)

Description: 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.

int **gnutls_tpm_get_registered** (*gnutls_tpm_key_list_t ** **list**)

void **gnutls_tpm_key_list_deinit** (*gnutls_tpm_key_list_t* **list**)

int **gnutls_tpm_key_list_get_url** (*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**)

Description: 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.

4.4.3. 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` can access TPM URLs.

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**)

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**)

int **gnutls_privkey_import_tpm_url** (*gnutls_privkey_t* **pkey**, *const char ** **url**, *const char ** **srk_password**, *const char ** **key_password**, *unsigned int* **flags**)

Description: 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.

int **gnutls_pubkey_import_tpm_url** (*gnutls_pubkey_t* **pkey**, *const char ** **url**, *const char ** **srk_password**, *unsigned int* **flags**)

Description: 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.

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`.

int **gnutls_tpm_get_registered** (*gnutls_tpm_key_list_t ** **list**)

void **gnutls_tpm_key_list_deinit** (*gnutls_tpm_key_list_t* **list**)

int **gnutls_tpm_key_list_get_url** (*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**)

Description: 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.

4.4.4. Invoking tpmtool

Program that allows handling cryptographic data from the TPM chip.

tpmtool help/usage (@option-?)

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>] ]...

None:

   -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
                                - prohibits the option 'legacy'
                                - requires the option 'generate-rsa'
       --legacy               Any generated key will be a legacy key
                                - prohibits the option 'signing'
                                - requires the option 'generate-rsa'
       --user                 Any registered key will be a user key
                                - prohibits the option 'system'
                                - requires the option 'register'
       --system               Any registered key will be a system key
                                - prohibits the option 'user'
                                - requires the option 'register'
       --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
       --outder               Use DER format for output keys
       --srk-well-known       SRK has well known password (20 bytes of zeros)

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.

Program that allows handling cryptographic data from the TPM chip.

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.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 not appear in combination with any of the following options: system.
- must appear in combination with the following options: register.

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

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 ArgumentType.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 ArgumentType.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. 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. The output will be in the TPM portable DER format.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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
```

5

How to use GnuTLS in applications

5.1. 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.

5.1.1. General idea

A brief description of how GnuTLS sessions operate is shown at Figure 5.1. 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.

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 subsection 2.5.4) 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.

[1]The first message in a TLS handshake

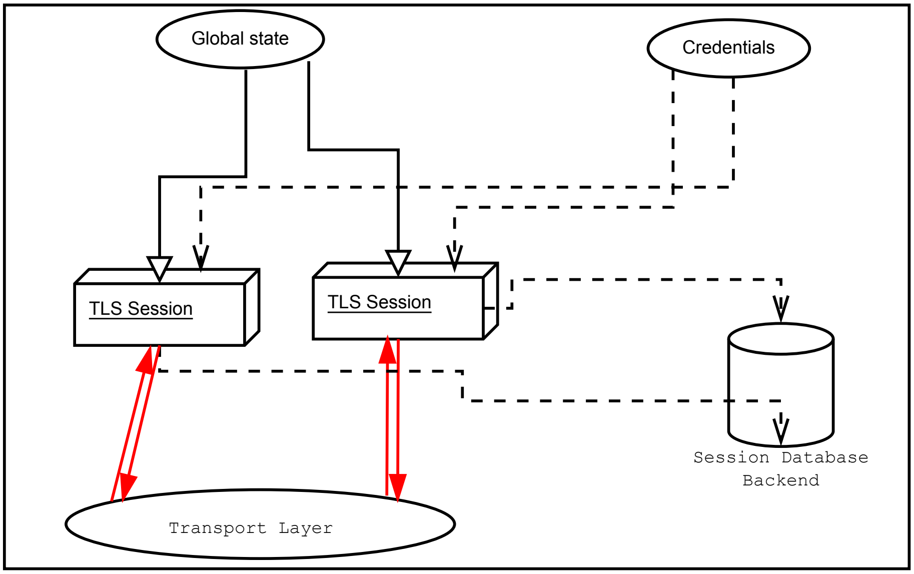

Figure 5.1.: High level design of GnuTLS.

5.1.2. 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`).

In GnuTLS, many objects are represented as opaque types that are initialized by passing an address to storage of that type to a pointer parameter of a function name `gnutls_@varobj_init`, and which have a counterpart function `gnutls_@varobj_deinit`. It is safe, but not mandatory, to pre-initialize the opaque storage to contain all zeroes (such as by using `calloc()` or `memset()`). If the initializer succeeds, the storage must be passed to the counterpart deinitializer when the object is no longer in use to avoid memory leaks. As of version 3.8.0, if the initializer function fails, it is safe, but not mandatory, to call the counterpart deinitializer, regardless of whether the storage was pre-initialized. However, this was not guaranteed in earlier versions; for maximum portability to older library versions, callers should either pre-initialize the storage to zero before initialization or refrain from calling the deinitializer if the initializer fails.

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`. 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`. All errors can be converted to a descriptive string using `gnutls_strerror`.

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 5.9). See Appendix D, for a description of the available error codes.

5.1.3. 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 [35] rules (since GnuTLS 3.5.7).

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;
```

5.1.4. 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` and `gnutls_global_set_log_function` are to be used. Those will print verbose information on the GnuTLS functions internal flow.

void **gnutls_global_set_log_level** (*int* **level**)

void **gnutls_global_set_log_function** (*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 5.1.

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 9.7) if set to one it will force the FIPS mode enablement.

Table 5.1.: 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`. 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** (*gnutls_audit_log_func* **log_func**)

Description: 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*);

5.1.5. 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`), 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` 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` and `gnutls_record_get_direction` 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`.[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.

[2] On special systems you could manually specify the locking system using the function `gnutls_global_set_mutex` before calling any other GnuTLS function. Setting mutexes manually is not recommended.

5.1.6. 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.

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`.

5.1.7. 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` with `GNUTLS_SHUT_WR` and the receiving process waiting for a return value of zero.

5.1.8. 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** (*gnutls_session_t* **session**, *gnutls_push_func* **push_func**)
>
> *void* **gnutls_transport_set_pull_function** (*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`. All callbacks should allocate and free memory using `gnutls_malloc` and `gnutls_free`.

5.2. 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.

5.2.1. 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.

5.2.2. Initialization

The GnuTLS library is initialized on load; prior to 3.3.0 was initialized by calling `gnutls_global_init`[3]. `gnutls_global_init` in versions after 3.3.0 is thread-safe (see subsection 5.1.5).

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`, after that, to ensure they don't disrupt GnuTLS' operation.

[3]The original behavior of requiring explicit initialization can obtained by setting the GNUTLS_NO_IMPLICIT_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.

5.2.3. 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`.

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
```

5.2.4. 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).

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])
```

5.3. 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 subsection 2.5.2). In this section we elaborate on the TLS or DTLS session initiation. Each session is initialized using `gnutls_init` 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**)

Description: This function initializes the provided session. Every session must be initialized before use, and after successful initialization and use must be deinitialized 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_DEFAULT_EXTENSIONS** flag. Note that it is never mandatory to use gnutls_deinit() after this function fails. Since gnutls 3.8.0, it is safe to unconditionally use gnutls_deinit() even after failure regardless of whether the memory was initialized prior to gnutls_init(); however, clients wanting to be portable to older versions of the library should either skip deinitialization on failure, or pre-initialize the memory passed in to gnutls_init() to all zeroes via memset() or similar.

Returns: **GNUTLS_E_SUCCESS** on success, or an error code.

After the session initialization details on the allowed ciphersuites and protocol versions should be set using the priority functions such as `gnutls_priority_set` and `gnutls_priority_set_direct`. We elaborate on them in section 5.10. 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`.

5.4. 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`). A mapping of the key exchange methods with the credential types is shown in Table 5.3.

int **gnutls_credentials_set** (*gnutls_session_t* **session**, *gnutls_credentials_type_t* **type**, *void ** **cred**)

Description: 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.

5.4.1. 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** (*gnutls_certificate_credentials_t ** **res**)

void **gnutls_certificate_free_credentials** (*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** (*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** (*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** (*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` which accept not only file names but URLs that specify objects stored in token, or system certificates and keys (see section 4.2). For these cases, another important function is `gnutls_certificate_set_pin_function`, that allows setting a callback function to retrieve a PIN if the input keys are protected by PIN.

void **gnutls_certificate_set_pin_function** (*gnutls_certificate_credentials_t* **cred**, *gnutls_pin_callback_t* **fn**, *void ** **userdata**)

Description: `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.`

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` in combination with `gnutls_pcert_import_x509_raw` and `gnutls_privkey_import_x509_raw`.

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 subsection 2.6.2).

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`. Both functions are shown below.

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**)

Description: 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).

void **gnutls_certificate_set_retrieve_function** (*gnutls_certificate_credentials_t* **cred**, *gnutls_certificate_retrieve_function ** **func**)

void **gnutls_certificate_set_retrieve_function2** (*gnutls_certificate_credentials_t* **cred**, *gnutls_certificate_retrieve_function2 ** **func**)

void **gnutls_certificate_set_retrieve_function3** (*gnutls_certificate_credentials_t* **cred**, *gnutls_certificate_retrieve_function3 ** **func**)

int **gnutls_sign_algorithm_get_requested** (*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`, 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** (*gnutls_pcert_st * **pcert**, *gnutls_x509_crt_t* **crt**, *unsigned int* **flags**)

int **gnutls_pcert_import_x509_raw** (*gnutls_pcert_st * **pcert**, *const gnutls_datum_t * **cert**, *gnutls_x509_crt_fmt_t* **format**, *unsigned int* **flags**)

void **gnutls_pcert_deinit** (*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 3.1 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`. 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`. 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**)

Description: 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.

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, *void* **gnutls_certificate_send_x509_rdn_sequence** (*gnutls_session_t* **session**, *int* **status**)

Description: 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.

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`.

int **gnutls_certificate_set_x509_key_file** (*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** (*gnutls_certificate_credentials_t* **res**, *const char ** **pkcs12file**, *gnutls_x509_crt_fmt_t* **type**, *const char ** **password**)

void **gnutls_certificate_set_retrieve_function2** (*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` which will load the trusted authorities from the system store.

int **gnutls_certificate_set_x509_system_trust** (*gnutls_certificate_credentials_t* **cred**)

Description: 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.

int **gnutls_certificate_set_x509_trust_file** (*gnutls_certificate_credentials_t* **cred**, *const char ** **cafile**, *gnutls_x509_crt_fmt_t* **type**)

int **gnutls_certificate_set_x509_trust_dir** (*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` is called prior to handshake.

Alternatively, one must set a callback function during the handshake using `gnutls_certificate_set_verify_function`, which will verify the peer's certificate once received. The verification should happen using `gnutls_certificate_verify_peers3` 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` to obtain the raw certificate of the peer and verify it using the functions discussed in subsection 3.1.1.

In both the automatic and the manual cases, the verification status returned can be printed using `gnutls_certificate_verification_status_print`.

void **gnutls_session_set_verify_cert** (*gnutls_session_t* **session**, *const char ** **hostname**, *unsigned* **flags**)

Description: 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.

int **gnutls_certificate_verify_peers3** (*gnutls_session_t* **session**, *const char ** **hostname**, *unsigned int ** **status**)

void **gnutls_certificate_set_verify_function** (*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` 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 subsubsection 3.1.4).

5.4.2. Raw public-keys

As of version 3.6.6 GnuTLS supports subsection 3.1.3. With raw public-keys only the public-key part (that is normally embedded in a certificate) is transmitted 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** (*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** (*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** (*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**)

5.4.3. 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` 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** (*gnutls_srp_server_credentials_t ** **sc**)

int **gnutls_srp_allocate_client_credentials** (*gnutls_srp_client_credentials_t ** **sc**)

void **gnutls_srp_free_server_credentials** (*gnutls_srp_server_credentials_t* **sc**)

void **gnutls_srp_free_client_credentials** (*gnutls_srp_client_credentials_t* **sc**)

int **gnutls_srp_set_client_credentials** (*gnutls_srp_client_credentials_t* **res**, *const char ** **username**, *const char ** **password**)

void **gnutls_srp_set_client_credentials_function** (*gnutls_srp_client_credentials_t* **cred**, *gnutls_srp_client_credentials_function ** **func**)

Description: 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` should be called, to set an appropriate callback.

int **gnutls_srp_set_server_credentials_file** (*gnutls_srp_server_credentials_t* **res**, *const char ** **password_file**, *const char ** **password_conf_file**)

Description: 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** (*gnutls_srp_server_credentials_t* **cred**, *gnutls_srp_server_credentials_function* * **func**)

Description: 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.

5.4.4. PSK

The initialization functions in PSK credentials differ between client and server.

int **gnutls_psk_allocate_server_credentials** (*gnutls_psk_server_credentials_t* * **sc**)

int **gnutls_psk_allocate_client_credentials** (*gnutls_psk_client_credentials_t* * **sc**)

void **gnutls_psk_free_server_credentials** (*gnutls_psk_server_credentials_t* **sc**)

void **gnutls_psk_free_client_credentials** (*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` 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** (*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** (*gnutls_psk_client_credentials_t* **cred**, *gnutls_psk_client_credentials_function * **func**)

Description: 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`. If a different password file format is to be used, then a callback should be set instead by `gnutls_psk_set_server_credentials_function`.

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`. The client can retrieve the hint, for example in the callback function, using `gnutls_psk_client_get_hint`.

int **gnutls_psk_set_server_credentials_file** (*gnutls_psk_server_credentials_t* **res**, *const char * **password_file**)

Description: 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. Each entry in the file consists of a username, followed by a colon (':') and a hex-encoded key. If the username contains a colon or any other special character, it can be hex-encoded preceded by a '#'.

Returns: On success, **GNUTLS_E_SUCCESS** (0) is returned, otherwise an error code is returned.

void **gnutls_psk_set_server_credentials_function** (*gnutls_psk_server_credentials_t* **cred**, *gnutls_psk_server_credentials_function ** **func**)

int **gnutls_psk_set_server_credentials_hint** (*gnutls_psk_server_credentials_t* **res**, *const char ** **hint**)

*const char ** **gnutls_psk_client_get_hint** (*gnutls_session_t* **session**)

Note: *there is no hint in TLS 1.3, so this function will return* **NULL** *if TLS 1.3 has been negotiated.*

5.4.5. 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 subsection 5.12.6 for more information. The initialization functions for the credentials are shown below.

int **gnutls_anon_allocate_server_credentials** (*gnutls_anon_server_credentials_t ** **sc**)

int **gnutls_anon_allocate_client_credentials** (*gnutls_anon_client_credentials_t ** **sc**)

void **gnutls_anon_free_server_credentials** (*gnutls_anon_server_credentials_t* **sc**)

void **gnutls_anon_free_client_credentials** (*gnutls_anon_client_credentials_t* **sc**)

5.5. 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` would be sufficient to specify the socket descriptor.

void **gnutls_transport_set_int** (*gnutls_session_t* **session**, *int* **fd**)

void **gnutls_transport_set_int2** (*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** (*gnutls_session_t* **session**, *gnutls_transport_ptr_t* **ptr**)

void **gnutls_transport_set_ptr2** (*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**)

Description: `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**)

Description: `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);`

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` with the intended errno value instead of setting `errno` directly.

GnuTLS currently only interprets the EINTR, EAGAIN and EMSGSIZE errno values and returns the corresponding GnuTLS error codes:

- `GNUTLS_E_INTERRUPTED`

void **gnutls_transport_set_pull_function** (*gnutls_session_t* **session**, *gnutls_pull_func* **pull_func**)

Description: 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**)

Description: 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.

void **gnutls_transport_set_errno** (*gnutls_session_t* **session**, *int* **err**)

Description: 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_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`** function. To check the retransmission timers the function **`gnutls_dtls_get_timeout`** is provided, which returns the time remaining until the next retransmission, or better the time until **`gnutls_handshake`** should be called again.

void **gnutls_transport_set_pull_timeout_function** (*gnutls_session_t* **session**, *gnutls_pull_timeout_func* **func**)

Description: 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.

unsigned int **gnutls_dtls_get_timeout** (*gnutls_session_t* **session**)

Description: 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.

5.5.1. 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`, `gnutls_record_recv`, 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`, 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`, either before the `poll` system call, or after a call to `gnutls_record_recv`. Data queued by `gnutls_record_send` (when interrupted) can be discarded using `gnutls_record_discard_queued`.

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.

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` function with the `GNUTLS_NONBLOCK` flag set (see section 5.3).

Datagram TLS protocol

When in non-blocking mode the function, the `gnutls_init` function must be called with the `GNUTLS_NONBLOCK` flag set (see section 5.3).

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` 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` must be called to decide which direction to check prior to restoring a function call.

int **gnutls_record_get_direction** (*gnutls_session_t* **session**)

Description: `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` through a multi-plexer, to be able to handle properly the DTLS handshake retransmission timers, the function `gnutls_dtls_get_timeout` should be used to estimate when to call `gnutls_handshake` if no data have been received.

5.5.2. 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 [7] 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`. 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` function call with error code `GNUTLS_E_PUSH_ERROR`.

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 subsection 2.6.8 TLS extension. This can be enabled by setting the GNUTLS_ENABLE_FALSE_START flag on `gnutls_init`.

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`, and the `gnutls_handshake` function will return early, allowing the server to send data earlier.

void **gnutls_transport_set_fastopen** (*gnutls_session_t* **session**, *int* **fd**, *struct sockaddr ** **connect_addr**, *socklen_t* **connect_addrlen**, *unsigned int* **flags**)

Description: 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.*

5.5.3. 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`. The client should call this function before calling `gnutls_handshake` and after calling `gnutls_session_set_data`.

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`. Note that this option only has effect on server.
2. Enable anti-replay measure. See subsection 5.5.4 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`. After receiving the EndOfEarlyData handshake message, the server can start retrieving the received data with `gnutls_record_recv_early_data`. You can call the function either after the handshake is complete, or through a handshake hook (`gnutls_handshake_set_hook_function`).

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`, right after calling `gnutls_session_set_data`.

After sending early data, to check whether the sent early data was accepted by the server, use `gnutls_session_get_flags` 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.

5.5.4. 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`.

The anti-replay mechanism shall be globally initialized with `gnutls_anti_replay_init`, and then attached to a session using `gnutls_anti_replay_enable`. It can be deinitialized with `gnutls_anti_replay_deinit`.

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` and `gnutls_anti_replay_set_ptr`.

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`. The cleanup can be implemented by iterating through the database entries and calling `gnutls_db_check_entry_expire_time`. 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(void)
{
  ...
  /* 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);
  ...
}
```

5.5.5. 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**, 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**. If successful the session should be initialized and associated with the cookie using **gnutls_dtls_prestate_set**, before proceeding to the handshake.

[4]A key of 128 bits or 16 bytes should be sufficient for this purpose.

int **gnutls_key_generate** (*gnutls_datum_t* * **key**, *unsigned int* **key_size**)

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**)

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**)

void **gnutls_dtls_prestate_set** (*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`.

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 [24].

void **gnutls_dtls_set_mtu** (*gnutls_session_t* **session**, *unsigned int* **mtu**)

unsigned int **gnutls_dtls_get_mtu** (*gnutls_session_t* **session**)

unsigned int **gnutls_dtls_get_data_mtu** (*gnutls_session_t* **session**)

5.5.6. DTLS and SCTP

Although DTLS can run under any reliable or unreliable layer, there are special requirements for SCTP according to [41]. We summarize the most important below, however for a full treatment we refer to [41].

- The MTU set via `gnutls_dtls_set_mtu` must be 2
  textasciicircum14.
- Replay detection must be disabled; use the flag `GNUTLS_NO_REPLAY_PROTECTION` with `gnutls_init`.
- Retransmission of messages must be disabled; use `gnutls_dtls_set_timeouts` 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.
- During a rehandshake, the caching of messages with unknown epoch is not handled by GnuTLS; this must be implemented in a special pull function.

5.6. 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**)

Description: 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.

In GnuTLS 3.5.0 and later it is recommended to use `gnutls_session_set_verify_cert` 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`, and then using `gnutls_certificate_verify_peers3` from it. See section 3.1 for more information.

void **gnutls_handshake_set_timeout** (*gnutls_session_t* **session**, *unsigned int* **ms**)

Description: 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().

void **gnutls_session_set_verify_cert** (*gnutls_session_t* **session**, *const char ** **hostname**, *unsigned* **flags**)

int **gnutls_certificate_verify_peers3** (*gnutls_session_t* **session**, *const char ** **hostname**, *unsigned int ** **status**)

5.7. 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` to check whether the error codes returned by these functions are fatal for the protocol or can be ignored.

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.

The `gnutls_record_check_pending` 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` reports no data to be read, `poll` should be called to check for data in the network buffers.

int **gnutls_record_get_direction** (*gnutls_session_t* **session**)

Once a TLS or DTLS session is no longer needed, it is recommended to use `gnutls_bye` 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` function.

ssize_t **gnutls_record_send** (*gnutls_session_t* **session**, *const void * * **data**, *size_t* **data_size**)

Description: 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**)

Description: 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**)

Description: 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.

ssize_t **gnutls_record_recv_seq** (*gnutls_session_t* **session**, *void* * **data**, *size_t* **data_size**, *unsigned char* * **seq**)

Description: 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**.

5.8. Buffered data transfer

Although `gnutls_record_send` 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.

size_t **gnutls_record_check_pending** (*gnutls_session_t* **session**)

Description: 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_bye** (*gnutls_session_t* **session**, *gnutls_close_request_t* **how**)

Description: 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**)

Description: 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.

5.9. 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` and `gnutls_record_recv`.

void **gnutls_record_cork** (*gnutls_session_t* **session**)

Description: 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.

int **gnutls_record_uncork** (*gnutls_session_t* **session**, *unsigned int* **flags**)

Description: 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.

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**)

Description: 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**)

Description: 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 section 2.4.

int **gnutls_alert_send** (*gnutls_session_t* **session**, *gnutls_alert_level_t* **level**, *gnutls_alert_description_t* **desc**)

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**)

Description: 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.

5.10. 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` or `gnutls_set_default_priority_append`), 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** (*gnutls_session_t* **session**)

int **gnutls_set_default_priority_append** (*gnutls_session_t* **session**, *const char ** **add_prio**, *const char *** **err_pos**, *unsigned* **flags**)

int **gnutls_priority_set_direct** (*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** (*gnutls_priority_t ** **priority_cache**, *const char ** **priorities**, *const char *** **err_pos**, *unsigned* **flags**)

int **gnutls_priority_init** (*gnutls_priority_t ** **priority_cache**, *const char ** **priorities**, *const char *** **err_pos**)

int **gnutls_priority_set** (*gnutls_session_t* **session**, *gnutls_priority_t* **priority**)

void **gnutls_priority_deinit** (*gnutls_priority_t* **priority_cache**)

Using Priority Strings

A priority string string may contain a single initial keyword such as in Table 5.4 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).

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 5.5; 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.

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 5.6 and Table 5.7.

Finally the ciphersuites enabled by any priority string can be listed using the `gnutls-cli` application (see ??), or by using the priority functions as in subsection 6.5.3.

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"
```

[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 5.11 for the acceptable security levels.

5.11. 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 5.8 we present some correspondence between key sizes of symmetric algorithms and public key algorithms based on [3]. Those can be used to generate certificates with appropriate key sizes as well as select parameters for Diffie-Hellman and SRP authentication.

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
*textasciicircum*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 [1] 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** (*gnutls_pk_algorithm_t* **algo**, *gnutls_sec_param_t* **param**)

Description: `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).`

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** (*gnutls_sec_param_t* **param**)

The following functions will set the minimum acceptable group size for Diffie-Hellman and SRP *gnutls_sec_param_t* **gnutls_pk_bits_to_sec_param** (*gnutls_pk_algorithm_t* **algo**, *unsigned int* **bits**)

Description: 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.

authentication.

void **gnutls_dh_set_prime_bits** (*gnutls_session_t* **session**, *unsigned int* **bits**)

void **gnutls_srp_set_prime_bits** (*gnutls_session_t* **session**, *unsigned int* **bits**)

5.12. Advanced topics

5.12.1. 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` 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 advertised 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(void)
{
  ...

  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**)

Description: 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.

5.12.2. 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 subsection 2.6.3) 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`.

int **gnutls_session_get_data2** (*gnutls_session_t* **session**, *gnutls_datum_t ** **data**)

int **gnutls_session_set_data** (*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**)

Description: `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.`

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` for expiration.

int **gnutls_session_get_id2** (*gnutls_session_t* **session**, *gnutls_datum_t* * **session_id**)

Description: 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.

void **gnutls_db_set_retrieve_function** (*gnutls_session_t* **session**, *gnutls_db_retr_func* **retr_func**)

void **gnutls_db_set_store_function** (*gnutls_session_t* **session**, *gnutls_db_store_func* **store_func**)

void **gnutls_db_set_ptr** (*gnutls_session_t* **session**, *void* * **ptr**)

void **gnutls_db_set_remove_function** (*gnutls_session_t* **session**, *gnutls_db_remove_func* **rem_func**)

int **gnutls_db_check_entry** (*gnutls_session_t* **session**, *gnutls_datum_t* **session_entry**)

Deprecated: This function is deprecated.

A server supporting session tickets must generate ticket encryption and authentication keys using `gnutls_session_ticket_key_generate`. Those keys should be associated with the GnuTLS session using `gnutls_session_ticket_enable_server`.

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` 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**)

Description: 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.

int **gnutls_session_ticket_key_generate** (*gnutls_datum_t ** **key**)

Description: Generate a random key to encrypt security parameters within SessionTicket.

Returns: On success, **GNUTLS_E_SUCCESS** (0) is returned, or an error code.

int **gnutls_session_resumption_requested** (*gnutls_session_t* **session**)

Description: 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`. 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`.

int **gnutls_session_ticket_send** (*gnutls_session_t* **session**, *unsigned* **nr**, *unsigned* **flags**)

Description: 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.

5.12.3. 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.

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**)

Description: 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.

In addition to the above the `gnutls_store_commitment` can be used to implement a key-pinning architecture as in [11]. This provides a way for web server to commit on a public key that is *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**)

Description: 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.

not yet active.

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**)

Description: 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.

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** (*gnutls_tdb_t * * **tdb**)

void **gnutls_tdb_deinit** (*gnutls_tdb_t* **tdb**)

void **gnutls_tdb_set_verify_func** (*gnutls_tdb_t* **tdb**, *gnutls_tdb_verify_func* **verify**)

void **gnutls_tdb_set_store_func** (*gnutls_tdb_t* **tdb**, *gnutls_tdb_store_func* **store**)

void **gnutls_tdb_set_store_commitment_func** (*gnutls_tdb_t* **tdb**, *gnutls_tdb_store_commitment_func* **cstore**)

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_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**)

*const char ** **dane_strerror** (*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.

In order to generate a DANE TLSA entry to use in a DNS server you may use danetool (see subsection 3.2.8).

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**)

Description: 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 verification. 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 advertised 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.

5.12.4. 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`.

The following paragraphs explain how to safely use the rehandshake process.

Client side

According to the TLS specification a client may initiate a rehandshake at any time. That can be achieved by calling `gnutls_handshake` 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` will return GNUTLS_E_REHANDSHAKE in the client, instructing it to call `gnutls_handshake`. 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`, 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` call. In that case the re-authentication will happen in the call of `gnutls_record_recv` that received the reauthentication request.

unsigned **gnutls_safe_renegotiation_status** (*gnutls_session_t* **session**)

Description: 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.

Server side

A server which wants to instruct the client to re-authenticate, should call `gnutls_rehandshake` 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` and the discussion in subsection 2.6.5). 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.

5.12.5. 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`. 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`, and only if a client has advertised 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`.

A client receiving a re-authentication request will "see" the error code `GNUTLS_E_REAUTH_REQUEST` at `gnutls_record_recv`. At this point, it should also call `gnutls_reauth`.

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` call. In *int* **gnutls_rehandshake** (*gnutls_session_t* **session**)

Description: 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.

that case the re-authentication will happen in the call of `gnutls_record_recv` that received the reauthentication request.

5.12.6. 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 [13].

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`, `gnutls_anon_set_server_known_dh_params`, or `gnutls_psk_set_server_known_dh_params`, 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** (*gnutls_certificate_credentials_t* **res**, *gnutls_sec_param_t* **sec_param**)

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

int **gnutls_anon_set_server_known_dh_params** (*gnutls_anon_server_credentials_t* **res**, *gnutls_sec_param_t* **sec_param**)

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

int **gnutls_psk_set_server_known_dh_params** (*gnutls_psk_server_credentials_t* **res**, *gnutls_sec_param_t* **sec_param**)

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

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 [13] 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** (*gnutls_dh_params_t* **dparams**, *unsigned int* **bits**)

int **gnutls_dh_params_import_pkcs3** (*gnutls_dh_params_t* **params**, *const gnutls_datum_t* * **pkcs3_params**, *gnutls_x509_crt_fmt_t* **format**)

void **gnutls_certificate_set_dh_params** (*gnutls_certificate_credentials_t* **res**, *gnutls_dh_params_t* **dh_params**)

Deprecated: This function is unnecessary and discouraged on GnuTLS 3.6.0 or later. Since 3.6.0, DH parameters are negotiated following RFC7919.

5.12.7. 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`. 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**)

Description: 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.

For example, after establishing a TLS session using `gnutls_handshake`, 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`, 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`. That however is not functional under newer protocol versions.

5.12.8. 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 [43].

In GnuTLS you can extract a channel binding using the `gnutls_session_channel_binding` function. Currently only the following types are supported:

- `GNUTLS_CB_TLS_UNIQUE`: corresponds to the `tls-unique` channel binding for TLS defined in [4]
- `GNUTLS_CB_TLS_EXPORTER`: corresponds to the `tls-exporter` channel binding for TLS defined in [42]

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");
    }
}
```

5.12.9. 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 5.10). 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.

5.12.10. 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.

enum **gnutls_init_flags_t**:

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_DEFAULT_EXTENSIONS	Do not enable any TLS extensions by default such as session tickets and OCSP certificate status request (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 advertised 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_X**25519 if all of them are enabled.
GNUTLS_POST_HANDSHAKE_AUTH	Enable post handshake authentication for server and client. When set and a server requests authentication after handshake **GNUTLS_E_REAUTH_REQUEST** will be returned by **gnutls_record_recv()**. A client

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 5.3.: Key exchange algorithms and the corresponding credential types.

Keyword	Description
@KEYWORD	Means that a compile-time specified system configuration file (see **??**) 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 profile is set to GNUTLS_PROFILE_HIGH (128-bits).
SECURE256	Currently alias for SECURE192. This option, will enable ciphers which use a 256-bit key but, due to limitations of the TLS protocol, the overall security level will be 192-bits (the security level depends on more factors than cipher key size).
SUITEB128	Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 128 bit security level, as well as the enabling of the corresponding verification profile.
SUITEB192	Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 192 bit security level, as well as the enabling of the corre-

Type	Keywords
Ciphers	Examples are AES-128-GCM, AES-256-GCM, AES-256-CBC, GOST28147-TC26Z-CNT; see also Table 2.1 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, RSA-PSK, RSA-EXPORT, 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-DTLS0.9, 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-EdDSA-Ed25519, SIGN-EdDSA-Ed448, 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-SECP192R1, GROUP-SECP224R1, GROUP-SECP256R1, GROUP-SECP384R1, GROUP-SECP521R1, GROUP-X25519, GROUP-X448, GROUP-GC256B, GROUP-GC512A, 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 available, restricting the groups to finite fields (DH), GOST curves and generic elliptic curves.
Elliptic curves (legacy)	CURVE-SECP192R1, CURVE-SECP224R1, CURVE-SECP256R1, CURVE-SECP384R1, CURVE-SECP521R1, CURVE-X25519, and CURVE-X448. Catch all which enables all curves from NORMAL priority is CURVE-ALL. Note that the

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 [20] 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_SHUFFLE_EXTENSIONS	will prevent randomizing the order of ClientHello extensions. By default, those extensions are randomized to make fingerprinting harder.
%NO_STATUS_REQUEST	will prevent sending of the TLS status_request extension in client side.
%NO_TICKETS	will prevent the advertizing of the TLS session ticket extension.
%NO_TICKETS_TLS12	will prevent the advertizing of the TLS session ticket extension in TLS 1.2. This is implied by the PFS keyword.
%NO_SESSION_HASH	will prevent the advertizing the TLS extended master secret (session hash) extension.
%FORCE_SESSION_HASH	negotiate the TLS extended master secret (session hash) extension. Specifying both %NO_SESSION_HASH and %FORCE_SESSION_HASH is not supported, and the behavior is undefined.
%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. 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. See RFC7507 for details.
%DISABLE_TLS13_COMPAT_MODE	will disable TLS 1.3 middlebox compatibility mode (RFC8446, Appendix D.4) for non-compliant middleboxes.
%VERIFY_ALLOW_BROKEN	will allow signatures with known to be broken algorithms (such as MD5 or SHA1) in certificate chains.
%VERIFY_ALLOW_SIGN_RSA_MD5	will allow RSA-MD5 signatures in certificate chains.
%VERIFY_ALLOW_SIGN_WITH_SHA1	will allow signatures with SHA1 hash algorithm in certificate chains.
%VERIFY_DISABLE_CRL_CHECKS	will disable CRL or OCSP checks in the verification of the certificate chain.
%VERIFY_ALLOW_X509_V1_CA_CRT	will allow V1 CAs in chains.

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 5.8.: Key sizes and security parameters.

`enum` **dane_verify_status_t:**

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.`

Table 5.9.: The DANE verification status flags.

6 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.

6.1. 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.

6.1.1. 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 subsection 6.3.7.

```
/* 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;
}
```

6.1.2. 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;
}
```

6.1.3. 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 subsection 6.1.1 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;
}
```

6.1.4. 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;
}
```

6.1.5. 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 attack\n");
                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;
}
```

6.2. Server examples

This section contains examples of TLS and SSL servers, using GnuTLS.

6.2.1. 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;
}
```

6.2.2. 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;
}
```

6.3. More advanced client and servers

This section has various, more advanced topics in client and servers.

6.3.1. 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;
}
```

6.3.2. 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;
}
```

6.3.3. 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`.

```
/* 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;
```

```
}
```

6.3.4. 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 = gnutls_calloc(cert_chain_length, sizeof(*cert));
        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);
        }

        for (i = 0; i < cert_chain_length; i++) {
                gnutls_x509_crt_deinit(cert[i]);
        }
        gnutls_free(cert);

        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;
}
```

6.3.5. 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;
}
```

6.3.6. 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;
}
```

6.3.7. 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 subsection 6.1.1 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;
}
```

6.3.8. 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;
}
```

6.3.9. 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;
}
```

6.3.10. 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;
}
```

6.3.11. 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;
}
```

6.3.12. 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);
```

```
}
```

6.3.13. 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);
}
```

6.4. 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);
                        if (!hostname) {
                                fprintf(stderr,
                                        "error: cannot allocate memory\n");
                                exit(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;
}
```

6.5. Miscellaneous examples

6.5.1. 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));
        }
}
```

6.5.2. 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);
        }
}
```

6.5.3. 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;
}
```

6.5.4. 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;
}
```

7

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 9.5), 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.

7.1. 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 Table 7.1. 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.

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.

[1] See @urefhttps://www.lysator.liu.se/ñisse/nettle/.

int **gnutls_aead_cipher_init** (*gnutls_aead_cipher_hd_t* * **handle**, *gnutls_cipher_algorithm_t* **cipher**, *const gnutls_datum_t* * **key**)

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**)

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**)

void **gnutls_aead_cipher_deinit** (*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**)

Description: `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.`

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].

[2]See the nettle manual https://www.lysator.liu.se/~nisse/nettle/nettle.html

int **gnutls_cipher_init** (*gnutls_cipher_hd_t* * **handle**, *gnutls_cipher_algorithm_t* **cipher**, *const gnutls_datum_t* * **key**, *const gnutls_datum_t* * **iv**)

int **gnutls_cipher_encrypt2** (*gnutls_cipher_hd_t* **handle**, *const void* * **ptext**, *size_t* **ptext_len**, *void* * **ctext**, *size_t* **ctext_len**)

int **gnutls_cipher_decrypt2** (*gnutls_cipher_hd_t* **handle**, *const void* * **ctext**, *size_t* **ctext_len**, *void* * **ptext**, *size_t* **ptext_len**)

void **gnutls_cipher_set_iv** (*gnutls_cipher_hd_t* **handle**, *void* * **iv**, *size_t* **ivlen**)

void **gnutls_cipher_deinit** (*gnutls_cipher_hd_t* **handle**)

int **gnutls_cipher_add_auth** (*gnutls_cipher_hd_t* **handle**, *const void* * **ptext**, *size_t* **ptext_size**)

int **gnutls_cipher_tag** (*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.

7.2. Public key algorithms

Public key cryptography algorithms such as RSA, DSA and ECDSA, are accessed using the abstract key API in section 4.1. This is a high level API with the advantage of transparently handling keys stored in memory and keys present in smart cards.

int **gnutls_privkey_init** (*gnutls_privkey_t* * **key**)

int **gnutls_privkey_import_url** (*gnutls_privkey_t* **key**, *const char* * **url**, *unsigned int* **flags**)

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**)

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**)

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**)

void **gnutls_privkey_deinit** (*gnutls_privkey_t* **key**)

int **gnutls_pubkey_init** (*gnutls_pubkey_t* * **key**)

int **gnutls_pubkey_import_url** (*gnutls_pubkey_t* **key**, *const char* * **url**, *unsigned int* **flags**)

int **gnutls_pubkey_import_x509** (*gnutls_pubkey_t* **key**, *gnutls_x509_crt_t* **crt**, *unsigned int* **flags**)

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**)

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**)

void **gnutls_pubkey_deinit** (*gnutls_pubkey_t* **key**)

Keys stored in memory can be imported using functions like `gnutls_privkey_import_x509_raw`, while keys on smart cards or HSMs should be imported using their PKCS#11 URL with `gnutls_privkey_import_url`.

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`), or better with the targeted to structures functions such as `gnutls_privkey_set_pin_function`.

7.2.1. 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` or with the more advanced `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**)

Description: 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 PKCS#8 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.

7.3. 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** (*gnutls_pkcs7_t ** **pkcs7**)

void **gnutls_pkcs7_deinit** (*gnutls_pkcs7_t* **pkcs7**)

int **gnutls_pkcs7_export2** (*gnutls_pkcs7_t* **pkcs7**, *gnutls_x509_crt_fmt_t* **format**, *gnutls_datum_t ** **out**)

int **gnutls_pkcs7_import** (*gnutls_pkcs7_t* **pkcs7**, *const gnutls_datum_t ** **data**, *gnutls_x509_crt_fmt_t* **format**)

int **gnutls_pkcs7_print** (*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` function is the data signing function.

int **gnutls_pkcs7_verify_direct** (*gnutls_pkcs7_t* **pkcs7**, *gnutls_x509_crt_t* **signer**, *unsigned* **idx**, *const gnutls_datum_t ** **data**, *unsigned* **flags**)

int **gnutls_pkcs7_verify** (*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**)

Description: 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.

@showenumdescgnutls_pkcs7_sign_flags,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** (*gnutls_pkcs7_t* **pkcs7**)

int **gnutls_pkcs7_get_signature_info** (*gnutls_pkcs7_t* **pkcs7**, *unsigned* **idx**, *gnutls_pkcs7_signature_info_st ** **info**)

int **gnutls_pkcs7_get_crt_count** (*gnutls_pkcs7_t* **pkcs7**)

int **gnutls_pkcs7_get_crt_raw2** (*gnutls_pkcs7_t* **pkcs7**, *unsigned* **indx**, *gnutls_datum_t ** **cert**)

int **gnutls_pkcs7_get_crl_count** (*gnutls_pkcs7_t* **pkcs7**)

int **gnutls_pkcs7_get_crl_raw2** (*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** (*gnutls_pkcs7_t* **pkcs7**, *const gnutls_datum_t ** **crt**)

int **gnutls_pkcs7_set_crt** (*gnutls_pkcs7_t* **pkcs7**, *gnutls_x509_crt_t* **crt**)

int **gnutls_pkcs7_set_crl_raw** (*gnutls_pkcs7_t* **pkcs7**, *const gnutls_datum_t ** **crl**)

int **gnutls_pkcs7_set_crl** (*gnutls_pkcs7_t* **pkcs7**, *gnutls_x509_crl_t* **crl**)

7.4. 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 Table 7.2. 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.

int **gnutls_hmac_init** (*gnutls_hmac_hd_t* * **dig**, *gnutls_mac_algorithm_t* **algorithm**, *const void* * **key**, *size_t* **keylen**)

int **gnutls_hmac** (*gnutls_hmac_hd_t* **handle**, *const void* * **ptext**, *size_t* **ptext_len**)

void **gnutls_hmac_output** (*gnutls_hmac_hd_t* **handle**, *void* * **digest**)

void **gnutls_hmac_deinit** (*gnutls_hmac_hd_t* **handle**, *void* * **digest**)

unsigned **gnutls_hmac_get_len** (*gnutls_mac_algorithm_t* **algorithm**)

int **gnutls_hmac_fast** (*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 Table 7.3.

int **gnutls_hash_init** (*gnutls_hash_hd_t* * **dig**, *gnutls_digest_algorithm_t* **algorithm**)

int **gnutls_hash** (*gnutls_hash_hd_t* **handle**, *const void* * **ptext**, *size_t* **ptext_len**)

void **gnutls_hash_output** (*gnutls_hash_hd_t* **handle**, *void* * **digest**)

void **gnutls_hash_deinit** (*gnutls_hash_hd_t* **handle**, *void* * **digest**)

unsigned **gnutls_hash_get_len** (*gnutls_digest_algorithm_t* **algorithm**)

int **gnutls_hash_fast** (*gnutls_digest_algorithm_t* **algorithm**, *const void* * **ptext**, *size_t* **ptext_len**, *void* * **digest**)

int **gnutls_fingerprint** (*gnutls_digest_algorithm_t* **algo**, *const gnutls_datum_t* * **data**, *void* * **result**, *size_t* * **result_size**)

7.5. Random number generation

Access to the random number generator is provided using the `gnutls_rnd` function. It allows obtaining random data of various levels.

> *int* **gnutls_rnd** (*gnutls_rnd_level_t* **level**, *void ** **data**, *size_t* **len**)
>
> **Description:** 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.

See section 9.6 for more information on the random number generator operation.

7.6. 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 subsection 4.1.2.

> *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**)
>
> **Description:** 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.
>
> **Deprecated:** since 3.7.0 it is no longer possible to override cipher implementation
>
> **Returns:** **GNUTLS_E_SUCCESS** on success, otherwise a negative error code.

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**)

Description: 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.

Deprecated: since 3.7.0 it is no longer possible to override cipher implementation

Returns: **GNUTLS_E_SUCCESS** on success, otherwise a negative error code.

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**)

Description: 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.

Deprecated: since 3.7.0 it is no longer possible to override cipher implementation

Returns: **GNUTLS_E_SUCCESS** on success, otherwise a negative error code.

enum gnutls_cipher_algorithm_t:

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).
GNUTLS_CIPHER_AES_128_CCM_8	AES in CCM mode with 64-bit tag and 128-bit keys (AEAD).
GNUTLS_CIPHER_AES_256_CCM_8	AES in CCM mode with 64-bit tag and 256-bit keys (AEAD).
GNUTLS_CIPHER_CHACHA20_POLY1305	The Chacha20 cipher with the Poly1305 authenticator (AEAD).
GNUTLS_CIPHER_GOST28147_TC26Z_CFB	GOST 28147-89 (Magma) cipher in CFB mode with TC26 Z S-box.
GNUTLS_CIPHER_GOST28147_CPA_CFB	GOST 28147-89 (Magma) cipher in CFB mode with CryptoPro A S-box.
GNUTLS_CIPHER_GOST28147_CPB_CFB	GOST 28147-89 (Magma) cipher in CFB mode with CryptoPro B S-box.
GNUTLS_CIPHER_GOST28147_CPC_CFB	GOST 28147-89 (Magma) cipher in CFB mode with CryptoPro C S-box.
GNUTLS_CIPHER_GOST28147_CPD_CFB	GOST 28147-89 (Magma) cipher in CFB mode with CryptoPro D S-box.
GNUTLS_CIPHER_AES_128_CFB8	AES in CFB8 mode with 128-bit keys.
GNUTLS_CIPHER_AES_192_CFB8	AES in CFB8 mode with 192-bit keys.
GNUTLS_CIPHER_AES_256_CFB8	AES in CFB8 mode with 256-bit keys.
GNUTLS_CIPHER_AES_128_XTS	AES in XTS mode with 128-bit key + 128bit tweak key.
GNUTLS_CIPHER_AES_256_XTS	AES in XTS mode with 256-bit key + 256bit tweak key. Note that the XTS ciphers are message oriented. The whole message needs to be provided with a single call, because cipher-stealing requires to know where the message actually terminates in order to be able to compute where the stealing occurs.

enum **gnutls_mac_algorithm_t**:

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).
GNUTLS_MAC_AES_CMAC_128	The AES-CMAC-128 MAC algorithm.
GNUTLS_MAC_AES_CMAC_256	The AES-CMAC-256 MAC algorithm.
GNUTLS_MAC_AES_GMAC_128	The AES-GMAC-128 MAC algorithm (requires nonce).
GNUTLS_MAC_AES_GMAC_192	The AES-GMAC-192 MAC algorithm (requires nonce).
GNUTLS_MAC_AES_GMAC_256	The AES-GMAC-256 MAC algorithm (requires nonce).
GNUTLS_MAC_GOST28147_TC26Z_-IMIT	The GOST 28147-89 working in IMIT mode with TC26 Z S-box.
GNUTLS_MAC_SHAKE_128	Reserved; unimplemented.
GNUTLS_MAC_SHAKE_256	Reserved; unimplemented.
GNUTLS_MAC_MAGMA_OMAC	GOST R 34.12-2015 (Magma) in OMAC (CMAC) mode.
GNUTLS_MAC_KUZNYECHIK_OMAC	GOST R 34.12-2015 (Kuznyechik) in OMAC (CMAC) mode.

Table 7.2.: The supported MAC and HMAC algorithms.

enum **gnutls_digest_algorithm_t**:

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.
GNUTLS_DIG_STREEBOG_256	GOST R 34.11-2001 (Streebog) algorithm, 256 bit.
GNUTLS_DIG_STREEBOG_512	GOST R 34.11-2001 (Streebog) algorithm, 512 bit.
GNUTLS_DIG_SHAKE_128	Reserved; unimplemented.
GNUTLS_DIG_SHAKE_256	Reserved; unimplemented.

Table 7.3.: The supported hash algorithms.

enum **gnutls_rnd_level_t**:

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.

Table 7.4.: The random number levels.

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**)

Description: 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.

Deprecated: since 3.7.0 it is no longer possible to override cipher implementation

Returns: **GNUTLS_E_SUCCESS** on success, otherwise a negative error code.

8 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.

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.

gnutls-cli help/usage (@option-?)

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]

None:

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
       --tofu                 Enable trust on first use authentication
       --strict-tofu          Fail to connect if a certificate is unknown or a known certificate has changed
       --dane                 Enable DANE certificate verification (DNSSEC)
       --local-dns            Use the local DNS server for DNSSEC resolving
       --ca-verification      Enable CA certificate verification
                                - enabled by default
                                - disabled as '--no-ca-verification'
       --ocsp                 Enable OCSP certificate verification
   -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            an alias for the 'starttls-proto' option
    --starttls-proto=str   The application protocol to be used to obtain the server's certificate (https, ft
                             - prohibits the option 'starttls'
    --starttls-name=str    The hostname presented to the application protocol for STARTTLS (for smtp, xmpp,
                             - prohibits the option 'starttls'
                             - requires the option 'starttls-proto'
-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 PEM format
    --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 verification
    --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
    --compress-cert=str    Compress certificate
-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
       --list-config          Reports the configuration of the 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
       --ca-auto-retrieve     Enable automatic retrieval of missing CA certificates
       --attime=str           Perform validation at the timestamp instead of the system time

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.

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.

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.NUMBER argument. Specifies the debug level.

tofu option.

This is the “enable trust on first use authentication” option. 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 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 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 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 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 ArgumentType.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 ArgumentType.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, section 8.

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 ArgumentType.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.

starttls-name option.

This is the "the hostname presented to the application protocol for starttls (for smtp, xmpp, lmtp)" option. This option takes a ArgumentType.STRING argument.

This option has some usage constraints. It:

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

Specify the hostname presented to the application protocol for STARTTLS.

save-ocsp-multi option.

This is the "save all ocsp responses provided by the peer in this file" option. This option takes a ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.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 ArgumentType.STRING argument. This option will set and enable the Application Layer Protocol Negotiation (ALPN) in the TLS protocol.

compress-cert option.

This is the “compress certificate” option. This option takes a ArgumentType.STRING argument. This option sets a supported compression method for certificate compression.

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
textasciicircum<cmd>
textasciicircum” option. Enable inline commands of the form
textasciicircum<cmd>
textasciicircum. 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 ArgumentType.STRING argument. Change the default delimiter (
textasciicircum) 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 ArgumentType.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 ArgumentType.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.

ca-auto-retrieve option.

This is the "enable automatic retrieval of missing ca certificates" option. This option enables the client to automatically retrieve the missing intermediate CA certificates in the certificate chain, based on the Authority Information Access (AIA) extension.

attime option.

This is the "perform validation at the timestamp instead of the system time" option. This option takes a ArgumentType.STRING argument "`timestamp`". timestamp is an instance in time encoded as Unix time or in a human readable timestring such as "29 Feb 2004", "2004-02-29". Full documentation available at <https://www.gnu.org/software/coreutils/manual/html_node/Date-input-formats.html> or locally via info '(coreutils) date invocation'.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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)-(AES-256-GCM)
- 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-SHA512
```

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;token=Test"

```

```
Object 0:
        URL: pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=client;type=cert
        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=client;type=cert"
$ MYKEY="pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=client;type=private"
$ 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.

Invoking gnutls-serv

Server program that listens to incoming TLS connections.

gnutls-serv help/usage (@option-?)

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>] ]...

None:

   -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
       --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:
                                  1 to 2147483648
       --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
       --crlf                 Do not replace CRLF by LF in Echo server mode
   -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
       --compress-cert=str    Compress certificate
   -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
       --x509certfile=str     X.509 Certificate file or PKCS #11 URL to use
       --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'
       --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
       --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
       --timeout=num          The timeout period for server
       --attime=str           Perform validation at the timestamp instead of the system time

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.

Server program that listens to incoming TLS connections.

Please send bug reports to:  <bugs@gnutls.org>
```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.NUMBER argument. Specifies the debug level.

sni-hostname option.

This is the “server’s hostname for server name extension” option. This option takes a ArgumentType.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 ArgumentType.STRING argument. 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.

compress-cert option.

This is the “compress certificate” option. This option takes a ArgumentType.STRING argument. This option sets a supported compression method for certificate compression.

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 ArgumentType.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 ArgumentType.STRING argument. 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 ArgumentType.STRING argument. 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, section 8.

x509dsacertfile option.

This is an alias for the `x509certfile` option, section 8.

x509ecckeyfile option.

This is an alias for the `x509keyfile` option, section 8.

x509ecccertfile option.

This is an alias for the `x509certfile` option, section 8.

rawpkkeyfile option.

This is the "private key file (pkcs #8 or pkcs #12) or pkcs #11 url to use" option. This option takes a ArgumentType.STRING argument. 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 ArgumentType.STRING argument.

This option has some usage constraints. It:

- 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 ArgumentType.STRING argument. 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 ArgumentType.FILE argument. This will override the default options in /etc/gnutls/pkcs11.conf

attime option.

This is the "perform validation at the timestamp instead of the system time" option. This option takes a ArgumentType.STRING argument "`timestamp`". timestamp is an instance in time encoded as Unix time or in a human readable timestring such as "29 Feb 2004", "2004-02-29".

Full documentation available at <https://www.gnu.org/software/coreutils/manual/html_node/Date-input-formats.html> or locally via info '(coreutils) date invocation'.

version option (-v).

This is the "output version information and exit" option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is 'v', a simple version. The 'c' mode will print copyright information and 'n' will print the full copyright notice.

help option (-h).

This is the "display extended usage information and exit" option. Display usage information and exit.

more-help option (-!).

This is the "extended usage information passed thru pager" option. Pass the extended usage information through a pager.

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
```

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.

gnutls-cli-debug help/usage (@option-?)

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>] ]... [hostname]

None:

   -d, --debug=num            Enable debugging
                                - it must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
   -p, --port=num             The port to connect to
                                - it must be in the range:
                                  0 to 65536
       --app-proto            an alias for the 'starttls-proto' option
       --starttls-proto=str   The application protocol to be used to obtain the server's certificate (https, ft
       --attime=str           Perform validation at the timestamp instead of the system time

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.

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.

Please send bug reports to:  <bugs@gnutls.org>

```

debug option (-d).

This is the “enable debugging” option. This option takes a ArgumentType.NUMBER argument. Specifies the debug level.

app-proto option.

This is an alias for the `starttls-proto` option, section 8.

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 ArgumentType.STRING argument. Specify the application layer protocol for STARTTLS. If the protocol is supported, gnutls-cli will proceed to the TLS negotiation.

attime option.

This is the “perform validation at the timestamp instead of the system time” option. This option takes a ArgumentType.STRING argument “`timestamp`”. timestamp is an instance in time encoded as Unix time or in a human readable timestring such as ”29 Feb 2004”, ”2004-02-29”. Full documentation available at <https://www.gnu.org/software/coreutils/manual/html_node/Date-input-formats.html> or locally via info ’(coreutils) date invocation’.

version option (-v).

This is the “output version information and exit” option. This option takes a ArgumentType.KEYWORD argument. Output version of program and exit. The default mode is ‘v’, a simple version. The ‘c’ mode will print copyright information and ‘n’ will print the full copyright notice.

help option (-h).

This is the “display extended usage information and exit” option. Display usage information and exit.

more-help option (-!).

This is the “extended usage information passed thru pager” option. Pass the extended usage information through a pager.

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
```

9

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.

9.1. The TLS Protocol

The main use case for the TLS protocol is shown in Figure 9.1. 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.

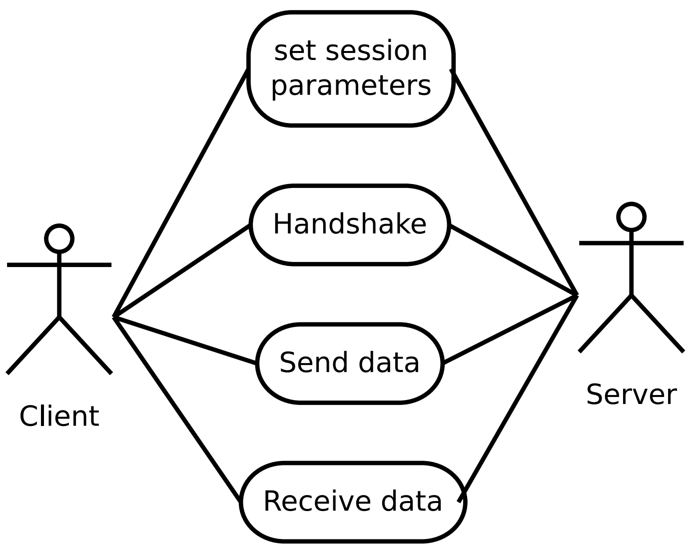

Figure 9.1.: TLS protocol use case.

9.2. 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 9.2.

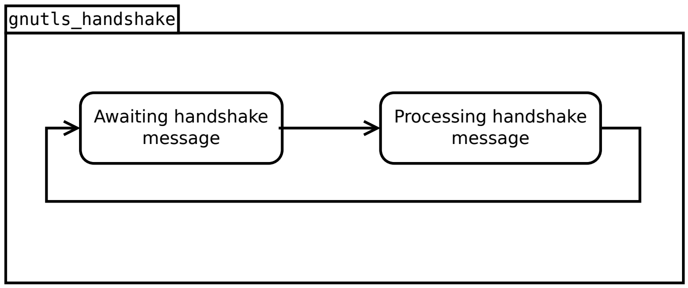

Figure 9.2.: GnuTLS handshake state machine.

Also the way the input is processed varies per ciphersuite. Several implementations of the internal handlers are available and `gnutls_handshake` 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 9.3.

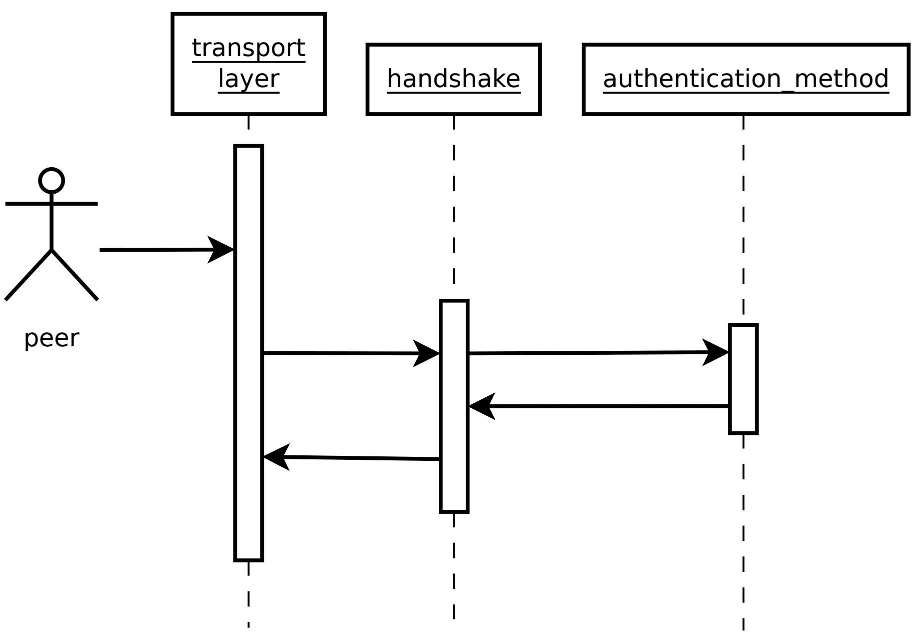

Figure 9.3.: GnuTLS handshake process sequence.

9.3. 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.

9.4. 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);
```

[1] such as the `gnutls_certificate_credentials_t` structures

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` or `gnutls_ext_register`.

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` or `gnutls_ext_register`.

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 D. 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 [37]. 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_size)
```

```
{
   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`, or `gnutls_supplemental_register` at some point in your program.

9.5. 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 9.4.

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]

[2] See https://lists.gnu.org/archive/html/gnutls-devel/2011-02/msg00079.html.

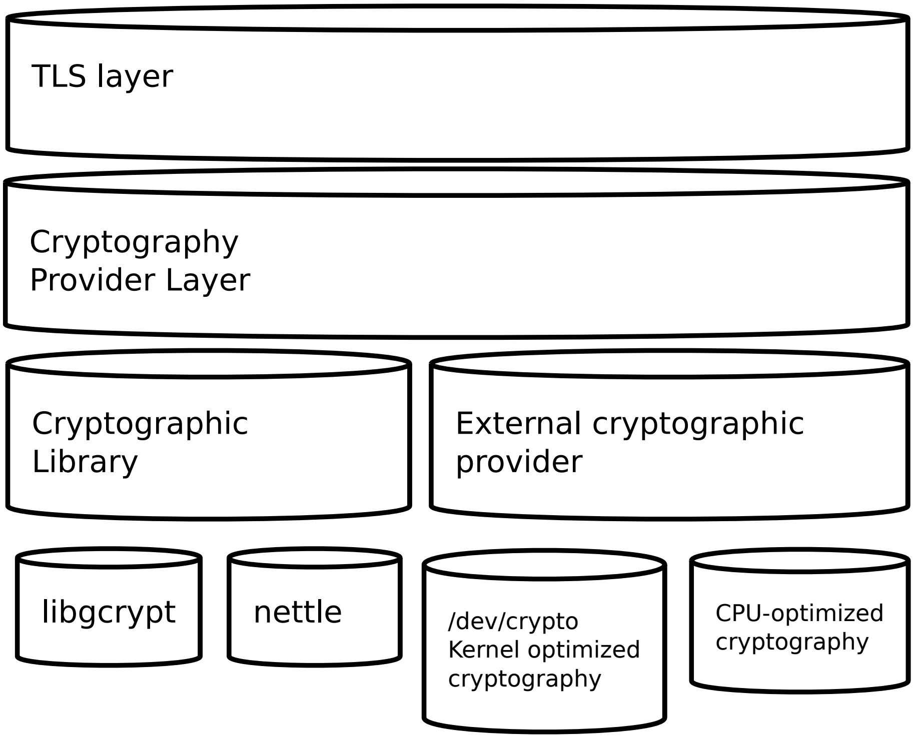

Figure 9.4.: GnuTLS cryptographic back-end design.

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

[3] Check https://home.gna.org/cryptodev-linux/ for the Linux kernel implementation of `/dev/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`: To register a cipher algorithm.
- `gnutls_crypto_register_aead_cipher`: To register an AEAD cipher algorithm.
- `gnutls_crypto_register_mac`: To register a MAC algorithm.
- `gnutls_crypto_register_digest`: To register a hash algorithm.

Those registration functions will only replace the specified algorithm and leave the rest of subsystem intact.

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 4.

9.6. 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 reseeded 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. 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 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 [17]. 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.

9.7. 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 Table 9.1, 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` 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` which can switch to alternative modes as in Table 9.1.

enum gnutls_fips_mode_t:

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_FIPS**140_LAX, it allows forbidden operations; any use of them results to a message to the audit callback functions.

Table 9.1.: 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);
```

Service indicator

The above restrictions may not cover all the requirements in every usage context, and as the FIPS140 standard evolves (like FIPS140-3), GnuTLS may not be able to add new restrictions without breaking compatibility.

Therefore an additional set of API functions is provided to communicate with the user whether any approved mode of operations is performed within a given context.

int **gnutls_fips140_context_init** (*gnutls_fips140_context_t* * **context**)

void **gnutls_fips140_context_deinit** (*gnutls_fips140_context_t* **context**)

int **gnutls_fips140_push_context** (*gnutls_fips140_context_t* **context**)

int **gnutls_fips140_pop_context** (**void**)

The `gnutls_fips140_context_t` represents the FIPS140-2 mode of operation. It can be attached to the current execution thread with `gnutls_fips140_push_context` and its internal state will be updated until it is detached with `gnutls_fips140_pop_context`. Afterwards `gnutls_fips140_get_operation_state` allows the user to examine whether any approved (or non-approved) security function is invoked.

gnutls_fips140_operation_state_t **gnutls_fips140_get_operation_state** (*gnutls_fips140_context_t* **context**)

Description: `Get the previous operation state of` **context** `in terms of FIPS.`

Returns: `a` *gnutls_fips*`140_operation_state_t`

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`.

- 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` has to be used to achieve the same functionality as described in subsection 5.5.1.

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 has to be used, as described in subsection 5.5.1
gnutls_session_get_server_random, gnutls_session_get_client_random	They are replaced by the safer function gnutls_session_get_random
gnutls_session_get_master_secret	Replaced by the keying material exporters discussed in subsection 5.12.7
gnutls_transport_set_global_errno	Replaced by using the system's errno facility or gnutls_transport_set_errno.
gnutls_x509_privkey_verify_data	Replaced by gnutls_pubkey_verify_data2.
gnutls_certificate_verify_peers	Replaced by gnutls_certificate_verify_peers2.
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.
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 is provided and is functionally equivalent and safer to use.
gnutls_pubkey_verify_data	The function gnutls_pubkey_verify_data2 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` 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`
`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`.
`gnutls_pubkey_verify_data`,	`gnutls_pubkey_verify_data2`.
`gnutls_x509_crt_get_verify_algorithm`,	No replacement; a similar function is `gnutls_x509_crt_get_signature_algorithm`.
`gnutls_pubkey_get_verify_algorithm`,	No replacement; a similar function is `gnutls_pubkey_get_preferred_hash_algorithm`.
`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`.
`gnutls_sign_callback_get`, `gnutls_sign_callback_set`	`gnutls_privkey_import_ext3`
`gnutls_x509_crt_verify_hash`	`gnutls_pubkey_verify_hash2`
`gnutls_x509_crt_verify_data`	`gnutls_pubkey_verify_data2`
`gnutls_privkey_sign_raw_data`	`gnutls_privkey_sign_hash` 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 [39] spec	The SRP key exchange is restricted to [39] 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`, `gnutls_x509_crl_sign`, `gnutls_x509_crq_sign`	These signing functions will no longer sign using SHA1, but with a secure hash algorithm.
`gnutls_certificate_set_ocsp_status_request_file`	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`.
The callback `gnutls_privkey_import_ext3` is not flexible enough for new signature algorithms such as RSA-PSS	It is replaced with `gnutls_privkey_import_ext4`
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` and `gnutls_reauth`.
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`.
`gnutls_pkcs11_privkey_generate3`, `gnutls_pkcs11_copy_secret_key`, `gnutls_pkcs11_copy_x509_privkey2`	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`, `gnutls_db_set_store_function`, `gnutls_db_set_remove_function`	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`.
`gnutls_session_get_data2`, `gnutls_session_get_data`	These functions may introduce a slight delay under TLS 1.3 for few milliseconds. Check output of `gnutls_session_get_flags` 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`

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.

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. 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 Linux kernel coding style[1].

  If you normally code using another coding standard, there is no problem, but you should run the indent-gnutls script found in the devel directory 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.

[1]see @urefhttps://www.kernel.org/doc/html/latest/process/coding-style.html.

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 9.7 for more information.

C

Supported 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_GOSTR341112_256_28147_CNT_IMIT	0xC1 0x02	TLS1.2

Table C.1.: The ciphersuites table

D
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.

Code	Name	Description
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.

Table D.1.: The error codes table

GNU Free Documentation License

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 noncommercially. 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.

Bibliography

[1] NIST Special Publication 800-57, Recommendation for Key Management - Part 1: General (Revised), March 2007.

[2] PKCS #11 Base Functionality v2.30: Cryptoki – Draft 4, July 2009.

[3] ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010), 2010.

[4] J. Altman, N. Williams, and L. Zhu. Channel bindings for TLS, July 2010. Available from https://www.ietf.org/rfc/rfc5929.

[5] R. J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2001.

[6] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport layer security (TLS) extensions, June 2003. Available from https://www.ietf.org/rfc/rfc3546.

[7] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. Tcp fast open. RFC 7413, December 2014.

[8] 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. RFC 5280 (Proposed Standard), May 2008. Available from https://www.ietf.org/rfc/rfc5280.

[9] T. Dierks and E. Rescorla. The TLS Protocol Version 1.2, August 2008. Available from https://www.ietf.org/rfc/rfc5246.

[10] P. Eronen and H. Tschofenig. Pre-shared key ciphersuites for TLS, December 2005. Available from https://www.ietf.org/rfc/rfc4279.

[11] C. Evans and C. Palmer. Public Key Pinning Extension for HTTP, December 2011. Available from https://tools.ietf.org/html/draft-ietf-websec-key-pinning-01.

[12] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0, August 2011. Available from https://www.ietf.org/rfc/rfc6101.

[13] D. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919 (Proposed Standard), Aug. 2016.

[14] P. Gutmann. Everything you never wanted to know about PKI but were forced to find out, 2002. Available from https://www.cs.auckland.ac.nz/~pgut001/pubs/pkitutorial.pdf.

[15] P. Hallam-Baker. X.509v3 transport layer security (tls) feature extension. RFC 7633, October 2015.

[16] R. Housley, T. Polk, W. Ford, and D. Solo. Internet X.509 public key infrastructure

certificate and certificate revocation list (CRL) profile, April 2002. Available from https://www.ietf.org/rfc/rfc3280.

[17] J. Kelsey and B. Schneier. Cryptanalytic attacks on pseudorandom number generators. Available from https://www.schneier.com/academic/paperfiles/paper-prngs.pdf.

[18] R. Khare and S. Lawrence. Upgrading to TLS within HTTP/1.1, May 2000. Available from https://www.ietf.org/rfc/rfc2817.

[19] R. Laboratories. PKCS 12 v1.0: Personal information exchange syntax, June 1999.

[20] A. Langley. A Transport Layer Security (TLS) ClientHello Padding Extension, October 2015. Available from https://www.ietf.org/rfc/rfc7685.

[21] A. Langley, N. Modadugu, and B. Moeller. Transport layer security (tls) false start. RFC 7918, August 2016.

[22] C. Latze and N. Mavrogiannopoulos. The TPMKEY URI Scheme, January 2013. Work in progress, available from https://tools.ietf.org/html/draft-mavrogiannopoulos-tpmuri-01.

[23] A. Lenstra, X. Wang, and B. de Weger. Colliding X.509 Certificates, 2005. Available from https://eprint.iacr.org/2005/067.

[24] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery, March 2007. Available from https://www.ietf.org/rfc/rfc4821.

[25] D. McGrew and E. Rescorla. Datagram transport layer security (dtls) extension to establish keys for the secure real-time transport protocol (srtp). RFC 5764, May 2010.

[26] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP, June 1999. Available from https://www.ietf.org/rfc/rfc2560.

[27] M. Nystrom and B. Kaliski. PKCS 10 v1.7: certification request syntax specification, November 2000. Available from https://www.ietf.org/rfc/rfc2986.

[28] J. Pechanec and D. J. Moffat. The PKCS 11 URI Scheme. RFC 7512 (Standards Track), Apr. 2015.

[29] M. T. R. Seggelmann 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.

[30] E. Rescola. HTTP over TLS, May 2000. Available from https://www.ietf.org/rfc/rfc2818.

[31] E. Rescorla and N. Modadugu. Datagram transport layer security, April 2006. Available from https://www.ietf.org/rfc/rfc4347.

[32] 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.

[33] R. L. Rivest. Can We Eliminate Certificate Revocation Lists?, February 1998. Available from https://people.csail.mit.edu/rivest/Rivest-CanWeEliminateCertificateRevocationLists.pdf.

[34] P. Saint-Andre and J. 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.

[35] P. Saint-Andre and A. Melnikov. Preparation, Enforcement, and Comparison of Internationalized Strings Representing Usernames and Passwords, August 2015. Available from https://www.ietf.org/rfc/rfc7613.

[36] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport layer security (TLS) session resumption without server-side state, January 2008. Available from https://www.ietf.org/rfc/rfc5077.

[37] S. Santesson. TLS Handshake Message for Supplemental Data, September 2006. Available from https://www.ietf.org/rfc/rfc4680.

[38] W. R. Stevens. *UNIX Network Programming, Volume 1*. Prentice Hall, 1998.

[39] D. Taylor, T. Perrin, T. Wu, and N. Mavrogiannopoulos. Using SRP for TLS authentication, November 2007. Available from https://www.ietf.org/rfc/rfc5054.

[40] S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. Internet X.509 public key infrastructure (PKI) proxy certificate profile, June 2004. Available from https://www.ietf.org/rfc/rfc3820.

[41] M. Tuexen, R. Seggelmann, and E. Rescorla. Datagram transport layer security (dtls) for stream control transmission protocol (sctp). RFC 6083, January 2011. https://www.rfc-editor.org/rfc/rfc6083.txt.

[42] S. Whited. Channel Bindings for TLS 1.3. RFC 9266, July 2022.

[43] N. Williams. On the use of channel bindings to secure channels, November 2007. Available from https://www.ietf.org/rfc/rfc5056.

[44] T. Wu. The stanford SRP authentication project. Available from https://srp.stanford.edu/.

[45] T. Wu. The SRP authentication and key exchange system, September 2000. Available from https://www.ietf.org/rfc/rfc2945.

[46] K. D. Zeilenga. Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names, June 2006. Available from https://www.ietf.org/rfc/rfc4514.

Index