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

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 [32], to public key infrastructure in [13] and to security engineering in [5].

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

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 1.2 (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 1.2, TLS 1.1, TLS 1.0 and SSL 3.0 protocols.
- Support for Datagram TLS 1.0.
- Support for handling and verification of X.509 and OpenPGP 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[1] library. The "Cryptographic back-end" is provided by the nettle[2] library.

1.1. Downloading and installing

GnuTLS is available for download at: `http://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.

[1] `http://www.gnu.org/software/libtasn1/`
[2] `http://www.lysator.liu.se/~nisse/nettle/`

GnuTLS depends on Libnettle, and you will need to install it before installing GnuTLS. Libnettle is available from http://www.lysator.liu.se/~nisse/nettle/. 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.

The compression library, libz, as well as p11-kit are a optional dependencies. You can get libz from http://www.zlib.net/ and p11-kit from http://p11-glue.freedesktop.org/.

The X.509 part of GnuTLS needs ASN.1 functionality, from 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 http://www.gnu.org/software/gnutls/download.html.

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 guarranteed to compile if some of these options are given.

```
--disable-srp-authentication
--disable-psk-authentication
--disable-anon-authentication
--disable-extra-pki
--disable-openpgp-authentication
--disable-openssl-compatibility
--disable-libdane
--without-p11-kit
--without-tpm
--disable-dtls-srtp-support
```

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

1.2. 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 chapter 3, and shared-key as well anonymous authentication in chapter 4. We elaborate on certificate authentication by demonstrating advanced usage of the API in chapter 5. The core of the TLS library is presented in chapter 7 and example applications are listed in chapter 8. In chapter 9 the usage of few included programs that may assist debugging is presented. The last chapter is chapter 10 that provides a short introduction to GnuTLS' internal architecture.

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" [27] 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, data authenticity, and optionally compression. 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 transport layer can be used by providing callbacks for GnuTLS to access the transport layer (for details see section 7.5).

[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.

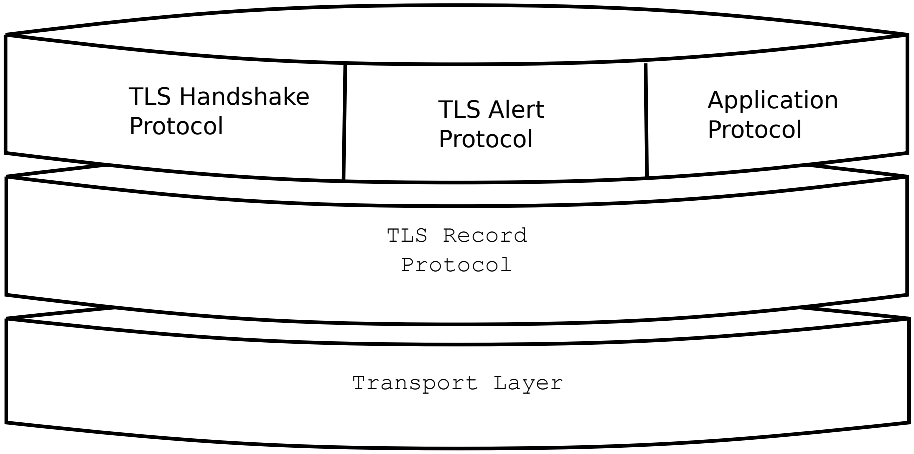

Figure 2.1.: The TLS protocol layers.

2.3. The TLS record protocol

The record protocol is the secure communications provider. Its purpose is to encrypt, authenticate and —optionally— compress 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 block encryption algorithms like `3DES`, `AES` or stream algorithms like `ARCFOUR_128`. Ciphers are encryption algorithms that use a single, secret, key to encrypt and decrypt data. Block algorithms in CBC mode also provide protection against statistical analysis of the data. Thus, if you're using the TLS protocol, a random number of blocks will be appended to data, to prevent eavesdroppers from guessing the actual data size.

The supported in GnuTLS ciphers and MAC algorithms are shown in Table 2.1 and Table 2.2.

Algorithm	Description
3DES_CBC	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	ARCFOUR_128 is a compatible algorithm with RSA's RC4 algorithm, which is considered to be a trade secret. It is a fast cipher but considered weak today.
ARCFOUR_40	This is the ARCFOUR cipher fed with a 40 bit key, which is considered weak.
AES_CBC	AES or RIJNDAEL is the block cipher algorithm that replaces the old DES algorithm. Has 128 bits block size and is used in CBC mode.
AES_GCM	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.
CAMELLIA_CBC	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.

Table 2.1.: Supported ciphers.

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 SHA256. Outputs 256 bits of data.
MAC_AEAD	This indicates that an authenticated encryption algorithm, such as GCM, is in use.

Table 2.2.: Supported MAC algorithms.

2.3.2. Compression algorithms used in the record layer

The TLS record layer also supports compression. The algorithms implemented in GnuTLS can be found in the table below. The included algorithms perform really good when text, or other compressible data are to be transferred, but offer nothing on already compressed data, such as compressed images, zipped archives etc. These compression algorithms, may be useful in high bandwidth TLS tunnels, and in cases where network usage has to be minimized. It should be noted however that compression increases latency.

The record layer compression in GnuTLS is implemented based on [14]. The supported algorithms are shown below.

Note that compression enables attacks such as traffic analysis, or even plaintext recovery under certain circumstances. To avoid some of these attacks GnuTLS allows each record to be com-

enum gnutls_compression_method_t:

GNUTLS_COMP_UNKNOWN	Unknown compression method.
GNUTLS_COMP_NULL	The NULL compression method (no compression).
GNUTLS_COMP_DEFLATE	The DEFLATE compression method from zlib.
GNUTLS_COMP_ZLIB	Same as **GNUTLS_COMP_DEFLATE**.

Table 2.3.: Supported compression algorithms

pressed independently (i.e., stateless compression), by using the "%STATELESS_COMPRESSION" priority string, in order to be used in cases where the attacker controlled data are pt in separate records.

2.3.3. Weaknesses and countermeasures

Some weaknesses that may affect the security of the record layer have been found in TLS 1.0 protocol. These weaknesses can be exploited by active attackers, and exploit the facts that

1. TLS has separate alerts for "decryption_failed" and "bad_record_mac"
2. The decryption failure reason can be detected by timing the response time.
3. The IV for CBC encrypted packets is the last block of the previous encrypted packet.

Those weaknesses were solved in TLS 1.1 [8] which is implemented in GnuTLS. For this reason we suggest to always negotiate the highest supported TLS version with the peer[2]. For a detailed discussion of the issues see the archives of the TLS Working Group mailing list and [21].

2.3.4. On record padding

The TLS protocol allows for random padding of records in CBC ciphers, to prevent statistical analysis based on the length of exchanged messages (see [9] section 6.2.3.2). GnuTLS appears to be one of few implementation that take advantage of this text, and pad records by a random length.

The TLS implementation in the Symbian operating system, frequently used by Nokia and Sony-Ericsson mobile phones, cannot handle non-minimal record padding. What happens when one of these clients handshake with a GnuTLS server is that the client will fail to compute the correct MAC for the record. The client sends a TLS alert (`bad_record_mac`) and disconnects. Typically this will result in error messages such as 'A TLS fatal alert has been received', 'Bad record MAC', or both, on the GnuTLS server side.

[2] If this is not possible then please consult subsection 7.11.6.

GnuTLS implements a work around for this problem. However, it has to be enabled specifically. It can be enabled by using `gnutls_record_disable_padding`, or `gnutls_priority_set` with the `%COMPAT` priority string (see section 7.9).

If you implement an application that have a configuration file, we recommend that you make it possible for users or administrators to specify a GnuTLS protocol priority string, which is used by your application via `gnutls_priority_set`. To allow the best flexibility, make it possible to have a different priority string for different incoming IP addresses.

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. Fatal alerts always terminate the current connection, and prevent future re-negotiations using the current session ID. All 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_USER_CANCELED	90	User canceled
GNUTLS_A_NO_RENEGOTIATION	100	No renegotiation is allowed
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

Table 2.4.: 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 handshake protocol of TLS 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.

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

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 follow.

- Certificate authentication: Authenticated key exchange using public key infrastructure and certificates (X.509 or OpenPGP).
- SRP authentication: Authenticated key exchange using a password.
- PSK authentication: Authenticated key exchange using a pre-shared key.
- 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 7.4.1.

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 7.11.1, subsection 8.1.9 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:

- Maximum fragment length negotiation
- Server name indication
- Session tickets
- HeartBeat
- Safe Renegotiation

and they will be 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**)

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 store session parameters. This complicates deployment, and could be avoiding 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 [30].

Since version 3.1.3 GnuTLS clients transparently support session tickets.

2.6.4. HeartBeat

This TLS extension allows to ping and receive confirmation from the peer, is described in [25]. 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_HERTBEAT_PING_RECEIVED` being returned from the receive function. Ping requests to peer can be send via `gnutls_heartbeat_ping`.

int **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 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 [28].

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 7.9). 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 section 5.6) to obtain its certificate revocation status and serve it to the clients. That way a client avoids an additional connection to the OCSP server.

void **gnutls_certificate_set_ocsp_status_request_function** (*gnutls_certificate_credentials_t* **sc**, *gnutls_status_request_ocsp_func* **ocsp_func**, *void * * **ptr**)

int **gnutls_certificate_set_ocsp_status_request_file** (*gnutls_certificate_credentials_t* **sc**, *const char** **response_file**, *unsigned int* **flags**)

int **gnutls_ocsp_status_request_enable_client** (*gnutls_session_t* **session**, *gnutls_datum_t * * **responder_id**, *size_t* **responder_id_size**, *gnutls_datum_t * * **extensions**)

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

A server is required to provide the OCSP server's response using the `gnutls_certificate_set_ocsp_status_request_file`. The response may be obtained periodically using the following command.

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

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

2.6.7. SRTP

The TLS protocol was extended in [20] 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.

To enable use the following functions.

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

Table 2.5.: Supported SRTP profiles

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.

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

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**)

2.7. How to use TLS in application protocols

This chapter is intended to provide some hints on how to use the 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. That way two separate ports were assigned, one for the non secure sessions, and one for the secured ones. This has the benefit that if a user requests a secure session then the client will try to connect to the secure port and fail otherwise. The only possible attack with this method is a denial of service one. The most famous example of this method is the famous "HTTP over TLS" or HTTPS protocol [26].

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

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

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,

[4]See also the Server Name Indication extension on subsection 2.6.2.
[5]See LDAP, IMAP etc.

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 that method [16].

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 see 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 dummy enough to send the confidential data in the clear.

How to avoid the above attack? As you may have already noticed this one 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. Certificate authentication

The most known authentication method of TLS are certificates. The PKIX [15] public key infrastructure is daily used by anyone using a browser today. GnuTLS supports both X.509 certificates [15] and OpenPGP certificates using a common API.

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

Key exchange	Description
RSA	The RSA algorithm is used to encrypt a key and send it to the peer. The certificate must allow the key to be used for encryption.
RSA_EXPORT	The RSA algorithm is used to encrypt a key and send it to the peer. In the EXPORT algorithm, the server signs temporary RSA parameters of 512 bits — which are considered weak — and sends them to the client.
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.

3.1. X.509 certificates

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

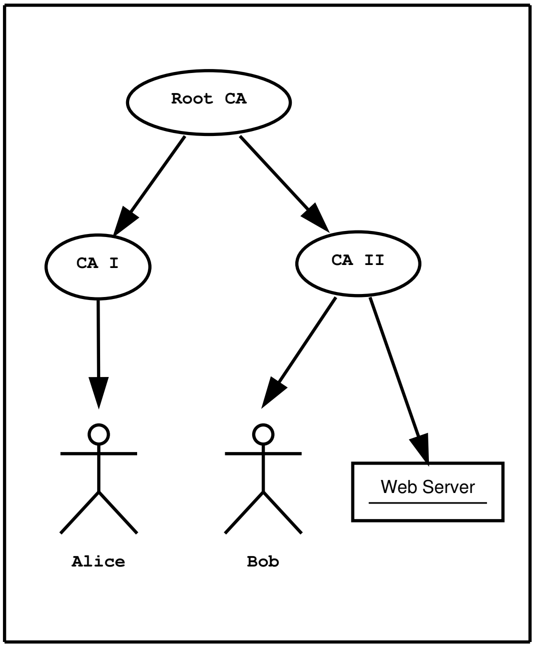

Figure 3.1.: An example of the X.509 hierarchical trust model.

One needs to trust one or more CAs for his secure communications. In that case only the certificates issued by the trusted authorities are acceptable. The framework is illustrated on Figure 3.1.

3.1.1. X.509 certificate structure

An X.509 certificate usually contains information about the certificate holder, the signer, a unique serial number, expiration dates and some other fields [15] 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 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

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.

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.

Certificate *extensions* are there to include information about the certificate's subject that did not fit in the typical certificate fields. 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.

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 8.4.2.

3.1.2. 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.

Extension	OID	Description
Subject key id	2.5.29.14	An identifier of the key of the subject.
Authority key id	2.5.29.35	An identifier of the authority's key used to sign the certificate.
Subject alternative name	2.5.29.17	Alternative names to subject's distinguished name.
Key usage	2.5.29.15	Constraints the key's usage of the certificate.
Extended key usage	2.5.29.37	Constraints the purpose of the certificate.
Basic constraints	2.5.29.19	Indicates whether this is a CA certificate or not, and specify the maximum path lengths of certificate chains.
CRL distribution points	2.5.29.31	This extension is set by the CA, in order to inform about the issued CRLs.
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.
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 [34].

Table 3.3.: X.509 certificate extensions.

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`.

3.1.3. 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_dn` which follows the rules in [38] 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_dn** (*gnutls_x509_crt_t* **cert**, *char ** **buf**, *size_t ** **buf_size**)

Description: This function will copy the name of the Certificate in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data. If **buf** is null then only the size will be filled.

Returns: **GNUTLS_E_SHORT_MEMORY_BUFFER** if the provided buffer is not long enough, and in that case the **buf_size** will be updated with the required size. On success 0 is returned.

int **gnutls_x509_crt_get_dn_by_oid** (*gnutls_x509_crt_t* **cert**, *const char ** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **buf_size**)

Description: This function will extract the part of the name of the Certificate subject specified by the given OID. The output, if the raw flag is not used, will be encoded as described in RFC4514. Thus a string that is ASCII or UTF-8 encoded, depending on the certificate data. Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 -- in hex format with a '#' prefix. You can check about known OIDs using gnutls_x509_dn_oid_known(). If **buf** is null then only the size will be filled. If the **raw_flag** is not specified the output is always null terminated, although the **buf_size** will not include the null character.

Returns: **GNUTLS_E_SHORT_MEMORY_BUFFER** if the provided buffer is not long enough, and in that case the **buf_size** will be updated with the required size. **GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE** if there are no data in the current index. On success 0 is returned.

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

int **gnutls_x509_crt_get_dn_oid** (*gnutls_x509_crt_t* **cert**, *int* **indx**, *void* * **oid**, *size_t* * **oid_size**)

Description: This function will extract the OIDs of the name of the Certificate subject specified by the given index. If **oid** is null then only the size will be filled. The **oid** returned will be null terminated, although **oid_size** will not account for the trailing null.

Returns: **GNUTLS_E_SHORT_MEMORY_BUFFER** if the provided buffer is not long enough, and in that case the **buf_size** will be updated with the required size. **GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE** if there are no data in the current index. On success 0 is returned.

int **gnutls_x509_crt_get_issuer_dn** (*gnutls_x509_crt_t* **cert**, *char* * **buf**, *size_t* * **buf_size**)

int **gnutls_x509_crt_get_issuer_dn_by_oid** (*gnutls_x509_crt_t* **cert**, *const char* * **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void* * **buf**, *size_t* * **buf_size**)

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

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

3.1.4. 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 6.1.1 for more information.

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

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.`

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**)

3.1.5. Verifying X.509 certificate paths

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

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 8.1.7. It is also possible to have a set of certificates that are trusted for a particular server but not to authorize other certificates. This purpose is served by the functions `gnutls_x509_trust_list_add_named_crt` and `gnutls_x509_trust_list_verify_named_crt`.

int **gnutls_x509_trust_list_add_cas** (*gnutls_x509_trust_list_t* **list**, *const gnutls_x509_crt_t ** **clist**, *int* **clist_size**, *unsigned int* **flags**)

Description: This function will add the given certificate authorities to the trusted list. The list of CAs must not be deinitialized during this structure's lifetime.

Returns: The number of added elements is returned.

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 only with gnutls_x509_trust_list_verify_named_crt(). 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 must not be deinitialized during the lifetime of the trusted list.

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

3.1.6. 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_crls** (*gnutls_x509_trust_list_t* **list**, *const gnutls_x509_crl_t ** **crl_list**, *int* **crl_size**, *unsigned int* **flags**, *unsigned int* **verification_flags**)

Description: This function will add the given certificate revocation lists to the trusted list. The list of CRLs must not be deinitialized during this structure's lifetime. This function must be called after gnutls_x509_trust_list_add_cas() to allow verifying the CRLs for validity.

Returns: The number of added elements is returned.

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 ** **verify**, *gnutls_verify_output_function* **func**)

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

Limitation: Pathlen constraints or key usage flags are not consulted.

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

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 ** **verify**, *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 **verify** parameter will hold an OR'ed sequence of **gnutls_certificate_status_t** 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. pkcs11 URLs are also accepted, instead of files, by this function.

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.`

Returns: `The number of added elements is returned.`

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 system this function returns` **GNUTLS_E_UNIMPLEMENTED_FEATURE**.

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_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**)

Then it is not required to setup a trusted list as above. The function `gnutls_certificate_verify_peers3` may then be used to verify the peer's certificate chain and identity. The flags are set similarly to the verification functions in the previous section.

There is also the possibility to pass some input to the verification functions in the form of flags. For `gnutls_x509_trust_list_verify_crt` the flags are passed straightforward, but `gnutls_certificate_verify_peers3` depends on the flags set by calling `gnutls_certificate_set_verify_flags`. All the available flags are part of the enumeration `gnutls_certificate_verify_flags` shown in Table 3.5.

3.2. OpenPGP certificates

The OpenPGP key authentication relies on a distributed trust model, called the "web of trust". The "web of trust" uses a decentralized system of trusted introducers, which are the same as a CA. OpenPGP allows anyone to sign anyone else's public key. When Alice signs Bob's key, she is introducing Bob's key to anyone who trusts Alice. If someone trusts Alice to introduce keys, then Alice is a trusted introducer in the mind of that observer. For example in Figure 3.2, David trusts Alice to be an introducer and Alice signed Bob's key thus Dave trusts Bob's key to be the real one.

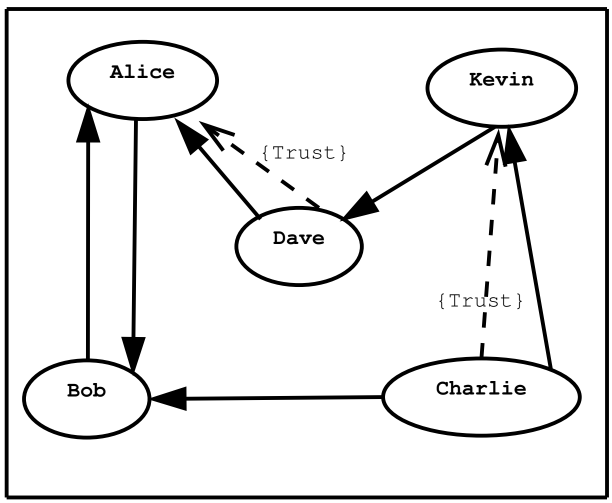

Figure 3.2.: An example of the OpenPGP trust model.

There are some key points that are important in that model. In the example Alice has to sign Bob's key, only if she is sure that the key belongs to Bob. Otherwise she may also make Dave falsely believe that this is Bob's key. Dave has also the responsibility to know who to trust. This model is similar to real life relations.

Just see how Charlie behaves in the previous example. Although he has signed Bob's key - because he knows, somehow, that it belongs to Bob - he does not trust Bob to be an introducer. Charlie decided to trust only Kevin, for some reason. A reason could be that Bob is lazy enough, and signs other people's keys without being sure that they belong to the actual owner.

3.2.1. OpenPGP certificate structure

In GnuTLS the OpenPGP certificate structures [7] are handled using the `gnutls_openpgp_crt_t` type. A typical certificate contains the user ID, which is an RFC 2822 mail and name address, a public key, possibly a number of additional public keys (called subkeys), and a number of signatures. The various fields are shown in Table 3.6.

The additional subkeys may provide key for various different purposes, e.g. one key to encrypt mail, and another to sign a TLS key exchange. Each subkey is identified by a unique key ID. The keys that are to be used in a TLS key exchange that requires signatures are called authentication keys in the OpenPGP jargon. The mapping of TLS key exchange methods to public keys is shown in Table 3.7.

The corresponding private keys are stored in the `gnutls_openpgp_privkey_t` type. All the prototypes for the key handling functions can be found in "`gnutls/openpgp.h`".

3.2.2. Verifying an OpenPGP certificate

The verification functions of OpenPGP keys, included in GnuTLS, are simple ones, and do not use the features of the "web of trust". For that reason, if the verification needs are complex, the assistance of external tools like GnuPG and GPGME[1] is recommended.

In GnuTLS there is a verification function for OpenPGP certificates, the `gnutls_openpgp_crt_verify_ring`. This checks an OpenPGP key against a given set of public keys (keyring) and returns the key status. The key verification status is the same as in X.509 certificates, although the meaning and interpretation are different. For example an OpenPGP key may be valid, if the self signature is ok, even if no signers were found. The meaning of verification status flags is the same as in the X.509 certificates (see Table 3.5).

int **gnutls_openpgp_crt_verify_ring** (*gnutls_openpgp_crt_t* **key**, *gnutls_openpgp_keyring_t* **keyring**, *unsigned int* **flags**, *unsigned int ** **verify**)

Description: Verify all signatures in the key, using the given set of keys (keyring). The key verification output will be put in **verify** and will be one or more of the *gnutls_certificate_status_t* enumerated elements bitwise or'd. Note that this function does not verify using any "web of trust". You may use GnuPG for that purpose, or any other external PGP application.

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

3.2.3. Verifying a certificate in the context of a TLS session

Similarly with X.509 certificates, one needs to specify the OpenPGP keyring file in the credentials structure. The certificates in this file will be used by `gnutls_certificate_verify_peers3` to verify the signatures in the certificate sent by the peer.

[1] http://www.gnupg.org/related_software/gpgme/

int **gnutls_openpgp_crt_verify_self** (*gnutls_openpgp_crt_t* **key**, *unsigned int* **flags**, *unsigned int ** **verify**)

Description: Verifies the self signature in the key. The key verification output will be put in **verify** and will be one or more of the gnutls_certificate_status_t enumerated elements bitwise or'd.

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

int **gnutls_certificate_set_openpgp_keyring_file** (*gnutls_certificate_credentials_t* **c**, *const char ** **file**, *gnutls_openpgp_crt_fmt_t* **format**)

Description: The function is used to set keyrings that will be used internally by various OpenPGP functions. For example to find a key when it is needed for an operations. The keyring will also be used at the verification functions.

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

3.3. 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.

3.3.1. Verifying a certificate using trust on first use authentication

It is possible to use a trust on first use (TOFU) authentication method in GnuTLS. That is the concept used by the SSH programs, where the public key of the peer is not verified, or verified in an out-of-bound way, but subsequent connections to the same peer require the public key to remain the same. Such a system in combination with the typical CA verification of a certificate, and OCSP revocation checks, can help to provide multiple factor verification, where a single point of failure is not enough to compromise the system. For example a server compromise may be detected using OCSP, and a CA compromise can be detected using the trust on first use method. Such a hybrid system with X.509 and trust on first use authentication is shown in subsection 8.1.2.

See subsection 7.11.2 on how to use the available functionality.

3.3.2. Verifying a certificate using DANE (DNSSEC)

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

The DANE functionality is provided by the `libgnutls-dane` library that is shipped with GnuTLS and the function prototypes are in `gnutls/dane.h`. See subsection 7.11.2 for information on how to use the library.

3.4. 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 he donated 1.000.000 euros to Bad Inc.

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

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

The last two requirements in the list are the most important in digital signatures. These protect against somebody who would like to generate two messages with the same hash output. When an algorithm is considered broken usually it means that the Collision resistance of the algorithm is less than brute force. Using the birthday paradox the brute force attack takes $2^{(\text{hash size})/2}$ operations. Today colliding certificates using the MD5 hash algorithm have been generated as shown in [18].

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

3.4.1. Trading security for interoperability

If you connect to a server and use GnuTLS' functions to verify the certificate chain, and get a `GNUTLS_CERT_INSECURE_ALGORITHM` validation error (see subsection 3.1.5), 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`

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.

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.

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 normaly be disabled, unless you know what this means.
GNUTLS_VERIFY_ALLOW_X509_V1_-CA_CRT	Allow trusted CA certicates with version 1. This is safer than **GNUTLS_VERIFY_ALLOW_ANY_X**509_V1_CA_CRT, and should be used instead. That way only signers in your trusted list will be allowed to have certicates of version 1. This is the default.
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. Must be used in combination with **GNUTLS_VERIFY_ALLOW_X**509_V1_CA_CRT.
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.

Table 3.5.: The `gnutls_certificate_verify_flags` enumeration.

Field	Description
version	The field that indicates the version of the OpenPGP structure.
user ID	An RFC 2822 string that identifies the owner of the key. There may be multiple user identifiers in a key.
public key	The main public key of the certificate.
expiration	The expiration time of the main public key.
public subkey	An additional public key of the certificate. There may be multiple subkeys in a certificate.
public subkey expiration	The expiration time of the subkey.

Table 3.6.: OpenPGP certificate fields.

Key exchange	Public key requirements
RSA	An RSA public key that allows encryption.
DHE_RSA	An RSA public key that is marked for authentication.
ECDHE_RSA	An RSA public key that is marked for authentication.
DHE_DSS	A DSA public key that is marked for authentication.

Table 3.7.: The types of (sub)keys required for the various TLS key exchange methods.

4. Shared-key and anonymous authentication

In addition to certificate authentication, the TLS protocol may be used with password, shared-key and anonymous authentication methods. The rest of this chapter discusses details of these methods.

4.1. SRP authentication

4.1.1. Authentication using SRP

GnuTLS supports authentication via the Secure Remote Password or SRP protocol (see [37, 36] 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.

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 [33]. The supported key exchange methods are shown below.

- 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_encode_alloc** (*const gnutls_datum_t* * **data**, *gnutls_datum_t* * **result**)

int **gnutls_srp_base64_decode_alloc** (*const gnutls_datum_t* * **b64_data**, *gnutls_datum_t* * **result**)

4.1.2. Invoking srptool

Simple program that emulates the programs in the Stanford SRP (Secure Remote Password) libraries using GnuTLS. It is intended for use in places where you don't expect SRP authentication to be the used for system users.

In brief, to use SRP you need to create two files. These are the password file that holds the users and the verifiers associated with them and the configuration file to hold the group parameters (called tpasswd.conf).

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

srptool help/usage (-h)

This is the automatically generated usage text for srptool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  srptool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
```

```
                                - It must be in the range:
                                  0 to 9999
   -i, --index                 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.
   -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:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. 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 an argument string. 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 an argument string. This generates a password configuration file (tpasswd.conf) containing the required for TLS parameters.

srptool exit status

One of the following exit values will be returned:

- 0 (EXIT_SUCCESS) Successful program execution.
- 1 (EXIT_FAILURE) The operation failed or the command syntax was not valid.

srptool See Also

gnutls-cli-debug (1), gnutls-serv (1), srptool (1), psktool (1), certtool (1)

srptool Examples

To create "`tpasswd.conf`" which holds the g and n values for SRP protocol (generator and a large prime), run:

```
$ srptool --create-conf /etc/tpasswd.conf
```

This command will create "`/etc/tpasswd`" and will add user 'test' (you will also be prompted for a password). Verifiers are stored by default in the way libsrp expects.

```
$ srptool --passwd /etc/tpasswd --passwd-conf /etc/tpasswd.conf -u test
```

This command will check against a password. If the password matches the one in "`/etc/tpasswd`" you will get an ok.

```
$ srptool --passwd /etc/tpasswd --passwd\-conf /etc/tpasswd.conf --verify -u test
```

4.2. PSK authentication

4.2.1. Authentication using PSK

Authentication using Pre-shared keys is a method to authenticate using usernames and binary keys. This protocol avoids making use of public key infrastructure and expensive calculations, thus it is suitable for constraint clients.

The implementation in GnuTLS is based on [10]. The supported PSK key exchange methods are:

- PSK: Authentication using the PSK protocol.
- 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.

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**)

4.2.2. Invoking psktool

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

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

psktool help/usage (-h)

This is the automatically generated usage text for psktool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  psktool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
   -s, --keysize=num          specify the key size in bytes
                                - It must be in the range:
                                  0 to 512
   -u, --username=str         specify a username
   -p, --passwd=str           specify a password file.
   -v, --version[=arg]        Output version information and exit
   -h, --help                 Display extended usage information and exit
   -!, --more-help            Extended usage information passed thru pager

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



Program that generates random keys for use with TLS-PSK.  The keys are
```

```
stored in hexadecimal format in a key file.

please send bug reports to:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. Specifies the debug level.

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 “`passwd.psk`” for use with GnuTLS run:

```
$ ./psktool -u psk_identity -p passwd.psk
Generating a random key for user ’psk_identity’
Key stored to passwd.psk
$ cat psks.txt
psk_identity:88f3824b3e5659f52d00e959bacab954b6540344
$
```

This command will create “`passwd.psk`” if it does not exist and will add user ’psk_identity’ (you will also be prompted for a password).

4.3. 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. Moreover it is useful when complete anonymity is required. Unless in one of the above cases, do not use anonymous authentication.

The available key exchange algorithms for anonymous authentication are shown below, but note that few public servers support them. They typically have to be explicitly enabled.

- 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.

5. 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.

5.1. PKCS #10 certificate requests

A certificate request is a structure, which contain information about an applicant of a certificate service. It usually contains a private key, a distinguished name and secondary data such as a challenge password. GnuTLS supports the requests defined in PKCS #10 [23]. Other formats of certificate requests are not currently supported.

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_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 6.1.

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.

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.

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.

Returns: **GNUTLS_E_SUCCESS** on success, otherwise a negative error code. **GNUTLS_E_ASN**1_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()).

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

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

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

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

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

```

  gnutls_global_init ();

  /* Initialize an empty certificate request, and
   * an empty private key.
   */
  gnutls_x509_crq_init (&crq);

  gnutls_x509_privkey_init (&key);

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

}
```

5.2. 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 5.1. Certificate revocation lists are used to complement the expiration date of a certificate, in order to account for other reasons of revocation, such as compromised keys, etc.

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

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

Table 5.1.: Certificate revocation list fields.

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**, *int* **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.

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

int **gnutls_x509_crl_get_version** (*gnutls_x509_crl_t* **crl**)

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

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.

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.

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.

Returns: On success, **GNUTLS_E_SUCCESS** (0) is returned, otherwise a negative error value. Since 2.12.0

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

int **gnutls_x509_crl_set_number** (*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**)

5.3. 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 [29].

The Online Certificate Status Protocol, or OCSP [22], is a widely implemented protocol to perform 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.

Note that in the context of a TLS session the server may provide an OCSP response that will used during the TLS certificate verification (see `gnutls_certificate_verify_peers3`). You may obtain this response using `gnutls_ocsp_status_request_get`.

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 create an OCSP request object, store some information about the certificate to check in the request, and then export 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_t* **req**, *gnutls_datum_t ** **data**)

int **gnutls_ocsp_req_print** (*gnutls_ocsp_req_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_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_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_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 5.2. Note that a response may contain information on multiple certificates.

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

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 5.2.: The most important OCSP response fields.

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_t* **resp**, *gnutls_datum_t* * **data**)

int **gnutls_ocsp_resp_print** (*gnutls_ocsp_resp_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.

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_get_single** (*gnutls_ocsp_resp_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.

int **gnutls_ocsp_resp_verify** (*gnutls_ocsp_resp_t* **resp**, *gnutls_x509_trust_list_t* **trustlist**, *unsigned int* * **verify**, *unsigned int* **flags**)

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

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

5.4. 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 6.2) 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.

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 5.3.: The revocation reasons

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_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 typical X.509, PKCS #8 and the openssl format. If the provided key is encrypted but no password was given, then **GNUTLS_E_DECRYPTION_FAILED** is returned.

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**)

Description: This function will import the given private key to the abstract *gnutls_privkey_t* structure. The supported formats are typical X.509, PKCS #8 and the openssl format.

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 [17] usually contains a user's private keys and certificates. It is commonly used in browsers to export and import the user's identities.

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 enum **gnutls_pkcs_encrypt_flags_t**:

GNUTLS_PKCS_PLAIN	Unencrypted private key.
GNUTLS_PKCS_USE_PKCS12_3DES	PKCS-12 3DES.
GNUTLS_PKCS_USE_PKCS12_ARCFOUR	PKCS-12 ARCFOUR.
GNUTLS_PKCS_USE_PKCS12_RC2_40	PKCS-12 RC2-40.
GNUTLS_PKCS_USE_PBES2_3DES	PBES2 3DES.
GNUTLS_PKCS_USE_PBES2_AES_128	PBES2 AES-128.
GNUTLS_PKCS_USE_PBES2_AES_192	PBES2 AES-192.
GNUTLS_PKCS_USE_PBES2_AES_256	PBES2 AES-256.
GNUTLS_PKCS_NULL_PASSWORD	Some schemas distinguish between an empty and a NULL password.

Table 5.4.: Encryption flags

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_bag_get_data** (*gnutls_pkcs12_bag_t* **bag**, *int* **indx**, *gnutls_datum_t ** **data**)

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

int **gnutls_pkcs12_bag_get_friendly_name** (*gnutls_pkcs12_bag_t* **bag**, *int* **indx**, *char *** **name**)

The functions below are used to generate a PKCS #12 structure. An example of their usage is also shown.

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 PKCS#12 blob in **p12blob** and extracts the private key, the corresponding certificate chain, and any additional certificates and a CRL. The **extra_certs_ret** and **extra_certs_ret_len** parameters are optional and both may be set to **NULL**. If either is non-**NULL**, then both must be. Encrypted PKCS#12 bags and PKCS#8 private keys are supported. However, only password based security, and the same password for all operations, are supported. PKCS#12 file may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. You should make sure the PKCS#12 file only contain one key/certificate pair and/or one CRL. It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all. 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.

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

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**, *int* **indx**, *const gnutls_datum_t ** **id**)

int **gnutls_pkcs12_bag_set_friendly_name** (*gnutls_pkcs12_bag_t* **bag**, *int* **indx**, *const char ** **name**)

```
/* 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 *fd;

  /* 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;
    }

  fd = fopen (OUTFILE, "w");
  if (fd == NULL)
    {
      fprintf (stderr, "cannot open file\n");
      return 1;
    }
  fwrite (pkcs12_struct, 1, pkcs12_struct_size, fd);
  fclose (fd);

  gnutls_pkcs12_bag_deinit (bag);
  gnutls_pkcs12_bag_deinit (key_bag);
  gnutls_pkcs12_deinit (pkcs12);

  return 0;
}
```

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`.

5.5. 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.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option *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.

descriptions for the `certtool` program. This software is released under the GNU General Public License, version 3 or later.

certtool help/usage (-h)

This is the automatically generated usage text for certtool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.5
USAGE:  certtool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
   -s, --generate-self-signed  Generate a self-signed certificate
   -c, --generate-certificate  Generate a signed certificate
       --generate-proxy       Generates a proxy certificate
       --generate-crl         Generate a CRL
   -u, --update-certificate   Update a signed certificate
   -p, --generate-privkey     Generate a private key
   -q, --generate-request     Generate a PKCS #10 certificate request
   -e, --verify-chain         Verify a PEM encoded certificate chain.
       --verify               Verify a PEM encoded certificate chain using a trusted list.
                                - requires these options:
                                load-ca-certificate
       --verify-crl           Verify a CRL using a trusted list.
                                - requires these options:
                                load-ca-certificate
```

```
       --generate-dh-params   Generate PKCS #3 encoded Diffie-Hellman parameters.
       --get-dh-params        Get 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=file    Loads a certificate request file
                                - file must pre-exist
       --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
       --password=str         Password to use
       --hex-numbers          Print big number in an easier format to parse
       --null-password        Enforce a NULL password
   -i, --certificate-info     Print information on the given certificate
       --certificate-pubkey   Print certificate's public key
       --pgp-certificate-info  Print information on the given OpenPGP certificate
       --pgp-ring-info        Print information on the given OpenPGP keyring structure
   -l, --crl-info             Print information on the given CRL structure
       --crq-info             Print information on the given certificate request
       --no-crq-extensions    Do not use extensions in certificate requests
       --p12-info             Print information on a PKCS #12 structure
       --p7-info              Print information on a PKCS #7 structure
       --smime-to-p7          Convert S/MIME to PKCS #7 structure
   -k, --key-info             Print information on a private key
       --pgp-key-info         Print information on an OpenPGP private key
       --pubkey-info          Print information on a public key
       --v1                   Generate an X.509 version 1 certificate (with no extensions)
       --to-p12               Generate a PKCS #12 structure
                                - requires these options:
                                load-certificate
       --to-p8                Generate a PKCS #8 structure
   -8, --pkcs8                Use PKCS #8 format for private keys
       --rsa                  Generate RSA key
       --dsa                  Generate DSA key
       --ecc                  Generate ECC (ECDSA) key
       --ecdsa                This is an alias for 'ecc'
       --hash=str             Hash algorithm to use for signing.
       --inder                Use DER format for input certificates and private keys.
                                - disabled as --no-inder
       --inraw                This is an alias for 'inder'
       --outder               Use DER format for output certificates and private keys
                                - disabled as --no-outder
       --outraw               This is an alias for 'outder'
       --bits=num             Specify the number of bits for key generate
       --sec-param=str        Specify the security level [low, legacy, normal, high, ultra].
       --disable-quick-random  No effect
       --template=file        Template file to use for non-interactive operation
                                - file must pre-exist
       --pkcs-cipher=str      Cipher to use for PKCS #8 and #12 operations
   -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.

please send bug reports to:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. Specifies the debug level.

verify-chain option (-e)

This is the “verify a pem encoded certificate chain.” option. The last certificate in the chain must be a self signed one.

verify option

This is the “verify a pem encoded certificate chain 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.

verify-crl option

This is the “verify a crl 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.

get-dh-params option

This is the “get the included pkcs #3 encoded diffie-hellman parameters.” option. Returns stored DH parameters in GnuTLS. Those parameters are used in the SRP protocol. The parameters returned by fresh generation are more efficient since GnuTLS 3.0.9.

load-privkey option

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

load-ca-privkey option

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

null-password option

This is the “enforce a null password” option. This option enforces a NULL password. This may be different than the empty password in some schemas.

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.

to-p12 option

This is the "generate a pkcs #12 structure" option.

This option has some usage constraints. It:

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

It requires a certificate, a private key and possibly a CA certificate to be specified.

rsa option

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

dsa option

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

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.

ecdsa option

This is an alias for the ecc option, section 5.5.

hash option

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

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 5.5.

outder option

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

outraw option

This is an alias for the outder option, section 5.5.

sec-param option

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

pkcs-cipher option

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

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)

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
```

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
```

Diffie-Hellman parameter generation

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

```
$ certtool --generate-dh-params --outfile dh.pem --sec-param normal
```

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 cert.pem --load-privkey key.pem  \
   --template cert.cfg \
   --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"

# The serial number of the certificate
serial = 007

# In how many days, counting from today, this certificate will expire.
expiration_days = 700

# X.509 v3 extensions

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

# A subject alternative name URI
#uri = "http://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"

# Challenge password used in certificate requests
challenge_passwd = 123456

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

# Whether this is a CA certificate or not
#ca

# for microsoft smart card logon
# key_purpose_oid = 1.3.6.1.4.1.311.20.2.2

### Other predefined key purpose OIDs

# Whether this certificate will be used for a TLS client
#tls_www_client

# Whether this certificate will be used for a TLS server
#tls_www_server

# Whether this certificate will be used to sign data (needed
# in TLS DHE ciphersuites).
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.
#encryption_key

# Whether this key will be used to sign other certificates.
#cert_signing_key

# Whether this key will be used to sign CRLs.
#crl_signing_key

# Whether this key will be used to sign code.
#code_signing_key

# Whether this key will be used to sign OCSP data.
#ocsp_signing_key

# Whether this key will be used for time stamping.
#time_stamping_key

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

### end of key purpose OIDs

# 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

# Path length contraint. 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 = http://my.ocsp.server/ocsp

# CA issuers URI
# ca_issuers_uri = http://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 = http://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 = http://www.example.com/another-policy-to-read


# Options for proxy certificates
# proxy_policy_language = 1.3.6.1.5.5.7.21.1


# Options for generating a CRL

```

```
# next CRL update will be in 43 days (wow)
#crl_next_update = 43

# this is the 5th CRL by this CA
#crl_number = 5

```

5.6. Invoking ocsptool

Ocsptool is a program that can parse and print information about OCSP requests/responses, generate requests and verify responses.

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

ocsptool help/usage (-h)

This is the automatically generated usage text for ocsptool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  ocsptool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --ask[=arg]            Ask an OCSP/HTTP server on a certificate validity
                                - requires these options:
                                load-cert
                                load-issuer
   -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     Generate an OCSP request
       --nonce                Don't add nonce to OCSP request
                                - disabled as --no-nonce
       --load-issuer=file     Read issuer certificate from file
                                - file must pre-exist
       --load-cert=file       Read certificate to check from file
                                - file must pre-exist
       --load-trust=file      Read OCSP trust anchors from file
```

```
                                - prohibits these options:
                                load-signer
                                - file must pre-exist
       --load-signer=file     Read OCSP response signer from file
                                - prohibits these options:
                                load-trust
                                - file must pre-exist
       --inder                Use DER format for input certificates and private keys
                                - disabled as --no-inder
   -Q, --load-request=file    Read DER encoded OCSP request from file
                                - file must pre-exist
   -S, --load-response=file   Read DER encoded OCSP response from file
                                - file must pre-exist
   -v, --version[=arg]        Output version information and exit
   -h, --help                 Display extended usage information and exit
   -!, --more-help            Extended usage information passed thru pager

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



Ocsptool is a program that can parse and print information about OCSP
requests/responses, generate requests and verify responses.

please send bug reports to:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. Specifies the debug level.

ask option

This is the “ask an ocsp/http server on a certificate validity” option. This option takes an optional argument string @fileserver name—url.

This option has some usage constraints. It:

- must appear in combination with the following options: load-cert, load-issuer.

Connects to the specified HTTP OCSP server and queries on the validity of the loaded certificate.

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 --print-cert > chain.pem
```

Use a text editor on `chain.pem` to create three files for each separate certificates, called `cert.pem` for the first certificate for the domain itself, secondly `issuer.pem` for the intermediate certificate and `root.pem` for the final root certificate.

The domain certificate normally contains a pointer to where the OCSP responder is located, in the Authority Information Access Information extension. For example, from `certtool -i < cert.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: http://ocsp.CAcert.org/
```

This means the CA support OCSP queries over HTTP. We are now ready to create a OCSP request for the certificate.

```
$ ocsptool --ask ocsp.CAcert.org --load-issuer issuer.pem \
           --load-cert cert.pem --outfile ocsp-response.der
```

The request is sent via HTTP to the OCSP server address specified. If the address is ommited ocsptool will use the address stored in the certificate.

5.7. Invoking danetool

Tool to generate DNS resource records for the DANE protocol.

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

danetool help/usage (-h)

This is the automatically generated usage text for danetool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.5
USAGE:  danetool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num             Enable debugging.
                                 - It must be in the range:
                                   0 to 9999
   -V, --verbose               More verbose output
                                 - may appear multiple times
       --infile=file           Input file
                                 - file must pre-exist
       --outfile=str           Output file
       --load-pubkey=str       Loads a public key file
       --load-certificate=str Loads a certificate file
       --hash=str              Hash algorithm to use for signing.
       --check=str             Check DANE TLSA entry.
       --local-dns             Use the local DNS server for DNSSEC resolving.
```

```
                              - disabled as --no-local-dns
       --inder                Use DER format for input certificates and private keys.
                              - disabled as --no-inder
       --inraw                This is an alias for 'inder'
       --tlsa-rr              Print the DANE RR data on a certificate or public key
                              - requires these options:
                              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=num             Specify the port number for the DANE data.
       --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                The provided certificate or public key is a local entity.
   -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 DNS resource records for the DANE protocol.

please send bug reports to:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. Specifies the debug level.

load-pubkey option

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

hash option

This is the “hash algorithm to use for signing.” option. This option takes an argument string. 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 an argument string. Obtains the DANE TLSA entry from the given hostname and prints information.

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.

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 5.7.

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 an argument string "`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 an argument string "`Protocol`". This command specifies the protocol for the service set in the DANE data.

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 the "the provided certificate or public key is a local entity." option. DANE distinguishes certificates and public keys offered via the DNSSEC to trusted and local entities. Use this flag if this is a local (and possibly unsigned) entity.

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 CA signed certificate use the following commands.

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

For a self signed certificate use:

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

The latter 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 signer of your certificate use:

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

To read a server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443
```

To verify a server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443 --load-certificate chain.pem
```

6. Hardware security modules and abstract key types

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 guarranteed. 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.

6.1. Abstract key types

Since there are many forms of a public or private keys supported by GnuTLS such as X.509, OpenPGP, 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 must be used.

```
#inlude <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_pkcs11_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);
}
```

6.1.1. Public keys

An abstract `gnutls_pubkey_t` can be initialized using the functions below. It can be imported through 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_openpgp** (*gnutls_pubkey_t* **key**, *gnutls_openpgp_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 or OpenPGP 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**)

int **gnutls_pubkey_import_openpgp_raw** (*gnutls_pubkey_t* **pkey**, *const gnutls_datum_t ** **data**, *gnutls_openpgp_crt_fmt_t* **format**, *const gnutls_openpgp_keyid_t* **keyid**, *unsigned int* **flags**)

An important function is `gnutls_pubkey_import_url` which will import public keys from URLs that identify objects stored in tokens (see section 6.2 and section 6.3). A function to check for a supported by GnuTLS URL is `gnutls_url_is_supported`.

int **gnutls_url_is_supported** (*const char** **url**)

Description: `Check whether url is supported. Depending on the system libraries GnuTLS may support pkcs11 or tpmkey 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, as well as a function that given a public key fingerprint would provide a memorable sketch.

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_get_pk_rsa_raw** (*gnutls_pubkey_t* **key**, *gnutls_datum_t* * **m**, *gnutls_datum_t* * **e**)

int **gnutls_pubkey_get_pk_dsa_raw** (*gnutls_pubkey_t* **key**, *gnutls_datum_t* * **p**, *gnutls_datum_t* * **q**, *gnutls_datum_t* * **g**, *gnutls_datum_t* * **y**)

int **gnutls_pubkey_get_pk_ecc_raw** (*gnutls_pubkey_t* **key**, *gnutls_ecc_curve_t* * **curve**, *gnutls_datum_t* * **x**, *gnutls_datum_t* * **y**)

int **gnutls_pubkey_get_pk_ecc_x962** (*gnutls_pubkey_t* **key**, *gnutls_datum_t** **parameters**, *gnutls_datum_t* * **ecpoint**)

6.1.2. Private keys

An abstract `gnutls_privkey_t` can be initialized using the functions below. It can be imported through 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_openpgp** (*gnutls_privkey_t* **pkey**, *gnutls_openpgp_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 or OpenPGP structures are shown below. Again, as with public keys, private keys can be imported from a hardware module using URLs.

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_import_openpgp_raw** (*gnutls_privkey_t* **pkey**, *const gnutls_datum_t* * **data**, *gnutls_openpgp_crt_fmt_t* **format**, *const gnutls_openpgp_keyid_t* **keyid**, *const char** **password**) *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_get_pk_algorithm** (*gnutls_privkey_t* **key**, *unsigned int ** **bits**)

gnutls_privkey_type_t **gnutls_privkey_get_type** (*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_ext2** (*gnutls_privkey_t* **pkey**, *gnutls_pk_algorithm_t* **pk**, *void** **userdata**, *gnutls_privkey_sign_func* **sign_func**, *gnutls_privkey_decrypt_func* **decrypt_func**, *gnutls_privkey_deinit_func* **deinit_func**, *unsigned int* **flags**)

Description: This function will associate the given callbacks with the *gnutls_privkey_t* structure. At least one of the two callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain **GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE**.

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

6.1.3. Operations

The abstract key types can be used to access signing and signature verification operations with the underlying keys.

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_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.

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.

Returns: In case of a verification failure **GNUTLS_E_PK_SIG_VERIFY_FAILED** is returned, and zero or positive code on success.

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.

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.

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.

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 request.

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

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**)

6.2. 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`.

Moreover PKCS #11 can be (ab)used to allow all applications in the same operating system to access shared cryptographic keys and certificates in a uniform way, as in Figure 6.1. That way applications could load their trusted certificate list, as well as user certificates from a common PKCS #11 module. Such a provider exists in the Gnome system, being the Gnome Keyring.

6.2.1. Initialization

To allow all the GnuTLS applications to access PKCS #11 tokens you can use a configuration per module, stored in `/etc/pkcs11/modules/`. These are the configuration files of p11-kit[2]. For example a file that will load the OpenSC module, could be named `/etc/pkcs11/modules/opensc` and contain the following:

```
module: /usr/lib/opensc-pkcs11.so
```

[1] http://www.opensc-project.org
[2] http://p11-glue.freedesktop.org/

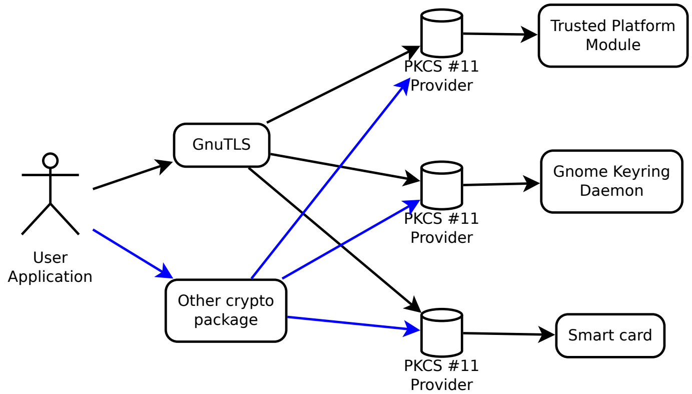

Figure 6.1.: PKCS #11 module usage.

If you use this file, then there is no need for other initialization in GnuTLS, except for the PIN and token functions (see next section). However, you may manually initialize the PKCS #11 subsystem if the default settings are not desirable.

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. Normally you don't need to call this function since it is being called by gnutls_global_init() using the **GNUTLS_PKCS**11_FLAG_AUTO. If other option is required then it must be called before it.

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

Note that PKCS #11 modules must be reinitialized on the child processes after a `fork`. GnuTLS provides `gnutls_pkcs11_reinit` to be called for this purpose.

6.2.2. 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.

int **gnutls_pkcs11_reinit** (**void**)

Description:
This function will reinitialize the PKCS 11 subsystem in gnutls. This is required by PKCS 11 when an application uses fork(). The reinitialization function must be called on the child.

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

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.

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.

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 6.1.: The `gnutls_pin_flag_t` enumeration.

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**)

6.2.3. Reading objects

All PKCS #11 objects are referenced by GnuTLS functions by URLs as described in [24]. 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;objecttype=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 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 access objects are shown below.

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* **crt**, *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 it returns null terminated string although **output_size** contains the size of the actual data only.

Returns: **GNUTLS_E_SUCCESS** (0) on success or a negative error code on error.

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

int **gnutls_x509_crt_import_pkcs11_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;

  obj_list_size = 0;
  ret = gnutls_pkcs11_obj_list_import_url (NULL, &obj_list_size, URL,
                                           GNUTLS_PKCS11_OBJ_ATTR_CRT_WITH_PRIVKEY,
                                           0);
  if (ret < 0 && ret != GNUTLS_E_SHORT_MEMORY_BUFFER)
    return -1;

/* no error checking from now on */
  obj_list = malloc (sizeof (*obj_list) * obj_list_size);

  gnutls_pkcs11_obj_list_import_url (obj_list, &obj_list_size, URL,
                                     GNUTLS_PKCS11_OBJ_ATTR_CRT_WITH_PRIVKEY,
                                     0);

/* 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);
    }

  return 0;
}
```

6.2.4. 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).

6.2.5. Using a PKCS #11 token with TLS

It is possible to use a PKCS #11 token to a TLS session, as shown in subsection 8.1.8. In addition the following functions can be used to load PKCS #11 key and certificates by specifying *int* **gnutls_pkcs11_copy_x509_privkey** (*const char ** **token_url**, *gnutls_x509_privkey_t* **key**, *const char ** **label**, *unsigned int* **key_usage**, *unsigned int* **flags**)

Description: This function will copy a private key into a PKCS #11 token specified by a URL. It is highly recommended flags to contain **GNUTLS_PKCS**11_OBJ_FLAG_MARK_SENSITIVE unless there is a strong reason not to.

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

int **gnutls_pkcs11_copy_x509_crt** (*const char ** **token_url**, *gnutls_x509_crt_t* **crt**, *const char ** **label**, *unsigned int* **flags**)

Description: This function will copy a certificate into a PKCS #11 token specified by a URL. The certificate can be marked as trusted or not.

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

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_file** (*gnutls_certificate_credentials_t* **res**, *const char ** **certfile**, *const char ** **keyfile**, *gnutls_x509_crt_fmt_t* **type**)

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.

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.

6.2.6. Invoking p11tool

Program that allows handling data from PKCS #11 smart cards and security modules.

To use PKCS #11 tokens with gnutls 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'.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `p11tool` program. This software is released under the GNU General Public License, version 3 or later.

p11tool help/usage (-h)

This is the automatically generated usage text for p11tool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  p11tool [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [url]

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
       --outfile=str          Output file
       --list-tokens          List all available tokens
       --export               Export the object specified by the URL
       --list-mechanisms      List all available mechanisms in a token
       --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-all-trusted     List all available certificates marked as trusted
       --initialize           Initializes a PKCS #11 token
       --write                Writes the loaded objects to a PKCS #11 token
       --delete               Deletes the objects matching the PKCS #11 URL
       --generate-rsa         Generate an RSA private-public key pair
       --generate-dsa         Generate an RSA private-public key pair
       --generate-ecc         Generate an RSA private-public key pair
       --label=str            Sets a label for the write operation
```

```
       --trusted              Marks the object to be written as trusted
                                - disabled as --no-trusted
       --private              Marks the object to be written as private
                                - disabled as --no-private
                                - enabled by default
       --login                Force login to token
                                - disabled as --no-login
       --detailed-url         Print detailed URLs
                                - disabled as --no-detailed-url
       --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
   -8, --pkcs8                Use PKCS #8 format for private keys
       --bits=num             Specify the number of bits for key generate
       --sec-param=str        Specify the security level
       --inder                Use DER/RAW format for input
                                - disabled as --no-inder
       --inraw                This is an alias for 'inder'
       --provider=file        Specify the PKCS #11 provider library
                                - file must pre-exist
   -v, --version[=arg]        Output version information and exit
   -h, --help                 Display extended usage information and exit
   -!, --more-help            Extended usage information passed thru pager

Options are specified by doubled hyphens and their name or by a single
hyphen and the flag character.
Operands and options may be intermixed.  They will be reordered.



Program that allows handling data from PKCS #11 smart cards and security
modules.

To use PKCS #11 tokens with gnutls 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'.

please send bug reports to:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. Specifies the debug level.

write option

This is the “writes the loaded objects to a pkcs #11 token” option. It can be used to write private keys, certificates or secret keys to a token.

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.

generate-dsa option

This is the “generate an rsa private-public key pair” option. Generates an RSA private-public key pair on the specified token.

generate-ecc option

This is the “generate an rsa private-public key pair” option. Generates an RSA private-public key pair on the specified token.

private option

This is the “marks the object to be written as private” option.

This option has some usage constraints. It:

- is enabled by default.

The written object will require a PIN to be used.

sec-param option

This is the “specify the security level” option. This option takes an argument string “`Security parameter`”. This is alternative to the bits option. Available options are [low, legacy, normal, high, ultra].

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 6.2.6.

provider option

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

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.

6.3. Trusted Platform Module (TPM)

In this section we present the Trusted Platform Module (TPM) support in GnuTLS. 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, which are the fact that you can use it to perform cryptographic operations the similarly to a PKCS #11 smart card. It allows for storing and using RSA keys but with slight differences from a PKCS #11 module that require different handling. The basic operations supported, and used by GnuTLS, are key generation and signing.

In GnuTLS the TPM functionality is available in `gnutls/tpm.h`.

6.3.1. 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 of the following form.

```
tpmkey:uuid=42309df8-d101-11e1-a89a-97bb33c23ad1;storage=user
```

It consists from a unique identifier of the key as well as the part of the flash memory the key is stored at. The two options for the storage field are 'user' and 'system'. The user keys are typically only available to the generating user and the system keys to all users. The stored in TPM keys are called registered keys.

The keys that are stored in the disk are exported from the TPM but in an encrypted form. To access them two passwords are required. The first is the TPM Storage Root Key (SRK), and the other is a key-specific password. Also those keys are identified by a URL of the form:

```
tpmkey:file=/path/to/file
```

When objects require a PIN to be accessed the same callbacks as with PKCS #11 objects are expected (see subsection 6.2.2).

6.3.2. 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.

6.3.3. 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_file` 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**)

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 permament storage using `gnutls_tpm_privkey_delete`.

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* structure. 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* structure. 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.

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**)

6.3.4. Invoking tpmtool

Program that allows handling cryptographic data from the TPM chip.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `tpmtool` program. This software is released under the GNU General *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.

Public License, version 3 or later.

tpmtool help/usage (-h)

This is the automatically generated usage text for tpmtool. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  tpmtool [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
       --infile=file          Input file
                                - file must pre-exist
       --outfile=str          Output file
       --generate-rsa         Generate an RSA private-public key pair
       --register             Any generated key will be registered in the TPM
                                - requires these options:
                                generate-rsa
       --signing              Any generated key will be a signing key
                                - requires these options:
                                generate-rsa
                                -- and prohibits these options:
                                legacy
       --legacy               Any generated key will be a legacy key
                                - requires these options:
                                generate-rsa
                                -- and prohibits these options:
                                signing
       --user                 Any registered key will be a user key
                                - requires these options:
                                register
                                -- and prohibits these options:
                                system
       --system               Any registred key will be a system key
                                - requires these options:
                                register
                                -- and prohibits these options:
                                user
```

```
       --pubkey=str           Prints the public key of the provided key
       --list                 Lists all stored keys in the TPM
       --delete=str           Delete the key identified by the given URL (UUID).
       --sec-param=str        Specify the security level [low, legacy, normal, high, ultra].
       --bits=num             Specify the number of bits for key generate
       --inder                Use the DER format for keys.
                                - disabled as --no-inder
       --outder               Use DER format for output keys
                                - disabled as --no-outder
   -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:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. 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 filesystem and protected by a PIN, or stored (registered) in the TPM chip flash.

user option

This is the “any registered key will be a user key” option.

This option has some usage constraints. It:

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

The generated key will be stored in a user specific persistent storage.

system option

This is the “any registred key will be a system key” option.

This option has some usage constraints. It:

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

The generated key will be stored in system persistent storage.

sec-param option

This is the “specify the security level [low, legacy, normal, high, ultra].” option. This option takes an argument string “`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.

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 filesystem 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 filesystem:

```
$ tpmtool --pubkey tpmkey:file=tmpkey.pem --outfile pubkey.pem
```

To list all keys stored in TPM use:

```
$ tpmtool --list
```

7. How to use GnuTLS in applications

7.1. 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.

7.1.1. General idea

A brief description of how GnuTLS sessions operate is shown at Figure 7.1. 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 indentity 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 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

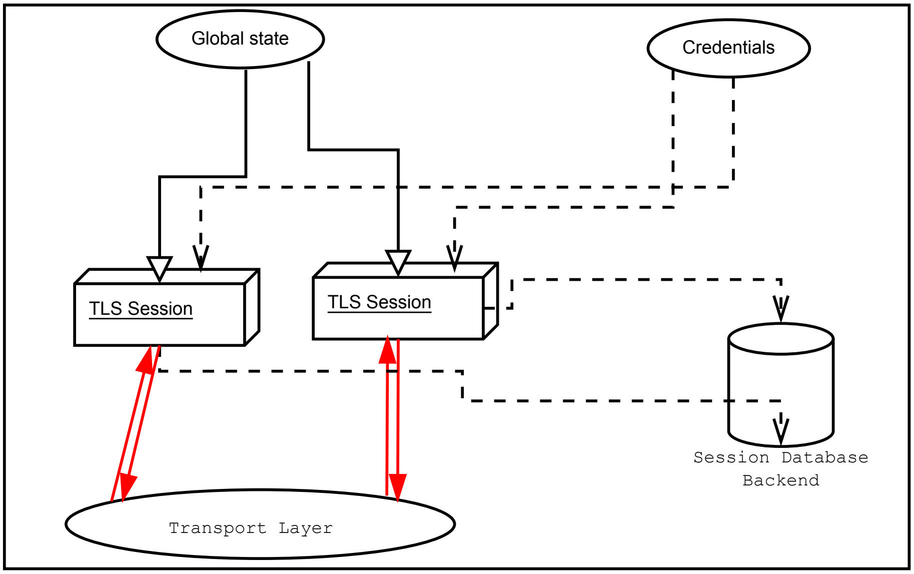

Figure 7.1.: High level design of GnuTLS.

7.1.2. Error handling

In GnuTLS most functions return an integer type as a result. In almost all cases a zero or a positive number means success, and a negative number indicates failure, or a situation that some action has to be taken. Thus negative error codes may be fatal or not.

Fatal errors terminate the connection 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 7.8). See Appendix D, for a description of the available error codes.

7.1.3. 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 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;
```

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;
```

7.1.4. 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**)

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 where you set the logging function gnutls is going to use. This is different from gnutls_global_set_log_function() because it will report the session of the event if any. Note that that session might 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*);

7.1.5. 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. If, however, an object needs to be shared across threads then access must be protected with a mutex. Read-only access to objects, for example the credentials holding structures, is also thread-safe.

The random generator of the cryptographic back-end, is not thread safe and requires mutex locks which are setup by GnuTLS. Applications can either call `gnutls_global_init` which will initialize the default operating system provided locks (i.e. `pthreads` on GNU/Linux and `CriticalSection` on Windows), or manually specify the locking system using the function `gnutls_global_set_mutex` before calling `gnutls_global_init`. Setting mutexes manually is recommended only for applications that have full control of the underlying libraries. If this is not the case, the use of the operating system defaults is recommended. An example of non-native thread usage is shown below.

```
#include <gnutls/gnutls.h>

int main()
{
   /* When the system mutexes are not to be used
    * gnutls_global_set_mutex() must be called explicitly
    */
   gnutls_global_set_mutex (mutex_init, mutex_deinit,
                            mutex_lock, mutex_unlock);
   gnutls_global_init();
}
```

7.1.6. 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_global_set_mutex** (*mutex_init_func* **init**, *mutex_deinit_func* **deinit**, *mutex_lock_func* **lock**, *mutex_unlock_func* **unlock**)

Description: `With this function you are allowed to override the default mutex locks used in some parts of gnutls and dependent libraries. This function should be used if you have complete control of your program and libraries. Do not call this function from a library. Instead only initialize gnutls and the default OS mutex locks will be used. This function must be called before gnutls_global_init().`

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`.

7.2. 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.

7.2.1. 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.

7.2.2. Initialization

GnuTLS must be initialized before it can be used. The library is initialized by calling `gnutls_global_init`. The resources allocated by the initialization process can be released if the application no longer has a need to call GnuTLS functions, this is done by calling `gnutls_global_deinit`.

In order to take advantage of the internationalization features in GnuTLS, such as translated error messages, the application must set the current locale using `setlocale` before initializing GnuTLS.

7.2.3. 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`.

7.2.4. 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 >= 3.0.0])

AC_SUBST([LIBGNUTLS_CFLAGS])
AC_SUBST([LIBGNUTLS_LIBS])
```

7.3. 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 current session to null. Every session must be initialized before use, so internal structures can be allocated. This function allocates structures which can only be free'd by calling gnutls_deinit(). Returns **GNUTLS_E_SUCCESS** (0) on success. **flags** can be one of **GNUTLS_CLIENT** and **GNUTLS_SERVER**. For a DTLS entity, the flags **GNUTLS_DATAGRAM** and **GNUTLS_NONBLOCK** are also available. The latter flag will enable a non-blocking operation of the DTLS timers. Note that since version 3.1.2 this function enables some common TLS extensions such as session tickets and OCSP certificate status request in client side by default. To prevent that use the **GNUTLS_NO_EXTENSIONS** flag.

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

After the session initialization details on the allowed ciphersuites and protocol versions should be set using the priority functions such as `gnutls_priority_set_direct`. We elaborate on them in section 7.9. 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`.

7.4. 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 7.1.

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

Description: Sets the needed credentials for the specified type. Eg 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.

general issues applying to both client and server certificates. The next section will elaborate on issues arising from client authentication only.

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.

Authentication method	Key exchange	Client credentials	Server credentials
Certificate	`KX_RSA`, `KX_DHE_RSA`, `KX_DHE_DSS`, `KX_ECDHE_RSA`, `KX_ECDHE_ECDSA`, `KX_RSA_EXPORT`	`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 7.1.: Key exchange algorithms and the corresponding credential types.

int **gnutls_certificate_set_x509_key_mem** (*gnutls_certificate_credentials_t* **res**, *const gnutls_datum_t ** **cert**, *const gnutls_datum_t ** **key**, *gnutls_x509_crt_fmt_t* **type**)

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**)

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_openpgp_key_mem** (*gnutls_certificate_credentials_t* **res**, *const gnutls_datum_t ** **cert**, *const gnutls_datum_t ** **key**, *gnutls_openpgp_crt_fmt_t* **format**)

int **gnutls_certificate_set_openpgp_key** (*gnutls_certificate_credentials_t* **res**, *gnutls_openpgp_crt_t* **crt**, *gnutls_openpgp_privkey_t* **pkey**)

int **gnutls_certificate_set_openpgp_key_file** (*gnutls_certificate_credentials_t* **res**, *const char ** **certfile**, *const char ** **keyfile**, *gnutls_openpgp_crt_fmt_t* **format**)

Note however, that since functions like `gnutls_certificate_set_x509_key_file` may accept URLs that specify objects stored in token, 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 by the token.

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_-`

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.

`import_x509_raw` and `gnutls_privkey_import_x509_raw`.

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 certificate/private key pair in the gnutls_certificate_credentials_t structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that wants to send more than its own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in **pcert_list**. The **pcert_list** and **key** will become part of the credentials structure and must not be deallocated. They will be automatically deallocated when res is deinitialized.

Returns: **GNUTLS_E_SUCCESS** (0) on success, or a negative error code.

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.

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**)

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 make the required structures 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_openpgp** (*gnutls_pcert_st** **pcert**, *gnutls_openpgp_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**)

int **gnutls_pcert_import_openpgp_raw** (*gnutls_pcert_st ** **pcert**, *const gnutls_datum_t** **cert**, *gnutls_openpgp_crt_fmt_t* **format**, *gnutls_openpgp_keyid_t* **keyid**, *unsigned int* **flags**)

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 `gnutls_certificate_server_set_request`. The request contains a list of the acceptable by the server 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 an 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.

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.

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 and OpenPGP certificates.

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

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_openpgp_keyring_file** (*gnutls_certificate_credentials_t* **c**, *const char ** **file**, *gnutls_openpgp_crt_fmt_t* **format**)

The peer's certificate is not automatically verified and one must call `gnutls_certificate_verify_peers3` after a successful handshake to verify the certificate's signature and the owner of the certificate. The verification status returned can be printed using `gnutls_certificate_verification_status_print`.

Alternatively the verification can occur during the handshake by using `gnutls_certificate_set_verify_function`.

The functions above 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 section 3.1.

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

Description: This function will verify the peer's certificate and store the status in the **status** variable as a bitwise or'd gnutls_certificate_status_t values or zero if the certificate is trusted. Note that value in **status** is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). If the **hostname** provided is non-NULL then this function will compare the hostname in the certificate against the given. If they do not match the **GNUTLS_CERT_UNEXPECTED_OWNER** status flag will be set. If available the OCSP Certificate Status extension will be utilized by this function. To avoid denial of service attacks some default upper limits regarding the certificate key size and chain size are set. To override them use gnutls_certificate_set_verify_limits().

Returns: a negative error code on error and **GNUTLS_E_SUCCESS** (0) on success.

void **gnutls_certificate_set_verify_function** (*gnutls_certificate_credentials_t* **cred**, *gnutls_certificate_verify_function ** **func**)

Description: This function sets a callback to be called when peer's certificate has been received in order to verify it on receipt rather than doing after the handshake is completed. The callback's function prototype is: int (*callback)(gnutls_session_t); If the callback function is provided then gnutls will call it, in the handshake, just after the certificate message has been received. To verify or obtain the certificate the gnutls_certificate_verify_peers2(), gnutls_certificate_type_get(), gnutls_certificate_get_peers() functions can be used. The callback function should return 0 for the handshake to continue or non-zero to terminate.

7.4.2. 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(). **username** and **password** should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep". 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* structure. 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* g, gnutls_datum_t* n); **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. In case the callback returned a negative number then gnutls will assume that the username does not exist. 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). The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

7.4.3. 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**)

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.

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(). **username** should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep". The callback function will be called once per handshake. The callback function should return 0 on success. -1 indicates an error.

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* structure. This password file holds usernames and keys and will be used for PSK authentication.

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

void **gnutls_psk_set_server_credentials_function** (*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**)

7.4.4. Anonymous

The key exchange methods for anonymous authentication might require Diffie-Hellman parameters to be generated by the server and associated with an anonymous credentials structure. Check subsection 7.11.3 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**)

7.5. 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_ptr2` would be sufficient to specify the socket descriptor.

void **gnutls_transport_set_ptr2** (*gnutls_session_t* **session**, *gnutls_transport_ptr_t* **recv_ptr**, *gnutls_transport_ptr_t* **send_ptr**)

Description: `Used to set the first argument of the transport function (for push and pull callbacks). In berkeley style sockets this function will set the connection descriptor. With this function you can use two different pointers for receiving and sending.`

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

If however another transport layer than TCP is selected, then the following functions have to be specified.

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 *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);

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
- GNUTLS_E_AGAIN
- GNUTLS_E_LARGE_PACKET

The EINTR and EAGAIN values are returned by interrupted system calls, or when non block-

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_errno** (*gnutls_session_t* **session**, *int* **err**)

Description: Store **err** in the session-specific errno variable. Useful values for **err** is EAGAIN and EINTR, 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).

ing 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. The callback function is used in DTLS only. **gnutls_pull_timeout_func** is of the form, ssize_t (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);

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.

7.5.1. 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., 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).

The `select` system call can also be used in combination with the GnuTLS functions. `select` allows monitoring of sockets and notifies on them being ready for reading or writing data. Note however that this system call cannot notify on data present in GnuTLS read buffers, it is only applicable to the kernel sockets API. Thus if you are using it for reading from a GnuTLS session, make sure that any cached data are read completely. That can be achieved by checking there are no data waiting to be read (using `gnutls_record_check_pending`), either before the `select` system call, or after a call to `gnutls_record_recv`. GnuTLS does not keep a write buffer, thus when writing no additional actions are required.

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 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 for some reason, that is, if a function returned **GNUTLS_E_INTERRUPTED** or **GNUTLS_E_AGAIN**. In such a case, you might want to call select() or poll() before calling the interrupted gnutls function again. To tell you whether a file descriptor should be selected for either reading or writing, gnutls_record_get_direction() returns 0 if the interrupted function was trying to read data, and 1 if it was trying to write data.

Returns: 0 if trying to read data, 1 if trying to write data.

Moreover, to prevent blocking from DTLS' retransmission timers to block a handshake, the `gnutls_init` function should be called with the `GNUTLS_NONBLOCK` flag set (see section 7.3).

7.5.2. DTLS sessions

Because datagram TLS can operate over connections where the peer of a server cannot be reliably verified, functionality is available to prevent denial of service attacks. GnuTLS requires a server to generate a secret key that is used to sign a cookie[2]. 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.

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 [19].

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**)

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

7.6. 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 does the handshake of the TLS/SSL protocol, and initializes the TLS connection. This function will fail if any problem is encountered, and will return a negative error code. In case of a client, if the client has asked to resume a session, but the server couldn't, then a full handshake will be performed. The non-fatal errors such as **GNUTLS_E_AGAIN** and **GNUTLS_E_INTERRUPTED** interrupt the handshake procedure, which should be resumed later. Call this function again, until it returns 0; cf. gnutls_record_get_direction() and gnutls_error_is_fatal(). If this function is called by a server after a rehandshake request then **GNUTLS_E_GOT_APPLICATION_DATA** or **GNUTLS_E_WARNING_ALERT_RECEIVED** may be returned. Note that these are non fatal errors, only in the specific case of a rehandshake. Their meaning is that the client rejected the rehandshake request or in the case of **GNUTLS_E_GOT_APPLICATION_DATA** it might also mean that some data were pending.

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

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

Description: This function sets the timeout for the handshake process to the provided value. Use an **ms** value of zero to disable timeout. Note that in order for the timeout to be enforced gnutls_transport_set_pull_timeout_function() must be set.

The handshake process doesn't ensure the verification of the peer's identity. When certificates are in use, this can be done, either after the handshake is complete, or during the handshake if `gnutls_certificate_set_verify_function` has been used. In both cases the `gnutls_certificate_verify_peers2` function can be used to verify the peer's certificate (see chapter 3 for more information).

int **gnutls_certificate_verify_peers2** (*gnutls_session_t* **session**, *unsigned int* * **status**)

7.7. 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.

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 full information. You can replace the default push function by using gnutls_transport_set_ptr2() with a call to send() with a MSG_DONTWAIT flag if blocking is a problem. If the EINTR is returned by the internal push function (the default is send()) 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 same parameters; alternatively you could provide a **NULL** pointer for data, and 0 for size. cf. gnutls_record_get_direction(). The errno value EMSGSIZE maps to **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.

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 `select`, i.e., `gnutls_record_check_pending` reports no data to be read, `select` 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, *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 a server requests a renegotiation, the client may receive an error code of **GNUTLS_E_REHANDSHAKE**. 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. If **EINTR** is returned by the internal push 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(). A server may also receive **GNUTLS_E_REHANDSHAKE** when a client has initiated a handshake. In that case the server can only initiate a handshake or terminate the connection.

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. 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 the record layer or the handshake layer.

Returns: 1 if the error code is fatal, for positive **error** values, 0 is returned. For unknown **error** values, -1 is returned.

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**.

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 guarranteed not to block.

Returns: Returns the size of the data or zero.

which allows distinguishing it from a malicious connection termination. A session can be deinitialized with the `gnutls_deinit` function.

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.

7.8. 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`.

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.

7.9. Priority strings

In order to specify cipher suite preferences on a TLS session there are priority functions that accept a string specifying the enabled for the handshake algorithms. That string may contain a single initial keyword such as in Table 7.2 and may be followed by additional algorithm or special keywords.

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 is returned.

Returns: the alert code to use for a particular error code.

int **gnutls_priority_set_direct** (*gnutls_session_t* **session**, *const char ** **priorities**, *const char *** **err_pos**)

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

Unless the initial keyword is "NONE" the defaults (in preference order) are for TLS protocols TLS 1.2, TLS1.1, TLS1.0, SSL3.0; for compression NULL; 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[3].

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).

[3]Except for the RSA-EXPORT which is only enabled in EXPORT level.

Keyword	Description
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.
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.
SECURE128	Means all known to be secure ciphersuites that offer a security level 128-bit or more and a message authenticity security level of 80 bits or more.
SECURE192	Means all the known to be secure ciphersuites that offer a security level 192-bit or more and a message authenticity security level of 128 bits or more.
SECURE256	Currently alias for SECURE192.
SUITEB128	Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 128 bit security level.
SUITEB192	Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 192 bit security level.
EXPORT	Means all ciphersuites are enabled, including the low-security 40 bit ciphers.
NONE	Means nothing is enabled. This disables even protocols and compression methods. It should be followed by the algorithms to be enabled.

Table 7.2.: Supported initial keywords.

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 algorithms and protocols are shown in Table 7.3. To avoid collisions in order to specify a compression algorithm in the priority string you have to prefix it with "COMP-", protocol versions with "VERS-", signature algorithms with "SIGN-" and certificate types with "CTYPE-". All other algorithms don't need a prefix. Each specified keyword 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 DHE key exchange methods are generally slower[4] than their elliptic curves counterpart (ECDHE). Moreover the plain Diffie-Hellman key exchange requires parameters

[4]It depends on the group used. Primes with lesser bits are always faster, but also easier to break. See section 7.10 for the acceptable security levels.

Type	Keywords
Ciphers	AES-128-CBC, AES-256-CBC, AES-128-GCM, CAMELLIA-128-CBC, CAMELLIA-256-CBC, ARCFOUR-128, 3DES-CBC ARCFOUR-40. Catch all name is CIPHER-ALL which will add all the algorithms from NORMAL priority.
Key exchange	RSA, DHE-RSA, DHE-DSS, SRP, SRP-RSA, SRP-DSS, PSK, DHE-PSK, ECDHE-RSA, ANON-ECDH, ANON-DH, RSA-EXPORT. The Catch all name is KX-ALL which will add all the algorithms from NORMAL priority.
MAC	MD5, SHA1, SHA256, AEAD (used with GCM ciphers only). All algorithms from NORMAL priority can be accessed with MAC-ALL.
Compression algorithms	COMP-NULL, COMP-DEFLATE. Catch all is COMP-ALL.
TLS versions	VERS-SSL3.0, VERS-TLS1.0, VERS-TLS1.1, VERS-TLS1.2, VERS-DTLS1.0. Catch all is VERS-TLS-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. Catch all is SIGN-ALL. This is only valid for TLS 1.2 and later.
Elliptic curves	CURVE-SECP192R1, CURVE-SECP224R1, CURVE-SECP256R1, CURVE-SECP384R1, CURVE-SECP521R1. Catch all is CURVE-ALL.

Table 7.3.: The supported algorithm keywords in priority strings.

to be generated and associated with a credentials structure by the server (see subsection 7.11.3).

The available special keywords are shown in Table 7.4 and Table 7.5.

Finally the ciphersuites enabled by any priority string can be listed using the `gnutls-cli` application (see section 9.1), or by using the priority functions as in subsection 8.4.3.

Example priority strings are:

```
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 except ARCFOUR-128:
    "NORMAL:-ARCFOUR-128"

Enabling the 128-bit secure ciphers, while disabling SSL 3.0 and enabling compression:
    "SECURE128:-VERS-SSL3.0:+COMP-DEFLATE"

Enabling the 128-bit and 192-bit secure ciphers, while disabling all TLS versions
except TLS 1.2:
```

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 would disable TLS record random padding and tolerate packets over the maximum allowed TLS record.
%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.
%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. This is the default.
%LATEST_RECORD_VERSION	will use the latest TLS version record version in client hello.

Table 7.4.: Special priority string keywords.

```
    "SECURE128:+SECURE192:-VERS-TLS-ALL:+VERS-TLS1.2"
```

7.10. 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 7.6 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^{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

Keyword	Description
%STATELESS_COMPRESSION	will disable keeping state across records when compressing. This may help to mitigate attacks when compression is used but an attacker is in control of input data. This has to be used only when the data that are possibly controlled by an attacker are placed in separate records.
%DISABLE_SAFE_RENEGOTIATION	will completely disable safe renegotiation completely. Do not use unless you know what you are doing.
%UNSAFE_RENEGOTIATION	will allow handshakes and re-handshakes without the safe renegotiation extension. Note that for clients this mode is insecure (you may be under attack), and for servers it will allow insecure clients to connect (which could be fooled by an attacker). Do not use unless you know what you are doing and want maximum compatibility.
%PARTIAL_RENEGOTIATION	will allow initial handshakes to proceed, but not re-handshakes. This leaves the client vulnerable to attack, and servers will be compatible with non-upgraded clients for initial handshakes. This is currently the default for clients and servers, for compatibility reasons.
%SAFE_RENEGOTIATION	will enforce safe renegotiation. Clients and servers will refuse to talk to an insecure peer. Currently this causes interoperability problems, but is required for full protection.
%VERIFY_ALLOW_SIGN_RSA_MD5	will allow RSA-MD5 signatures 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.

Table 7.5.: More priority string keywords.

Security bits	RSA, DH and SRP parameter size	ECC key size	Security parameter	Description
<72	<1008	<160	`INSECURE`	Considered to be insecure
72	1008	160	`WEAK`	Short term protection against small organizations
80	1248	160	`LOW`	Very short term protection against agencies
96	1776	192	`LEGACY`	Legacy standard level
112	2432	224	`NORMAL`	Medium-term protection
128	3248	256	`HIGH`	Long term protection
256	15424	512	`ULTRA`	Foreseeable future

Table 7.6.: Key sizes and security parameters.

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).`

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.`

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 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**)

7.11. Advanced topics

7.11.1. Session resumption

Client side

To reduce time and roundtrips spent in a handshake the client can request session resumption from a server that previously shared a session with. For that 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_get_id2** (*gnutls_session_t* **session**, *gnutls_datum_t ** **session_id**)
>
> *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: `Check whether session is resumed or not.`

Returns: `non zero if this session is resumed, or a zero if this is a new session.`

Server side

In order to support resumption a server can store the session security parameters in a local database or by using session tickets (see subsection 2.6.3) to delegate storage to the client. Because session tickets might not be supported by all clients, servers could combine the two methods.

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.

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**)

A server utilizing tickets should 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`.

A server enabling both session tickets and a storage for session data would use session tickets when clients support it and the storage otherwise.

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 SessionTicket. **key** must be initialized with gnutls_session_ticket_key_generate().

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.

7.11.2. 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 functionlity to use an SSH-like trust on first use authentication. The available functions to store and verify public keys are listed below.

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 not yet active.

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

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.

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 the provided certificate using a list of stored public keys. The **service** field if non-NULL should be a port number. The **retrieve** 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.

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.

specified as below.

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 the provided certificate to the list of stored public keys. The key will be considered valid until the provided expiration time. The **store** 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.

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

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 **store** 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.

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_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 verication. If the **q** parameter is provided it will be used for caching entries.

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

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 facilitate caching. The following flags are returned by the verify functions to indicate the status of the verification.

enum **dane_verify_status_t**:

DANE_VERIFY_CA_CONSTRAINS_-VIOLATED	The CA constrains was violated.
DANE_VERIFY_CERT_DIFFERS	The certificate obtained via DNS differs.
DANE_VERIFY_NO_DANE_INFO	No DANE data were found in the DNS record.

Table 7.7.: The DANE verification status flags.

In order to generate a DANE TLSA entry to use in a DNS server you may use danetool (see section 5.7).

7.11.3. Parameter generation

Several TLS ciphersuites require additional parameters that need to be generated or provided by the application. The Diffie-Hellman based ciphersuites (ANON-DH or DHE), require the group parameters to be provided. Those can either be be generated on the fly using `gnutls_dh_params_generate2` or imported from pregenerated data using `gnutls_dh_params_import_pkcs3`. The parameters can be used in a TLS session by calling `gnutls_certificate_set_dh_params` or `gnutls_anon_set_server_dh_params` for anonymous sessions.

int **gnutls_dh_params_generate2** (*gnutls_dh_params_t* **params**, *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**)

void **gnutls_anon_set_server_dh_params** (*gnutls_anon_server_credentials_t* **res**, *gnutls_dh_params_t* **dh_params**)

Due to the time-consuming calculations required for the generation of Diffie-Hellman parameters we suggest against performing generation of them within an application. The `certtool` tool can be used to generate or export known safe values that can be stored in code or in a configuration file to provide the ability to replace. We also recommend the usage of `gnutls_sec_param_to_pk_bits` (see section 7.10) to determine the bit size of the generated parameters.

Note that the information stored in the generated PKCS #3 structure changed with GnuTLS 3.0.9. Since that version the `privateValueLength` member of the structure is set, allowing the server utilizing the parameters to use keys of the size of the security parameter. This provides better performance in key exchange.

The ciphersuites that involve the RSA-EXPORT key exchange require additional parameters. Those ciphersuites are rarely used today because they are by design insecure, thus if you have no requirement for them, the rest of this section can be skipped. The RSA-EXPORT key exchange requires 512-bit RSA keys to be generated. It is recommended those parameters to be refreshed (regenerated) in short intervals. The following functions can be used for these parameters.

int **gnutls_rsa_params_generate2** (*gnutls_rsa_params_t* **params**, *unsigned int* **bits**)

void **gnutls_certificate_set_rsa_export_params** (*gnutls_certificate_credentials_t* **res**, *gnutls_rsa_params_t* **rsa_params**)

int **gnutls_rsa_params_import_pkcs1** (*gnutls_rsa_params_t* **params**, *const gnutls_datum_t ** **pkcs1_params**, *gnutls_x509_crt_fmt_t* **format**)

int **gnutls_rsa_params_export_pkcs1** (*gnutls_rsa_params_t* **params**, *gnutls_x509_crt_fmt_t* **format**, *unsigned char ** **params_data**, *size_t ** **params_data_size**)

To allow renewal of the parameters within an application without accessing the credentials, which are a shared structure, an alternative interface is available using a callback function.

void **gnutls_certificate_set_params_function** (*gnutls_certificate_credentials_t* **res**, *gnutls_params_function ** **func**)

Description: `This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for certificate authentication. The callback should return` **GNUTLS_E_SUCCESS** `(0) on success.`

7.11.4. Keying material exporters

The TLS PRF can be used by other protocols to derive keys based on the TLS master secret. The API to use is `gnutls_prf`. The function needs to be provided with the label in the parameter `label`, and the extra data to mix in the `extra` parameter. Depending on whether you want to mix in the client or server random data first, you can set the `server_random_first` parameter.

For example, after establishing a TLS session using `gnutls_handshake`, you can invoke the TLS PRF with this call:

```
#define MYLABEL "EXPORTER-FOO"
#define MYCONTEXT "some context data"
char out[32];
rc = gnutls_prf (session, strlen (MYLABEL), MYLABEL, 0,
                 strlen (MYCONTEXT), MYCONTEXT, 32, out);
```

If you don't want to mix in the client/server random, there is a low-level TLS PRF interface called `gnutls_prf_raw`.

7.11.5. 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 [35].

In GnuTLS you can extract a channel binding using the `gnutls_session_channel_binding` function. Currently only the type `GNUTLS_CB_TLS_UNIQUE` is supported, which corresponds to the `tls-unique` channel binding for TLS defined in [4].

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");
    }
}
```

7.11.6. 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 7.9). A conservative priority string that would disable certain TLS protocol options that are known to cause compatibility problems, is shown below.

`NORMAL:%COMPAT`

For broken peers that do not tolerate TLS version numbers over TLS 1.0 another priority string is:

`NORMAL:-VERS-TLS-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. Note however that there are known attacks against those protocol versions, especially over the CBC-mode ciphersuites. To mitigate them another priority string that only allows the stream cipher ARCFOUR is below.

`NORMAL:-VERS-TLS-ALL:+VERS-TLS1.0:+VERS-SSL3.0:-CIPHER-ALL:+ARCFOUR-128:%COMPAT`

7.11.7. 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.

8. 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.

8.1. 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.

8.1.1. Simple 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 or OpenPGP 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.

```
/* 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 "examples.h"

/* A very basic TLS client, with X.509 authentication and server certificate
 * verification. Note that error checking for missing files etc. is missing
 * for simplicity.
 */

#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];
  const char *err;
  gnutls_certificate_credentials_t xcred;

  gnutls_global_init ();

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  /* sets the trusted cas file
   */
  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
   */
  gnutls_init (&session, GNUTLS_CLIENT);

  gnutls_session_set_ptr (session, (void *) "my_host_name");

  gnutls_server_name_set (session, GNUTLS_NAME_DNS, "my_host_name",
                          strlen("my_host_name"));

  /* Use default priorities */
  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);
    }

  /* put the x509 credentials to the current session
   */
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

  /* connect to the peer
   */
  sd = tcp_connect ();

  gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) 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
    {
      printf ("- Handshake was completed\n");
    }

  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);

  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 ret, 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.
   */
  ret = gnutls_certificate_verify_peers3 (session, hostname, &status);
  if (ret < 0)
    {
      printf ("Error\n");
      return GNUTLS_E_CERTIFICATE_ERROR;
    }

  type = gnutls_certificate_type_get (session);

  ret = gnutls_certificate_verification_status_print( status, type, &out, 0);
  if (ret < 0)
    {
      printf ("Error\n");
      return GNUTLS_E_CERTIFICATE_ERROR;
    }

  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;
}

```

8.1.2. Simple 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 "examples.h"

/* 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.
   */
  ret = gnutls_certificate_verify_peers3 (session, hostname, &status);
  if (ret < 0)
    {
      printf ("Error\n");
      return GNUTLS_E_CERTIFICATE_ERROR;
    }

  type = gnutls_certificate_type_get (session);

  ret = gnutls_certificate_verification_status_print( status, type, &out, 0);
  if (ret < 0)
    {
      printf ("Error\n");
      return GNUTLS_E_CERTIFICATE_ERROR;
    }

  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)
    {
      ret = gnutls_store_pubkey(NULL, NULL, hostname, "https",
                                type, &cert_list[0], 0, 0);
      if (ret < 0)
        printf("gnutls_store_pubkey: %s\n", gnutls_strerror(ret));
    }

  /* notify gnutls to continue handshake normally */
  return 0;
}

```

8.1.3. Simple 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 <gnutls/gnutls.h>

/* A very basic TLS client, with anonymous authentication.
 */

#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_ptr (session, (gnutls_transport_ptr_t) 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
    {
      printf ("- Handshake was completed\n");
    }

  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);

  gnutls_bye (session, GNUTLS_SHUT_RDWR);

end:

  tcp_close (sd);

  gnutls_deinit (session);

  gnutls_anon_free_client_credentials (anoncred);

  gnutls_global_deinit ();

  return 0;
}
```

8.1.4. Simple 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 <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/dtls.h>

/* A very basic Datagram TLS client, over UDP with X.509 authentication.
 */

#define MAX_BUF 1024
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#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];
  const char *err;
  gnutls_certificate_credentials_t xcred;

  gnutls_global_init ();

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  /* sets the trusted cas file */
  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);
  gnutls_certificate_set_verify_function (xcred, verify_certificate_callback);

  /* Initialize TLS session */
  gnutls_init (&session, GNUTLS_CLIENT | GNUTLS_DATAGRAM);

  /* Use default priorities */
  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);
    }

  /* put the x509 credentials to the current session */
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);
  gnutls_server_name_set (session, GNUTLS_NAME_DNS, "my_host_name",
                          strlen("my_host_name"));

  /* connect to the peer */
  sd = udp_connect ();

  gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

  /* set the connection MTU */
  gnutls_dtls_set_mtu (session, 1000);
  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
    {
      printf ("- Handshake was completed\n");
    }

  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);

  /* It is suggested not to use GNUTLS_SHUT_RDWR in DTLS
   * connections because the peer's closure message might
   * be lost */
  gnutls_bye (session, GNUTLS_SHUT_WR);

end:

  udp_close (sd);

  gnutls_deinit (session);

  gnutls_certificate_free_credentials (xcred);

  gnutls_global_deinit ();

  return 0;
}
```

8.1.5. 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)
{
  const char *tmp;
  gnutls_credentials_type_t cred;
  gnutls_kx_algorithm_t kx;
  int dhe, ecdh;

  dhe = ecdh = 0;

  /* print the key exchange's algorithm name
```

```
   */
  kx = gnutls_kx_get (session);
  tmp = gnutls_kx_get_name (kx);
  printf ("- Key Exchange: %s\n", tmp);

  /* Check the authentication type used and switch
   * to the appropriate.
   */
  cred = gnutls_auth_get_type (session);
  switch (cred)
    {
    case GNUTLS_CRD_IA:
      printf ("- TLS/IA session\n");
      break;


#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);

    }                           /* switch */

  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));

  /* print the protocol's name (ie TLS 1.0)
   */
  tmp = gnutls_protocol_get_name (gnutls_protocol_get_version (session));
  printf ("- Protocol: %s\n", tmp);

  /* print the certificate type of the peer.
   * ie X.509
   */
  tmp =
    gnutls_certificate_type_get_name (gnutls_certificate_type_get (session));

  printf ("- Certificate Type: %s\n", tmp);

  /* print the compression algorithm (if any)
   */
  tmp = gnutls_compression_get_name (gnutls_compression_get (session));
  printf ("- Compression: %s\n", tmp);

  /* print the name of the cipher used.
   * ie 3DES.
   */
  tmp = gnutls_cipher_get_name (gnutls_cipher_get (session));
  printf ("- Cipher: %s\n", tmp);

  /* Print the MAC algorithms name.
   * ie SHA1
   */
  tmp = gnutls_mac_get_name (gnutls_mac_get (session));
  printf ("- MAC: %s\n", tmp);

  return 0;
}
```

8.1.6. 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 <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 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)
{
  int ret;
  gnutls_datum_t data;

  ret = gnutls_load_file (CERT_FILE, &data);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading certificate file.\n");
      exit (1);
    }
```

```

  ret = gnutls_pcert_import_x509_raw (&pcrt, &data, GNUTLS_X509_FMT_PEM, 0);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading certificate file: %s\n",
               gnutls_strerror (ret));
      exit (1);
    }

  gnutls_free(data.data);

  ret = gnutls_load_file (KEY_FILE, &data);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading key file.\n");
      exit (1);
    }

  gnutls_privkey_init (&key);

  ret = gnutls_privkey_import_x509_raw (key, &data, GNUTLS_X509_FMT_PEM, NULL, 0);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading key file: %s\n",
               gnutls_strerror (ret));
      exit (1);
    }

  gnutls_free(data.data);
}

int
main (void)
{
  int ret, sd, ii;
  gnutls_session_t session;
  gnutls_priority_t priorities_cache;
  char buffer[MAX_BUF + 1];
  gnutls_certificate_credentials_t xcred;
  /* Allow connections to servers that have OpenPGP keys as well.
   */

  gnutls_global_init ();

  load_keys ();

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  /* priorities */
  gnutls_priority_init (&priorities_cache, "NORMAL", NULL);


  /* sets the trusted cas file
   */
  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);

  gnutls_certificate_set_retrieve_function2 (xcred, cert_callback);
```

```

  /* Initialize TLS session
   */
  gnutls_init (&session, GNUTLS_CLIENT);

  /* Use default priorities */
  gnutls_priority_set (session, priorities_cache);

  /* put the x509 credentials to the current session
   */
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

  /* connect to the peer
   */
  sd = tcp_connect ();

  gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

  /* Perform the TLS handshake
   */
  ret = gnutls_handshake (session);

  if (ret < 0)
    {
      fprintf (stderr, "*** Handshake failed\n");
      gnutls_perror (ret);
      goto end;
    }
  else
    {
      printf ("- Handshake was completed\n");
    }

  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);

  gnutls_bye (session, GNUTLS_SHUT_RDWR);

end:

```

```
  tcp_close (sd);

  gnutls_deinit (session);

  gnutls_certificate_free_credentials (xcred);
  gnutls_priority_deinit (priorities_cache);

  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;

}
```

8.1.7. Verifying a certificate

An example is listed below which uses the high level verification functions to verify a given certificate list.

```
/* 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 "examples.h"

/* 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;

  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.
   */
  gnutls_x509_trust_list_init (&tlist, 0);

  gnutls_x509_trust_list_add_cas (tlist, ca_list, ca_list_size, 0);
  gnutls_x509_trust_list_add_crls (tlist, crl_list, crl_list_size,
                                   GNUTLS_TL_VERIFY_CRL, 0);

  cert = malloc (sizeof (*cert) * cert_chain_length);

  /* Import all the certificates in the chain to
   * native certificate format.
   */
  for (i = 0; i < cert_chain_length; i++)
    {
      gnutls_x509_crt_init (&cert[i]);
      gnutls_x509_crt_import (cert[i], &cert_chain[i], GNUTLS_X509_FMT_DER);
    }

  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)
    {
      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");

      if (output & GNUTLS_CERT_SIGNER_NOT_FOUND)
        fprintf (stderr, ": no issuer was found");
      if (output & GNUTLS_CERT_SIGNER_NOT_CA)
        fprintf (stderr, ": issuer is not a CA");
      if (output & GNUTLS_CERT_NOT_ACTIVATED)
        fprintf (stderr, ": not yet activated\n");
      if (output & GNUTLS_CERT_EXPIRED)
        fprintf (stderr, ": expired\n");

      fprintf (stderr, "\n");
    }
  else
    fprintf (stderr, "Trusted\n");

  /* Check if the name in the first certificate matches our destination!
```

```
   */
  if (!gnutls_x509_crt_check_hostname (cert[0], hostname))
    {
      printf ("The certificate's owner does not match hostname '%s'\n",
              hostname);
    }

  gnutls_x509_trust_list_deinit (tlist, 1);

  return;
}

static int
print_details_func (gnutls_x509_crt_t cert,
                    gnutls_x509_crt_t issuer, gnutls_x509_crl_t crl,
                    unsigned int verification_output)
{
  char name[512];
  char issuer_name[512];
  size_t name_size;
  size_t issuer_name_size;

  issuer_name_size = sizeof (issuer_name);
  gnutls_x509_crt_get_issuer_dn (cert, issuer_name, &issuer_name_size);

  name_size = sizeof (name);
  gnutls_x509_crt_get_dn (cert, name, &name_size);

  fprintf (stdout, "\tSubject: %s\n", name);
  fprintf (stdout, "\tIssuer: %s\n", issuer_name);

  if (issuer != NULL)
    {
      issuer_name_size = sizeof (issuer_name);
      gnutls_x509_crt_get_dn (issuer, issuer_name, &issuer_name_size);

      fprintf (stdout, "\tVerified against: %s\n", issuer_name);
    }

  if (crl != NULL)
    {
      issuer_name_size = sizeof (issuer_name);
      gnutls_x509_crl_get_issuer_dn (crl, issuer_name, &issuer_name_size);

      fprintf (stdout, "\tVerified against CRL of: %s\n", issuer_name);
    }

  fprintf (stdout, "\tVerification output: %x\n\n", verification_output);

  return 0;
}
```

8.1.8. 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.

```
/* 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 <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 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: ");
  if (password == NULL || password[0] == 0)
    {
      fprintf (stderr, "No password given\n");
      exit (1);
    }

  len = MIN (pin_max, strlen (password));
  memcpy (pin, password, len);
  pin[len] = 0;

  return 0;
}

int
main (void)
{
  int ret, sd, ii;
  gnutls_session_t session;
  gnutls_priority_t priorities_cache;
  char buffer[MAX_BUF + 1];
  gnutls_certificate_credentials_t xcred;
  /* Allow connections to servers that have OpenPGP keys as well.
   */

  gnutls_global_init ();
  /* PKCS11 private key operations might require PIN.
   * Register a callback.
   */
  gnutls_pkcs11_set_pin_function (pin_callback, NULL);

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  /* priorities */
  gnutls_priority_init (&priorities_cache, "NORMAL", NULL);

  /* sets the trusted cas file
   */
  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);

  gnutls_certificate_set_x509_key_file (xcred, CERT_URL, KEY_URL, GNUTLS_X509_FMT_DER);

  /* Initialize TLS session
   */
  gnutls_init (&session, GNUTLS_CLIENT);

  /* Use default priorities */
  gnutls_priority_set (session, priorities_cache);

  /* put the x509 credentials to the current session
   */
```

```
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

  /* connect to the peer
   */
  sd = tcp_connect ();

  gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

  /* Perform the TLS handshake
   */
  ret = gnutls_handshake (session);

  if (ret < 0)
    {
      fprintf (stderr, "*** Handshake failed\n");
      gnutls_perror (ret);
      goto end;
    }
  else
    {
      printf ("- Handshake was completed\n");
    }

  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);

  gnutls_bye (session, GNUTLS_SHUT_RDWR);

end:

  tcp_close (sd);

  gnutls_deinit (session);

  gnutls_certificate_free_credentials (xcred);
  gnutls_priority_deinit (priorities_cache);

  gnutls_global_deinit ();

  return 0;
```

```
}
```

8.1.9. 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 <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 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_certificate_credentials_t xcred;

  /* variables used in session resuming
   */
  int t;
  char *session_data = NULL;
  size_t session_data_size = 0;

  gnutls_global_init ();

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);

  for (t = 0; t < 2; t++)
    {                           /* connect 2 times to the server */

      sd = tcp_connect ();
```

```

      gnutls_init (&session, GNUTLS_CLIENT);

      gnutls_priority_set_direct (session, "PERFORMANCE:!ARCFOUR-128", NULL);

      gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);

      if (t > 0)
        {
          /* if this is not the first time we connect */
          gnutls_session_set_data (session, session_data, session_data_size);
          free (session_data);
        }

      gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) 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
        {
          printf ("- Handshake was completed\n");
        }

      if (t == 0)
        {                       /* the first time we connect */
          /* get the session data size */
          gnutls_session_get_data (session, NULL, &session_data_size);
          session_data = malloc (session_data_size);

          /* put session data to the session variable */
          gnutls_session_get_data (session, session_data, &session_data_size);

        }
      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");
            }
```

```
      }

      /* This function was defined in a previous example
       */
      /* print_info(session); */

      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);

      gnutls_bye (session, GNUTLS_SHUT_RDWR);

    end:

      tcp_close (sd);

      gnutls_deinit (session);

    }                           /* for() */

  gnutls_certificate_free_credentials (xcred);

  gnutls_global_deinit ();

  return 0;
}
```

8.1.10. Simple 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;

  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_ptr (session, (gnutls_transport_ptr_t) 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
    {
      printf ("- Handshake was completed\n");
    }

  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;
}
```

8.1.11. Simple 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;
}
```

8.1.12. 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);
}
```

8.1.13. 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, optval;
  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);
}
```

8.2. Server examples

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

8.2.1. 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>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"
```

```

/* This is a sample TLS 1.0 echo server, using X.509 authentication.
 */

#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */

/* These are global */
gnutls_certificate_credentials_t x509_cred;
gnutls_priority_t priority_cache;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set (session, priority_cache);

  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.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_IGNORE);

  return session;
}

static gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{
  int bits = gnutls_sec_param_to_pk_bits (GNUTLS_PK_DH, GNUTLS_SEC_PARAM_LOW);

  /* Generate Diffie-Hellman parameters - for use with DHE
   * kx algorithms. When short bit length is used, it might
   * be wise to regenerate parameters often.
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, bits);

  return 0;
}

int
main (void)
{
  int 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;
  char buffer[MAX_BUF + 1];
```

```
  int optval = 1;

  /* this must be called once in the program
   */
  gnutls_global_init ();

  gnutls_certificate_allocate_credentials (&x509_cred);
  /* gnutls_certificate_set_x509_system_trust(xcred); */
  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);
    }

  generate_dh_params ();

  gnutls_priority_init (&priority_cache, "PERFORMANCE:%SERVER_PRECEDENCE", NULL);


  gnutls_certificate_set_dh_params (x509_cred, dh_params);

  /* 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 (;;)
    {
      session = initialize_tls_session ();

      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_ptr (session, (gnutls_transport_ptr_t) 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 (;;)
        {
          memset (buffer, 0, MAX_BUF + 1);
          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)
            {
              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, strlen (buffer));
            }
        }
      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_certificate_free_credentials (x509_cred);
  gnutls_priority_deinit (priority_cache);

```

```
  gnutls_global_deinit ();

  return 0;

}
```

8.2.2. Echo server with OpenPGP authentication

The following example is an echo server which supports OpenPGP key authentication. You can easily combine this functionality —that is have a server that supports both X.509 and OpenPGP certificates— but we separated them to keep these examples as simple as possible.

```
/* 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 <gnutls/openpgp.h>

#define KEYFILE "secret.asc"
#define CERTFILE "public.asc"
#define RINGFILE "ring.gpg"

/* This is a sample TLS 1.0-OpenPGP 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 */
#define DH_BITS 1024

/* These are global */
gnutls_certificate_credentials_t cred;
gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{

  /* Generate Diffie-Hellman parameters - for use with DHE
   * kx algorithms. These should be discarded and regenerated
   * once a day, once a week or once a month. Depending on the
   * security requirements.
```

```
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, DH_BITS);

  return 0;
}

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set_direct (session, "NORMAL:+CTYPE-OPENPGP", NULL);

  /* request client certificate if any.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_REQUEST);

  gnutls_dh_set_prime_bits (session, DH_BITS);

  return session;
}

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;
  char buffer[MAX_BUF + 1];
  int optval = 1;
  char name[256];

  strcpy (name, "Echo Server");

  /* this must be called once in the program
   */
  gnutls_global_init ();

  gnutls_certificate_allocate_credentials (&cred);
  gnutls_certificate_set_openpgp_keyring_file (cred, RINGFILE,
                                               GNUTLS_OPENPGP_FMT_BASE64);

  gnutls_certificate_set_openpgp_key_file (cred, CERTFILE, KEYFILE,
                                           GNUTLS_OPENPGP_FMT_BASE64);

  generate_dh_params ();

  gnutls_certificate_set_dh_params (cred, dh_params);

  /* 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 (;;)
    {
      session = initialize_tls_session ();

      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_ptr (session, (gnutls_transport_ptr_t) sd);
      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 (;;)
        {
          memset (buffer, 0, MAX_BUF + 1);
          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)
            {
              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, strlen (buffer));
            }
        }
      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_certificate_free_credentials (cred);

  gnutls_global_deinit ();

  return 0;

}
```

8.2.3. 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"

/* 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 */

/* These are global */
gnutls_srp_server_credentials_t srp_cred;
gnutls_certificate_credentials_t cert_cred;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  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);

  /* request client certificate if any.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_IGNORE);

  return session;
}

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;
  char buffer[MAX_BUF + 1];
  int optval = 1;
  char name[256];

  strcpy (name, "Echo Server");

  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 (;;)
    {
      session = initialize_tls_session ();

      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_ptr (session, (gnutls_transport_ptr_t) 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");
      printf ("- User %s was connected\n", gnutls_srp_server_get_username(session));

      /* print_info(session); */

```

```
      for (;;)
        {
          memset (buffer, 0, MAX_BUF + 1);
          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)
            {
              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, strlen (buffer));
            }
        }
      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_srp_free_server_credentials (srp_cred);
  gnutls_certificate_free_credentials (cert_cred);

  gnutls_global_deinit ();

  return 0;

}
```

8.2.4. 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 */
#define DH_BITS 1024

/* These are global */
gnutls_anon_server_credentials_t anoncred;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set_direct (session, "NORMAL:+ANON-ECDH:+ANON-DH", NULL);

  gnutls_credentials_set (session, GNUTLS_CRD_ANON, anoncred);

  gnutls_dh_set_prime_bits (session, DH_BITS);

  return session;
}

static gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{

  /* Generate Diffie-Hellman parameters - for use with DHE
   * kx algorithms. These should be discarded and regenerated
   * once a day, once a week or once a month. Depending on the
   * security requirements.
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, DH_BITS);

  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;
  char buffer[MAX_BUF + 1];
  int optval = 1;

  /* this must be called once in the program
   */
  gnutls_global_init ();

  gnutls_anon_allocate_server_credentials (&anoncred);

  generate_dh_params ();

  gnutls_anon_set_server_dh_params (anoncred, dh_params);

  /* 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 (;;)
    {
      session = initialize_tls_session ();

      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_ptr (session, (gnutls_transport_ptr_t) ((ptrdiff_t) 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 (;;)
        {
          memset (buffer, 0, MAX_BUF + 1);
          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)
            {
              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, strlen (buffer));
            }
        }
      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;

}
```

8.2.5. 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 MAX_BUFFER 1024
#define PORT 5556

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);
static gnutls_session_t initialize_tls_session (void);
static int generate_dh_params (void);

/* Use global credentials and parameters to simplify
 * the example. */
static gnutls_certificate_credentials_t x509_cred;
```

```
static gnutls_priority_t priority_cache;
static gnutls_dh_params_t dh_params;

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);
    }

  generate_dh_params ();

  gnutls_certificate_set_dh_params (x509_cred, dh_params);

  gnutls_priority_init (&priority_cache,
                        "PERFORMANCE:-VERS-TLS-ALL:+VERS-DTLS1.0:%SERVER_PRECEDENCE",
                        NULL);

  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;

      session = initialize_tls_session ();
      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);

      do
        {
          ret = gnutls_handshake (session);
        }
      while (ret < 0 && gnutls_error_is_fatal (ret) == 0);

      if (ret < 0)
        {
          fprintf (stderr, "Error in handshake(): %s\n",
                   gnutls_strerror (ret));
          gnutls_deinit (session);
          continue;
        }

      printf ("- Handshake was completed\n");

      for (;;)
        {
          do
            {
              ret = gnutls_record_recv_seq (session, buffer, MAX_BUFFER,
                                            sequence);
            }
          while (ret == GNUTLS_E_AGAIN || ret == GNUTLS_E_INTERRUPTED);

          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 */
          ret = gnutls_record_send (session, buffer, ret);
          if (ret < 0)
            {
              fprintf (stderr, "Error in send(): %s\n",
                       gnutls_strerror (ret));
              break;
            }
        }

      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 = 0;
  tv.tv_usec = ms * 1000;

  while(tv.tv_usec >= 1000000)
    {
      tv.tv_usec -= 1000000;
      tv.tv_sec++;
    }

  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;
}

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER | GNUTLS_DATAGRAM);

  gnutls_priority_set (session, priority_cache);

  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, x509_cred);

  return session;
}

static int
generate_dh_params (void)
{
  int bits = gnutls_sec_param_to_pk_bits (GNUTLS_PK_DH, GNUTLS_SEC_PARAM_LOW);

  /* Generate Diffie-Hellman parameters - for use with DHE
   * kx algorithms. When short bit length is used, it might
   * be wise to regenerate parameters often.
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, bits);

  return 0;
}
```

8.3. 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);
static int
_verify_response (gnutls_datum_t * data, gnutls_x509_crt_t cert,
                  gnutls_x509_crt_t signer);

/* 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;

    gnutls_global_init ();

    if (argc > 4)
        hostname = argv[4];

    cert = load_cert (cert_file);
    issuer = load_cert (issuer_file);
    signer = load_cert (signer_file);

    if (hostname == NULL)
      {

```

```
          for (seq = 0;; seq++)
            {
                ret = gnutls_x509_crt_get_authority_info_access (cert, seq,
                                                                 GNUTLS_IA_OCSP_URI,
                                                                 &tmp,
                                                                 NULL);
                if (ret == GNUTLS_E_UNKNOWN_ALGORITHM)
                    continue;
                if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
                  {
                      fprintf (stderr,
                               "No URI was found in the certificate.\n");
                      exit (1);
                  }
                if (ret < 0)
                  {
                      fprintf (stderr, "error: %s\n",
                               gnutls_strerror (ret));
                      exit (1);
                  }

                printf ("CA issuers URI: %.*s\n", tmp.size, tmp.data);

                hostname = malloc (tmp.size + 1);
                memcpy (hostname, tmp.data, tmp.size);
                hostname[tmp.size] = 0;

                gnutls_free (tmp.data);
                break;
            }

      }

    /* Note that the OCSP servers hostname might be available
     * using gnutls_x509_crt_get_authority_info_access() in the issuer's
     * certificate */

    memset (&ud, 0, sizeof (ud));
    fprintf (stderr, "Connecting to %s\n", hostname);

    _generate_request (&req, cert, issuer);

#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);

    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_binary_file (cert_file, &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_ocsp_req_t req;
    int ret;
    unsigned char noncebuf[23];
    gnutls_datum_t nonce = { noncebuf, sizeof (noncebuf) };

    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_rnd (GNUTLS_RND_RANDOM, nonce.data, nonce.size);
    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_ocsp_resp_t resp;
    int ret;
    unsigned verify;

    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_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_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;
}
```

8.4. Miscellaneous examples

8.4.1. 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));
    }
}
```

8.4.2. 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);

    }
}
```

8.4.3. 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]);
}
```

9. 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.

9.1. Invoking gnutls-cli

Simple client program to set up a TLS connection to some other computer. It sets up a TLS connection and forwards data from the standard input to the secured socket and vice versa.

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

gnutls-cli help/usage (-h)

This is the automatically generated usage text for gnutls-cli. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  gnutls-cli [ -<flag> [<val>] | --<name>[{=| }<val>] ]... [hostname]

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
   -V, --verbose              More verbose output
                                - may appear multiple times
       --tofu                 Enable trust on first use authentication
                                - disabled as --no-tofu
       --dane                 Enable DANE certificate verification (DNSSEC)
                                - disabled as --no-dane
       --local-dns            Use the local DNS server for DNSSEC resolving.
                                - disabled as --no-local-dns
       --ca-verification      Disable CA certificate verification
                                - disabled as --no-ca-verification
                                - enabled by default
       --ocsp                 Enable OCSP certificate verification
                                - disabled as --no-ocsp
   -r, --resume               Establish a session and resume
   -b, --heartbeat            Activate heartbeat support
```

```
   -e, --rehandshake          Establish a session and rehandshake
       --noticket             Don't accept session tickets
   -s, --starttls             Connect, establish a plain session and start TLS.
   -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
       --crlf                 Send CR LF instead of LF
       --x509fmtder           Use DER format for certificates to read from
   -f, --fingerprint          Send the openpgp fingerprint, instead of the key
       --disable-extensions   Disable all the TLS extensions
       --print-cert           Print peer's certificate in PEM format
       --recordsize=num       The maximum record size to advertize
                                - It must be in the range:
                                  0 to 4096
       --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
       --pgpkeyfile=file      PGP Key file to use
                                - file must pre-exist
       --pgpkeyring=file      PGP Key ring file to use
                                - file must pre-exist
       --pgpcertfile=file     PGP Public Key (certificate) 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
       --pgpsubkey=str        PGP subkey to use (hex or auto)
       --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
       --benchmark-ciphers    Benchmark individual ciphers
       --benchmark-soft-ciphers  Benchmark individual software ciphers (no hw acceleration)
       --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
   -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:  bug-gnutls@gnu.org
```

debug option (-d)

This is the “enable debugging.” option. This option takes an argument number. 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.

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 “disable ca certificate verification” option.

This option has some usage constraints. It:

- is enabled by default.

This option will 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.

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.

disable-extensions option

This is the “disable all the tls extensions” option. This option disables all TLS extensions. Deprecated option. Use the priority string.

dh-bits option

This is the “the minimum number of bits allowed for dh” option. This option takes an argument number. 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 an argument string. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as PERFORMANCE, NORMAL, SECURE128, SECURE256.

Check the GnuTLS manual on section “Priority strings” for more information on allowed keywords

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.

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.

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
```

9.2. Invoking gnutls-serv

Server program that listens to incoming TLS connections.

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

gnutls-serv help/usage (-h)

This is the automatically generated usage text for gnutls-serv. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  gnutls-serv [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num            Enable debugging.
                                - It must be in the range:
                                  0 to 9999
       --noticket             Don't accept session tickets
   -g, --generate             Generate Diffie-Hellman and RSA-export parameters
   -q, --quiet                Suppress some messages
       --nodb                 Do not use a resumption database
       --http                 Act as an HTTP server
       --echo                 Act as an Echo server
   -u, --udp                  Use DTLS (datagram TLS) over UDP
       --mtu=num              Set MTU for datagram TLS
                                - It must be in the range:
                                  0 to 17000
       --srtp-profiles=str    Offer SRTP profiles
   -a, --disable-client-cert  Do not request a client certificate
   -r, --require-client-cert  Require a client 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
       --pgpkeyfile=file      PGP Key file to use
                                - file must pre-exist
       --pgpkeyring=file      PGP Key ring file to use
                                - file must pre-exist
       --pgpcertfile=file     PGP Public Key (certificate) 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
       --x509dsakeyfile=str   Alternative X.509 key file or PKCS #11 URL to use
       --x509dsacertfile=str  Alternative X.509 Certificate file or PKCS #11 URL to use
```

```
       --x509ecckeyfile=str   Alternative X.509 key file or PKCS #11 URL to use
       --x509ecccertfile=str  Alternative X.509 Certificate file or PKCS #11 URL to use
       --pgpsubkey=str        PGP subkey to use (hex or auto)
       --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=file   The OCSP response to send to client
                                - file must pre-exist
   -p, --port=num             The port to connect to
   -l, --list                 Print a list of the supported algorithms and modes
   -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:  bug-gnutls@gnu.org
```

debug option (-d)

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

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 an argument string. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as PERFORMANCE, NORMAL, SECURE128, SECURE256.

Check the GnuTLS manual on section "Priority strings" for more information on allowed keywords

ocsp-response option

This is the “the ocsp response to send to client” option. This option takes an argument file. If the client requested an OCSP response, return data from this file to the client.

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.

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
```

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 DSA, you can also create a DSA key and certificate for the server. These credentials will be used in the final example below.

```
$ certtool --generate-privkey --dsa > x509-server-key-dsa.pem
$ certtool --generate-certificate --load-privkey x509-server-key-dsa.pem \
  --load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
  --template server.tmpl --outfile x509-server-dsa.pem
...
```

The next step is to create OpenPGP credentials for the server.

```
gpg --gen-key
...enter whatever details you want, use 'test.gnutls.org' as name...
```

Make a note of the OpenPGP key identifier of the newly generated key, here it was `5D1D14D8`. You will need to export the key for GnuTLS to be able to use it.

```
gpg -a --export 5D1D14D8 > openpgp-server.txt
gpg --export 5D1D14D8 > openpgp-server.bin
gpg --export-secret-keys 5D1D14D8 > openpgp-server-key.bin
gpg -a --export-secret-keys 5D1D14D8 > openpgp-server-key.txt
```

Let's start the server with support for OpenPGP credentials:

```
gnutls-serv --http \
            --pgpkeyfile openpgp-server-key.txt \
            --pgpcertfile openpgp-server.txt
```

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 \
            --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 \
            --pskpasswd psk-passwd.txt
```

Finally, we start the server with all the earlier parameters and you get this command:

```
gnutls-serv --http \
            --x509cafile x509-ca.pem \
            --x509keyfile x509-server-key.pem \
            --x509certfile x509-server.pem \
            --x509dsakeyfile x509-server-key-dsa.pem \
            --x509dsacertfile x509-server-dsa.pem \
            --pgpkeyfile openpgp-server-key.txt \
            --pgpcertfile openpgp-server.txt \
            --srppasswdconf srp-tpasswd.conf \
            --srppasswd srp-passwd.txt \
            --pskpasswd psk-passwd.txt
```

9.3. Invoking gnutls-cli-debug

TLS debug client. It sets up multiple TLS connections to a server and queries its capabilities. It was created to assist in debugging GnuTLS, but it might be useful to extract a TLS server's capabilities. It connects to a TLS server, performs tests and print the server's capabilities. If called with the '-v' parameter more checks will be performed. Can be used to check for servers with special needs or bugs.

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

gnutls-cli-debug help/usage (-h)

This is the automatically generated usage text for gnutls-cli-debug. The text printed is the same whether for the `help` option (-h) or the `more-help` option (-!). `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 - Ver. 3.1.4
USAGE:  gnutls-cli-debug [ -<flag> [<val>] | --<name>[{=| }<val>] ]...

   -d, --debug=num             Enable debugging.
                                 - It must be in the range:
                                   0 to 9999
   -V, --verbose               More verbose output
                                 - may appear multiple times
   -p, --port=num              The port to connect to
                                 - It must be in the range:
                                   0 to 65536
   -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:  bug-gnutls@gnu.org
```

debug option (-d)

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

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

```
$ ../src/gnutls-cli-debug localhost
Resolving 'localhost'...
Connecting to '127.0.0.1:443'...
Checking for SSL 3.0 support... yes
Checking whether %COMPAT is required... no
Checking for TLS 1.0 support... yes
Checking for TLS 1.1 support... no
Checking fallback from TLS 1.1 to... TLS 1.0
Checking for TLS 1.2 support... no
Checking whether we need to disable TLS 1.0... N/A
Checking for Safe renegotiation support... yes
Checking for Safe renegotiation support (SCSV)... yes
Checking for HTTPS server name... not checked
Checking for version rollback bug in RSA PMS... no
Checking for version rollback bug in Client Hello... no
Checking whether the server ignores the RSA PMS version... no
Checking whether the server can accept Hello Extensions... yes
Checking whether the server can accept small records (512 bytes)... yes
Checking whether the server can accept cipher suites not in SSL 3.0 spec... yes
Checking whether the server can accept a bogus TLS record version in the client hello... yes
Checking for certificate information... N/A
Checking for trusted CAs... N/A
Checking whether the server understands TLS closure alerts... partially
Checking whether the server supports session resumption... yes
Checking for export-grade ciphersuite support... no
Checking RSA-export ciphersuite info... N/A
Checking for anonymous authentication support... no
Checking anonymous Diffie-Hellman group info... N/A
Checking for ephemeral Diffie-Hellman support... no
Checking ephemeral Diffie-Hellman group info... N/A
Checking for ephemeral EC Diffie-Hellman support... yes
Checking ephemeral EC Diffie-Hellman group info...
```

```
 Curve SECP256R1
Checking for AES-GCM cipher support... no
Checking for AES-CBC cipher support... yes
Checking for CAMELLIA cipher support... no
Checking for 3DES-CBC cipher support... yes
Checking for ARCFOUR 128 cipher support... yes
Checking for ARCFOUR 40 cipher support... no
Checking for MD5 MAC support... yes
Checking for SHA1 MAC support... yes
Checking for SHA256 MAC support... no
Checking for ZLIB compression support... no
Checking for max record size... no
Checking for OpenPGP authentication support... no
```

10. 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.

10.1. The TLS Protocol

The main use case for the TLS protocol is shown in Figure 10.1. 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.

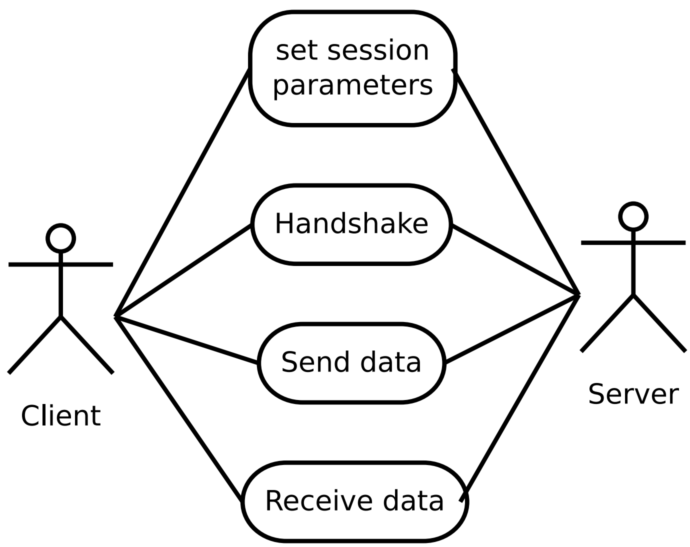

Figure 10.1.: TLS protocol use case.

10.2. 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 10.2.

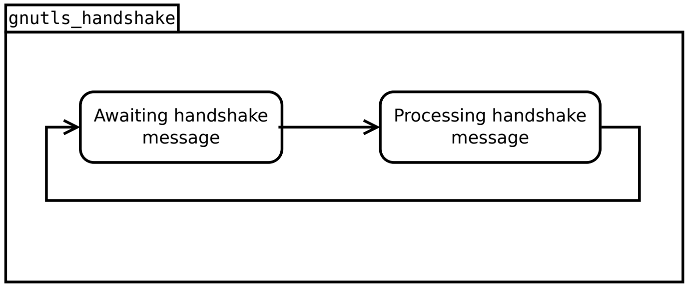

Figure 10.2.: GnuTLS handshake state machine.

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 10.3.

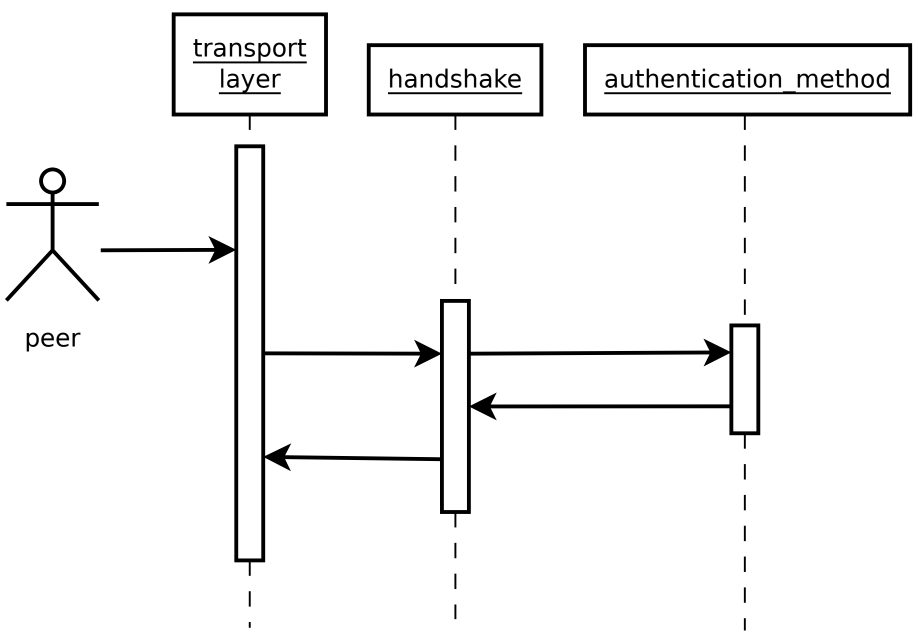

Figure 10.3.: GnuTLS handshake process sequence.

10.3. 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.

10.4. TLS Extension Handling

As with authentication methods, the TLS extensions handlers can be implemented using 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,
```

[1] such as the `gnutls_certificate_credentials_t` structures

```
                                        gnutls_buffer_st *extdata);
```

Here there are two functions, one for receiving the extension data and one for sending. These functions have to check internally whether they operate in client or server side.

A simple example of an extension handler can be seen in `ext/srp.c` in GnuTLS' source code. After implementing these functions, together with the extension number they handle, they have to be registered using `_gnutls_ext_register` in `gnutls_extensions.c` typically within `_gnutls_ext_init`.

Adding a new TLS extension

Adding support for a new TLS 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 sake of discussion, let's consider adding support for the hypothetical TLS extension `foobar`.

Add `configure` option like `--enable-foobar` or `--disable-foobar`.

This step is useful when the extension code is large and it might be desirable to disable the extension under some circumstances. Otherwise it can be safely skipped.

Whether to chose 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 IANA extension value to `extensions_t` in `gnutls_int.h`.

A good name for the value would be GNUTLS_EXTENSION_FOOBAR. Check with http://www.iana.org/assignments/tls-extensiontype-values for allocated values. For experiments, you could pick a number but remember that some consider it a bad idea to deploy such modified version since it will lead to interoperability problems in the future when the

IANA allocates that number to someone else, or when the foobar protocol is allocated another number.

Add an entry to _gnutls_extensions in gnutls_extensions.c.

A typical entry would be:

```
  int ret;

#if ENABLE_FOOBAR
  ret = _gnutls_ext_register (&foobar_ext);
  if (ret != GNUTLS_E_SUCCESS)
    return ret;
#endif
```

Most likely you'll need to add an `#include "ext/foobar.h"`, that will contain something like like:

```
  extension_entry_st foobar_ext = {
    .name = "FOOBAR",
    .type = GNUTLS_EXTENSION_FOOBAR,
    .parse_type = GNUTLS_EXT_TLS,
    .recv_func = _foobar_recv_params,
    .send_func = _foobar_send_params,
    .pack_func = _foobar_pack,
    .unpack_func = _foobar_unpack,
    .deinit_func = NULL
  }
```

The GNUTLS_EXTENSION_FOOBAR is the integer value you added to `gnutls_int.h` earlier. In this structure you specify the functions to read the extension from the hello message, the function to send the reply to, and two more functions to pack and unpack from stored session data (e.g. when resumming a session). The `deinit` function will be called to deinitialize the extension's private parameters, if any.

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 extension.

The functions you are responsible to add are those mentioned in the previous step. They should be added in a file such as `ext/foobar.c` and headers should be placed in `ext/foobar.h`. As a starter, you could add this:

```
int
_foobar_recv_params (gnutls_session_t session, const opaque * data,
                     size_t data_size)
{
  return 0;
}

```

```
int
_foobar_send_params (gnutls_session_t session, gnutls_buffer_st* data)
{
  return 0;
}

int
_foobar_pack (extension_priv_data_t epriv, gnutls_buffer_st * ps)
{
   /* Append the extension's internal state to buffer */
   return 0;
}

int
_foobar_unpack (gnutls_buffer_st * ps, extension_priv_data_t * epriv)
{
   /* Read the internal state from buffer */
   return 0;
}
```

The _foobar_recv_params function is responsible for parsing incoming extension data (both in the client and server).

The _foobar_send_params function is responsible for sending extension data (both in the client and server).

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_ext_set_session_data and _gnutls_ext_get_session_data. You can check simple examples at ext/max_record.c and ext/server_name.c extensions. That private information can be saved and restored across session resumption if the following functions are set:

The _foobar_pack function is responsible for packing internal extension data to save them in the session resumption storage.

The _foobar_unpack function is responsible for restoring session data from the session resumption storage.

Recall that both the client and server, send and receive parameters, and your code most likely will need to do different things depending on which mode it is 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 opaque * 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 files, you'll need to add them to `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 enable/disable the extension.

It might be desirable to allow users of the extension to request 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 LGPLv3+ applies. The implementation of the function should lie in the `ext/foobar.c` file.

To make the API available in the shared library you need to add the symbol in `lib/libgnutls.map`, so that the symbol is 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 [31]. 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, the that flags `session->security_parameters.do_send_supplemental` and `session->security_parameters.do_recv_supplemental` are set:

```
int
_gnutls_foobar_recv_params (gnutls_session_t session, const opaque * data,
                                 size_t _data_size)
{
   ...
   session->security_parameters.do_recv_supplemental=1;
   ...
```

```
}

int
_gnutls_foobar_send_params (gnutls_session_t session, gnutls_buffer_st *extdata)
{
   ...
   session->security_parameters.do_send_supplemental=1;
   ...
}
```

Furthermore add the functions _foobar_supp_recv_params and _foobar_supp_send_params to _foobar.h and _foobar.c. 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_st *buf)
{
   unsigned char *msg = "hello world";
   int len = strlen(msg);

   _gnutls_buffer_append_data_prefix(buf, 8, msg, len);

   return len;
}
```

Afterwards, add the new supplemental data handshake message to lib/gnutls_supplemental.c by adding a new entry to the _gnutls_supplemental[] structure:

```
gnutls_supplemental_entry _gnutls_supplemental[] =
{
  {"foobar",
   GNUTLS_SUPPLEMENTAL_FOOBAR_DATA,
   _foobar_supp_recv_params,
   _foobar_supp_send_params},
  {0, 0, 0, 0}
};
```

You have to include your foobar.h header file as well:

```
#include "foobar.h"
```

Lastly, add the new supplemental data type to `lib/includes/gnutls/gnutls.h`:

```
typedef enum
{
    GNUTLS_SUPPLEMENTAL_USER_MAPPING_DATA = 0,
    GNUTLS_SUPPLEMENTAL_FOOBAR_DATA = 1
} gnutls_supplemental_data_format_type_t;
```

Heartbeat extension.

One such extension is HeartBeat protocol (RFC6520: https://tools.ietf.org/html/rfc6520) implementation. To enable it use option –heartbeat with example client and server supplied with gnutls:

```
./doc/credentials/gnutls-http-serv --priority "NORMAL:-CIPHER-ALL:+NULL" -d 100 --heartbeat --echo
./src/gnutls-cli --priority "NORMAL:-CIPHER-ALL:+NULL" -d 100 localhost -p 5556 --insecure --heartbeat
```

After that pasting

```
**HEARTBEAT**
```

command into gnutls-cli will trigger corresponding command on the server and it will send HeartBeat Request with random length to client.

Another way is to run capabilities check with:

```
./doc/credentials/gnutls-http-serv -d 100 --heartbeat
./src/gnutls-cli-debug localhost -p 5556
```

10.5. 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. Unfortunately the cryptographic library that GnuTLS is based on takes no advantage of these capabilities. For this reason GnuTLS handles this internally by following a layered approach to accessing cryptographic operations as in Figure 10.4.

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.

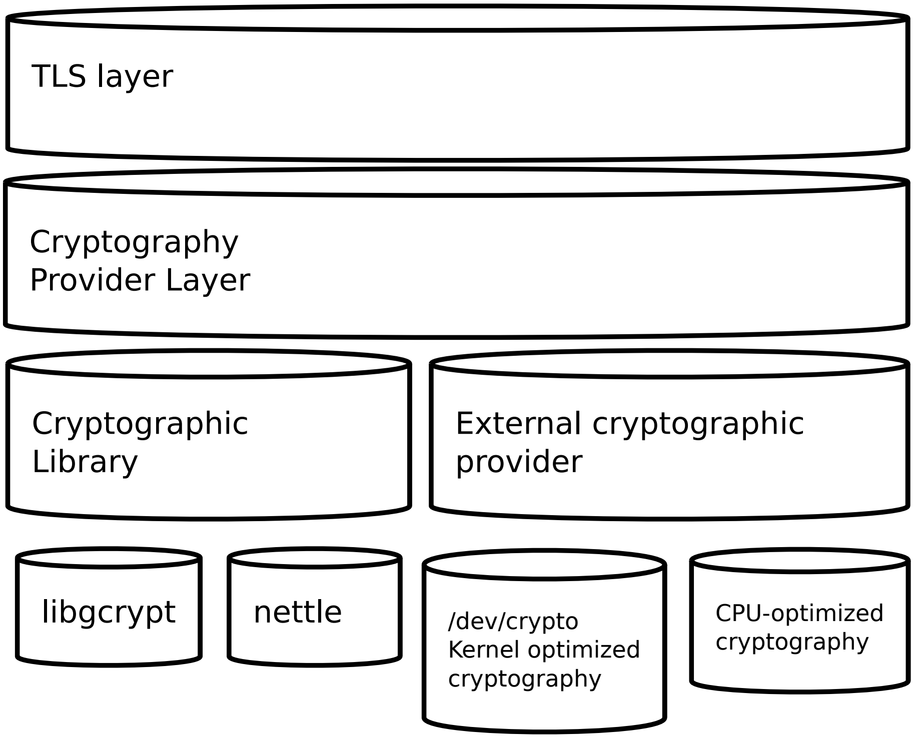

Figure 10.4.: GnuTLS cryptographic back-end design.

Cryptographic library layer

The Cryptographic library layer, currently supports only libnettle. Older versions of GnuTLS used to support libgcrypt, but it was switched with nettle mainly for performance reasons[2] and secondary because it is a simpler library to use. In the future other cryptographic libraries might be supported as well.

External cryptography provider

Systems that include a cryptographic co-processor, typically come with kernel drivers to utilize the operations from software. For this reason GnuTLS provides a layer where each individual algorithm used can be replaced by another implementation, i.e., the one provided by the driver. The FreeBSD, OpenBSD and Linux kernels[3] include already a number of hardware assisted implementations, and also provide an interface to access them, called `/dev/crypto`. GnuTLS will take advantage of this interface if compiled with special options. That is because in

[2] See http://lists.gnu.org/archive/html/gnutls-devel/2011-02/msg00079.html.
[3] Check http://home.gna.org/cryptodev-linux/ for the Linux kernel implementation of `/dev/crypto`.

most systems where hardware-assisted cryptographic operations are not available, using this interface might actually harm performance.

In systems that include cryptographic instructions with the CPU's instructions set, using the kernel interface will introduce an unneeded layer. For this reason GnuTLS includes such optimizations found in popular processors such as the AES-NI or VIA PADLOCK instruction sets. This is achieved using a mechanism that detects CPU capabilities and overrides parts of crypto back-end at runtime. The next section discusses the registration of a detected algorithm optimization. For more information please consult the GnuTLS source code in `lib/accelerated/`.

Overriding specific algorithms

When an optimized implementation of a single algorithm is available, say a hardware assisted version of AES-CBC then the following (internal) functions, from `crypto-backend.h`, can be used to register those algorithms.

- `gnutls_crypto_single_cipher_register`: To register a cipher algorithm.
- `gnutls_crypto_single_digest_register`: To register a hash (digest) or MAC algorithm.

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

Overriding the cryptographic library

In some systems, that might contain a broad acceleration engine, it might be desirable to override big parts of the cryptographic back-end, or even all of them. The following functions are provided for this reason.

- `gnutls_crypto_cipher_register`: To override the cryptographic algorithms back-end.
- `gnutls_crypto_digest_register`: To override the digest algorithms back-end.
- `gnutls_crypto_rnd_register`: To override the random number generator back-end.
- `gnutls_crypto_bigint_register`: To override the big number number operations back-end.
- `gnutls_crypto_pk_register`: To override the public key encryption back-end. This is tied to the big number operations so either none or both of them should be overriden.

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 7.5.1.

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 7.5.1
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 7.11.4
gnutls_transport_set_global_errno	Replaced by using the system's errno fascility or gnutls_transport_set_errno.
gnutls_x509_privkey_verify_data	Replaced by gnutls_pubkey_verify_data.
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.
@funcintrefgnutls_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.

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 help-gnutls@gnu.org. Archives of the mailing list discussions, and an interface to manage subscriptions, is available through the World Wide Web at http://lists.gnu.org/mailman/listinfo/help-gnutls.

A mailing list for developers are also available, see http://www.gnu.org/software/gnutls/lists.html. Bug reports should be sent to bug-gnutls@gnu.org, see section B.3.

B.2. Commercial Support

Commercial support is available for users of GnuTLS. The kind of support that can be purchased may include:

- Implement new features. Such as a new TLS extension.
- Port GnuTLS to new platforms. This could include porting to an embedded platforms that may need memory or size optimization.
- Integrating TLS as a security environment in your existing project.
- System design of components related to TLS.

If you are interested, please write to:

```
Simon Josefsson Datakonsult
Hagagatan 24
113 47 Stockholm
Sweden

E-mail: simon@josefsson.org
```

If your company provides support related to GnuTLS and would like to be mentioned here, contact the authors.

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:

bug-gnutls@gnu.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 GNU Standards document.

  If you normally code using another coding standard, there is no problem, but you should use indent to reformat the code before submitting your work.

- Use the unified diff format diff -u.

- Return errors. No reason whatsoever should abort the execution of the library. Even memory allocation errors, e.g. when malloc return NULL, should work although result in an error code.
- Design with thread safety in mind. Don't use global variables. Don't even write to per-handle global variables unless the documented behaviour of the function you write is to write to the per-handle global variable.
- Avoid using the C math library. It causes problems for embedded implementations, and in most situations it is very easy to avoid using it.
- Document your functions. Use comments before each function headers, that, if properly formatted, are extracted into Texinfo manuals and GTK-DOC web pages.
- Supply a ChangeLog and NEWS entries, where appropriate.

C. Supported Ciphersuites

Ciphersuite name	TLS ID	Since
TLS_DH_ANON_ARCFOUR_MD5	0x00 0x18	SSL3.0
TLS_DH_ANON_3DES_EDE_CBC_SHA1	0x00 0x1B	SSL3.0
TLS_DH_ANON_AES_128_CBC_SHA1	0x00 0x34	SSL3.0
TLS_DH_ANON_AES_256_CBC_SHA1	0x00 0x3A	SSL3.0
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_PSK_SHA_ARCFOUR_SHA1	0x00 0x8A	TLS1.0
TLS_PSK_SHA_3DES_EDE_CBC_SHA1	0x00 0x8B	TLS1.0
TLS_PSK_SHA_AES_128_CBC_SHA1	0x00 0x8C	TLS1.0
TLS_PSK_SHA_AES_256_CBC_SHA1	0x00 0x8D	TLS1.0
TLS_PSK_AES_128_CBC_SHA256	0x00 0xAE	TLS1.0
TLS_PSK_AES_128_GCM_SHA256	0x00 0xA8	TLS1.2
TLS_PSK_NULL_SHA256	0x00 0xB0	TLS1.0
TLS_DHE_PSK_SHA_ARCFOUR_SHA1	0x00 0x8E	TLS1.0
TLS_DHE_PSK_SHA_3DES_EDE_CBC_SHA1	0x00 0x8F	TLS1.0
TLS_DHE_PSK_SHA_AES_128_CBC_SHA1	0x00 0x90	TLS1.0
TLS_DHE_PSK_SHA_AES_256_CBC_SHA1	0x00 0x91	TLS1.0
TLS_DHE_PSK_AES_128_CBC_SHA256	0x00 0xB2	TLS1.0
TLS_DHE_PSK_AES_128_GCM_SHA256	0x00 0xAA	TLS1.2
TLS_DHE_PSK_NULL_SHA256	0x00 0xB4	TLS1.0
TLS_SRP_SHA_3DES_EDE_CBC_SHA1	0xC0 0x1A	TLS1.0
TLS_SRP_SHA_AES_128_CBC_SHA1	0xC0 0x1D	TLS1.0
TLS_SRP_SHA_AES_256_CBC_SHA1	0xC0 0x20	TLS1.0
TLS_SRP_SHA_DSS_3DES_EDE_CBC_SHA1	0xC0 0x1C	TLS1.0
TLS_SRP_SHA_RSA_3DES_EDE_CBC_SHA1	0xC0 0x1B	TLS1.0
TLS_SRP_SHA_DSS_AES_128_CBC_SHA1	0xC0 0x1F	TLS1.0
TLS_SRP_SHA_RSA_AES_128_CBC_SHA1	0xC0 0x1E	TLS1.0
TLS_SRP_SHA_DSS_AES_256_CBC_SHA1	0xC0 0x22	TLS1.0
TLS_SRP_SHA_RSA_AES_256_CBC_SHA1	0xC0 0x21	TLS1.0
TLS_DHE_DSS_ARCFOUR_SHA1	0x00 0x66	TLS1.0
TLS_DHE_DSS_3DES_EDE_CBC_SHA1	0x00 0x13	SSL3.0

TLS_DHE_DSS_AES_128_CBC_SHA1	0x00 0x32	SSL3.0
TLS_DHE_DSS_AES_256_CBC_SHA1	0x00 0x38	SSL3.0
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_RSA_3DES_EDE_CBC_SHA1	0x00 0x16	SSL3.0
TLS_DHE_RSA_AES_128_CBC_SHA1	0x00 0x33	SSL3.0
TLS_DHE_RSA_AES_256_CBC_SHA1	0x00 0x39	SSL3.0
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_RSA_NULL_MD5	0x00 0x01	SSL3.0
TLS_RSA_NULL_SHA1	0x00 0x02	SSL3.0
TLS_RSA_NULL_SHA256	0x00 0x3B	TLS1.2
TLS_RSA_EXPORT_ARCFOUR_40_MD5	0x00 0x03	SSL3.0
TLS_RSA_ARCFOUR_SHA1	0x00 0x05	SSL3.0
TLS_RSA_ARCFOUR_MD5	0x00 0x04	SSL3.0
TLS_RSA_3DES_EDE_CBC_SHA1	0x00 0x0A	SSL3.0
TLS_RSA_AES_128_CBC_SHA1	0x00 0x2F	SSL3.0
TLS_RSA_AES_256_CBC_SHA1	0x00 0x35	SSL3.0
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_DHE_RSA_AES_128_GCM_SHA256	0x00 0x9E	TLS1.2
TLS_DHE_DSS_AES_128_GCM_SHA256	0x00 0xA2	TLS1.2
TLS_DH_ANON_AES_128_GCM_SHA256	0x00 0xA6	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_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_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_AES_128_CBC_SHA256	0xC0 0x23	TLS1.2
TLS_ECDHE_RSA_AES_128_CBC_SHA256	0xC0 0x27	TLS1.2
TLS_ECDHE_ECDSA_AES_128_GCM_SHA256	0xC0 0x2B	TLS1.2
TLS_ECDHE_RSA_AES_128_GCM_SHA256	0xC0 0x2F	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	TLS1.0
TLS_ECDHE_PSK_AES_256_CBC_SHA384	0xC0 0x38	TLS1.0
TLS_ECDHE_PSK_NULL_SHA256	0xC0 0x3A	TLS1.0
TLS_ECDHE_PSK_NULL_SHA384	0xC0 0x3B	TLS1.0
TLS_ECDHE_ECDSA_AES_256_GCM_SHA384	0xC0 0x2C	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_PSK_WITH_AES_256_GCM_SHA384	0x00 0xA9	TLS1.2
TLS_DHE_PSK_WITH_AES_256_GCM_SHA384	0x00 0xAB	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` indicate 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 record packet with illegal version was received.
-9	GNUTLS_E_UNEXPECTED_PACKET_-LENGTH	A TLS packet with unexpected length was received.
-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 requested session has expired.
-30	GNUTLS_E_DB_ERROR	Error in Database backend.
-31	GNUTLS_E_SRP_PWD_ERROR	Error in password 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.
-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 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.
-102	GNUTLS_E_WARNING_IA_IPHF_RECEIVED	Received a TLS/IA Intermediate Phase Finished message
-103	GNUTLS_E_WARNING_IA_FPHF_RECEIVED	Received a TLS/IA Final Phase Finished message
-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 SRP username supplied is unknown.
-110	GNUTLS_E_PREMATURE_TERMINATION	The TLS connection was non-properly terminated.
-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.
-292	GNUTLS_E_HEARTBEAT_PONG_-RECEIVED	A heartbeat pong message was received.
-293	GNUTLS_E_HEARTBEAT_PING_RECEIVED	A heartbeat ping message was received.
-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.
-340	GNUTLS_E_NO_CERTIFICATE_STATUS	There is no certificate status (OCSP).
-341	GNUTLS_E_OCSP_RESPONSE_ERROR	The OCSP response is invalid

Table D.1.: The error codes table

GNU Free Documentation License

Version 1.3, 3 November 2008

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.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

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

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Bibliography

[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 http://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 http://www.ietf.org/rfc/rfc3546.

[7] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP message format, November 2007. Available from http://www.ietf.org/rfc/rfc4880.

[8] T. Dierks and E. Rescorla. The TLS protocol version 1.1, April 2006. Available from http://www.ietf.org/rfc/rfc4346.

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

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

[11] C. Evans and C. Palmer. Public Key Pinning Extension for HTTP, December 2011. Available from http://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 http://www.ietf.org/rfc/rfc6101.

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

[14] S. Hollenbeck. Transport layer security protocol compression methods, May 2004. Available from http://www.ietf.org/rfc/rfc3749.

[15] 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 http://www.ietf.org/rfc/rfc3280.

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

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

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

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

[20] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP), May 2010. Available from http://www.ietf.org/rfc/rfc5764.

[21] B. Moeller. Security of CBC ciphersuites in SSL/TLS: Problems and countermeasures, 2002. Available from http://www.openssl.org/~bodo/tls-cbc.txt.

[22] 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 http://www.ietf.org/rfc/rfc2560.

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

[24] J. Pechanec and D. J. Moffat. The PKCS 11 URI Scheme, August 2011. Work in progress, available from http://tools.ietf.org/html/draft-pechanec-pkcs11uri-05.

[25] M. T. R. Seggelmann and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, February 2012. Available from http://www.ietf.org/rfc/rfc6520.

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

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

[28] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport layer security (TLS) renegotiation indication extension, February 2010. Available from http://www.ietf.org/rfc/rfc5746.

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

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

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

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

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

[34] 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 http://www.ietf.org/rfc/rfc3820.

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

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

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

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

Index