1 SSL User's Guide

The Secure Socket Layer (SSL) application provides secure communication over sockets.

1.1 Introduction

1.1.1 Purpose

Transport Layer Security (TLS) and its predecessor, the Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communications security over a computer network. The protocols use X.509 certificates and hence public key (asymmetric) cryptography to authenticate the counterpart with whom they communicate, and to exchange a symmetric key for payload encryption. The protocol provides data/message confidentiality (encryption), integrity (through message authentication code checks) and host verification (through certificate path validation). DTLS (Datagram Transport Layer Security) that is based on TLS but datagram oriented instead of stream oriented.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language, the concepts of OTP, and has a basic understanding of SSL/TLS/DTLS.

1.2 TLS/DTLS and TLS Predecessor, SSL

The Erlang SSL application implements the SSL/TLS/DTLS protocol for the currently supported versions, see the ssl(3) manual page.

By default SSL/TLS is run over the TCP/IP protocol even though you can plug in any other reliable transport protocol with the same Application Programming Interface (API) as the gen_tcp module in Kernel. DTLS is by default run over UDP/IP, which means that application data has no delivery guarantees. Other transports, such as SCTP, may be supported in future releases.

If a client and a server wants to use an upgrade mechanism, such as defined by RFC 2817, to upgrade a regular TCP/IP connection to an TLS connection, this is supported by the Erlang SSL application API. This can be useful for, for example, supporting HTTP and HTTPS on the same port and implementing virtual hosting. Note this is a TLS feature only.

1.2.1 Security Overview

To achieve authentication and privacy, the client and server perform a TLS/DTLS handshake procedure before transmitting or receiving any data. During the handshake, they agree on a protocol version and cryptographic algorithms, generate shared secrets using public key cryptographies, and optionally authenticate each other with digital certificates.

1.2.2 Data Privacy and Integrity

A symmetric key algorithm has one key only. The key is used for both encryption and decryption. These algorithms are fast, compared to public key algorithms (using two keys, one public and one private) and are therefore typically used for encrypting bulk data.

The keys for the symmetric encryption are generated uniquely for each connection and are based on a secret negotiated in the TLS/DTLS handshake.
The TLS/DTLS handshake protocol and data transfer is run on top of the TLS/DTLS Record Protocol, which uses a keyed-hash Message Authentication Code (MAC), or a Hash-based MAC (HMAC), to protect the message data integrity. From the TLS RFC: "A Message Authentication Code is a one-way hash computed from a message and some secret data. It is difficult to forge without knowing the secret data. Its purpose is to detect if the message has been altered."

1.2.3 Digital Certificates

A certificate is similar to a driver's license, or a passport. The holder of the certificate is called the **subject**. The certificate is signed with the private key of the issuer of the certificate. A chain of trust is built by having the issuer in its turn being certified by another certificate, and so on, until you reach the so called root certificate, which is self-signed, that is, issued by itself.

Certificates are issued by Certification Authorities (CAs) only. A handful of top CAs in the world issue root certificates. You can examine several of these certificates by clicking through the menus of your web browser.

1.2.4 Peer Authentication

Authentication of the peer is done by public key path validation as defined in RFC 3280. This means basically the following:

- Each certificate in the certificate chain is issued by the previous one.
- The certificates attributes are valid.
- The root certificate is a trusted certificate that is present in the trusted certificate database kept by the peer.

The server always sends a certificate chain as part of the TLS handshake, but the client only sends one if requested by the server. If the client does not have an appropriate certificate, it can send an "empty" certificate to the server.

The client can choose to accept some path evaluation errors, for example, a web browser can ask the user whether to accept an unknown CA root certificate. The server, if it requests a certificate, does however not accept any path validation errors. It is configurable if the server is to accept or reject an "empty" certificate as response to a certificate request.

1.2.5 TLS Sessions

From the TLS RFC: "A TLS session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection."

Session data is by default kept by the SSL application in a memory storage, hence session data is lost at application restart or takeover. Users can define their own callback module to handle session data storage if persistent data storage is required. Session data is also invalidated after 24 hours from it was saved, for security reasons. The amount of time the session data is to be saved can be configured.

By default the TLS/DTLS clients try to reuse an available session and by default the TLS/DTLS servers agree to reuse sessions when clients ask for it.

1.3 Using SSL application API

To see relevant version information for ssl, call `ssl:versions/0`.

To see all supported cipher suites, call `ssl:cipher_suites(all)` . The available cipher suites for a connection depend on your certificate. Specific cipher suites that you want your connection to use can also be specified. Default is to use the strongest available.
1.3.1 Setting up Connections

This section shows a small example of how to set up client/server connections using the Erlang shell. The returned value of the `sslsocket` is abbreviated with `[...]` as it can be fairly large and is opaque.

Minimal Example

**Note:**
The minimal setup is not the most secure setup of SSL/TLS.

To set up client/server connections:

**Step 1:** Start the server side:

```
1 server> ssl:start().
ok
```

**Step 2:** Create an TLS listen socket: (To run DTLS add the option `{protocol, dtls}`)

```
2 server> {ok, ListenSocket} = 
ssl:listen(9999, [{certfile, "cert.pem"}, {keyfile, "key.pem"}, {reuseaddr, true}]).
{ok, sslsocket, [\ldots]}  
```

**Step 3:** Do a transport accept on the TLS listen socket:

```
3 server> {ok, TLSTransportSocket} = ssl:transport_accept(ListenSocket).
{ok, sslsocket, [\ldots]}
```

**Step 4:** Start the client side:

```
1 client> ssl:start().
ok
```

To run DTLS add the option `{protocol, dtls}` to third argument.

```
2 client> {ok, Socket} = ssl:connect("localhost", 9999, [], infinity).
{ok, sslsocket, [\ldots]}
```

**Step 5:** Do the TLS handshake:

```
4 server> {ok, Socket} = ssl:handshake(TLSTransportSocket).
ok
```

**Step 6:** Send a message over TLS:

```
5 server> ssl:send(Socket, "foo").
ok
```

**Step 7:** Flush the shell message queue to see that the message was sent on the server side:

```
3 client> flush().
Shell got {ssl,sslsocket,[\ldots]},"foo"
ok
```
Upgrade Example - TLS only

**Note:**
To upgrade a TCP/IP connection to an SSL connection, the client and server must agree to do so. The agreement can be accomplished by using a protocol, for example, the one used by HTTP specified in RFC 2817.

To upgrade to an SSL connection:

**Step 1:** Start the server side:

1 server> ssl:start().
ok

**Step 2:** Create a normal TCP listen socket:

2 server> {ok, ListenSocket} = gen_tcp:listen(9999, [{reuseaddr, true}]).
{ok, #Port<0.475>}

**Step 3:** Accept client connection:

3 server> {ok, Socket} = gen_tcp:accept(ListenSocket).
{ok, #Port<0.476>}

**Step 4:** Start the client side:

1 client> ssl:start().
ok

2 client> {ok, Socket} = gen_tcp:connect("localhost", 9999, [], infinity).

**Step 5:** Ensure `active` is set to `false` before trying to upgrade a connection to an SSL connection, otherwise SSL handshake messages can be delivered to the wrong process:

4 server> inet:setopts(Socket, [{active, false}]).
ok

**Step 6:** Do the TLS handshake:

5 server> {ok, TLSSocket} = ssl:handshake(Socket, [{cacertfile, "cacerts.pem"}, {certfile, "cert.pem"}, {keyfile, "key.pem"}]).
{ok, {sslsocket,...}}

**Step 7:** Upgrade to an TLS connection. The client and server must agree upon the upgrade. The server must call `ssl:handshake/2` before the client calls `ssl:connect/3`.

3 client> {ok, TLSSocket} = ssl:connect(Socket, [{cacertfile, "cacerts.pem"}, {certfile, "cert.pem"}, {keyfile, "key.pem"}], infinity).
{ok, {sslsocket,...}}

**Step 8:** Send a message over TLS:

4 client> ssl:send(TLSSocket, "foo").
ok

**Step 9:** Set `active` true on the TLS socket:

4 server> ssl:setopts(TLSSocket, [{active, true}]).
ok
1.3 Using SSL application API

**Step 10:** Flush the shell message queue to see that the message was sent on the client side:

```erlang
5 server> flush().
Shell got {ssl,{sslsocket,[[...]],"foo"},ok
```

### 1.3.2 Customizing cipher suites

Fetch default cipher suite list for an TLS/DTLS version. Change default to all to get all possible cipher suites.

```erlang
1> Default = \ssl\c\ipher\ suites(default, 'tlsv1.2').
[#{cipher => aes_256_gcm, key_exchange => ecdhe_ecdsa,
   mac => aead, prf => sha384}, ....]
```

In OTP 20 it is desirable to remove all cipher suites that uses rsa kexchange (removed from default in 21)

```erlang
2> NoRSA = \ssl\:filter_cipher_suites(Default,
    [{key_exchange, fun(rsa) -> false;
        (_) -> true end}].
    [...]
```

Pick just a few suites

```erlang
3> Suites = \ssl\:filter_cipher_suites(Default,
    [{key_exchange, fun(ecdh_ecdsa) -> true;
        (_) -> false end},
    {cipher, fun(aes_128_cbc) -> true;
        (_) -> false end}].
    [#{cipher => aes_128_cbc, key_exchange => ecdh_ecdsa,
       mac => sha256, prf => sha256},
     #{cipher => aes_256_cbc, key_exchange => ecdhe_ecdsa,
       mac => sha384, prf => sha384}, ...
```

Make some particular suites the most preferred, or least preferred by changing prepend to append.

```erlang
4> \ssl\:prepend_cipher_suites(Suites, Default).
[#{cipher => aes_128_cbc, key_exchange => ecdh_ecdsa,
   mac => sha256, prf => sha256},
 #\{cipher => aes_128_cbc, key_exchange => ecdhe_ecdsa,
   mac => sha384, prf => sha384}, ...
```

### 1.3.3 Using an Engine Stored Key

Erlang ssl application is able to use private keys provided by OpenSSL engines using the following mechanism:

```erlang
1> \ssl\:start().
ok
```

Load a crypto engine, should be done once per engine used. For example dynamically load the engine called MyEngine:

```erlang
2> {ok, EngineRef} = \crypto\:engine_load("dynamic",
    [{<<"SO_PATH">>, "/tmp/user/engines/MyEngine"},<<"LOAD">>},[]).
     {ok,#Ref<0.2399045421.3028942852.173962>}
```

Create a map with the engine information and the algorithm used by the engine:
3> PrivKey =
   #{algorithm => rsa,
    engine => EngineRef,
    key_id => "id of the private key in Engine"}.

Use the map in the ssl key option:

4> {ok, SSLSocket} =
   ssl:connect("localhost", 9999,
              [{cacertfile, "cacerts.pem"},
               {certfile, "cert.pem"},
               {key, PrivKey}], infinity).

See also  crypto documentation

1.4 Using TLS for Erlang Distribution

This section describes how the Erlang distribution can use TLS to get extra verification and security.

The Erlang distribution can in theory use almost any connection-based protocol as bearer. However, a module that implements the protocol-specific parts of the connection setup is needed. The default distribution module is inet_tcp_dist in the Kernel application. When starting an Erlang node distributed, net_kernel uses this module to set up listen ports and connections.

In the SSL application, an extra distribution module, inet_tls_dist, can be used as an alternative. All distribution connections will use TLS and all participating Erlang nodes in a distributed system must use this distribution module.

The security level depends on the parameters provided to the TLS connection setup. Erlang node cookies are however always used, as they can be used to differentiate between two different Erlang networks.

To set up Erlang distribution over TLS:

• **Step 1:** Build boot scripts including the SSL application.
• **Step 2:** Specify the distribution module for net_kernel.
• **Step 3:** Specify the security options and other SSL options.
• **Step 4:** Set up the environment to always use TLS.

The following sections describe these steps.

1.4.1 Building Boot Scripts Including the SSL Application

Boot scripts are built using the systools utility in the SASL application. For more information on systools, see the SASL documentation. This is only an example of what can be done.

The simplest boot script possible includes only the Kernel and STDLIB applications. Such a script is located in the bin directory of the Erlang distribution. The source for the script is found under the Erlang installation top directory under releases/\(<OTP \ version>/\start_clean.rel.

Do the following:

• Copy that script to another location (and preferably another name).
• Add the applications Crypto, Public Key, and SSL with their current version numbers after the STDLIB application.

The following shows an example .rel file with TLS added:
1.4 Using TLS for Erlang Distribution

The version numbers differ in your system. Whenever one of the applications included in the script is upgraded, change the script.

Do the following:

• Build the boot script.

  Assuming the .rel file is stored in a file start_ssl.rel in the current directory, a boot script can be built as follows:

  ```erl
  1> systools:make_script("start_ssl",[]).
  There is now a start_ssl.boot file in the current directory.
  ```

  Do the following:

  • Test the boot script. To do this, start Erlang with the -boot command-line parameter specifying this boot script (with its full path, but without the .boot suffix). In UNIX it can look as follows:

  ```bash
  $ erl -boot /home/me/ssl/start_ssl
  Erlang (BEAM) emulator version 5.0
  Eshell V5.0  (abort with ^G)
  1> whereis(ssl_manager).
  <0.41.0>
  The whereis function-call verifies that the SSL application is started.
  ```

  As an alternative to building a bootscript, you can explicitly add the path to the SSLebin directory on the command line. This is done with command-line option -pa. This works as the SSL application does not need to be started for the distribution to come up, as a clone of the SSL application is hooked into the Kernel application. So, as long as the SSL application code can be reached, the distribution starts. The -pa method is only recommended for testing purposes.

  Note:

  The clone of the SSL application must enable the use of the SSL code in such an early bootstage as needed to set up the distribution. However, this makes it impossible to soft upgrade the SSL application.

1.4.2 Specifying Distribution Module for net_kernel

The distribution module for SSL/TLS is named inet_tls_dist and is specified on the command line with option -proto_dist. The argument to -proto_dist is to be the module name without suffix _dist. So, this distribution module is specified with -proto_dist inet_tls on the command line.

Extending the command line gives the following:

```bash
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_tls
```

For the distribution to be started, give the emulator a name as well:
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_tls -sname ssl_test
Erlang (BEAM) emulator version 5.0 [source]
Eshell V5.0 (abort with ^G)
(ssl_test@myhost)1>

However, a node started in this way refuses to talk to other nodes, as no TLS parameters are supplied (see the next section).

1.4.3 Specifying SSL/TLS Options

The SSL/TLS distribution options can be written into a file that is consulted when the node is started. This file name is then specified with the command line argument -ssl_dist_optfile.

Any available SSL/TLS option can be specified in an options file, but note that options that take a fun() has to use the syntax fun Mod:Func/Arity since a function body cannot be compiled when consulting a file.

Do not tamper with the socket options list, binary, active, packet, nodelay and deliver since they are used by the distribution protocol handler itself. Other raw socket options such as packet_size may interfere severely, so beware!

For SSL/TLS to work, at least a public key and a certificate must be specified for the server side. In the following example, the PEM file "/home/me/ssl/erlserver.pem" contains both the server certificate and its private key.

Create a file named for example "/home/me/ssl/ssl_test@myhost.conf":

```erl
[{server,
  [{certfile, "/home/me/ssl/erlserver.pem"},
   {secure_renegotiate, true}]}],

{client,
  [{secure_renegotiate, true}]}].
```

And then start the node like this (line breaks in the command are for readability, and shall not be there when typed):

$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_tls
   -ssl_dist_optfile "/home/me/ssl/ssl_test@myhost.conf"
   -sname ssl_test

The options in the {server, Opts} tuple are used when calling ssl:ssl_accept/3, and the options in the {client, Opts} tuple are used when calling ssl:connect/4.

For the client, the option {server_name_indication, atom_to_list(TargetNode)} is added when connecting. This makes it possible to use the client option {verify, verify_peer}, and the client will verify that the certificate matches the node name you are connecting to. This only works if the the server certificate is issued to the name atom_to_list(TargetNode).

For the server it is also possible to use the option {verify, verify_peer} and the server will only accept client connections with certificates that are trusted by a root certificate that the server knows. A client that presents an untrusted certificate will be rejected. This option is preferably combined with {fail_if_no_peer_cert, true} or a client will still be accepted if it does not present any certificate.

A node started in this way is fully functional, using TLS as the distribution protocol.

1.4.4 Specifying SSL/TLS Options (Legacy)

As in the previous section the PEM file "/home/me/ssl/erlserver.pem" contains both the server certificate and its private key.

On the erl command line you can specify options that the SSL/TLS distribution adds when creating a socket.
1.4 Using TLS for Erlang Distribution

The simplest SSL/TLS options in the following list can be specified by adding the prefix `server_` or `client_` to the option name:

- `certfile`
- `keyfile`
- `password`
- `cacertfile`
- `verify`
- `verify_fun` (write as `(Module, Function, InitialUserState)`)  
- `crl_check`
- `crl_cache` (write as Erlang term)
- `reuse_sessions`
- `secure_renegotiate`
- `depth`
- `hibernate_after`
- `ciphers` (use old string format)

Note that `verify_fun` needs to be written in a different form than the corresponding SSL/TLS option, since funs are not accepted on the command line.

The server can also take the options `dhfile` and `fail_if_no_peer_cert` (also prefixed). `client_`-prefixed options are used when the distribution initiates a connection to another node. `server_`-prefixed options are used when accepting a connection from a remote node.

Raw socket options, such as `packet` and `size` must not be specified on the command line.

The command-line argument for specifying the SSL/TLS options is named `-ssl_dist_opt` and is to be followed by pairs of SSL options and their values. Argument `-ssl_dist_opt` can be repeated any number of times.

An example command line doing the same as the example in the previous section can now look as follows (line breaks in the command are for readability, and shall not be there when typed):

```
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_tls  
   -ssl_dist_opt server_certfile "/home/me/ssl/erlserver.pem"  
   -ssl_dist_opt server_secure_renegotiate true client_secure_renegotiate true  
   -sname ssl_test
```

Erlang (BEAM) emulator version 5.0 [source]

Eshell V5.0 (abort with ^G)

(ssl_test@myhost)1>

1.4.5 Setting up Environment to Always Use SSL/TLS (Legacy)

A convenient way to specify arguments to Erlang is to use environment variable `ERL_FLAGS`. All the flags needed to use the SSL/TLS distribution can be specified in that variable and are then interpreted as command-line arguments for all subsequent invocations of Erlang.

In a Unix (Bourne) shell, it can look as follows (line breaks are for readability, they are not to be there when typed):
The `init:get_arguments()` call verifies that the correct arguments are supplied to the emulator.

### 1.4.6 Using SSL/TLS distribution over IPv6

It is possible to use SSL/TLS distribution over IPv6 instead of IPv4. To do this, pass the option `-proto_dist inet6_tls` instead of `-proto_dist inet_tls` when starting Erlang, either on the command line or in the `ERL_FLAGS` environment variable.

An example command line with this option would look like this:

```bash
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet6_tls
   -ssl_dist_opt file "/home/me/ssl/ssl_test@myhost.conf"
   -sname ssl_test
```

A node started in this way will only be able to communicate with other nodes using SSL/TLS distribution over IPv6.

### 1.5 Standards Compliance

#### 1.5.1 Purpose

This section describes the current state of standards compliance of the ssl application.

#### 1.5.2 Common (pre TLS 1.3)

- For security reasons RSA key exchange cipher suites are no longer supported by default, but can be configured. (OTP 21)
- For security reasons DES cipher suites are no longer supported by default, but can be configured. (OTP 20)
- For security reasons 3DES cipher suites are no longer supported by default, but can be configured. (OTP 21)
- Renegotiation Indication Extension RFC 5746 is supported
- Ephemeral Diffie-Hellman cipher suites are supported, but not Diffie Hellman Certificates cipher suites.
- Elliptic Curve cipher suites are supported if the Crypto application supports it and named curves are used.
- Export cipher suites are not supported as the U.S. lifted its export restrictions in early 2000.
- IDEA cipher suites are not supported as they have become deprecated by the TLS 1.2 specification so it is not motivated to implement them.
- Compression is not supported.
1.5 Standards Compliance

1.5.3 Common

- CRL validation is supported.
- Policy certificate extensions are not supported.
- ‘Server Name Indication’ extension (RFC 6066) is supported.
- Application Layer Protocol Negotiation (ALPN) and its successor Next Protocol Negotiation (NPN) are supported.
- It is possible to use Pre-Shared Key (PSK) and Secure Remote Password (SRP) cipher suites, but they are not enabled by default.

1.5.4 SSL 2.0
For security reasons SSL-2.0 is not supported. Interoperability with SSL-2.0 enabled clients dropped. (OTP 21)

1.5.5 SSL 3.0
For security reasons SSL-3.0 is no longer supported by default, but can be configured. (OTP 19)

1.5.6 TLS 1.0
For security reasons TLS-1.0 is no longer supported by default, but can be configured. (OTP 22)

1.5.7 TLS 1.1
For security reasons TLS-1.1 is no longer supported by default, but can be configured. (OTP 22)

1.5.8 TLS 1.2
Supported

1.5.9 DTLS 1.0
For security reasons DTLS-1.0 (based on TLS 1.1) is no longer supported by default, but can be configured. (OTP 22)

1.5.10 DTLS 1.2
Supported (based on TLS 1.2)

1.5.11 DTLS 1.3
Not yet supported

1.5.12 TLS 1.3
OTP-22 introduces basic support for TLS 1.3. Basic functionality covers a simple TLS 1.3 handshake with support of the mandatory extensions (supported_groups, signature_algorithms, key_share, supported_versions and signature_algorithms_cert). The current implementation supports a selective set of cryptographic algorithms:

- Key Exchange: ECDHE
- Groups: all standard groups supported for the Diffie-Hellman key exchange
- Ciphers: TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256 and TLS_AES_128_CCM_SHA256
- Signature Algorithms: rsa_pkcs1_sha256, rsa_pkcs1_sha384, rsa_pkcs1_sha512, ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, ecdsa_secp521r1_sha512, rsa_pss_rsa_sha256, rsa_pss_rsa_sha384, rsa_pss_rsa_sha512, rsa_pkcs1_sha1 and ecdsa_sha1
• Certificates: RSA (it MUST use the rsaEncryption OID) and ECDSA keys

Other notable features:
• PSK and session resumption not supported
• Early data and 0-RTT not supported
• Key and Initialization Vector Update not supported

For more detailed information see the Standards Compliance below.

The following table describes the current state of standards compliance for TLS 1.3.
(C = Compliant, NC = Non-Compliant, PC = Partially-Compliant, NA = Not Applicable)

<table>
<thead>
<tr>
<th>Section</th>
<th>Feature</th>
<th>State</th>
<th>Since</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3. Updates Affecting TLS 1.2</td>
<td></td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>Version downgrade protection mechanism</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>RSASSA-PSS signature schemes</td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>supported_versions (ClientHello) extension</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>signature_algorithms_cert extension</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>2. Protocol Overview</td>
<td></td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>(EC)DHE</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>PSK-only</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PSK with (EC)DHE</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>2.1. Incorrect DHE share</td>
<td>HelloRetryRequest</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>2.2. Resumption and Pre-Shared Key (PSK)</td>
<td></td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>2.3. 0-RTT Data</td>
<td></td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>4.1.1. Cryptographic Negotiation</td>
<td></td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>supported_groups extension</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>signature_algorithms extension</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
## 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Pre-Shared Key Extension</th>
<th>Client Hello</th>
<th>Client</th>
<th>PC</th>
<th>22.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_shared_key extension</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.1.2. Client Hello

- **server_name (RFC6066)**: NC
- **max_fragment_length (RFC6066)**: NC
- **status_request (RFC6066)**: NC
- **supported_groups (RFC7919)**: C | 22.1 |
- **signature_algorithms (RFC8446)**: C | 22.1 |
- **use_srtp (RFC5764)**: NC
- **heartbeat (RFC6520)**: NC
- **application_layer_protocol_negotiation (RFC7301)**: C | 22.1 |
- **signed_certificate_timestamp (RFC6962)**: NC
- **client_certificate_type (RFC7250)**: NC
- **server_certificate_type (RFC7250)**: NC
- **padding (RFC7685)**: NC
- **key_share (RFC8446)**: C | 22.1 |
- **pre_shared_key (RFC8446)**: NC
- **psk_key_exchange_modes (RFC8446)**: NC
- **early_data (RFC8446)**: NC
- **cookie (RFC8446)**: NC
- **supported_versions (RFC8446)**: C | 22.1 |
- **certificateAuthorities (RFC8446)**: NC
<table>
<thead>
<tr>
<th>Feature</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid_filters (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>post_handshake_auth (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>signature_algorithms_cert (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td>server_name (RFC6066)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>max_fragment_length (RFC6066)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>status_request (RFC6066)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>supported_groups (RFC7919)</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>signature_algorithms (RFC8446)</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>use_srtp (RFC5764)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>heartbeat (RFC6520)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>application_layer_protocol_negotiation</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>signed_certificate_timestamp (RFC6962)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>client_certificate_type (RFC7250)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>server_certificate_type (RFC7250)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>padding (RFC7685)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>key_share (RFC8446)</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>pre_shared_key (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>psk_key_exchange_modes (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>early_data (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>cookie (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>
## 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Feature</th>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>supported_versions (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>certificateAuthorities (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>oid_filters (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>post_handshake_auth (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>signature_algorithms_cert (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
</tbody>
</table>

### 4.1.3. Server Hello

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version downgrade protection</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>key_share (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>pre_shared_key (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>supported_versions (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
</tbody>
</table>

### 4.2.1. Supported Versions

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>supported_versions (RFC8446)</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>------------</td>
<td>--------</td>
</tr>
<tr>
<td>4.2.2. Cookie</td>
<td>Client</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td>NC</td>
</tr>
<tr>
<td>4.2.3. Signature Algorithms</td>
<td>Client</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha384</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha512</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_secp256r1_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_secp384r1_sha384</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_secp521r1_sha512</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_rsa_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_rsa_sha384</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_rsa_sha512</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ed25519</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>ed448</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_pss_sha256</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_pss_sha384</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>rsa_pss_pss_sha512</td>
<td>NC</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha1</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_sha1</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Server</td>
<td>PC</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha384</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>rsa_pkcs1_sha512</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_secp256r1_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ecdsa_secp384r1_sha384</td>
<td>C</td>
</tr>
</tbody>
</table>
## 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>Required Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecdsa_secp521r1_sha512</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>rsa_pss_rsaec_sha256</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>rsa_pss_rsaec_sha384</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>rsa_pss_rsaec_sha512</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>ed25519</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>ed448</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>rsa_pss_pss_sha256</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>rsa_pss_pss_sha384</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>rsa_pss_pss_sha512</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>rsa_pks1_sha1</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>ecdsa_sha1</td>
<td>C</td>
<td>22</td>
</tr>
</tbody>
</table>

### 4.2.4. Certificate Authorities

<table>
<thead>
<tr>
<th>Type</th>
<th>Required Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>NC</td>
</tr>
<tr>
<td>Server</td>
<td>NC</td>
</tr>
</tbody>
</table>

### 4.2.5. OID Filters

<table>
<thead>
<tr>
<th>Type</th>
<th>Required Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>NC</td>
</tr>
<tr>
<td>Server</td>
<td>NC</td>
</tr>
</tbody>
</table>

### 4.2.6. Post-Handshake Client Authentication

<table>
<thead>
<tr>
<th>Type</th>
<th>Required Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>NC</td>
</tr>
<tr>
<td>Server</td>
<td>NC</td>
</tr>
</tbody>
</table>

### 4.2.7. Supported Groups

<table>
<thead>
<tr>
<th>Type</th>
<th>Required Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>C 22.1</td>
</tr>
<tr>
<td>secp256r1</td>
<td>C 22.1</td>
</tr>
<tr>
<td>secp384r1</td>
<td>C 22.1</td>
</tr>
<tr>
<td>secp521r1</td>
<td>C 22.1</td>
</tr>
<tr>
<td>x25519</td>
<td>C 22.1</td>
</tr>
<tr>
<td>x448</td>
<td>C 22.1</td>
</tr>
<tr>
<td>ffdhe2048</td>
<td>C 22.1</td>
</tr>
<tr>
<td>ffdhe3072</td>
<td>C 22.1</td>
</tr>
<tr>
<td>ffdhe4096</td>
<td>C 22.1</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Client</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ffdhe6144</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe8192</td>
<td>C</td>
</tr>
<tr>
<td>secp256r1</td>
<td>C</td>
</tr>
<tr>
<td>secp384r1</td>
<td>C</td>
</tr>
<tr>
<td>secp521r1</td>
<td>C</td>
</tr>
<tr>
<td>x25519</td>
<td>C</td>
</tr>
<tr>
<td>x448</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe2048</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe3072</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe4096</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe6144</td>
<td>C</td>
</tr>
<tr>
<td>ffdhe8192</td>
<td>C</td>
</tr>
</tbody>
</table>

4.2.8. Key Share

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>22.1</td>
</tr>
</tbody>
</table>

4.2.9. Pre-Shared Key Exchange Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.10. Early Data Indication

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.11. Pre-Shared Key Extension

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.11.1. Ticket Age

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.11.2. PSK Binder

<table>
<thead>
<tr>
<th>Mode</th>
<th>Client</th>
<th>Server</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 4.2.11.3. Processing Order

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3.1. Encrypted Extensions

<table>
<thead>
<tr>
<th></th>
<th>Client</th>
<th>Server</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>server_name (RFC6066)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>max_fragment_length (RFC6066)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>supported_groups (RFC7919)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>use_srtp (RFC5764)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>heartbeat (RFC6520)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>application_layer_protocol_negotiation (RFC7301)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>client_certificate_type (RFC7250)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>server_certificate_type (RFC7250)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>early_data (RFC8446)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>supported_versions (RFC8446)</td>
<td>NC</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_name (RFC6066)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>max_fragment_length (RFC6066)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>supported_groups (RFC7919)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>use_srtp (RFC5764)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heartbeat (RFC6520)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>application_layer_protocol_negotiation (RFC7301)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>client_certificate_type (RFC7250)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Client Status</th>
<th>Server Status</th>
<th>22.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_certificate_type (RFC7250)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>early_data (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supported_versions (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4.3.2. Certificate Request</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>status_request (RFC6066)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signature_algorithms (RFC8446)</td>
<td>C</td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td>signed_certificate_timestamp (RFC6962)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>certificateAuthorities (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oid_filters (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signature_algorithms_cert (RFC8446)</td>
<td>C</td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td></td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td>status_request (RFC6066)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signature_algorithms (RFC8446)</td>
<td>C</td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td>signed_certificate_timestamp (RFC6962)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>certificateAuthorities (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oid_filters (RFC8446)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signature_algorithms_cert (RFC8446)</td>
<td>C</td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td><strong>4.4.1. The Transcript Hash</strong></td>
<td></td>
<td></td>
<td>22.1</td>
</tr>
<tr>
<td><strong>4.4.2. Certificate</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>status_request (RFC6066)</td>
<td>NC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Standards Compliance</th>
<th>Client</th>
<th>Server</th>
<th>PC</th>
<th>C</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed_certificate_timestamp (RFC6962)</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>status_request (RFC6066)</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>signed_certificate_timestamp (RFC6962)</td>
<td>NC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.2.1. OCSP Status and SCT Extensions

<table>
<thead>
<tr>
<th>Standards Compliance</th>
<th>Client</th>
<th>Server</th>
<th>NC</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

### 4.4.2.2. Server Certificate Selection

<table>
<thead>
<tr>
<th>Standards Compliance</th>
<th>Client</th>
<th>Server</th>
<th>PC</th>
<th>C</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>The certificate type MUST be X.509v3, unless explicitly negotiated otherwise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm from the client's &quot;signature_algorithms&quot; extension (currently RSA, ECDSA, or EdDSA).</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The certificate MUST allow the key to be used for signing with a signature scheme indicated in the client's &quot;signature_algorithms&quot;/&quot;signature_algorithms_cert&quot; extensions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The &quot;server_name&quot; and &quot;certificateAuthorities&quot; extensions are used to guide certificate selection. As servers MAY require the presence of the &quot;server_name&quot; extension, clients SHOULD send this extension, when applicable.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 4.4.2.3. Client Certificate Selection

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The certificate type MUST be X.509v3, unless explicitly negotiated otherwise</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>If the “certificateAuthorities” extension in the CertificateRequest message was present, at least one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>The certificates MUST be signed using an acceptable signature algorithm</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>If the CertificateRequest message contained a non-empty &quot;oid_filters&quot; extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>

### 4.4.2.4. Receiving a Certificate Message

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>Server</td>
<td>C</td>
<td>22</td>
</tr>
</tbody>
</table>

### 4.4.3. Certificate Verify

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>Server</td>
<td>C</td>
<td>22</td>
</tr>
</tbody>
</table>

### 4.4.4. Finished

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>Server</td>
<td>C</td>
<td>22</td>
</tr>
</tbody>
</table>

### 4.5. End of Early Data

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>Server</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>

### 4.6.1. New Session Ticket Message

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>early_data (RFC8446)</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>
### 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Server</th>
<th>Client</th>
<th>NC</th>
</tr>
</thead>
<tbody>
<tr>
<td>early_data (RFC8446)</td>
<td>NC</td>
<td>NC</td>
</tr>
</tbody>
</table>

#### 4.6.2. Post-Handshake Authentication

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>NC</th>
</tr>
</thead>
</table>

#### 4.6.3. Key and Initialization Vector Update

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>NC</th>
</tr>
</thead>
</table>

#### 5.1. Record Layer

| MUST NOT be interleaved with other record types | C | 22 |
| MUST NOT span key changes | C | 22 |
| MUST NOT send zero-length fragments | C | 22 |
| Alert messages MUST NOT be fragmented | C | 22 |

#### 5.2. Record Payload Protection

| C | 22 |

#### 5.3. Per-Record Nonce

| C | 22 |

#### 5.4. Record Padding

| MAY choose to pad | NC |
| MUST NOT send Handshake and Alert records that have a zero-length TLSInnerPlaintext.content | NC |
| The padding sent is automatically verified | C | 22 |

#### 5.5. Limits on Key Usage

| NC |

#### 6.1. Closure Alerts

<p>| close_notify | NC |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Compliance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2. Error Alerts</td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td>7.1. Key Schedule</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>7.2. Updating Traffic Secrets</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>7.3. Traffic Key Calculation</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>7.5. Exporters</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>8.0-RTT and Anti-Replay</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>8.1. Single-Use Tickets</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>8.2. Client Hello Recording</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>8.3. Freshness Checks</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>9.1. Mandatory-to-Implement Cipher Suites</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td></td>
<td>MUST implement the TLS_AES_128_GCM_SHA256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>SHOULD implement the TLS_AES_256_GCM_SHA384</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>SHOULD implement the TLS_CHACHA20_POLY1305_SHA256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Digital signatures</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>MUST support rsa_pkcs1_sha256 (for certificates)</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>MUST support rsa_pss_rsaes_sha256 (for CertificateVerify and certificates)</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>MUST support ecdsa_secp256r1_sha256</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>Key Exchange</td>
<td>C</td>
</tr>
</tbody>
</table>
### 9.2. Mandatory-to-Implement Extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Requirement</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUST send and use these extensions</td>
<td>PC</td>
<td>22.1</td>
</tr>
<tr>
<td>&quot;supported_versions&quot; is REQUIRED for ClientHello, ServerHello and HelloRetryRequest</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>&quot;signature_algorithms&quot; is REQUIRED for certificate authentication</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>&quot;supported_groups&quot; is REQUIRED for ClientHello messages using (EC)DHE key exchange</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>&quot;key_share&quot; is REQUIRED for (EC)DHE key exchange</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>&quot;pre_shared_key&quot; is REQUIRED for PSK key agreement</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>
### 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Compliance</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;psk_key_exchange_modes&quot; is REQUIRED for PSK key agreement</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td><strong>TLS 1.3 ClientHello</strong></td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>If not containing a &quot;pre_shared_key&quot; extension, it MUST contain both a</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>&quot;signature_algorithms&quot; extension and a &quot;supported_groups&quot; extension.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>If containing a &quot;supported_groups&quot; extension, it MUST also contain a &quot;key_share&quot; extension, and vice versa. An empty KeyShare.client_shares vector is permitted.</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td><strong>TLS 1.3 ServerHello</strong></td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td>MUST support the use of the &quot;server_name&quot; extension</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td><strong>9.3. Protocol Invariants</strong></td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>MUST correctly handle extensible fields</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>A client sending a ClientHello MUST support all parameters advertised in it. Otherwise, the server may fail to interoperate by selecting one of those parameters.</td>
<td>C</td>
<td>22.1</td>
</tr>
<tr>
<td>A server receiving a ClientHello MUST correctly ignore all unrecognized cipher suites, extensions, and other parameters. Otherwise, it may fail to interoperate with newer clients. In TLS 1.3, a client receiving</td>
<td>C</td>
<td>22.1</td>
</tr>
</tbody>
</table>
a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.

<table>
<thead>
<tr>
<th>A middlebox which terminates a TLS connection MUST behave as a compliant TLS server</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A middlebox which forwards ClientHello parameters it does not understand MUST NOT process any messages beyond that ClientHello. It MUST forward all subsequent traffic unmodified. Otherwise, it may fail to interoperate with newer clients and servers.</td>
<td>NA</td>
</tr>
</tbody>
</table>

**B.4. Cipher Suites**

<table>
<thead>
<tr>
<th>Cipher Suite</th>
<th>PC</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_AES_128_GCM_SHA256</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TLS_AES_256_GCM_SHA384</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TLS_CHACHA20_POLY1305_SHA256</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TLS_AES_128_CCM_SHA256</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TLS_AES_128_CCM_8_SHA256</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>TLS_AES_128_CCM_8_SHA256</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>C.1. Random Number Generation and Seeding</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>C.2. Certificates and Authentication</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>C.3. Implementation Pitfalls</td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td>C.4. Client Tracking Prevention</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>C.5. Unauthenticated Operation</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>D.1. Negotiating with an Older Server</td>
<td>NC</td>
<td></td>
</tr>
</tbody>
</table>
### 1.5 Standards Compliance

<table>
<thead>
<tr>
<th>Section</th>
<th>Compliance</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.2. Negotiating with an Older Client</td>
<td>C</td>
<td>22</td>
</tr>
<tr>
<td>D.3. 0-RTT Backward Compatibility</td>
<td>NC</td>
<td></td>
</tr>
<tr>
<td>D.4. Middlebox Compatibility Mode</td>
<td>PC</td>
<td>22</td>
</tr>
<tr>
<td>D.5. Security Restrictions Related to Backward Compatibility</td>
<td>C</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 5.1: Standards Compliance
2 Reference Manual
The ssl application is an implementation of the SSL, TLS and DTLS protocols in Erlang.
For current statement of standards compliance see the User’s Guide.

DEPENDENCIES
The SSL application uses the public_key, asn1 and Crypto application to handle public keys and encryption, hence these applications must be loaded for the SSL application to work. In an embedded environment this means they must be started with `application:start/[1,2]` before the SSL application is started.

CONFIGURATION
The application environment configuration parameters in this section are defined for the SSL application. For more information about configuration parameters, see the `application(3)` manual page in Kernel.

The environment parameters can be set on the command line, for example:

```
erl -ssl protocol_version "['tlsv1.2', 'tlsv1.1']"
```

```erlang
protocol_version = ssl:ssl_tls_protocol()<optional>
```
Protocol supported by started clients and servers. If this option is not set, it defaults to all TLS protocols currently supported by the SSL application. This option can be overridden by the version option to `ssl:connect/2` and `ssl:listen/2`.

```erlang
dtls_protocol_version = ssl:dtls_protocol()<optional>
```
Protocol supported by started clients and servers. If this option is not set, it defaults to all DTLS protocols currently supported by the SSL application. This option can be overridden by the version option to `ssl:connect/2` and `ssl:listen/2`.

```erlang
session_lifetime = integer() <optional>
```
Maximum lifetime of the session data in seconds. Defaults to 24 hours which is the maximum recommended lifetime by RFC 5246. However sessions may be invalidated earlier due to the maximum limitation of the session cache table.

```erlang
session_cb = atom() <optional>
```
Name of the session cache callback module that implements the `ssl_session_cache_api` behavior. Defaults to `ssl_session_cache`.

```erlang
session_cb_init_args = proplist:proplist() <optional>
```
List of extra user-defined arguments to the `init` function in the session cache callback module. Defaults to `[]`.

```erlang
session_cache_client_max = integer() <optional>
```
Limits the growth of the clients session cache, that is how many sessions towards servers that are cached to be used by new client connections. If the maximum number of sessions is reached, the current cache entries will be invalidated regardless of their remaining lifetime. Defaults to 1000. Recommended ssl-8.2.1 or later for this option to work as intended.

```erlang
session_cache_server_max = integer() <optional>
```
Limits the growth of the servers session cache, that is how many client sessions are cached by the server. If the maximum number of sessions is reached, the current cache entries will be invalidated regardless of their remaining lifetime. Defaults to 1000. Recommended ssl-8.2.1 or later for this option to work as intended.
ssl_pem_cache_clean = integer() <optional>

   Number of millisecond between PEM cache validations. Defaults to 2 minutes.
ssl:clear_pem_cache()

bypass_pem_cache = boolean() <optional>

   Introduced in ssl-8.0.2. Disables the PEM-cache. Can be used as a workaround for the PEM-cache bottleneck
   before ssl-8.1.1. Defaults to false.
alert_timeout = integer() <optional>

   Number of milliseconds between sending of a fatal alert and closing the connection. Waiting a little while
   improves the peers chances to properly receiving the alert so it may shutdown gracefully. Defaults to 5000
   milliseconds.

internal_active_n = integer() <optional>

   For TLS connections this value is used to handle the internal socket. As the implementation was changed from
   an active once to an active N behavior (N = 100), for performance reasons, this option exist for possible tweaking
   or restoring of the old behavior (internal_active_n = 1) in unforeseen scenarios. The option will not affect erlang
   distribution over TLS that will always run in active N mode. Added in ssl-9.1 (OTP-21.2).

ERROR LOGGER AND EVENT HANDLERS
The SSL application uses the default OTP error logger to log unexpected errors and TLS/DTLS alerts. The logging
of TLS/DTLS alerts may be turned off with the log_alert option.

SEE ALSO

application(3)
ssl

Erlang module

This module contains interface functions for the SSL/TLS/DTLS protocol. For detailed information about the supported standards see ssl(6).

Data Types

Types used in SSL/TLS/DTLS

socket() = gen_tcp:socket()

sslsocket() = any()

An opaque reference to the TLS/DTLS connection, may be used for equality matching.

tls_option() = tls_client_option() | tls_server_option()

tls_client_option() =
  client_option() |
  common_option() |
  socket_option() |
  transport_option()

tls_server_option() =
  server_option() |
  common_option() |
  socket_option() |
  transport_option()

socket_option() =
  gen_tcp:connect_option() |
  gen_tcp:listen_option() |
  gen_udp:option() |

The default socket options are [[mode, list], {packet, 0}, {header, 0}, {active, true}].

For valid options, see the inet(3), gen_tcp(3) and gen_udp(3) manual pages in Kernel. Note that stream oriented options such as packet are only relevant for SSL/TLS and not DTLS

active_msgs() =
  {ssl, sslsocket(), Data :: binary() | list()} |
  {ssl_closed, sslsocket()} |
  {ssl_error, sslsocket(), Reason :: any()} |
  {ssl_passive, sslsocket()}

When a TLS/DTLS socket is in active mode (the default), data from the socket is delivered to the owner of the socket in the form of messages as described above.

The ssl_passive message is sent only when the socket is in {active, N} mode and the counter dropped to 0. It indicates that the socket has transitioned to passive ({active, false}) mode.

transport_option() =
  {cb_info,
    {CallbackModule :: atom(),
     DataTag :: atom(),
     ClosedTag :: atom(),
     ErrTag :: atom()}} |
{cb_info,
   {CallbackModule :: atom(),
   DataTag :: atom(),
   ClosedTag :: atom(),
   ErrTag :: atom(),
   PassiveTag :: atom()}}

Defaults to \{gen_tcp, tcp, tcp_closed, tcp_error, tcp_passive\} for TLS (for backward compatibility a four tuple will be converted to a five tuple with the last element "second_element_passive") and \{gen_udp, udp, udp_closed, udp_error\} for DTLS (might also be changed to five tuple in the future).

Can be used to customize the transport layer. The tag values should be the values used by the underlying transport in its active mode messages. For TLS the callback module must implement a reliable transport protocol, behave as \texttt{gen_tcp}, and have functions corresponding to \texttt{inet:setopts/2}, \texttt{inet:getopts/2}, \texttt{inet:peername/1}, \texttt{inet:sockname/1}, and \texttt{inet:port/1}. The callback \texttt{gen_tcp} is treated specially and calls \texttt{inet} directly. For DTLS this feature must be considered experimental.

\begin{verbatim}
host() = hostname() | ip_address()
hostname() = string()
ip_address() = inet:ip_address()
protocol_version() = tls_version() | dtls_version()
tls_version() = 'tlsv1.2' | 'tlsv1.3' | tls_legacy_version()
dtls_version() = 'dtlsv1.2' | dtls_legacy_version()
tls_legacy_version() = tlsv1 | 'tlsv1.1' | sslv3
dtls_legacy_version() = dtlsv1
prf_random() = client_random | server_random
verify_type() = verify_none | verify_peer
ciphers() = [erl_cipher_suite()] | string()
erl_cipher_suite() =
   #\{key_exchange := kex_algo(),
     cipher := cipher(),
     mac := hash() | aead,
     prf := hash() | default_prf\}
cipher() =
aes_128_cbc | aes_256_cbc | aes_128_gcm | aes_256_gcm |
aes_128_ccm | aes_256_ccm | aes_128_ccm_8 | aes_256_ccm_8 |
chacha20_poly1305 |

legacy_cipher() = rc4_128 | des_cbc | '3des_ede_cbc'
cipher_filters() =
   [{key_exchange | cipher | mac | prf, algo_filter()}]
hash() = sha | sha2() | legacy_hash()
sha2() = sha224 | sha256 | sha384 | sha512
legacy_hash() = md5
old_cipher_suite() =
   \{kex_algo(), cipher(), hash()\} |
   \{kex_algo(), cipher(), hash() | aead, hash()\}
signature_algs() = [[hash(), sign_algo()]]
sign_algo() = rsa | dsa | ecdsa
sign_scheme() =
\end{verbatim}
ssl

rsa_pkcs1_sha256 | rsa_pkcs1_sha384 | rsa_pkcs1_sha512 |
ecdsa_secp256r1_sha256 | ecdsa_secp384r1_sha384 |
ecdsa_secp521r1_sha512 | rsa_pss_rsae_sha256 |
rsa_pss_rsae_sha384 | rsa_pss_rsaes_sha512 |
rsa_pss_pss_sha256 | rsa_pss_pss_sha384 | rsa_pss_pss_sha512 |
rsa_pkcs1_sha1 | ecdsa_sha1
group() =
  secp256r1 | secp384r1 | secp521r1 | ff dhe2048 | ff dhe3072 | ff dhe4096 | ff dhe6144 | ff dhe8192
kex_algo() =
  rsa | dhe_rsa | dhe_dss | ecdhe_ecdsa | ecdh_ecdsa |
  ecdh_rsa | srp_rsa | srp_dss | psk | dhe_psk | rsa_psk |
  dh_anon | ecdh_anon | srp_anon | any
algo_filter() =
  fun((kex_algo() | cipher() | hash() | aead | default_prf) ->
    true | false)
named_curve() =
  sect571r1 | sect571k1 | secp521r1 | brainpoolP512r1 |
  sect409k1 | sect409r1 | brainpoolP384r1 | secp384r1 |
  sect283k1 | sect283r1 | brainpoolP256r1 | secp256k1 |
  secp256r1 | sect239k1 | sect233k1 | sect233r1 | secp224k1 |
  secp224r1 | sect193r1 | sect193r2 | secp192k1 | secp192r1 |
  sect163k1 | sect163r1 | sect163r2 | secp160k1 | secp160r1 |
  secp160r2
psk_identity() = string()
srp_identity() = {Username :: string(), Password :: string()}
srp_param_type() =
  srp_1024 | srp_1536 | srp_2048 | srp_3072 | srp_4096 |
  srp_6144 | srp_8192
app_level_protocol() = binary()
protocol_extensions() =
  #{renegotiation_info => binary(),
    signature_algs => signature_algs(),
    alpn => app_level_protocol(),
    srp => binary(),
    next_protocol => app_level_protocol(),
    ec_point_formats => [0..2],
    elliptic_curves => [public_key:oid()],
    sni => hostname()}
error_alert() =
  {tls_alert, {tls_alert(), Description :: string()}}
tls_alert() =
  close_notify | unexpected_message | bad_record_mac |
  record_overflow | handshake_failure | bad_certificate |
  unsupported_certificate | certificate_revoked |
  certificate_expired | certificate_unknown |
  illegal_parameter | unknown_ca | access_denied |
  decode_error | decrypt_error | export_restriction |
  protocol_version | insufficient_security | internal_error |
  inappropriate_fallback | user_canceled | no renegotiation |
unsupported_extension | certificate_unobtainable | 
unrecognized_name | bad_certificate_status_response | 
bad_certificate_hash_value | unknown_psk_identity | 
no_application_protocol
reason() = any()

TLS/DTLS OPTION DESCRIPTIONS - COMMON for SERVER and CLIENT

common_option() =
{protocol, protocol()} | 
{handshake, handshake_completion()} | 
{cert, cert()} | 
{certfile, cert_pem()} | 
{key, key()} | 
{keyfile, key_pem()} | 
{password, key_password()} | 
{ciphers, cipher_suites()} | 
{eccs, [named_curve()]} | 
{signature_algs_cert, signature_schemes()} | 
{supported_groups, supported_groups()} | 
{secure_renegotiate, secure_renegotiation()} | 
{depth, allowed_cert_chain_length()} | 
{verify_fun, custom_verify()} | 
{crl_check, crl_check()} | 
{crl_cache, crl_cache_opts()} | 
{max_handshake_size, handshake_size()} | 
{partial_chain, root_fun()} | 
{versions, protocol_versions()} | 
{user_lookup_fun, custom_user_lookup()} | 
{log_level, logging_level()} | 
{log_alert, log_alert()} | 
{hibernate_after, hibernate_after()} | 
{padding_check, padding_check()} | 
{beast_mitigation, beast_mitigation()} | 
{ssl_imp, ssl_imp()}

protocol() = tls | dtls
Choose TLS or DTLS protocol for the transport layer security. Defaults to tls. For DTLS other transports than UDP are not yet supported.

handshake_completion() = hello | full
Defaults to full. If hello is specified the handshake will pause after the hello message and give the user a possibility make decisions based on hello extensions before continuing or aborting the handshake by calling handshake_continue/3 or handshake_cancel/1

cert() = public_key:der_encoded()
The DER-encoded users certificate. If this option is supplied, it overrides option certfile.

cert_pem() = file:filename()
Path to a file containing the user certificate on PEM format.

key() =
{RSAPrivateKey' | 'DSAPrivateKey' | 'ECPrivateKey' | 
'PrivateKeyInfo',
**public_key:der_encoded()**

```erlang
#{algorithm := rsa | dss | ecdsa, 
  engine := crypto:engine_ref(),
  key_id := crypto:key_id(),
  password => crypto:password()}
```

The DER-encoded user's private key or a map referring to a crypto engine and its key reference that optionally can be password protected, see also crypto:engine_load/4 and Crypto's Users Guide. If this option is supplied, it overrides option keyfile.

**key_pem() = file:filename()**

Path to the file containing the user's private PEM-encoded key. As PEM-files can contain several entries, this option defaults to the same file as given by option certfile.

**key_password() = string()**

String containing the user's password. Only used if the private keyfile is password-protected.

**cipher_suites() = ciphers()**

Supported cipher suites. The function cipher_suites/2 can be used to find all ciphers that are supported by default. cipher_suites(all, 'tls1.2') can be called to find all available cipher suites. Pre-Shared Key (RFC 4279 and RFC 5487), Secure Remote Password (RFC 5054), RC4, 3DES, DES cipher suites, and anonymous cipher suites only work if explicitly enabled by this option; they are supported/enabled by the peer also. Anonymous cipher suites are supported for testing purposes only and are not be used when security matters.

**signature_schemes() = [sign_scheme()]**

In addition to the signature_algorithms extension from TLS 1.2, **TLS 1.3 (RFC 5246 Section 4.2.3)** adds the signature_algorithms_cert extension which enables having special requirements on the signatures used in the certificates that differs from the requirements on digital signatures as a whole. If this is not required this extension is not needed.

The client will send a signature_algorithms_cert extension (ClientHello), if TLS version 1.3 or later is used, and the signature_algs_cert option is explicitly specified. By default, only the signature_algs extension is sent.

The signature schemes shall be ordered according to the client's preference (favorite choice first).

**supported_groups() = [group()]**

TLS 1.3 introduces the "supported_groups" extension that is used for negotiating the Diffie-Hellman parameters in a TLS 1.3 handshake. Both client and server can specify a list of parameters that they are willing to use.

If it is not specified it will use a default list ([x25519, x448, secp256r1, secp384r1]) that is filtered based on the installed crypto library version.

**secure_renegotiation() = boolean()**

Specifies if to reject renegotiation attempt that does not live up to RFC 5746. By default secure_renegotiate is set to true, that is, secure renegotiation is enforced. If set to false secure renegotiation will still be used if possible, but it falls back to insecure renegotiation if the peer does not support RFC 5746.

**allowed_cert_chain_length() = integer()**

Maximum number of non-self-issued intermediate certificates that can follow the peer certificate in a valid certification path. So, if depth is 0 the PEER must be signed by the trusted ROOT-CA directly; if 1 the path can be PEER, CA, ROOT-CA; if 2 the path can be PEER, CA, CA, ROOT-CA, and so on. The default value is 1.

**custom_verify() = {Verifyfun :: function(), InitialUserState :: any()}**

The verification fun is to be defined as follows:
The verification fun is called during the X509-path validation when an error or an extension unknown to the SSL application is encountered. It is also called when a certificate is considered valid by the path validation to allow access to each certificate in the path to the user application. It differentiates between the peer certificate and the CA certificates by using `valid_peer` or `valid` as second argument to the verification fun. See the `public_key User's Guide` for definition of `#'OTPCertificate'{}` and `#'Extension'{}`.

- If the verify callback fun returns `{fail, Reason}`, the verification process is immediately stopped, an alert is sent to the peer, and the TLS/DTLS handshake terminates.
- If the verify callback fun returns `{valid, UserState}`, the verification process continues.
- If the verify callback fun always returns `{valid, UserState}`, the TLS/DTLS handshake does not terminate regarding verification failures and the connection is established.
- If called with an extension unknown to the user application, return value `{unknown, UserState}` is to be used.

Note that if the fun returns `unknown` for an extension marked as critical, validation will fail.

Default option `verify_fun` in `verify_peer` mode:

```erlang
{fun(_,{bad_cert, _} = Reason, _) ->
   {fail, Reason};
   (_,{extension, _}, UserState) ->
   {unknown, UserState};
   (_, valid, UserState) ->
   {valid, UserState};
   (_, valid_peer, UserState) ->
   {valid, UserState}
end, [{}].
```

Default option `verify_fun` in mode `verify_none`:

```erlang
{fun(_, {bad_cert, _}, UserState) ->
   {valid, UserState};
   (_,{extension, #'Extension'{critical = true}}, UserState) ->
   {valid, UserState};
   (_,{extension, _}, UserState) ->
   {unknown, UserState};
   (_, valid, UserState) ->
   {valid, UserState};
   (_, valid_peer, UserState) ->
   {valid, UserState}
end, [{}].
```

The possible path validation errors are given on form `{bad_cert, Reason}` where `Reason` is:

- `unknown_ca`
  No trusted CA was found in the trusted store. The trusted CA is normally a so called ROOT CA, which is a self-signed certificate. Trust can be claimed for an intermediate CA (trusted anchor does not have to be self-signed according to X-509) by using option `partial_chain`.

- `selfsigned_peer`
  The chain consisted only of one self-signed certificate.
PKIX X-509-path validation error

For possible reasons, see public_key:pkix_path_validation/3

**crl_check() = boolean() | peer | best_effort**

Perform CRL (Certificate Revocation List) verification (public_key:pkix_crls_validate/3) on all the certificates during the path validation (public_key:pkix_path_validation/3) of the certificate chain. Defaults to false.

- **peer**
  - check is only performed on the peer certificate.
- **best_effort**
  - if certificate revocation status cannot be determined it will be accepted as valid.

The CA certificates specified for the connection will be used to construct the certificate chain validating the CRLs.

The CRLs will be fetched from a local or external cache. See ssl_crl_cache_api(3).

**crl_cache_opts() = [any()]**

Specify how to perform lookup and caching of certificate revocation lists. Module defaults to ssl_crl_cache with DbHandle being internal and an empty argument list.

There are two implementations available:

**ssl_crl_cache**

This module maintains a cache of CRLs. CRLs can be added to the cache using the function ssl_crl_cache:insert/1, and optionally automatically fetched through HTTP if the following argument is specified:

- `{http, timeout()}`

  Enables fetching of CRLs specified as http URIs in X509 certificate extensions. Requires the OTP inets application.

**ssl_crl_hash_dir**

This module makes use of a directory where CRLs are stored in files named by the hash of the issuer name.

The file names consist of eight hexadecimal digits followed by .rN, where N is an integer, e.g. 1a2b3c4d.r0. For the first version of the CRL, N starts at zero, and for each new version, N is incremented by one. The OpenSSL utility c_rehash creates symlinks according to this pattern.

For a given hash value, this module finds all consecutive .r* files starting from zero, and those files taken together make up the revocation list. CRL files whose nextUpdate fields are in the past, or that are issued by a different CA that happens to have the same name hash, are excluded.

The following argument is required:

- `{dir, string()}`

  Specifies the directory in which the CRLs can be found.

**root_fun() = function()**

fun(Chain::[public_key:der_encoded()]) ->
  (trusted_ca, DerCert::public_key:der_encoded() | unknown_ca)

Claim an intermediate CA in the chain as trusted. TLS then performs public_key:pkix_path_validation/3 with the selected CA as trusted anchor and the rest of the chain.

**protocol_versions() = [protocol_version()]**

TLS protocol versions supported by started clients and servers. This option overrides the application environment option protocol_version and dtls_protocol_version. If the environment option is not set, it defaults to all versions, except SSL-3.0, supported by the SSL application. See also ssl(6).
custom_user_lookup() =
  {Lookupfun :: function(), UserState :: any()}

The lookup fun is to defined as follows:

fun(psk, PSKIdentity :: string(), UserState :: term()) ->
  {ok, SharedSecret :: binary()} | error;
fun(srp, Username :: string(), UserState :: term()) ->
  {ok, {SRPParams :: srp_param_type(), Salt :: binary(),
       DerivedKey :: binary()}} | error.

For Pre-Shared Key (PSK) cipher suites, the lookup fun is called by the client and server to determine the shared secret. When called by the client, PSKIdentity is set to the hint presented by the server or to undefined. When called by the server, PSKIdentity is the identity presented by the client.

For Secure Remote Password (SRP), the fun is only used by the server to obtain parameters that it uses to generate its session keys. DerivedKey is to be derived according to RFC 2945 and RFC 5054: crypto:sha([Salt, crypto:sha([Username, <<$:>>, Password]])

session_id() = binary()
Identifies a TLS session.

log_alert() = boolean()
If set to false, error reports are not displayed. Deprecated in OTP 22, use {log_level, logging_level()} instead.

logging_level() = logger:level()
Specifies the log level for TLS/DTLS. At verbosity level notice and above error reports are displayed in TLS/DTLS. The level debug triggers verbose logging of TLS/DTLS protocol messages.

hibernate_after() = timeout()
When an integer-value is specified, TLS/DTLS-connection goes into hibernation after the specified number of milliseconds of inactivity, thus reducing its memory footprint. When undefined is specified (this is the default), the process never goes into hibernation.

handshake_size() = integer()
Integer (24 bits unsigned). Used to limit the size of valid TLS handshake packets to avoid DoS attacks. Defaults to 256*1024.

padding_check() = boolean()
Affects TLS-1.0 connections only. If set to false, it disables the block cipher padding check to be able to interoperate with legacy software.

**Warning:**
Using {padding_check, boolean()} makes TLS vulnerable to the Poodle attack.

beast_mitigation() = one_n_minus_one | zero_n | disabled
Affects SSL-3.0 and TLS-1.0 connections only. Used to change the BEAST mitigation strategy to interoperate with legacy software. Defaults to one_n_minus_one.

one_n_minus_one - Perform l/n-1 BEAST mitigation.
zero_n - Perform 0/n BEAST mitigation.
disabled - Disable BEAST mitigation.
Warning:

Using (beast_mitigation, disabled) makes SSL-3.0 or TLS-1.0 vulnerable to the BEAST attack.

ssl_imp() = new | old

Deprecated since OTP-17, has no affect.

**TLS/DTLS OPTION DESCRIPTIONS - CLIENT**

```plaintext
client_option() =
  {verify, client_verify_type()} | 
  {reuse_session, client_reuse_session()} | 
  {reuse_sessions, client_reuse_sessions()} | 
  {cacerts, client_cacerts()} | 
  {cacertfile, client_cafile()} | 
  {alpn_advertised_protocols, client_alpn()} | 
  {client_preferred_next_protocols,} | 
  {psk_identity, client_psk_identity()} | 
  {srp_identity, client_srp_identity()} | 
  {server_name_indication, sni()} | 
  {customize_hostname_check, customize_hostname_check()} | 
  {signature_algs, client_signature_algs()} | 
  {fallback, fallback()}
```

```plaintext
client_verify_type() = verify_type()
```

In mode verify_none the default behavior is to allow all x509-path validation errors. See also option verify_fun.

```plaintext
client_reuse_session() = session_id()
```

Reuses a specific session earlier saved with the option {reuse_sessions, save} since OTP-21.3

```plaintext
client_reuse_sessions() = boolean() | save
```

When save is specified a new connection will be negotiated and saved for later reuse. The session ID can be fetched with connection_information/2 and used with the client option reuse_session The boolean value true specifies that if possible, automatized session reuse will be performed. If a new session is created, and is unique in regard to previous stored sessions, it will be saved for possible later reuse. Since OTP-21.3

```plaintext
client_cacerts() = [public_key:der_encoded()]
```

The DER-encoded trusted certificates. If this option is supplied it overrides option cacertfile.

```plaintext
client_cafile() = file:filename()
```

Path to a file containing PEM-encoded CA certificates. The CA certificates are used during server authentication and when building the client certificate chain.

```plaintext
client_alpn() = [app_level_protocol()]
```

The list of protocols supported by the client to be sent to the server to be used for an Application-Layer Protocol Negotiation (ALPN). If the server supports ALPN then it will choose a protocol from this list; otherwise it will fail the connection with a "no_application_protocol" alert. A server that does not support ALPN will ignore this value.

The list of protocols must not contain an empty binary.

The negotiated protocol can be retrieved using the negotiated_protocol/1 function.

```plaintext
client_preferred_next_protocols() =
  {Precedence :: server | client,
```
Indicates that the client is to try to perform Next Protocol Negotiation.

If precedence is server, the negotiated protocol is the first protocol to be shown on the server advertised list, which is also on the client preference list.

If precedence is client, the negotiated protocol is the first protocol to be shown on the client preference list, which is also on the server advertised list.

If the client does not support any of the server advertised protocols or the server does not advertise any protocols, the client falls back to the first protocol in its list or to the default protocol (if a default is supplied). If the server does not support Next Protocol Negotiation, the connection terminates if no default protocol is supplied.

client_psk_identity() = psk_identity()

Specifies the identity the client presents to the server. The matching secret is found by calling user_lookup_fun

client_srp_identity() = srp_identity()

Specifies the username and password to use to authenticate to the server.

sni() = hostname() | disable

Specify the hostname to be used in TLS Server Name Indication extension. If not specified it will default to the Host argument of `connect/3,4` unless it is of type inet:ipaddress().

The HostName will also be used in the hostname verification of the peer certificate using `public_key:pkix_verify_hostname/2`

The special value `disable` prevents the Server Name Indication extension from being sent and disables the hostname verification check `public_key:pkix_verify_hostname/2`

customize_hostname_check() = list()

Customizes the hostname verification of the peer certificate, as different protocols that use TLS such as HTTP or LDAP may want to do it differently, for possible options see `public_key:pkix_verify_hostname/3`

fallback() = boolean()

Send special cipher suite TLS_FALLBACK_SCSV to avoid undesired TLS version downgrade. Defaults to false

**Warning:**

Note this option is not needed in normal TLS usage and should not be used to implement new clients. But legacy clients that retries connections in the following manner

```erlang
ssl:connect(Host, Port, [...{versions, ['tlsv2', 'tlsv1.1', 'tlsv1', 'sslv3']}])
ssl:connect(Host, Port, [...{versions, [tlsv1.1', 'tlsv1', 'sslv3']}, {fallback, true}])
ssl:connect(Host, Port, [...{versions, ['tlsv1', 'sslv3']}, {fallback, true}])
ssl:connect(Host, Port, [...{versions, ['sslv3']}, {fallback, true}])
```

may use it to avoid undesired TLS version downgrade. Note that TLS_FALLBACK_SCSV must also be supported by the server for the prevention to work.
client_signature_algs() = signature_algs()

In addition to the algorithms negotiated by the cipher suite used for key exchange, payload encryption, message authentication and pseudo random calculation, the TLS signature algorithm extension Section 7.4.1.4.1 in RFC 5246 may be used, from TLS 1.2, to negotiate which signature algorithm to use during the TLS handshake. If no lower TLS versions than 1.2 are supported, the client will send a TLS signature algorithm extension with the algorithms specified by this option. Defaults to

\[
\text{
\begin{verbatim}
  \%
  \{ sha512, ecdsa, sha384, ecdsa, sha256, ecdsa, sha224, ecdsa, sha224, rsa, 
  \}\%
  \{ sha, ecdsa, sha, rsa, sha, dsa, 
\end{verbatim}
}\]

The algorithms should be in the preferred order. Selected signature algorithm can restrict which hash functions that may be selected. Default support for {md5, rsa} removed in ssl-8.0

**TLS/DTLS OPTION DESCRIPTIONS - SERVER**

server_option() =

{cacerts, server_cacerts()} |
{cacertfile, server_cacertfile()} |
{dh, dh_der()} |
{dhfile, dh_file()} |
{verify, server_verify_type()} |
{fail_if_no_peer_cert, fail_if_no_peer_cert()} |
{reuse_sessions, server_reuse_sessions()} |
{reuse_session, server_reuse_session()} |
{alpn_preferred_protocols, server_alpn()} |
{next_protocols_advertised, server_next_protocol()} |
{psk_identity, server_psk_identity()} |
{honor_cipher_order, boolean()} |
{sni_hosts, sni_hosts()} |
{sni_fun, sni_fun()} |
{honor_cipher_order, honor_cipher_order()} |
{honor_ecc_order, honor_ecc_order()} |
{client_renegotiation, client_renegotiation()} |
{signature_algs, server_signature_algs()}

server_cacerts() = \[public_key:der_encoded()\]

The DER-encoded trusted certificates. If this option is supplied it overrides option cacertfile.

server_cacertfile() = file:filename()

Path to a file containing PEM-encoded CA certificates. The CA certificates are used to build the server certificate chain and for client authentication. The CAs are also used in the list of acceptable client CAs passed to the client when a certificate is requested. Can be omitted if there is no need to verify the client and if there are no intermediate CAs for the server certificate.
dh_der() = binary()
The DER-encoded Diffie-Hellman parameters. If specified, it overrides option dh_file.

**Warning:**
The dh_der option is not supported by TLS 1.3. Use the supported_groups option instead.

dh_file() = file:filename()
Path to a file containing PEM-encoded Diffie Hellman parameters to be used by the server if a cipher suite using Diffie Hellman key exchange is negotiated. If not specified, default parameters are used.

**Warning:**
The dh_file option is not supported by TLS 1.3. Use the supported_groups option instead.

server_verify_type() = verify_type()
A server only does x509-path validation in mode verify_peer, as it then sends a certificate request to the client (this message is not sent if the verify option is verify_none). You can then also want to specify option fail_if_no_peer_cert.

fail_if_no_peer_cert() = boolean()
Used together with {verify, verify_peer} by an TLS/DTLS server. If set to true, the server fails if the client does not have a certificate to send, that is, sends an empty certificate. If set to false, it fails only if the client sends an invalid certificate (an empty certificate is considered valid). Defaults to false.

server_reuse_sessions() = boolean()
The boolean value true specifies that the server will agree to reuse sessions. Setting it to false will result in an empty session table, that is no sessions will be reused. See also option reuse_session

server_reuse_session() = function()
Enables the TLS/DTLS server to have a local policy for deciding if a session is to be reused or not. Meaningful only if reuse_sessions is set to true. SuggestedSessionId is a binary(), PeerCert is a DER-encoded certificate, Compression is an enumeration integer, and CipherSuite is of type ciphersuite().

server_alpn() = [app_level_protocol()]
Indicates the server will try to perform Application-Layer Protocol Negotiation (ALPN).
The list of protocols is in order of preference. The protocol negotiated will be the first in the list that matches one of the protocols advertised by the client. If no protocol matches, the server will fail the connection with a "no_application_protocol" alert.
The negotiated protocol can be retrieved using the negotiated_protocol/1 function.

server_next_protocol() = [app_level_protocol()]
List of protocols to send to the client if the client indicates that it supports the Next Protocol extension. The client can select a protocol that is not on this list. The list of protocols must not contain an empty binary. If the server negociates a Next Protocol, it can be accessed using the negotiated_next_protocol/1 method.

server_psk_identity() = psk_identity()
Specifies the server identity hint, which the server presents to the client.

honor_cipher_order() = boolean()
If set to true, use the server preference for cipher selection. If set to false (the default), use the client preference.
sni_hosts() = 
    [hostname(), [server_option() | common_option()]]

If the server receives a SNI (Server Name Indication) from the client matching a host listed in the sni_hosts option, the specific options for that host will override previously specified options. The option sni_fun, and sni_hosts are mutually exclusive.

sni_fun() = function()

If the server receives a SNI (Server Name Indication) from the client, the given function will be called to retrieve [server_option()] for the indicated server. These options will be merged into predefined [server_option()] list. The function should be defined as: fun(ServerName :: string()) -> [server_option()] and can be specified as a fun or as named fun module:function/1 The option sni_fun, and sni_hosts are mutually exclusive.

client_renegotiation() = boolean()

In protocols that support client-initiated renegotiation, the cost of resources of such an operation is higher for the server than the client. This can act as a vector for denial of service attacks. The SSL application already takes measures to counter-act such attempts, but client-initiated renegotiation can be strictly disabled by setting this option to false. The default value is true. Note that disabling renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

honor_cipher_order() = boolean()

If true, use the server's preference for cipher selection. If false (the default), use the client's preference.

honor_ecc_order() = boolean()

If true, use the server's preference for ECC curve selection. If false (the default), use the client's preference.

server_signature_algs() = signature_algs()

The algorithms specified by this option will be the ones accepted by the server in a signature algorithm negotiation, introduced in TLS-1.2. The algorithms will also be offered to the client if a client certificate is requested. For more details see the corresponding client option.

Exports

append_cipher_suites(Deferred, Suites) -> ciphers()

Types:
    Deferred = ciphers() | cipher_filters()
    Suites = ciphers()

Make Deferred suites become the least preferred suites, that is put them at the end of the cipher suite list Suites after removing them from Suites if present. Deferred may be a list of cipher suits or a list of filters in which case the filters are use on Suites to extract the Deferred cipher list.

cipher_suites() -> [old_cipher_suite()] | [string()]
cipher_suites(Type) -> [old_cipher_suite()] | string()

Types:
    Type = erlang | openssl | all

Deprecated in OTP 21, use cipher_suites/2 instead.

cipher_suites(Supported, Version) -> ciphers()

Types:
Supported = default | all | anonymous
Version = protocol_version()

Returns all default or all supported (except anonymous), or all anonymous cipher suites for a TLS version.

cipher_suites(Supported, Version, StringType :: rfc | openssl) ->
[string()]

Types:
  Supported = default | all | anonymous
  Version = protocol_version()

Same as cipher_suites/2 but lists RFC or OpenSSL string names instead of erl_cipher_suite().

eccs() -> NamedCurves
eccs(Version) -> NamedCurves

Types:
  Version = protocol_version()
  NamedCurves = [named_curve()]

Returns a list of supported ECCs. eccs() is equivalent to calling eccs(Protocol) with all supported protocols
and then deduplicating the output.

clear_pem_cache() -> ok

PEM files, used by ssl API-functions, are cached. The cache is regularly checked to see if any cache entries should be
invalidated, however this function provides a way to unconditionally clear the whole cache.

connect(TCPSocket, TLSOptions) ->
  {ok, sslsocket()}
  | {error, reason()}
  | {option_not_a_key_value_tuple, any()}

connect(TCPSocket, TLSOptions, Timeout) ->
  {ok, sslsocket()}
  | {error, reason()}

Types:
  TCPSocket = socket()
  TLSOptions = [tls_client_option()]
  Timeout = timeout()

Upgrades a gen_tcp, or equivalent, connected socket to an TLS socket, that is, performs the client-side TLS
handshake.

Note:

| If the option verify is set to verify_peer the option server_name_indication shall also be specified,   |
| if it is not no Server Name Indication extension will be sent, and public_key:pkix_verify_hostname/2 will be called |
| with the IP-address of the connection as ReferenceID, which is probably not what you want. |

If the option {handshake, hello} is used the handshake is paused after receiving the server hello message and
the success response is {ok, SslSocket, Ext} instead of {ok, SslSocket}. Thereafter the handshake is
continued or canceled by calling handshake_continue/3 or handshake_cancel/1.
If the option `active` is set to `once`, `true` or an integer value, the process owning the `sslsocket` will receive messages of type `active_msgs()`.

```erlang
connect(Host, Port, TLSOptions) ->
    {ok, sslsocket()} |
    {ok, sslsocket(), Ext :: protocol_extensions()} |
    {error, reason()} |
    {option_not_a_key_value_tuple, any()}
connect(Host, Port, TLSOptions, Timeout) ->
    {ok, sslsocket()} |
    {ok, sslsocket(), Ext :: protocol_extensions()} |
    {error, reason()} |
    {option_not_a_key_value_tuple, any()}
```

**Types:**

- `Host = host()`
- `Port = inet:port_number()`
- `TLSOptions = [tls_client_option()]`
- `Timeout = timeout()`

Opens an TLS/DTLS connection to `Host`, `Port`.

When the option `verify` is set to `verify_peer` the check `public_key:pkix_verify_hostname/2` will be performed in addition to the usual x509-path validation checks. If the check fails the error `{bad_cert, hostname_check_failed}` will be propagated to the path validation fun `verify_fun`, where it is possible to do customized checks by using the full possibilities of the `public_key:pkix_verify_hostname/3` API. When the option `server_name_indication` is provided, its value (the DNS name) will be used as `ReferenceID` to `public_key:pkix_verify_hostname/2`. When no `server_name_indication` option is given, the `Host` argument will be used as Server Name Indication extension. The `Host` argument will also be used for the `public_key:pkix_verify_hostname/2` check and if the `Host` argument is an `inet:ip_address()` the `ReferenceID` used for the check will be `{ip, Host}` otherwise `dns_id` will be assumed with a fallback to `ip` if that fails.

**Note:**

According to good practices certificates should not use IP-addresses as "server names". It would be very surprising if this happen outside a closed network.

If the option `{handshake, hello}` is used the handshake is paused after receiving the server hello message and the success response is `{ok, SslSocket, Ext}` instead of `{ok, SslSocket}`. Thereafter the handshake is continued or canceled by calling `handshake_continue/3` or `handshake_cancel/1`.

If the option `active` is set to `once`, `true` or an integer value, the process owning the `sslsocket` will receive messages of type `active_msgs()`.

```erlang
close(SslSocket) -> ok | {error, Reason}
```

**Types:**

- `SslSocket = sslsocket()`
- `Reason = any()`

Closes an TLS/DTLS connection.
close(SslSocket, How) -> ok | \{ok, port()\} | \{error, Reason\}
Types:
  SslSocket = \texttt{sslsocket()}
  How = \texttt{timeout()} | \{NewController :: pid(), timeout()\}
  Reason = any()

Closes or downgrades an TLS connection. In the latter case the transport connection will be handed over to the NewController process after receiving the TLS close alert from the peer. The returned transport socket will have the following options set: \[\{\text{active, false}\}, \{\text{packet, 0}\}, \{\text{mode, binary}\}\]

controlling_process(SslSocket, NewOwner) -> ok | \{error, Reason\}
Types:
  SslSocket = \texttt{sslsocket()}
  NewOwner = pid()
  Reason = any()

Assigns a new controlling process to the SSL socket. A controlling process is the owner of an SSL socket, and receives all messages from the socket.

connection_information(SslSocket) ->
\{ok, Result\} | \{error, reason()\}
Types:
  SslSocket = \texttt{sslsocket()}
  Result = \[\{\text{OptionName, OptionValue}\}\]
  OptionName = atom()
  OptionValue = any()

Returns the most relevant information about the connection, ssl options that are undefined will be filtered out. Note that values that affect the security of the connection will only be returned if explicitly requested by connection_information/2.

\begin{quote}
\textbf{Note:}

The legacy Item = cipher_suite is still supported and returns the cipher suite on its (undocumented) legacy format. It should be replaced by selected_cipher_suite.
\end{quote}

connection_information(SslSocket, Items) ->
\{ok, Result\} | \{error, reason()\}
Types:
  SslSocket = \texttt{sslsocket()}
  Items = \[\text{OptionName}\]
  Result = \[\{\text{OptionName, OptionValue}\}\]
  OptionName = atom()
  OptionValue = any()

Returns the requested information items about the connection, if they are defined.

Note that client_random, server_random and master_secret are values that affect the security of connection. Meaningful atoms, not specified above, are the ssl option names.
filter_cipher_suites(Suites, Filters) -> Ciphers
Types:
  Suites = ciphers()
  Filters = cipher_filters()
  Ciphers = ciphers()
Removes cipher suites if any of the filter functions returns false for any part of the cipher suite. This function also calls default filter functions to make sure the cipher suites are supported by crypto. If no filter function is supplied for some part the default behaviour is fun(Algorithm) -> true.

format_error(Reason :: {error, Reason}) -> string()
Types:
  Reason = any()
Presents the error returned by an SSL function as a printable string.

getopts(SslSocket, OptionNames) ->
  {ok, [gen_tcp:option()]} | {error, reason()}
Types:
  SslSocket = sslsocket()
  OptionNames = [gen_tcp:option_name()]
Gets the values of the specified socket options.

getstat(SslSocket) -> {ok, OptionValues} | {error, inet:posix()}
getstat(SslSocket, Options) ->
  {ok, OptionValues} | {error, inet:posix()}
Types:
  SslSocket = sslsocket()
  Options = [inet:stat_option()]
  OptionValues = [{inet:stat_option(), integer()}]
Gets one or more statistic options for the underlying TCP socket.
See inet:getstat/2 for statistic options description.

handshake(HsSocket) ->
  {ok, SslSocket} |
  {ok, SslSocket, Ext} |
  {error, Reason}
handshake(HsSocket, Timeout) ->
  {ok, SslSocket} |
  {ok, SslSocket, Ext} |
  {error, Reason}
Types:
HsSocket = sslsocket()
Timeout = timeout()
SslSocket = sslsocket()
Ext = protocol_extensions()
Reason = closed | timeout | error_alert()

Performs the SSL/TLS/DTLS server-side handshake.
Returns a new TLS/DTLS socket if the handshake is successful.

If the option active is set to once, true or an integer value, the process owning the sslsocket will receive messages of type active_msgs()

handshake(Socket, Options) ->
  {ok, SslSocket} |
  {ok, SslSocket, Ext} |
  {error, Reason}

handshake(Socket, Options, Timeout) ->
  {ok, SslSocket} |
  {ok, SslSocket, Ext} |
  {error, Reason}

Types:
  Socket = socket() | sslsocket()
  SslSocket = sslsocket()
  Options = [server_option()]
  Timeout = timeout()
  Ext = protocol_extensions()
  Reason = closed | timeout | {options, any()} | error_alert()

If Socket is an ordinary socket(): upgrades a gen_tcp, or equivalent, socket to an SSL socket, that is, performs the SSL/TLS server-side handshake and returns a TLS socket.

Warning:
The Socket shall be in passive mode ({active, false}) before calling this function or else the behavior of this function is undefined.

If Socket is an sslsocket(): provides extra SSL/TLS/DTLS options to those specified in listen/2 and then performs the SSL/TLS/DTLS handshake. Returns a new TLS/DTLS socket if the handshake is successful.

If option (handshake, hello) is specified the handshake is paused after receiving the client hello message and the success response is {ok, SslSocket, Ext} instead of {ok, SslSocket}. Thereafter the handshake is continued or canceled by calling handshake_continue/3 or handshake_cancel/1.

If the option active is set to once, true or an integer value, the process owning the sslsocket will receive messages of type active_msgs()

handshake_cancel(Sslsocket :: #sslsocket{}) -> any()

Cancel the handshake with a fatal USER_CANCELED alert.

handshake_continue(HsSocket, Options) ->
handshake_continue(HsSocket, Options, Timeout) ->
    {ok, SslSocket} | {error, Reason}

Types:
    HsSocket = sslsocket()
    Options = [tls_client_option() | tls_server_option()]
    Timeout = timeout()
    SslSocket = sslsocket()
    Reason = closed | timeout | error_alert()

Continue the SSL/TLS handshake possibly with new, additional or changed options.

listen(Port, Options) -> {ok, ListenSocket} | {error, reason()}

Types:
    Port = inet:port_number()
    Options = [tls_server_option()]
    ListenSocket = sslsocket()

Creates an SSL listen socket.

negotiated_protocol(SslSocket) -> {ok, Protocol} | {error, Reason}

Types:
    SslSocket = sslsocket()
    Protocol = binary()
    Reason = protocol_not_negotiated

Returns the protocol negotiated through ALPN or NPN extensions.

peercert(SslSocket) -> {ok, Cert} | {error, reason()}

Types:
    SslSocket = sslsocket()
    Cert = binary()

The peer certificate is returned as a DER-encoded binary. The certificate can be decoded with public_key:pkix_decode_cert/2

peername(SslSocket) -> {ok, {Address, Port}} | {error, reason()}

Types:
    SslSocket = sslsocket()
    Address = inet:ip_address()
    Port = inet:port_number()

Returns the address and port number of the peer.

prepend_cipher_suites(Preferred, Suites) -> ciphers()

Types:
Preferred = \textit{ciphers()} \mid \textit{cipher\_filters()}
Suites = \textit{ciphers()}

Make \textit{Preferred} suites become the most preferred suites that is put them at the head of the cipher suite list \textit{Suites} after removing them from \textit{Suites} if present. \textit{Preferred} may be a list of cipher suits or a list of filters in which case the filters are use on \textit{Suites} to extract the preferred cipher list.

\texttt{prf(SslSocket, Secret, Label, Seed, WantedLength) \rightarrow \{ok, binary()\} \mid \{error, \textit{reason()\}}}

\textbf{Types:}
- \texttt{SslSocket = \textit{sslsocket()}}
- \texttt{Secret = binary() \mid master\_secret}
- \texttt{Label = binary()}
- \texttt{Seed = [binary() \mid \textit{prf\_random()}]}
- \texttt{WantedLength = integer() \geq 0}

Uses the Pseudo-Random Function (PRF) of a TLS session to generate extra key material. It either takes user-generated values for \texttt{Secret} and \texttt{Seed} or atoms directing it to use a specific value from the session security parameters. Can only be used with TLS/DTLS connections; \texttt{\{error, undefined\}} is returned for SSLv3 connections.

\texttt{recv(SslSocket, Length) \rightarrow \{ok, Data\} \mid \{error, \textit{reason()\}}}
\texttt{recv(SslSocket, Length, Timeout) \rightarrow \{ok, Data\} \mid \{error, \textit{reason()\}}}

\textbf{Types:}
- \texttt{SslSocket = \textit{sslsocket()}}
- \texttt{Length = integer()}
- \texttt{Data = binary() \mid list() \mid \textit{HttpPacket}}
- \texttt{Timeout = timeout()}
- \texttt{HttpPacket = any()}

See the description of \texttt{HttpPacket} in \texttt{erlang:decode\_packet/3} in ERTS.

Receives a packet from a socket in passive mode. A closed socket is indicated by return value \texttt{\{error, closed\}}.

Argument \texttt{Length} is meaningful only when the socket is in mode \texttt{raw} and denotes the number of bytes to read. If \texttt{Length = 0}, all available bytes are returned. If \texttt{Length > 0}, exactly \texttt{Length} bytes are returned, or an error; possibly discarding less than \texttt{Length} bytes of data when the socket gets closed from the other side.

Optional argument \texttt{Timeout} specifies a time-out in milliseconds. The default value is \texttt{infinity}.

\texttt{renegotiate(SslSocket) \rightarrow ok \mid \{error, \textit{reason()\}}}

\textbf{Types:}
- \texttt{SslSocket = \textit{sslsocket()}}

Initiates a new handshake. A notable return value is \texttt{\{error, renegotiation\_rejected\}} indicating that the peer refused to go through with the renegotiation, but the connection is still active using the previously negotiated session.

\texttt{send(SslSocket, Data) \rightarrow ok \mid \{error, \textit{reason()\}}}

\textbf{Types:}
SslSocket = sslsocket()
Data = iodata()

Writes Data to SslSocket.

A notable return value is \{error, closed\} indicating that the socket is closed.

setopts(SslSocket, Options) -> ok | \{error, reason()\}

Types:
  SslSocket = sslsocket()
  Options = [gen_tcp:option()]

Sets options according to Options for socket SslSocket.

shutdown(SslSocket, How) -> ok | \{error, reason()\}

Types:
  SslSocket = sslsocket()
  How = read | write | read_write

Immediately closes a socket in one or two directions.

How == write means closing the socket for writing, reading from it is still possible.

To be able to handle that the peer has done a shutdown on the write side, option \{exit_on_close, false\} is useful.

ssl_accept(SslSocket) -> ok | \{error, Reason\}

ssl_accept(Socket, TimeoutOrOptions) ->
  ok | \{ok, sslsocket()\} | \{error, Reason\}

Types:
  Socket = sslsocket() | socket()
  TimeoutOrOptions = timeout() | [tls_server_option()]
  Reason = timeout | closed | \{options, any()\} | error_alert()

Deprecated in OTP 21, use handshake/[1,2] instead.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
</table>

handshake/[1,2] always returns a new socket.

ssl_accept(Socket, Options, Timeout) ->
  ok | \{ok, sslsocket()\} | \{error, Reason\}

Types:
  Socket = sslsocket() | socket()
  Options = [tls_server_option()]
  Timeout = timeout()
  Reason = timeout | closed | \{options, any()\} | error_alert()

Deprecated in OTP 21, use handshake/[2,3] instead.
Note:
handshake/[2,3] always returns a new socket.

sockname(SslSocket) -> {ok, {Address, Port}} | {error, reason()}
Types:
  SslSocket = sslsocket()
  Address = inet:ip_address()
  Port = inet:port_number()
Returns the local address and port number of socket SslSocket.

start() -> ok | {error, reason()}
start(Type) -> ok | {error, Reason}
Starts the SSL application. Default type is temporary.

stop() -> ok
Stops the SSL application.

str_to_suite(CipherSuiteName) -> erl_cipher_suite()
Types:
  CipherSuiteName =
    string() |
    {error, {not_recognized, CipherSuiteName :: string()}}
Converts an RFC or OpenSSL name string to an erl_cipher_suite().
Returns an error if the cipher suite is not supported or the name is not a valid cipher suite name.

suite_to_openssl_str(CipherSuite) -> string()
Types:
  CipherSuite = erl_cipher_suite()
Converts erl_cipher_suite() to OpenSSL name string.
PRE TLS-1.3 these names differ for RFC names

suite_to_str(CipherSuite) -> string()
Types:
  CipherSuite = erl_cipher_suite()
Converts erl_cipher_suite() to RFC name string.

transport_accept(ListenSocket) ->
  {ok, SslSocket} | {error, reason()}
transport_accept(ListenSocket, Timeout) ->
  {ok, SslSocket} | {error, reason()}
Types:
ListenSocket = sslsocket()
Timeout = timeout()
SslSocket = sslsocket()

Accepts an incoming connection request on a listen socket. ListenSocket must be a socket returned from listen/2. The socket returned is to be passed to handshake/[2,3] to complete handshaking, that is, establishing the SSL/TLS/DTLS connection.

**Warning:**

Most API functions require that the TLS/DTLS connection is established to work as expected.

The accepted socket inherits the options set for ListenSocket in listen/2.

The default value for Timeout is infinity. If Timeout is specified and no connection is accepted within the given time, {error, timeout} is returned.

versions() -> [VersionInfo]
Types:

VersionInfo =
    {ssl_app, string()} |
    {supported | available, [tls_version()]} |
    {supported_dtls | available_dtls, [dtls_version()]} |

Returns version information relevant for the SSL application.

app_vsn
The application version of the SSL application.

supported
SSL/TLS versions supported by default. Overridden by a version option on connect/[2,3,4], listen/2, and ssl_accept/[1,2,3]. For the negotiated SSL/TLS version, see connection_information/1.

supported_dtls
DTLS versions supported by default. Overridden by a version option on connect/[2,3,4], listen/2, and ssl_accept/[1,2,3]. For the negotiated DTLS version, see connection_information/1.

available
All SSL/TLS versions supported by the SSL application. TLS 1.2 requires sufficient support from the Crypto application.

available_dtls
All DTLS versions supported by the SSL application. DTLS 1.2 requires sufficient support from the Crypto application.

SEE ALSO

inet(3) and gen_tcp(3) gen_udp(3)
ssl_crl_cache

Erlang module

Implements an internal CRL (Certificate Revocation List) cache. In addition to implementing the `ssl_crl_cache_api` behaviour the following functions are available.

**DATA TYPES**

**crl_src()** =
- `{file, file:filename()}` | `{der, public_key:der_encoded()}

**uri()** = `uri_string:uri_string()`

**Exports**

`delete(Entries) -> ok | {error, Reason}`

Types:
- `Entries` = `crl_src()`
- `Reason` = `crl_reason()`

Delete CRLs from the ssl applications local cache.

`insert(CRLSrc) -> ok | {error, Reason}`
- `CRLSrc` = `crl_src()`

`insert(URI, CRLSrc) -> ok | {error, Reason}`
- `URI` = `uri()`
- `Reason` = `term()`

Insert CRLs, available to fetch on DER format from URI, into the ssl applications local cache.
ssl_crl_cache_api

Erlang module

When SSL/TLS performs certificate path validation according to RFC 5280 it should also perform CRL validation checks. To enable the CRL checks the application needs access to CRLs. A database of CRLs can be set up in many different ways. This module provides the behavior of the API needed to integrate an arbitrary CRL cache with the erlang ssl application. It is also used by the application itself to provide a simple default implementation of a CRL cache.

Data Types

crl_cache_ref() = any()
Reference to the CRL cache.
dist_point() = #'DistributionPoint'{}
For description see X509 certificates records

Exports

fresh_crl(DistributionPoint, CRL) -> FreshCRL
Types:
    DistributionPoint = dist_point()
    CRL = [public_key:der_encoded()]
    FreshCRL = [public_key:der_encoded()]
fun fresh_crl/2 will be used as input option update_crl to public_key:pkix_crls_validate/3

lookup(DistributionPoint, Issuer, DbHandle) -> not_available | CRLs
lookup(DistributionPoint, DbHandle) -> not_available | CRLs
Types:
    DistributionPoint = dist_point()
    Issuer = public_key:issuer_name()
    DbHandle = crl_cache_ref()
    CRLs = [public_key:der_encoded()]
Lookup the CRLs belonging to the distribution point Distributionpoint. This function may choose to only look in the cache or to follow distribution point links depending on how the cache is administrated.
The Issuer argument contains the issuer name of the certificate to be checked. Normally the returned CRL should be issued by this issuer, except if the cRLEissuer field of DistributionPoint has a value, in which case that value should be used instead.
In an earlier version of this API, the lookup function received two arguments, omitting Issuer. For compatibility, this is still supported: if there is no lookup/3 function in the callback module, lookup/2 is called instead.

select(Issuer, DbHandle) -> CRLs
Types:
    Issuer = public_key:issuer_name()
    DbHandle = cache_ref()
Select the CRLs in the cache that are issued by Issuer
ssl_session_cache_api

Erlang module

Defines the API for the TLS session cache so that the data storage scheme can be replaced by defining a new callback module implementing this API.

Data Types

session_cache_ref() = any()
session_cache_key() = {partial_key(), ssl:session_id()}
A key to an entry in the session cache.
partial_key()
The opaque part of the key. Does not need to be handled by the callback.
session()
The session data that is stored for each session.

Exports

delete(Cache, Key) -> _
Types:
    Cache = session_cache_ref()
    Key = session_cache_key()
Deletes a cache entry. Is only called from the cache handling process.
foldl(Fun, Acc0, Cache) -> Acc
Types:
    Fun = fun()
    Acc0 = Acc = term()
    Cache = session_cache_ref()
Calls Fun(Elem, AccIn) on successive elements of the cache, starting with AccIn == Acc0. Fun/2 must return a new accumulator, which is passed to the next call. The function returns the final value of the accumulator. Acc0 is returned if the cache is empty.
init(Args) -> Cache
Types:
    Cache = session_cache_ref()
    Args = proplists:proplist()
Includes property {role, client | server}. Currently this is the only predefined property, there can also be user-defined properties. See also application environment variable session_cb_init_args.
Performs possible initializations of the cache and returns a reference to it that is used as parameter to the other API functions. Is called by the cache handling processes init function, hence putting the same requirements on it as a normal process init function. This function is called twice when starting the SSL application, once with the role client and once with the role server, as the SSL application must be prepared to take on both roles.
ssl_session_cache_api

lookup(Cache, Key) -> Entry
Types:
  Cache = session_cache_ref()
  Key = session_cache_key()
  Session = session() | undefined
Looks up a cache entry. Is to be callable from any process.

select_session(Cache, PartialKey) -> [Session]
Types:
  Cache = session_cache_ref()
  PartialKey = partial_key()
  Session = session()
Selects sessions that can be reused. Is to be callable from any process.

size(Cache) -> integer()
Types:
  Cache = session_cache_ref()
Returns the number of sessions in the cache. If size exceeds the maximum number of sessions, the current cache entries will be invalidated regardless of their remaining lifetime. Is to be callable from any process.

terminate(Cache) -> _
Types:
  Cache = session_cache_ref()
  As returned by init/0
Takes care of possible cleanup that is needed when the cache handling process terminates.

update(Cache, Key, Session) -> _
Types:
  Cache = session_cache_ref()
  Key = session_cache_key()
  Session = session()
Caches a new session or updates an already cached one. Is only called from the cache handling process.