

Secure Socket Layer

Copyright © 1999-2023 Ericsson AB. All Rights Reserved.

Secure Socket Layer 10.8.6

January 17, 2023

Copyright © 1999-2023 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

January 17, 2023

1 SSL User's Guide

The SSL application implements Transport Layer Security (TLS), formerly known as the Secure Socket Layer (SSL), that is it 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 TLS/DTLS.

1.2 TLS/DTLS and TLS Predecessor, SSL

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

By default 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 a 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 Authenticity 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 - PRE TLS-1.3

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 when session database exceeds its limit or 24 hours after being saved (RFC max lifetime recommendation). 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. See also Session Reuse Pre TLS-1.3

1.2.6 TLS-1.3 session tickets

In TLS 1.3 the session reuse is replaced by a new session tickets mechanism based on the pre shared key concept. This mechanism also obsoletes the session tickets from RFC5077, not implemented by this application. See also Session Tickets and Session Resumption in TLS-1.3

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, 'tlsv1.3')` . The available cipher suites for a connection depend on the TLS version and pre TLS-1.3 also on the certificate. To see the default cipher suite list change `all` to `default`. Note that TLS 1.3 and previous versions do not have any cipher suites in common, for listing cipher suites for a specific version use `ssl:cipher_suites(exclusive, 'tlsv1.3')` . Specific cipher suites that you want your connection to use can also be specified. Default is to use the strongest available.

The following sections shows small examples 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 to the user except for the purpose of pattern matching.

Note:

Note that client certificate verification is optional for the server and needs additional conguration on both sides to work. The Certificate and keys, in the examples, are provided using the certs_keys option introduced in OTP-25.

1.3.1 Basic Client

```
 1 > ssl:start(), ssl:connect("google.com", 443, [{verify, verify_peer},
    {cacerts, public_key:cacerts_get()}]).
   {ok,{sslsocket, [...]}}
```

1.3.2 Basic Connection

Step 1: Start the server side:

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

Step 2: with alternative certificates, in this example the EDDSA certificate will be preferred if TLS-1.3 is negotiated and the RSA certificate will always be used for TLS-1.2 as it does not support the EDDSA algorithm:

```
2 server> {ok, ListenSocket} =
ssl:listen(9999, [{certs_keys, [#{certfile => "eddsacert.pem",
                                  keyfile => "eddsakey.pem"},
           #{certfile => "rsacert.pem",
                                  keyfile => "rsakey.pem",
             password => "foobar"}
          ]},{reuseaddr, true}]).
{ok,{sslsocket, [...]}}
```

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

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

Note:

ssl:transport_accept/1 and ssl:handshake/2 are separate functions so that the handshake part can be called in a new erlang process dedicated to handling the connection

Step 4: Start the client side:

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

Be sure to configure trusted certificates to use for server certificate verification.

```
2 client> {ok, Socket} = ssl:connect("localhost", 9999,
      [{verify, verify_peer},
      {cacertfile, "cacerts.pem"}, {active, once}], infinity).
{ok,{sslsocket, [...]}}
```

Step 5: Do the TLS handshake:

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

Note:

A real server should use ssl:handshake/2 that has a timeout to avoid DoS attacks. In the example the timeout defaults to infinty.

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 sent on the server side is recived by the client side:

```
3 client> flush().
Shell got {ssl,{sslsocket,[...]},"foo"}
ok
```

1.3.3 Upgrade Example - TLS only

Upgrading a a TCP/IP connection to a TLS connections is mostly used when there is a desire have unencrypted communication first and then later secure the communication channel by using TLS. Note that the client and server need to agree to do the upgrade in the protocol doing the communication. This is concept is often referenced as `STARTLS` and used in many protocols such as `SMTP`, `FTPS` and `HTTPS` via a proxy.

Warning:

Maximum security recommendations are however moving away from such solutions.

To upgrade to a TLS connection:

Step 1: Start the server side:

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

Step 2: Create a normal TCP listen socket and ensure `active` is set to `false` and not set to any active mode otherwise TLS handshake messages can be delivered to the wrong process.

```
2 server> {ok, ListenSocket} = gen_tcp:listen(9999, [{reuseaddr, true},
  {active, false}]).
  {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: Do the TLS handshake:

```
4 server> {ok, TLSSocket} = ssl:handshake(Socket, [{verify, verify_peer},
  {fail_if_no_peer_cert, true},
  {cacertfile, "cacerts.pem"},
  {certs_keys, [#{certfile => "cert.pem", keyfile => "key.pem"}]}]).
  {ok,{sslsocket,[...]}}
```

Step 6: Upgrade to a TLS connection. The client and server must agree upon the upgrade. The server must be prepared to be a TLS server before the client can do a successful connect.

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

Step 7: Send a message over TLS:

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

Step 8: Set `active once` on the TLS socket:

```
5 server> ssl:setopts(TLSSocket, [{active, once}]).
      ok
```

Step 9: Flush the shell message queue to see that the message sent on the client side is recived by the server side:

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

1.3.4 Customizing cipher suites

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

```
1>  Default = ssl:cipher_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 key exchange (removed from default in 21)

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

Pick just a few suites

```
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_128_cbc,key_exchange => ecdh_ecdsa,mac => sha,
    prf => default_prf}]
```

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

```
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 => ecdh_ecdsa,mac => sha,
    prf => default_prf},
  #{cipher => aes_256_cbc,key_exchange => ecdhe_ecdsa,
    mac => sha384,prf => sha384}, ...]
```

1.3.5 Using an Engine Stored Key

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

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

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

```
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"},
     {certs_keys, [#{certfile => "cert.pem", key => PrivKey}]}
     ], infinity).
```

See also crypto documentation

1.3.6 Session Reuse pre TLS 1.3

Clients can request to reuse a session established by a previous full handshake between that client and server by sending the id of the session in the initial handshake message. The server may or may not agree to reuse it. If agreed the server will send back the id and if not it will send a new id. The ssl application has several options for handling session reuse.

On the client side the ssl application will save session data to try to automate session reuse on behalf of the client processes on the Erlang node. Note that only verified sessions will be saved for security reasons, that is session resumption relies on the certificate validation to have been run in the original handshake. To minimize memory consumption only unique sessions will be saved unless the special `save` value is specified for the following option `{reuse_sessions, boolean() | save}` in which case a full handshake will be performed and that specific session will have been saved before the handshake returns. The session id and even an opaque binary containing the session data can be retrieved using `ssl:connection_information/1` function. A saved session (guaranteed by the save option) can be explicitly reused using `{reuse_session, SessionId}`. Also it is possible for the client to reuse a session that is not saved by the ssl application using `{reuse_session, {SessionId, SessionData}}`.

Note:

When using explicit session reuse, it is up to the client to make sure that the session being reused is for the correct server and has been verified.

Here follows a client side example, divide into several steps for readability.

Step 1 - Automated Session Reuse

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

2> {ok, C1} = ssl:connect("localhost", 9999, [{verify, verify_peer},
           {versions, ['tlsv1.2']},
           {cacertfile, "cacerts.pem"}]).
{ok,{sslsocket,{gen_tcp,#Port<0.7>,tls_connection,undefined}, ...}}

3> ssl:connection_information(C1, [session_id]).
{ok,[{session_id,<<95,32,43,22,35,63,249,22,26,36,106,
                   152,49,52,124,56,130,192,137,161,
                   146,145,164,232,...>>}]}

%% Reuse session if possible, note that if C2 is really fast the session
%% data might not be available for reuse.
4> {ok, C2} = ssl:connect("localhost", 9999, [{verify, verify_peer},
           {versions, ['tlsv1.2']},
           {cacertfile, "cacerts.pem"},
           {reuse_sessions, true}]).
{ok,{sslsocket,{gen_tcp,#Port<0.8>,tls_connection,undefined}, ...]}}

%% C2 got same session ID as client one, session was automatically reused.
5> ssl:connection_information(C2, [session_id]).
{ok,[{session_id,<<95,32,43,22,35,63,249,22,26,36,106,
                   152,49,52,124,56,130,192,137,161,
                   146,145,164,232,...>>}]}
```

Step 2- Using `save` Option

```
%% We want save this particular session for
%% reuse although it has the same basis as C1
6> {ok, C3} = ssl:connect("localhost", 9999, [{verify, verify_peer},
           {versions, ['tlsv1.2']},
           {cacertfile, "cacerts.pem"},
           {reuse_sessions, save}]).
{ok,{sslsocket,{gen_tcp,#Port<0.9>,tls_connection,undefined}, ...]}}

%% A full handshake is performed and we get a new session ID
7> {ok, [{session_id, ID}]} = ssl:connection_information(C3, [session_id]).
{ok,[{session_id,<<91,84,27,151,183,39,84,90,143,141,
                   121,190,66,192,10,1,27,192,33,95,78,
                   8,34,180,...>>}]}

%% Use automatic session reuse
8> {ok, C4} = ssl:connect("localhost", 9999, [{verify, verify_peer},
           {versions, ['tlsv1.2']},
           {cacertfile, "cacerts.pem"},
           {reuse_sessions, true}]).
{ok,{sslsocket,{gen_tcp,#Port<0.10>,tls_connection,
                        undefined}, ...]}}

%% The "saved" one happened to be selected, but this is not a guarantee
9> ssl:connection_information(C4, [session_id]).
{ok,[{session_id,<<91,84,27,151,183,39,84,90,143,141,
                   121,190,66,192,10,1,27,192,33,95,78,
                   8,34,180,...>>}]}

%% Make sure to reuse the "saved" session
10> {ok, C5} = ssl:connect("localhost", 9999, [{verify, verify_peer},
            {versions, ['tlsv1.2']},
            {cacertfile, "cacerts.pem"},
            {reuse_session, ID}]).
{ok,{sslsocket,{gen_tcp,#Port<0.11>,tls_connection,
                        undefined}, ...]}}

11> ssl:connection_information(C5, [session_id]).
{ok,[{session_id,<<91,84,27,151,183,39,84,90,143,141,
                   121,190,66,192,10,1,27,192,33,95,78,
                   8,34,180,...>>}]}
```

Step 3 - Explicit Session Reuse

```
%% Perform a full handshake and the session will not be saved for reuse
12> {ok, C9} =
ssl:connect("localhost", 9999, [{verify, verify_peer},
    {versions, ['tlsv1.2']},
                {cacertfile, "cacerts.pem"},
                             {reuse_sessions, false},
                         {server_name_indication, disable}]).
{ok,{sslsocket,{gen_tcp,#Port<0.14>,tls_connection, ...}}

%% Fetch session ID and data for C9 connection
12> {ok, [{session_id, ID1}, {session_data, SessData}]} =
 ssl:connection_information(C9, [session_id, session_data]).
{ok,[{session_id,<<9,233,4,54,170,88,170,180,17,96,202,
                   85,85,99,119,47,9,68,195,50,120,52,
                   130,239,...>>},
     {session_data,<<131,104,13,100,0,7,115,101,115,115,105,
                     111,110,109,0,0,0,32,9,233,4,54,170,...>>}]}

%% Explicitly reuse the session from C9
13> {ok, C10} = ssl:connect("localhost", 9999, [{verify, verify_peer},
      {versions, ['tlsv1.2']},
      {cacertfile, "cacerts.pem"},
      {reuse_session, {ID1, SessData}}]).
{ok,{sslsocket,{gen_tcp,#Port<0.15>,tls_connection,
                        undefined}, ...}}

14> ssl:connection_information(C10, [session_id]).
{ok,[{session_id,<<9,233,4,54,170,88,170,180,17,96,202,
                   85,85,99,119,47,9,68,195,50,120,52,
                   130,239,...>>}]}
```

Step 4 - Not Possible to Reuse Explicit Session by ID Only

```
%% Try to reuse the session from C9 using only the id
15> {ok, E} = ssl:connect("localhost", 9999, [{verify, verify_peer},
                  {versions, ['tlsv1.2']},
                  {cacertfile, "cacerts.pem"},
           {reuse_session, ID1}]).
{ok,{sslsocket,{gen_tcp,#Port<0.18>,tls_connection,
                        undefined}, ...}}

%% This will fail (as it is not saved for reuse)
%% and a full handshake will be performed, we get a new id.
16>  ssl:connection_information(E, [session_id]).
{ok,[{session_id,<<87,46,43,126,175,68,160,153,37,29,
                   196,240,65,160,254,88,65,224,18,63,
                   18,17,174,39,...>>}]}
```

On the server side the the `{reuse_sessions, boolean()}` option determines if the server will save session data and allow session reuse or not. This can be further customized by the option `{reuse_session, fun()}` that may introduce a local policy for session reuse.

1.3.7 Session Tickets and Session Resumption in TLS 1.3

TLS 1.3 introduces a new secure way of resuming sessions by using session tickets. A session ticket is an opaque data structure that is sent in the pre_shared_key extension of a ClientHello, when a client attempts to resume a session with keying material from a previous successful handshake.

Session tickets can be stateful or stateless. A stateful session ticket is a database reference (session ticket store) and used with stateful servers, while a stateless ticket is a self-encrypted and self-authenticated data structure with cryptographic keying material and state data, enabling session resumption with stateless servers.

The choice between stateful or stateless depends on the server requirements as the session tickets are opaque for the clients. Generally, stateful tickets are smaller and the server can guarantee that tickets are only used once. Stateless tickets contain additional data, require less storage on the server side, but they offer different guarantees against anti-replay. See also Anti-Replay Protection in TLS 1.3

Session tickets are sent by servers on newly established TLS connections. The number of tickets sent and their lifetime are configurable by application variables. See also SSL's configuration.

Session tickets are protected by application traffic keys, and in stateless tickets, the opaque data structure itself is self-encrypted.

An example with automatic and manual session resumption:

```
      {ok, _} = application:ensure_all_started(ssl).
      LOpts = [{certs_keys, [#{certfile => "cert.pem",
                               keyfile => "key.pem"}]},
               {versions, ['tlsv1.2','tlsv1.3']},
               {session_tickets, stateless}].
      {ok, LSock} = ssl:listen(8001, LOpts).
      {ok, CSock} = ssl:transport_accept(LSock).
```

Step 2 (client): Start the client and connect to server:

```
      {ok, _} = application:ensure_all_started(ssl).
      COpts = [{cacertfile, "cert.pem"},
               {versions, ['tlsv1.2','tlsv1.3']},
               {log_level, debug},
               {session_tickets, auto}].
      ssl:connect("localhost", 8001, COpts).
```

Step 3 (server): Start the TLS handshake:

```
      ssl:handshake(CSock).
```

A connection is established using a full handshake. Below is a summary of the exchanged messages:

```
      >>> TLS 1.3 Handshake, ClientHello ...
      <<< TLS 1.3 Handshake, ServerHello ...
      <<< Handshake, EncryptedExtensions ...
      <<< Handshake, Certificate ...
      <<< Handshake, CertificateVerify ...
      <<< Handshake, Finished ...
      >>> Handshake, Finished ...
      <<< Post-Handshake, NewSessionTicket ...
```

At this point the client has stored the received session tickets and ready to use them when establishing new connections to the same server.

Step 4 (server): Accept a new connection on the server:

```
      {ok, CSock2} = ssl:transport_accept(LSock).
```

Step 5 (client): Make a new connection:

```
      ssl:connect("localhost", 8001, COpts).
```

Step 6 (server): Start the handshake:

```
      ssl:handshake(CSock2).
```

The second connection is a session resumption using keying material from the previous handshake:

```
>>> TLS 1.3 Handshake, ClientHello ...
<<< TLS 1.3 Handshake, ServerHello ...
<<< Handshake, EncryptedExtensions ...
<<< Handshake, Finished ...
>>> Handshake, Finished ...
<<< Post-Handshake, NewSessionTicket ...
```

Manual handling of session tickets is also supported. In manual mode, it is the responsibility of the client to handle received session tickets.

Step 7 (server): Accept a new connection on the server:

```
{ok, CSock3} = ssl:transport_accept(LSock).
```

Step 8 (client): Make a new connection to server:

```
{ok, _} = application:ensure_all_started(ssl).
COpts2 = [{cacertfile, "cacerts.pem"},
          {versions, ['tlsv1.2','tlsv1.3']},
          {log_level, debug},
          {session_tickets, manual}].
ssl:connect("localhost", 8001, COpts).
```

Step 9 (server): Start the handshake:

```
ssl:handshake(CSock3).
```

After the handshake is performed, the user process receivess messages with the tickets sent by the server.

Step 10 (client): Receive a new session ticket:

```
Ticket = receive {ssl, session_ticket, {_, TicketData}} -> TicketData end.
```

Step 11 (server): Accept a new connection on the server:

```
{ok, CSock4} = ssl:transport_accept(LSock).
```

Step 12 (client): Initiate a new connection to the server with the session ticket received in Step 10:

```
{ok, _} = application:ensure_all_started(ssl).
COpts2 = [{cacertfile, "cert.pem"},
          {versions, ['tlsv1.2','tlsv1.3']},
          {log_level, debug},
          {session_tickets, manual},
          {use_ticket, [Ticket]}].
ssl:connect("localhost", 8001, COpts).
```

Step 13 (server): Start the handshake:

```
ssl:handshake(CSock3).
```

1.3.8 Early Data in TLS 1.3

TLS 1.3 allows clients to send data on the first flight if the endpoints have a shared crypographic secret (pre-shared key). This means that clients can send early data if they have a valid session ticket received in a previous successful handshake. For more information about session resumption see Session Tickets and Session Resumption in TLS 1.3.

The security properties of Early Data are weaker than other kinds of TLS data. This data is not forward secret, and it is vulnerable to replay attacks. For available mitigation strategies see Anti-Replay Protection in TLS 1.3.

In normal operation, clients will not know which, if any, of the available mitigation strategies servers actually implement, and hence must only send early data which they deem safe to be replayed. For example, idempotent HTTP

operations, such as HEAD and GET, can usually be regarded as safe but even they can be exploited by a large number of replays causing resource limit exhaustion and other similar problems.

An example of sending early data with automatic and manual session ticket handling:

Warning:

The Early Data feature is experimental in this version of OTP.

Server (with NSS key logging)

```
   early_data_server() ->
       application:load(ssl),
       {ok, _} = application:ensure_all_started(ssl),
       Port = 11029,
       LOpts = [{certs_keys, [#{certfile => "cert.pem", keyfile => "key.pem"}]},
       {reuseaddr, true},
       {versions, ['tlsv1.2','tlsv1.3']},
       {session_tickets, stateless},
       {early_data, enabled},
       {keep_secrets, true} %% Enable NSS key log (debug option)
       ],
       {ok, LSock} = ssl:listen(Port, LOpts),
       %% Accept first connection
       {ok, CSock0} = ssl:transport_accept(LSock),
       {ok, _} = ssl:handshake(CSock0),
       %% Accept second connection
       {ok, CSock1} = ssl:transport_accept(LSock),
       {ok, Sock} = ssl:handshake(CSock1),
       Sock.
```

Exporting the secrets (optional)

```
   {ok, [{keylog, KeylogItems}]} = ssl:connection_information(Sock, [keylog]).
   file:write_file("key.log", [[KeylogItem,$\n] || KeylogItem <- KeylogItems]).
```

Client (automatic ticket handling):

```
   early_data_auto() ->
       %% First handshake 1-RTT - get session tickets
application:load(ssl),
{ok, _} = application:ensure_all_started(ssl),
Port = 11029,
Data = <<"HEAD / HTTP/1.1\r\nHost: \r\nConnection: close\r\n">>,
COpts0 = [{cacertfile, "cacerts.pem"},
          {versions, ['tlsv1.2', 'tlsv1.3']},
          {session_tickets, auto}],
       {ok, Sock0} = ssl:connect("localhost", Port, COpts0),

       %% Wait for session tickets
timer:sleep(500),
%% Close socket if server cannot handle multiple
       %% connections e.g. openssl s_server
ssl:close(Sock0),

       %% Second handshake 0-RTT
COpts1 = [{cacertfile,  "cacerts.pem"},
          {versions, ['tlsv1.2', 'tlsv1.3']},
   {session_tickets, auto},
   {early_data, Data}],
       {ok, Sock} = ssl:connect("localhost", Port, COpts1),
Sock.
```

Client (manual ticket handling):

```
   early_data_manual() ->
       %% First handshake 1-RTT - get session tickets
application:load(ssl),
{ok, _} = application:ensure_all_started(ssl),
Port = 11029,
Data = <<"HEAD / HTTP/1.1\r\nHost: \r\nConnection: close\r\n\r\n">>,
COpts0 = [{cacertfile, "cacerts.pem"},
          {versions, ['tlsv1.2', 'tlsv1.3']},
          {session_tickets, manual}],
       {ok, Sock0} = ssl:connect("localhost", Port, COpts0),

       %% Wait for session tickets
Ticket =
    receive
        {ssl, session_ticket, Ticket0} ->
     Ticket0
           end,

      %% Close socket if server cannot handle multiple connections
      %% e.g. openssl s_server
      ssl:close(Sock0),

      %% Second handshake 0-RTT
      COpts1 = [{cacertfile, "cacerts.pem"},
                {versions, ['tlsv1.2', 'tlsv1.3']},
  {session_tickets, manual},
  {use_ticket, [Ticket]},
  {early_data, Data}],
      {ok, Sock} = ssl:connect("localhost", Port, COpts1),
      Sock.
```

1.3.9 Anti-Replay Protection in TLS 1.3

The TLS 1.3 protocol does not provide inherent protection for replay of 0-RTT data but describes mechanisms that SHOULD be implemented by compliant server implementations. The implementation of TLS 1.3 in the SSL application employs all standard methods to prevent potential threats.

Single-use tickets

This mechanism is available with stateful session tickets. Session tickets can only be used once, subsequent use of the same ticket results in a full handshake. Stateful servers enforce this rule by maintaining a database of outstanding valid tickets.

Client Hello Recording

This mechanism is available with stateless session tickets. The server records a unique value derived from the ClientHello (PSK binder) in a given time window. The ticket's age is verified by using both the "obsfuscated_ticket_age" and an additional timestamp encrypted in the ticket data. As the used datastore allows false positives, apparent replays will be answered by doing a full 1-RTT handshake.

Freshness Checks

This mechanism is available with the stateless session tickets. As the ticket data has an embedded timestamp, the server can determine if a ClientHello was sent reasonably recently and accept the 0-RTT handshake, otherwise if falls back to a full 1-RTT handshake. This mechanism is tightly coupled with the previous one, it prevents storing an unlimited number of ClientHellos.

The current implementation uses a pair of Bloom filters to implement the last two mechanisms. Bloom filters are fast, memory-efficient, probabilistic data structures that can tell if an element may be in a set or if it is definitely not in the set.

If the option anti_replay is defined in the server, a pair of Bloom filters (**current** and **old**) are used to record incoming ClientHello messages (it is the unique binder value that is actually stored). The **current** Bloom filter is used for `WindowSize` seconds to store new elements. At the end of the time window the Bloom filters are rotated (the **current** Bloom filter becomes the **old** and an empty Bloom filter is set as **current**.

The Anti-Replay protection feature in stateless servers executes in the following steps when a new ClientHello is received:

- Reported ticket age (obfuscated ticket age) shall be less than ticket lifetime.
- Actual ticket age shall be less than the ticket lifetime (stateless session tickets contain the servers timestamp when the ticket was issued).
- Ticket shall be used within specified time window (freshness checks).
- If all above checks passed both **current** and **old** Bloom filters are checked to detect if binder was already seen. Being a probabilistic data structure, false positives can occur and they trigger a full handshake.
- If the binder is not seen, the binder is validated. If the binder is valid, the server proceeds with the 0-RTT handshake.

1.3.10 Using DTLS

Using DTLS has basically the same API as TLS. You need to add the option {protocol, dtls} to the connect and listen functions. For example

```
 client> {ok, Socket} = ssl:connect("localhost", 9999, [{protocol, dtls},
{verify, verify_peer},{cacertfile, "cacerts.pem"}], infinity).
{ok,{sslsocket, [...]}}
```

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:

```
      {release, {"OTP  APN 181 01","R15A"}, {erts, "5.9"},
      [{kernel,"2.15"},
      {stdlib,"1.18"},
      {crypto, "2.0.3"},
      {public_key, "0.12"},
      {asn1, "4.0"},
      {ssl, "5.0"}
      ]}.
```

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:

```
   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:

```
$ 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 SSL `ebin` 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 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:

```
$ 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 TLS Options

The 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 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 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"`:

```
[{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:handshake/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 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 TLS distribution adds when creating a socket.

The simplest 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 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 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 TLS (Legacy)

A convenient way to specify arguments to Erlang is to use environment variable ERL_FLAGS. All the flags needed to use the 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):

```
$ ERL_FLAGS="-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"
$ export ERL_FLAGS
$ erl -sname ssl_test
Erlang (BEAM) emulator version 5.0 [source]

Eshell V5.0  (abort with ^G)
(ssl_test@myhost)1> init:get_arguments().
[{root,["/usr/local/erlang"]},
 {progname,["erl "]},
 {sname,["ssl_test"]},
 {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"]
 {home,["/home/me"]}]
```

The `init:get_arguments()` call verifies that the correct arguments are supplied to the emulator.

1.4.6 Using TLS distribution over IPv6

It is possible to use 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:

```
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet6_tls
  -ssl_dist_optfile "/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 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.

1.5.3 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 at all. (OTP 23)

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 support for TLS 1.3. 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: all cipher suites are supported
- Signature Algorithms: All algorithms form RFC 8446
- Certificates: RSA, ECDSA and EDDSA keys

Other notable features:

- PSK and session resumption is supported (stateful and stateless tickets)
- Anti-replay protection using Bloom-filters with stateless tickets
- Early data and 0-RTT is supported
- Key and Initialization Vector Update is 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)

Section	Feature	State	Since
1.3. Updates Affecting TLS 1.2		**C**	**24.1**
	Version downgrade protection mechanism	**C**	**22**
	RSASSA-PSS signature schemes	**C**	**24.1**
	supported_versions (ClientHello) extension	**C**	**22**
	signature_algorithms_cert extension	**C**	**24.1**
2. Protocol Overview		**PC**	**22**
	(EC)DHE	**C**	**22**
	PSK-only	**NC**	
	PSK with (EC)DHE	**C**	**22.2**
2.1. Incorrect DHE share	HelloRetryRequest	**C**	**22**
2.2. Resumption and Pre-Shared Key (PSK)		**C**	**22.2**
2.3. 0-RTT Data		**PC**	**23.3**

4.1.1. Cryptographic Negotiation		**C**	**22.2**
	supported_groups extension	**C**	**22**
	signature_algorithms extension	**C**	**22**
	pre_shared_key extension	**C**	**22.2**
4.1.2. Client Hello	**Client**	**PC**	**22.1**
	server_name (RFC6066)	**C**	**23.2**
	max_fragment_length (RFC6066)	**C**	**23.0**
	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)	**C**	**22.2**

	psk_key_exchange_modes (RFC8446)	**C**	**22.2**
	early_data (RFC8446)	**C**	**23.3**
	cookie (RFC8446)	**C**	**23.1**
	supported_versions (RFC8446)	**C**	**22.1**
	certificate_authorities (RFC8446)	**C**	24.3
	oid_filters (RFC8446)	**NC**	
	post_handshake_auth (RFC8446)	**NC**	
	signature_algorithms_cert (RFC8446)	**C**	**22.1**
	Server	**PC**	**22**
	server_name (RFC6066)	**C**	**23.2**
	max_fragment_length (RFC6066)	**C**	**23.0**
	status_request (RFC6066)	**NC**	
	supported_groups (RFC7919)	**C**	**22**
	signature_algorithms (RFC8446)	**C**	**22**
	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**
	pre_shared_key (RFC8446)	**C**	**22.2**
	psk_key_exchange_modes (RFC8446)	**C**	**22.2**
	early_data (RFC8446)	**C**	**23.3**
	cookie (RFC8446)	**C**	**23.1**
	supported_versions (RFC8446)	**C**	**22**
	oid_filters (RFC8446)	**NC**	
	post_handshake_auth (RFC8446)	**NC**	
	signature_algorithms_cert (RFC8446)	**C**	**22**
4.1.3. Server Hello	**Client**	**C**	**22.2**
	Version downgrade protection	**C**	**22.1**
	key_share (RFC8446)	**C**	**22.1**
	pre_shared_key (RFC8446)	**C**	**22.2**
	supported_versions (RFC8446)	**C**	**22.1**
	Server	**C**	**22.2**
	Version downgrade protection	**C**	**22**
	key_share (RFC8446)	**C**	**22**

	pre_shared_key (RFC8446)	**C**	**22.2**
	supported_versions (RFC8446)	**C**	**22**
4.1.4. Hello Retry Request	**Server**	**C**	**22**
	key_share (RFC8446)	**C**	**22**
	cookie (RFC8446)	**C**	**23.1**
	supported_versions (RFC8446)	**C**	**22**
4.2.1. Supported Versions	**Client**	**C**	**22.1**
	Server	**C**	**22**
4.2.2. Cookie	**Client**	**C**	**23.1**
	Server	**C**	**23.1**
4.2.3. Signature Algorithms	**Client**	**C**	**24**
	rsa_pkcs1_sha256	**C**	**22.1**
	rsa_pkcs1_sha384	**C**	**22.1**
	rsa_pkcs1_sha512	**C**	**22.1**
	ecdsa_secp256r1_sha256	**C**	**22.1**
	ecdsa_secp384r1_sha384	**C**	**22.1**
	ecdsa_secp521r1_sha512	**C**	**22.1**
	rsa_pss_rsae_sha256	**C**	**22.1**
	rsa_pss_rsae_sha384	**C**	**22.1**
	rsa_pss_rsae_sha512	**C**	**22.1**
	ed25519	**C**	**24**
	ed448	**C**	**24**
	rsa_pss_pss_sha256	**C**	**23**

	rsa_pss_pss_sha384	**C**	**23**	
	rsa_pss_pss_sha512	**C**	**23**	
	rsa_pkcs1_sha1	**C**	**22.1**	
	ecdsa_sha1	**C**	**22.1**	
	Server	**C**	**24**	
	rsa_pkcs1_sha256	**C**	**22**	
	rsa_pkcs1_sha384	**C**	**22**	
	rsa_pkcs1_sha512	**C**	**22**	
	ecdsa_secp256r1_sha256	**C**	**22.1**	
	ecdsa_secp384r1_sha384	**C**	**22.1**	
	ecdsa_secp521r1_sha512	**C**	**22.1**	
	rsa_pss_rsae_sha256	**C**	**22**	
	rsa_pss_rsae_sha384	**C**	**22**	
	rsa_pss_rsae_sha512	**C**	**22**	
	ed25519	**C**	**24**	
	ed448	**C**	**24**	
	rsa_pss_pss_sha256	**C**	**23**	
	rsa_pss_pss_sha384	**C**	**23**	
	rsa_pss_pss_sha512	**C**	**23**	
	rsa_pkcs1_sha1	**C**	**22**	
	ecdsa_sha1	**C**	**22**	
4.2.4. Certificate Authorities	**Client**	**NC**		
	Server	**NC**		
4.2.5. OID Filters	**Client**	**NC**		
	Server	**NC**		

4.2.6. Post-Handshake Client Authentication	**Client**	**NC**	
	Server	**NC**	
4.2.7. Supported Groups	**Client**	**C**	**22.1**
	secp256r1	**C**	**22.1**
	secp384r1	**C**	**22.1**
	secp521r1	**C**	**22.1**
	x25519	**C**	**22.1**
	x448	**C**	**22.1**
	ffdhe2048	**C**	**22.1**
	ffdhe3072	**C**	**22.1**
	ffdhe4096	**C**	**22.1**
	ffdhe6144	**C**	**22.1**
	ffdhe8192	**C**	**22.1**
	Server	**C**	**22**
	secp256r1	**C**	**22**
	secp384r1	**C**	**22**
	secp521r1	**C**	**22**
	x25519	**C**	**22**
	x448	**C**	**22**
	ffdhe2048	**C**	**22**
	ffdhe3072	**C**	**22**
	ffdhe4096	**C**	**22**
	ffdhe6144	**C**	**22**
	ffdhe8192	**C**	**22**
4.2.8. Key Share	**Client**	**C**	**22.1**

	Server	**C**	**22**
4.2.9. Pre-Shared Key Exchange Modes	**Client**	**C**	**22.2**
	Server	**C**	**22.2**
4.2.10. Early Data Indication	**Client**	**C**	**23.3**
	Server	**C**	**23.3**
4.2.11. Pre-Shared Key Extension	**Client**	**C**	**22.2**
	Server	**C**	**22.2**
4.2.11.1. Ticket Age	**Client**	**C**	**22.2**
	Server	**C**	**22.2**
4.2.11.2. PSK Binder	**Client**	**C**	**22.2**
	Server	**C**	**22.2**
4.2.11.3. Processing Order	**Client**	**NC**	
	Server	**NC**	
4.3.1. Encrypted Extensions	**Client**	**PC**	**22.1**
	server_name (RFC6066)	**C**	**23.2**
	max_fragment_length (RFC6066)	**C**	**23.0**
	supported_groups (RFC7919)	**NC**	
	use_srtp (RFC5764)	**NC**	
	heartbeat (RFC6520)	**NC**	
	application_layer_protocol_negotiation (RFC7301)	**NC**	
	client_certificate_type (RFC7250)	**NC**	

	server_certificate_type (RFC7250)	**NC**	
	early_data (RFC8446)	**C**	**23.3**
	supported_versions (RFC8446)	**NC**	
	Server	**PC**	**22**
	server_name (RFC6066)	**C**	**23.2**
	max_fragment_length (RFC6066)	**C**	**23.0**
	supported_groups (RFC7919)	**NC**	
	use_srtp (RFC5764)	**NC**	
	heartbeat (RFC6520)	**NC**	
	application_layer_protocol_negotiation (RFC7301)	**NC**	
	client_certificate_type (RFC7250)	**NC**	
	server_certificate_type (RFC7250)	**NC**	
	early_data (RFC8446)	**C**	**23.3**
	supported_versions (RFC8446)	**NC**	
4.3.2. Certificate Request	**Client**	**PC**	**22.1**
	status_request (RFC6066)	**NC**	
	signature_algorithms (RFC8446)	**C**	**22.1**
	signed_certificate_timestamp (RFC6962)	**NC**	
	certificate_authorities (RFC8446)	**C**	24.3

	oid_filters (RFC8446)	**NC**	
	signature_algorithms_cert (RFC8446)	**C**	**22.1**
	Server	**PC**	**22**
	status_request (RFC6066)	**NC**	
	signature_algorithms (RFC8446)	**C**	**22**
	signed_certificate_timestamp (RFC6962)	**NC**	
	certificate_authorities (RFC8446)	**C**	24.3
	oid_filters (RFC8446)	**NC**	
	signature_algorithms_cert (RFC8446)	**C**	**22**
4.4.1. The Transcript Hash		**C**	**22**
4.4.2. Certificate	**Client**	**PC**	**22.1**
	Arbitrary certificate chain orderings	**C**	**22.2**
	Extraneous certificates in chain	**C**	**23.2**
	status_request (RFC6066)	**NC**	
	signed_certificate_timestamp (RFC6962)	**NC**	
	Server	**PC**	**22**
	status_request (RFC6066)	**NC**	
	signed_certificate_timestamp (RFC6962)	**NC**	

4.4.2.1. OCSP Status and SCT Extensions	**Client**	**NC**	
	Server	**NC**	
4.4.2.2. Server Certificate Selection		**C**	**24.3**
	The certificate type MUST be X.509v3, unless explicitly negotiated otherwise	**C**	**22**
	The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm from the client's "signature_algorithms" extension (currently RSA, ECDSA, or EdDSA).	**C**	**22**
	The certificate MUST allow the key to be used for signing with a signature scheme indicated in the client's "signature_algorithms" extensions	**C**	**22** "signature_algorithms_cert"
	The "server_name" and "certificate_authorities" extensions are used to guide certificate selection. As servers MAY require the presence of the "server_name" extension, clients SHOULD send this extension, when applicable.	**C**	**24.3**

4.4.2.3. Client Certificate Selection		**PC**	**22.1**
	The certificate type MUST be X.509v3, unless explicitly negotiated otherwise	**C**	**22.1**
	If the "certificate_authorities" 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.	**C**	**24.3**
	The certificates MUST be signed using an acceptable signature algorithm	**C**	**22.1**
	If the CertificateRequest message contained a non-empty "oid_filters" extension, the end-entity certificate MUST match the extension OIDs that are recognized by the client	**NC**	
4.4.2.4. Receiving a Certificate Message	**Client**	**C**	**22.1**
	Server	**C**	**22**
4.4.3. Certificate Verify	**Client**	**C**	**22.1**
	Server	**C**	**22**
4.4.4. Finished	**Client**	**C**	**22.1**
	Server	**C**	**22**

4.5. End of Early Data	**Client**	**C**	**23.3**
	Server	**C**	**23.3**
4.6.1. New Session Ticket Message	**Client**	**C**	**23.3**
	early_data (RFC8446)	**C**	**23.3**
	Server	**C**	**23.3**
	early_data (RFC8446)	**C**	**23.3**
4.6.2. Post-Handshake Authentication	**Client**	**NC**	
	Server	**NC**	
4.6.3. Key and Initialization Vector Update	**Client**	**C**	**22.3**
	Server	**C**	**22.3**
5.1. Record Layer		**C**	**22**
	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		**PC**	**22**	
	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		**C**	**22.3**	
6.1. Closure Alerts		**22**		
	close_notify	**C**	**22**	
	user_cancelled	**C**	**22**	
6.2. Error Alerts		**PC**	**22**	
7.1. Key Schedule		**C**	**22**	
7.2. Updating Traffic Secrets		**C**	**22**	
7.3. Traffic Key Calculation		**C**	**22**	
7.5. Exporters		**NC**		
8. 0-RTT and Anti-Replay		**C**	**22.2**	
8.1. Single-Use Tickets		**C**	**22.2**	
8.2. Client Hello Recording		**C**	**22.2**	
8.3. Freshness Checks		**C**	**22.2**	
9.1. Mandatory-to-Implement Cipher Suites		**C**	**22.1**	

	MUST implement the TLS_AES_128_GCM_SHA256	**C**	**22**
	SHOULD implement the TLS_AES_256_GCM_SHA384	**C**	**22**
	SHOULD implement the TLS_CHACHA20_POLY1305_SHA256	**C**	**22**
	Digital signatures	**C**	**22.1**
	MUST support rsa_pkcs1_sha256 (for certificates)	**C**	**22**
	MUST support rsa_pss_rsae_sha256 (for CertificateVerify and certificates)	**C**	**22**
	MUST support ecdsa_secp256r1_sha256	**C**	**22.1**
	Key Exchange	**C**	**22**
	MUST support key exchange with secp256r1	**C**	**22**
	SHOULD support key exchange with X25519	**C**	**22**
9.2. Mandatory-to-Implement Extensions		**C**	**23.2**
	Supported Versions	**C**	**22**
	Cookie	**C**	**23.1**
	Signature Algorithms	**C**	**22**
	Signature Algorithms Certificate	**C**	**22**
	Negotiated Groups	**C**	**22**
	Key Share	**C**	**22**

	Server Name Indication	**C**	**23.2**
	MUST send and use these extensions	**C**	**22.2**
	"supported_versions" is REQUIRED for ClientHello, ServerHello and HelloRetryRequest	**C**	**22.1**
	"signature_algorithms" is REQUIRED for certificate authentication	**C**	**22**
	"supported_groups" is REQUIRED for ClientHello messages using (EC)DHE key exchange	**C**	**22**
	"key_share" is REQUIRED for (EC)DHE key exchange	**C**	**22**
	"pre_shared_key" is REQUIRED for PSK key agreement	**C**	**22.2**
	"psk_key_exchange_modes" is REQUIRED for PSK key agreement	**C**	**22.2**
	TLS 1.3 ClientHello	**C**	**22.1**
	If not containing a "pre_shared_key" extension, it MUST contain both a "signature_algorithms" extension and a "supported_groups" extension.	**C**	**22.1**
	If containing a "supported_groups" extension, it MUST also contain	**C**	**22.1**

	a "key_share" extension, and vice versa. An empty KeyShare.client_shares vector is permitted.		
	TLS 1.3 ServerHello	**C**	**23.2**
	MUST support the use of the "server_name" extension	**C**	**23.2**
9.3. Protocol Invariants		**C**	**22.1**
	MUST correctly handle extensible fields	**C**	**22.1**
	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.	**C**	**22.1**
	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 a CertificateRequest or NewSessionTicket MUST also ignore all unrecognized extensions.	**C**	**22.1**
	A middlebox which terminates a TLS connection MUST behave as	**NA**	

	a compliant TLS server		
	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.	**NA**	
B.4. Cipher Suites		**C**	**23**
	TLS_AES_128_GCM_SHA256	**C**	**22**
	TLS_AES_256_GCM_SHA384	**C**	**22**
	TLS_CHACHA20_POLY1305_SHA256	**C**	**22**
	TLS_AES_128_CCM_SHA256	**C**	**22**
	TLS_AES_128_CCM_8_SHA256	**C**	**23**
C.1. Random Number Generation and Seeding		**C**	**22**
C.2. Certificates and Authentication		**C**	**22**
C.3. Implementation Pitfalls		**PC**	**22**
C.4. Client Tracking Prevention		**C**	**22.2**
C.5. Unauthenticated Operation		**C**	**22**
D.1. Negotiating with an Older Server		**C**	**22.2**

D.2. Negotiating with an Older Client		**C**	**22**
D.3. 0-RTT Backward Compatibility		**NC**	
D.4. Middlebox Compatibility Mode		**C**	**23**
D.5. Security Restrictions Related to Backward Compatibility		**C**	**22**

Table 5.1: Standards Compliance

2 Reference Manual

ssl

Application

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']"
```

`protocol_version =` ssl:tls_version() | [ssl:tls_version()] `<optional>`

Protocol supported by started clients and servers. If this option is not set, it defaults to all TLS protocols currently supported, more might be configurable, by the SSL application. This option can be overridden by the version option to `ssl:connect/[2,3]` and `ssl:listen/2`.

`dtls_protocol_version =` ssl:dtls_version() | [ssl:dtls_version()] `<optional>`

Protocol supported by started clients and servers. If this option is not set, it defaults to all DTLS protocols currently supported, more might be configurable, by the SSL application. This option can be overridden by the version option to `ssl:connect/[2,3]` and `ssl:listen/2`.

`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.

`session_cb = atom() <optional>`

Deprecated Since OTP-23.3 replaced by `client_session_cb` and `server_session_cb`

`client_session_cb = atom() <optional>`

Since OTP-23.3 Name client of the session cache callback module that implements the `ssl_session_cache_api` behavior. Defaults to `ssl_client_session_cache_db`.

`server_session_cb = atom() <optional>`

Since OTP-23.3 Name of the server session cache callback module that implements the `ssl_session_cache_api` behavior. Defaults to `ssl_server_session_cache_db`.

`session_cb_init_args = proplist:proplist() <optional>`

Deprecated Since OTP-23.3 replaced by `client_session_cb_init_args` and `server_session_cb_init_args`

`client_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 `[]`.

`server_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 `[]`.

`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.

`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 milliseconds between PEM cache validations. Defaults to 2 minutes.

Note: The cache can be reloaded by calling ssl:clear_pem_cache/0.

`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).

`server_session_tickets_amount = integer() <optional>`

Number of session tickets sent by the server. It must be greater than 0. Defaults to 3.

`server_session_ticket_lifetime = integer() <optional>`

Lifetime of session tickets sent by the server. Servers must not use any value greater than 604800 seconds (7 days). Expired tickets are automatically removed. Defaults to 7200 seconds (2 hours).

`server_session_ticket_store_size = integer() <optional>`

Sets the maximum size of the server session ticket store (stateful tickets). Defaults to 1000. Size limit is enforced by dropping old tickets.

`server_session_ticket_max_early_data = integer() <optional>`

Sets the maximum size of the early data that the server accepts and also configures its NewSessionTicket messages to include this same size limit in their early_data_indication extension. Defaults to 16384. Size limit is enforced by both client and server.

`client_session_ticket_lifetime = integer() <optional>`

Lifetime of session tickets in the client ticket store. Expired tickets are automatically removed. Defaults to 7200 seconds (2 hours).

`client_session_ticket_store_size = integer() <optional>`

Sets the maximum size of the client session ticket store. Defaults to 1000. Size limit is enforced by dropping old tickets.

ERROR LOGGER AND EVENT HANDLERS

The SSL application uses OTP logger. TLS/DTLS alerts are logged on notice level. Unexpected errors are logged on error level. These log entries will by default end up in the default Erlang log. The option `log_level` may be used to in run-time to set the log level of a specific TLS connection, which is handy when you want to use level debug to inspect the TLS handshake setup.

SEE ALSO

application(3)

ssl

Erlang module

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

Data Types

Types used in 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 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 `gen_tcp`, and have functions corresponding to `inet:setopts/2`, `inet:getopts/2`, `inet:peername/1`, `inet:sockname/1`, and `inet:port/1`. The callback `gen_tcp` is treated specially and calls `inet` directly. For DTLS this feature must be considered experimental.

```
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'
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()
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()}
sign_algo() = rsa | dsa | ecdsa | eddsa
sign_scheme() =
    eddsa_ed25519 | eddsa_ed448 | ecdsa_secp256r1_sha256 |
```

```
    ecdsa_secp384r1_sha384 | ecdsa_secp521r1_sha512 |
    rsassa_pss_scheme() |
    sign_scheme_legacy()
rsassa_pss_scheme() =
    rsa_pss_rsae_sha256 | rsa_pss_rsae_sha384 |
    rsa_pss_rsae_sha512 | rsa_pss_pss_sha256 |
    rsa_pss_pss_sha384 | rsa_pss_pss_sha512
sign_scheme_legacy() =
    rsa_pkcs1_sha256 | rsa_pkcs1_sha384 | rsa_pkcs1_sha512 |
    rsa_pkcs1_sha1 | ecdsa_sha1
group() =
    secp256r1 | secp384r1 | secp521r1 | ffdhe2048 | ffdhe3072 |
    ffdhe4096 | ffdhe6144 | ffdhe8192
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(),
      max_frag_enum => 1..4,
      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()
bloom_filter_window_size() = integer()
bloom_filter_hash_functions() = integer()
bloom_filter_bits() = integer()
client_session_tickets() = disabled | manual | auto
server_session_tickets() = disabled | stateful | stateless
```

TLS/DTLS OPTION DESCRIPTIONS - COMMON for SERVER and CLIENT

```
common_option() =
    {protocol, protocol()} |
    {handshake, handshake_completion()} |
    {cert, cert() | [cert()]} |
    {certfile, cert_pem()} |
    {key, key()} |
    {keyfile, key_pem()} |
    {password, key_pem_password()} |
    {certs_keys, certs_keys()} |
    {ciphers, cipher_suites()} |
    {eccs, [named_curve()]} |
    {signature_algs, signature_algs()} |
    {signature_algs_cert, sign_schemes()} |
    {supported_groups, supported_groups()} |
    {secure_renegotiate, secure_renegotiation()} |
    {keep_secrets, keep_secrets()} |
    {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()} |
    {session_tickets, session_tickets()} |
    {key_update_at, key_update_at()} |
    {receiver_spawn_opts, spawn_opts()} |
```

```
    {sender_spawn_opts, spawn_opts()}
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 user certificate. Note that the cert option may also be a list of DER-encoded certificates where the first one is the user certificate, and the rest of the certificates constitutes the certificate chain. For maximum interoperability the certificates in the chain should be in the correct order, the chain will be sent as is to the peer. If chain certificates are not provided, certificates from client_cacerts(), server_cacerts(), or client_cafile(), server_cafile() are used to construct the chain. If this option is supplied, it overrides option `certfile`.

```
cert_pem() = file:filename()
```

Path to a file containing the user certificate on PEM format or possible several certificates where the first one is the user certificate and the rest of the certificates constitutes the certificate chain. For more details see cert(),

```
key() =
    {'RSAPrivateKey' | 'DSAPrivateKey' | 'ECPrivateKey' |
     'PrivateKeyInfo',
     public_key:der_encoded()} |
    #{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/3 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_pem_password() = iodata() | fun(() -> iodata())
```

String containing the user's password or a function returning same type. Only used if the private keyfile is password-protected.

```
certs_keys() = [cert_key_conf()]
```

A list of a certificate (or possible a certificate and its chain) and the associated key of the certificate, that may be used to authenticate the client or the server. The certificate key pair that is considered best and matches negotiated parameters for the connection will be selected. Different signature algorithms are prioritized in the order `eddsa, ecdsa, rsa_pss_pss, rsa and dsa` . If more than one key is supplied for the same signing algorithm (which is probably an unusual use case) they will prioritized by strength unless it is a so called `engine key` that will be favoured over other keys. As engine keys cannot be inspected, supplying more than one engine key will make no sense. This offers flexibility to for instance configure a newer certificate that is expected to be used in most cases and an older but acceptable certificate that will only be used to communicate with legacy systems. Note that there is a trade off between the induced overhead and the flexibility so alternatives should be chosen for good reasons. If the `certs_keys` option is specified it overrides all single certificate and key options. For examples see the Users Guide

Note:

`eddsa` certificates are only supported by TLS-1.3 that does not support `dsa` certificates. `rsa_pss_pss` (RSA certificates using Probabilistic Signature Scheme) are supported in TLS-1.2 and TLS-1.3, but some TLS-1.2 implementations may not support `rsa_pss_pss`.

```
cert_key_conf() =
    #{cert => cert(),
      key => key(),
      certfile => cert_pem(),
      keyfile => key_pem(),
      password => key_pem_password()}
```

A certificate (or possibly a certificate and its chain) and its associated key on one of the possible formats. For the PEM file format there may also be a password associated with the file containg the key.

`cipher_suites() = ciphers()`

A list of cipher suites that should be supported

The function ssl:cipher_suites/2 can be used to find all cipher suites that are supported by default and all cipher suites that may be configured.

If you compose your own `cipher_suites()` make sure they are filtered for cryptolib support ssl:filter_cipher_suites/2 Additionally the functions ssl:append_cipher_suites/2 , ssl:prepend_cipher_suites/2, ssl:suite_to_str/1, ssl:str_to_suite/1, and ssl:suite_to_openssl_str/1 also exist to help creating customized cipher suite lists.

Note:

Note that TLS-1.3 and TLS-1.2 cipher suites are not overlapping sets of cipher suites so to support both these versions cipher suites from both versions need to be included. Also if the supplied list does not comply with the configured versions or cryptolib so that the list becomes empty, this option will fallback on its appropriate default value for the configured versions.

Non-default cipher suites including anonymous cipher suites (PRE TLS-1.3) are supported for interop/testing purposes and may be used by adding them to your cipher suite list. Note that they must also be supported/enabled by the peer to actually be used.

`signature_algs() = [{hash(), sign_algo()} | sign_scheme()]`

Explicitly list acceptable signature algorithms for certificates and handshake messages in the preferred order. The client will send its list as the client hello `signature_algorithm` extension introduced in TLS-1.2, see **Section 7.4.1.4.1 in RFC 5246**. Previously these algorithms where implicitly chosen and partly derived from the cipher suite.

In TLS-1.2 a somewhat more explicit negotiation is made possible using a list of {hash(), sign_algo()} pairs.

In TLS-1.3 these algorithm pairs are replaced by so called signature schemes sign_scheme() and completely decoupled from the cipher suite.

Signature algorithms used for certificates may be overridden by the signature schemes (algorithms) supplied by the `signature_algs_cert` option.

TLS-1.2 default is Default_TLS_12_Alg_Pairs interleaved with rsa_pss_schemes since ssl-11.0 (OTP-25) pss_pss is prefered over pss_rsae that is prefered over rsa

`Default_TLS_12_Alg_Pairs =`

```
[
%% SHA2
{sha512, ecdsa},
{sha512, rsa},
{sha384, ecdsa},
{sha384, rsa},
{sha256, ecdsa},
{sha256, rsa},
{sha224, ecdsa},
{sha224, rsa},
%% SHA
{sha, ecdsa},
{sha, rsa},
{sha, dsa}
]
```

Support for {md5, rsa} was removed from the the TLS-1.2 default in ssl-8.0 (OTP-22)

`rsa_pss_schemes =`

```
[rsa_pss_pss_sha512,
rsa_pss_pss_sha384,
rsa_pss_pss_sha256,
rsa_pss_rsae_sha512,
rsa_pss_rsae_sha384,
rsa_pss_rsae_sha256]
```

`TLS_13_Legacy_Schemes =`

```
 [
 %% Legacy algorithms only applicable to certificate signatures
rsa_pkcs1_sha512, %% Corresponds to {sha512, rsa}
rsa_pkcs1_sha384, %% Corresponds to {sha384, rsa}
rsa_pkcs1_sha256, %% Corresponds to {sha256, rsa}
ecdsa_sha1,       %% Corresponds to {sha, ecdsa}
rsa_pkcs1_sha1    %% Corresponds to {sha, rsa}
]
```

`Default_TLS_13_Schemes =`

```
 [
 %% EDDSA
eddsa_ed25519,
eddsa_ed448

%% ECDSA
ecdsa_secp521r1_sha512,
ecdsa_secp384r1_sha384,
ecdsa_secp256r1_sha256] ++

%% RSASSA-PSS
rsa_pss_schemes()
```

EDDSA was made highest priority in ssl-11.0 (OTP-25)

TLS-1.3 default is

```
Default_TLS_13_Schemes ++ Legacy_TLS_13_Schemes
```

If both TLS-1.3 and TLS-1.2 are supported the default will be

```
Default_TLS_13_Schemes ++ Default_TLS_12_Alg_Pairs
```

so appropriate algorithms can be chosen for the negotiated version.

> **Note:**
>
> TLS-1.2 algorithms will not be negotiated for TLS-1.3, but TLS-1.3 RSASSA-PSS rsassa_pss_scheme() signature schemes may be negotiated also for TLS-1.2 from 24.1 (fully working from 24.1.3). However if TLS-1.3 is negotiated when both TLS-1.3 and TLS-1.2 is supported using defaults, the corresponding TLS-1.2 algorithms to the TLS-1.3 legacy signature schemes will be considered as the legacy schemes and applied only to certificate signatures.

`sign_schemes() = [sign_scheme()]`

Explicitly list acceptable signature schemes (algorithms) in the preferred order. Overrides the algorithms supplied in `signature_algs` option for certificates.

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 need.

The client will send a `signature_algorithms_cert` extension (in the client hello message), if TLS version 1.2 (back-ported to TLS 1.2 in 24.1) or later is used, and the signature_algs_cert option is explicitly specified. By default, only the signature_algs extension is sent.

> **Note:**
>
> Note that supported signature schemes for TLS-1.2 are sign_scheme_legacy() and rsassa_pss_scheme()

`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 10.

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

The verification fun is to be defined as follows:

```
fun(OtpCert :: #'OTPCertificate'{},
    Event, InitialUserState :: term()) ->
 {valid, UserState :: term()} |
 {fail, Reason :: term()} | {unknown, UserState :: term()}.

fun(OtpCert :: #'OTPCertificate'{}, DerCert :: public_key:der_encoded(),
    Event, InitialUserState :: term()) ->
 {valid, UserState :: term()} |
 {fail, Reason :: term()} | {unknown, UserState :: term()}.

Types:
      Event = {bad_cert, Reason :: atom() |
               {revoked, atom()}} |
       {extension, #'Extension'{}} |
               valid |
               valid_peer
```

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 `Event` 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`:

```
{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`:

```
{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() =
    {Module :: atom(),
     {DbHandle :: internal | term(), Args :: list()}}
```

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 inX509 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, 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 :: binary(), UserState :: term()) ->
 {ok, SharedSecret :: binary()} | error;
fun(srp, Username :: binary(), 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`, TLS/DTLS Alert reports are not displayed. Deprecated in OTP 22, use {log_level, logging_level()} instead.

`logging_level() = logger:level() | none | all`

Specifies the log level for a TLS/DTLS connection. Alerts are logged on `notice` level, which is the default level. The level `debug` triggers verbose logging of TLS/DTLS protocol messages. See also ssl(6)

`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 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 1/n-1 BEAST mitigation.

`zero_n` - Perform 0/n BEAST mitigation.

`disabled` - Disable BEAST mitigation.

Warning:

Using `{beast_mitigation, disabled}` makes TLS-1.0 vulnerable to the BEAST attack.

```
ssl_imp() = new | old
```

Deprecated since OTP-17, has no effect.

```
session_tickets() =
    client_session_tickets() | server_session_tickets()
```

Configures the session ticket functionality in TLS 1.3 client and server.

```
key_update_at() = integer() >= 1
```

Configures the maximum amount of bytes that can be sent on a TLS 1.3 connection before an automatic key update is performed.

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. The current default ensures that data integrity will not be breached with probability greater than 1/2^57. For more information see **Limits on Authenticated Encryption Use in TLS**.

Warning:

The default value of this option shall provide the above mentioned security guarantees and it shall be reasonable for most applications (~353 TB).

```
middlebox_comp_mode() = boolean()
```

Configures the middlebox compatibility mode on a TLS 1.3 connection.

A significant number of middleboxes misbehave when a TLS 1.3 connection is negotiated. Implementations can increase the chance of making connections through those middleboxes by making the TLS 1.3 handshake more like a TLS 1.2 handshake.

The middlebox compatibility mode is enabled (`true`) by default.

```
spawn_opts() = [erlang:spawn_opt_option()]
```

Configures spawn options of TLS sender and receiver processes.

Setting up garbage collection options can be helpful for trade-offs between CPU usage and Memory usage. See `erlang:spawn_opt/2`.

For dist connections, default sender option is `[...{priority, max}]`, this priority option cannot be changed. For all connections, `...link` is added to receiver and cannot be changed.

```
keep_secrets() = boolean()
```

Configures a TLS 1.3 connection for keylogging

In order to retrieve keylog information on a TLS 1.3 connection, it must be configured in advance to keep the client_random and various handshake secrets.

The keep_secrets functionality is disabled (`false`) by default.

Added in OTP 23.2

TLS/DTLS OPTION DESCRIPTIONS - CLIENT

```
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,
     client_preferred_next_protocols()} |
    {psk_identity, client_psk_identity()} |
    {srp_identity, client_srp_identity()} |
    {server_name_indication, sni()} |
    {max_fragment_length, max_fragment_length()} |
    {customize_hostname_check, customize_hostname_check()} |
    {fallback, fallback()} |
    {middlebox_comp_mode, middlebox_comp_mode()} |
    {certificate_authorities, client_certificate_authorities()} |
    {session_tickets, client_session_tickets()} |
    {use_ticket, use_ticket()} |
    {early_data, client_early_data()}
client_verify_type() = verify_type()
```

Defaults to `verify_none` as additional options are needed to be able to perform the certificate verification. A warning will be emitted unless `verify_none` is explicitly configured. Usually the applications will want to configure `verify_peer` together with an appropriate `cacert` or `cacertfile` option. For example an `HTTPS` client would normally use the option `{cacerts, public_key:cacerts_get()}` (available since OTP-25) to access the CA certificates provided by the OS. Using verify_none means that all x509-certificate path validation errors will be ignored. See also option verify_fun.

```
client_reuse_session() =
    session_id() | {session_id(), SessionData :: binary()}
```

Reuses a specific session. The session should be referred by its session id if it is earlier saved with the option `{reuse_sessions, save}` since OTP-21.3 or explicitly specified by its session id and associated data since OTP-22.3. See also SSL's Users Guide, Session Reuse pre TLS 1.3.

```
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, automated 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.

```
client_certificate_authorities() = boolean()
```

If set to true, sends the certificate authorities extension in TLS-1.3 client hello. The default is false. Note that setting it to true may result in a big overhead if you have many trusted CA certificates. Since OTP-24.3.

```
client_cacerts() =
    [public_key:der_encoded()] | [public_key:combined_cert()]
```

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

```
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.

Note:

When PEM caching is enabled, files provided with this option will be checked for updates at fixed time intervals specified by the ssl_pem_cache_clean environment parameter.

Note:

Alternatively, CA certificates can be provided as a DER-encoded binary with client_cacerts option.

```
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.

```
client_preferred_next_protocols() =
    {Precedence :: server | client,
     ClientPrefs :: [app_level_protocol()]} |
    {Precedence :: server | client,
     ClientPrefs :: [app_level_protocol()],
     Default :: app_level_protocol()}
```

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.

```
max_fragment_length() = undefined | 512 | 1024 | 2048 | 4096
```

Specifies the maximum fragment length the client is prepared to accept from the server. See **RFC 6066**

```
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

```
ssl:connect(Host, Port, [...{versions, ['tlsv2', 'tlsv1.1', 'tlsv1']}])
ssl:connect(Host, Port, [...{versions, [tlsv1.1', 'tlsv1']}, {fallback, true}])
ssl:connect(Host, Port, [...{versions, ['tlsv1']}, {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_session_tickets() = disabled | manual | auto`

Configures the session ticket functionality. Allowed values are `disabled`, `manual` and `auto`. If it is set to `manual` the client will send the ticket information to user process in a 3-tuple:

`{ssl, session_ticket, {SNI, TicketData}}`

where `SNI` is the ServerNameIndication and `TicketData` is the extended ticket data that can be used in subsequent session resumptions.

If it is set to `auto`, the client automatically handles received tickets and tries to use them when making new TLS connections (session resumption with pre-shared keys).

Note:

This option is supported by TLS 1.3 and above. See also SSL's Users Guide, Session Tickets and Session Resumption in TLS 1.3

`use_ticket() = [binary()]`

Configures the session tickets to be used for session resumption. It is a mandatory option in `manual` mode (`session_tickets = manual`).

Note:

Session tickets are only sent to user if option **session_tickets** is set to `manual`

This option is supported by TLS 1.3 and above. See also SSL's Users Guide, Session Tickets and Session Resumption in TLS 1.3

```
client_early_data() = binary()
```

Configures the early data to be sent by the client.

In order to be able to verify that the server has the intention to process the early data, the following 3-tuple is sent to the user process:

```
{ssl, SslSocket, {early_data, Result}}
```

where `Result` is either `accepted` or `rejected`.

Warning:

It is the responsibility of the user to handle a rejected Early Data and to resend when it is appropriate.

TLS/DTLS OPTION DESCRIPTIONS - SERVER

```
server_option() =
    {cacerts, server_cacerts()} |
    {cacertfile, server_cafile()} |
    {dh, dh_der()} |
    {dhfile, dh_file()} |
    {verify, server_verify_type()} |
    {fail_if_no_peer_cert, fail_if_no_peer_cert()} |
    {certificate_authorities, server_certificate_authorities()} |
    {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()} |
    {sni_hosts, sni_hosts()} |
    {sni_fun, sni_fun()} |
    {honor_cipher_order, honor_cipher_order()} |
    {honor_ecc_order, honor_ecc_order()} |
    {client_renegotiation, client_renegotiation()} |
    {session_tickets, server_session_tickets()} |
    {anti_replay, anti_replay()} |
    {cookie, cookie()} |
    {early_data, server_early_data()}
server_cacerts() =
    [public_key:der_encoded()] | [public_key:combined_cert()]
```

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

```
server_certificate_authorities() = boolean()
```

Determines if a TLS-1.3 server should include the authorities extension in its certificate request message that will be sent if the option `verify` is set to `verify_peer`. Defaults to `true`.

A reason to exclude the extension would be if the server wants to communicate with clients incapable of sending complete certificate chains that adhere to the extension, but the server still has the capability to recreate a chain that it can verify.

`server_cafile() = 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.

Note:

When PEM caching is enabled, files provided with this option will be checked for updates at fixed time intervals specified by the ssl_pem_cache_clean environment parameter.

Note:

Alternatively, CA certificates can be provided as a DER-encoded binary with server_cacerts option.

`dh_der() = binary()`

The DER-encoded Diffie-Hellman parameters. If specified, it overrides option `dhfile`.

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()`

Client certificates are an optional part of the TLS protocol. A server only does x509-certificate path validation in mode `verify_peer`. By default the server is in `verify_none` mode an hence will not send an certificate request to the client. When using `verify_peer` you may also want to specify the options fail_if_no_peer_cert and certificate_authorities.

`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 negotiates 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_session_tickets() = disabled | stateful | stateless`

Configures the session ticket functionality. Allowed values are `disabled`, `stateful` and `stateless`.

If it is set to `stateful` or `stateless`, session resumption with pre-shared keys is enabled and the server will send stateful or stateless session tickets to the client after successful connections.

A stateful session ticket is a database reference to internal state information. A stateless session ticket is a self-encrypted binary that contains both cryptographic keying material and state data.

> **Note:**
>
> This option is supported by TLS 1.3 and above. See also SSL's Users Guide, Session Tickets and Session Resumption in TLS 1.3

```
anti_replay() =
    '10k' | '100k' |
    {bloom_filter_window_size(),
     bloom_filter_hash_functions(),
     bloom_filter_bits()}
```

Configures the server's built-in anti replay feature based on Bloom filters.

Allowed values are the pre-defined `'10k'`, `'100k'` or a custom 3-tuple that defines the properties of the bloom filters: `{WindowSize, HashFunctions, Bits}`. `WindowSize` is the number of seconds after the current Bloom filter is rotated and also the window size used for freshness checks. `HashFunctions` is the number hash functions and `Bits` is the number of bits in the bit vector. `'10k'` and `'100k'` are simple defaults with the following properties:

- `'10k'`: Bloom filters can hold 10000 elements with 3% probability of false positives. `WindowSize`: 10, `HashFunctions`: 5, `Bits`: 72985 (8.91 KiB).
- `'100k'`: Bloom filters can hold 100000 elements with 3% probability of false positives. `WindowSize`: 10, `HashFunctions`: 5, `Bits`: 729845 (89.09 KiB).

> **Note:**
>
> This option is supported by TLS 1.3 and above and only with stateless session tickets. Ticket lifetime, the number of tickets sent by the server and the maximum number of tickets stored by the server in stateful mode are configured by application variables. See also SSL's Users Guide, Anti-Replay Protection in TLS 1.3

```
cookie() = boolean()
```

If `true` (default), the server sends a cookie extension in its HelloRetryRequest messages.

> **Note:**
>
> The cookie extension has two main purposes. It allows the server to force the client to demonstrate reachability at their apparent network address (thus providing a measure of DoS protection). This is primarily useful for non-connection-oriented transports. It also allows to offload the server's state to the client. The cookie extension is enabled by default as it is a mandatory extension in RFC8446.

```
server_early_data() = disabled | enabled
```

Configures if the server accepts (`enabled`) or rejects (`rejects`) early data sent by a client. The default value is `disabled`.

> **Warning:**
>
> This option is a placeholder, early data is not yet implemented on the server side.

```
connection_info() =
    [common_info() |
```

```
     curve_info() |
     ssl_options_info() |
     security_info()]
common_info() =
    {protocol, protocol_version()} |
    {session_id, session_id()} |
    {session_resumption, boolean()} |
    {selected_cipher_suite, erl_cipher_suite()} |
    {sni_hostname, term()} |
    {srp_username, term()}
curve_info() = {ecc, {named_curve, term()}}
ssl_options_info() = tls_option()
security_info() =
    {client_random, binary()} |
    {server_random, binary()} |
    {master_secret, binary()}
connection_info_items() = [connection_info_item()]
connection_info_item() =
    protocol | session_id | session_resumption |
    selected_cipher_suite | sni_hostname | srp_username | ecc |
    client_random | server_random | master_secret | keylog |
    tls_options_name()
tls_options_name() = atom()
```

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 suites or a list of filters in which case the filters are use on `Suites` to extract the Deferred cipher list.

```
cipher_suites(Description, Version) -> ciphers()
```

Types:

```
Description =
    default | all | exclusive | anonymous | exclusive_anonymous
Version = protocol_version()
```

Lists all possible cipher suites corresponding to `Description` that are available. The `exclusive` and `exclusive_anonymous` option will exclusively list cipher suites first supported in `Version` whereas the other options are inclusive from the lowest possible version to `Version`. The `all` options includes all suites except the anonymous and no anonymous suites are supported by default.

Note:

TLS-1.3 has no overlapping cipher suites with previous TLS versions, that is the result of `cipher_suites(all, 'tlsv1.3').` contains a separate set of suites that can be used with TLS-1.3 an other set that can be used if a lower version is negotiated. PRE TLS-1.3 so called PSK and SRP suites need extra configuration to work see user lookup function. No anonymous suites are supported by TLS-1.3.

Also note that the cipher suites returned by this function are the cipher suites that the OTP ssl application can support provided that they are supported by the cryptolib linked with the OTP crypto application. Use ssl:filter_cipher_suites(Suites, []). to filter the list for the current cryptolib. Note that cipher suites may be filtered out because they are too old or too new depending on the cryptolib

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

Types:

```
   Description = default | all | exclusive | 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 for performance reasons. The cache is automatically checked at regular intervals to see if any cache entries should be invalidated.

This function provides a way to unconditionally clear the entire cache, thereby forcing a reload of previously cached PEM files.

```
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 a 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()

```
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 a 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 happened 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()

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

Types:

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

Closes a TLS/DTLS connection.

```
close(SslSocket, How) ->
        ok | {ok, port()} | {ok, port(), Data} | {error, Reason}
```

Types:

```
SslSocket = sslsocket()
How = timeout() | {NewController :: pid(), timeout()}
Data = binary()
Reason = any()
```

Closes or downgrades a 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: `[{active, false}, {packet, 0}, {mode, binary}]`.

In case of downgrade, the close function might return some binary data that should be treated by the user as the first bytes received on the downgraded connection.

```
controlling_process(SslSocket, NewOwner) -> ok | {error, Reason}
```

Types:

```
SslSocket = 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 = sslsocket()
Result = connection_info()
```

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.

Note:

The legacy `Item = cipher_suite` was removed in OTP-23. Previously it returned the cipher suite on its (undocumented) legacy format. It is replaced by `selected_cipher_suite`.

```
connection_information(SslSocket, Items) ->
                          {ok, Result} | {error, reason()}
```

Types:

```
SslSocket = sslsocket()
Items = connection_info_items()
Result = connection_info()
```

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

Note that client_random, server_random, master_secret and keylog are values that affect the security of connection. Meaningful atoms, not specified above, are the ssl option names.

In order to retrieve keylog and other secret information from a TLS 1.3 connection, keep_secrets must be configured in advance and set to `true`.

Note:

If only undefined options are requested the resulting list can be empty.

```
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. If no filter function is supplied for some part the default behaviour regards it as if there was a filter function that returned true. For examples see Customizing cipher suites Additionally, this function also filters the cipher suites to exclude cipher suites not supported by the cryptolib used by the OTP crypto application. That is calling ssl:filter_cipher_suites(Suites, []) will be equivalent to only applying the filters for cryptolib support.

```
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 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()

Warning:

Not setting the timeout makes the server more vulnerable to DoS attacks.

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

Warning:

The ordinary `Socket` shall be in passive mode ({active, false}) before calling this function, and before the client tries to connect with TLS, or else the behavior of this function is undefined. The best way to ensure this is to create the ordinary listen socket in passive mode.

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

Warning:

Not setting the timeout makes the server more vulnerable to DoS attacks.

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) ->
                      {ok, SslSocket} | {error, Reason}
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 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 = public_key:der_encoded()
```

The peer certificate is returned as a DER-encoded binary. The certificate can be decoded with public_key:pkix_decode_cert/2 Suggested further reading about certificates is public_key User's Guide and ssl User's Guide

```
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 = ciphers() | cipher_filters()
Suites = ciphers()
```

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

```
prf(SslSocket, Secret, Label, Seed, WantedLength) ->
       {ok, binary()} | {error, reason()}
```

Types:

```
SslSocket = sslsocket()
Secret = binary() | master_secret
Label = binary()
Seed = [binary() | prf_random()]
WantedLength = integer() >= 0
```

Uses the Pseudo-Random Function (PRF) of a TLS session to generate extra key material. It either takes user-generated values for `Secret` and `Seed` or atoms directing it to use a specific value from the session security parameters.

```
recv(SslSocket, Length) -> {ok, Data} | {error, reason()}
recv(SslSocket, Length, Timeout) -> {ok, Data} | {error, reason()}
```

Types:

```
SslSocket = sslsocket()
Length = integer()
Data = binary() | list() | HttpPacket
Timeout = timeout()
HttpPacket = any()
```

See the description of `HttpPacket` in `erlang:decode_packet/3` in ERTS.

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

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

Optional argument `Timeout` specifies a time-out in milliseconds. The default value is `infinity`.

```
renegotiate(SslSocket) -> ok | {error, reason()}
```

Types:

```
SslSocket = sslsocket()
```

Initiates a new handshake. A notable return value is `{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.

```
update_keys(SslSocket, Type) -> ok | {error, reason()}
```

Types:

```
SslSocket = sslsocket()
Type = write | read_write
```

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. If the amount of data surpasses those limits, a key update is triggered and a new set of keys are installed. See also the option key_update_at.

This function can be used to explicitly start a key update on a TLS 1.3 connection. There are two types of the key update: if **Type** is set to **write**, only the writing key is updated; if **Type** is set to **read_write**, both the reading and writing keys are updated.

```
send(SslSocket, Data) -> ok | {error, reason()}
```

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.

```
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 :: permanent | transient | temporary) ->
         ok | {error, reason()}
```

Starts the SSL application. Default type is `temporary`.

```
stop() -> ok
```

Stops the SSL application.

```
str_to_suite(CipherSuiteName) ->
               erl_cipher_suite() |
               {error, {not_recognized, CipherSuiteName}}
```

Types:

```
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 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 | implemented, [tls_version()]} |
    {supported_dtls | available_dtls | implemented_dtls,
     [dtls_version()]}
```

Lists information, mainly concerning TLS/DTLS versions, in runtime for debugging and testing purposes.

`app_vsn`
: The application version of the SSL application.

`supported`
: TLS versions supported with current application environment and crypto library configuration. Overridden by a version option on connect/[2,3,4], listen/2, and handshake/[2,3]. For the negotiated TLS version, see connection_information/1 .

`supported_dtls`
: DTLS versions supported with current application environment and crypto library configuration. Overridden by a version option on connect/[2,3,4], listen/2, and handshake/[2,3]. For the negotiated DTLS version, see connection_information/1 .

`available`
: All TLS versions supported with the linked crypto library.

`available_dtls`
: All DTLS versions supported with the linked crypto library.

`implemented`
: All TLS versions supported by the SSL application if linked with a crypto library with the necessary support.

`implemented_dtls`
: All DTLS versions supported by the SSL application if linked with a crypto library with the necessary support.

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

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

Types:

CRLSrc = crl_src()]}

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 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

```
logger_info() =
    {logger:level(),
     Report :: #{description => string(), reason => term()},
     logger:metadata()}
```

Information for ssl applications use of Logger(3)

Exports

```
Module:fresh_crl(DistributionPoint, CRL) -> FreshCRL
Module:fresh_crl(DistributionPoint, CRL) -> FreshCRL | {LoggerInfo, FreshCRL}
```

Types:

`DistributionPoint = dist_point()`
`CRL = [public_key:der_encoded()]`
`FreshCRL = [public_key:der_encoded()]`
`LoggerInfo = {logger, logger_info() }}`

`fun fresh_crl/2` will be used as input option `update_crl` to public_key:pkix_crls_validate/3

It is possible to return logger info that will be used by the TLS connection to produce log events.

```
Module:lookup(DistributionPoint, Issuer, DbHandle) -> not_available | CRLs |
{LoggerInfo, CRLs}
Module:lookup(DistributionPoint, Issuer, DbHandle) -> not_available | CRLs
Module:lookup(DistributionPoint, DbHandle) -> not_available | CRLs
```

Types:

`DistributionPoint = dist_point()`
`Issuer = public_key:issuer_name()`
`DbHandle = crl_cache_ref()`
`CRLs = [public_key:der_encoded()]`
`LoggerInfo = {logger, logger_info() }}`

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 `cRLIssuer` 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.

It is possible to return logger info that will be used by the TLS connection to produce log events.

```
Module:select(Issuer, DbHandle) -> CRLs | {LoggerInfo, CRLs}
Module:select(Issuer, DbHandle) -> CRLs
```

Types:

```
Issuer = public_key:issuer_name() | list()
DbHandle = cache_ref()
LoggerInfo = {logger, logger_info() }
```

Select the CRLs in the cache that are issued by `Issuer` unless the value is a list of so called general names, see X509 certificates records, originating form `#'DistributionPoint'.cRLissuer` and representing different mechanism to obtain the CRLs. The cache callback needs to use the appropriate entry to retrieve the CRLs or return an empty list if it does not exist.

It is possible to return logger info that will be used by the TLS connection to produce log events.

ssl_session_cache_api

Erlang module

Defines the API for the TLS session cache (pre TLS-1.3) 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

`Module:delete(Cache, Key) -> _`

Types:

`Cache = session_cache_ref()`
`Key = session_cache_key()`

Deletes a cache entry. Is only called from the cache handling process.

`Module: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.

> **Note:**
>
> Since OTP-23.3 this functions is only used on the client side and does not need to implemented for a server cache.

`Module: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.

```
Module: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.

```
Module:select_session(Cache, PartialKey) -> [Session]
```

Types:

`Cache = session_cache_ref()`
`PartialKey = partial_key()`
`Session = session()`

Selects sessions that can be reused, that is sessions that include `PartialKey` in its key. Is to be callable from any process.

> **Note:**
>
> Since OTP-23.3 This functions is only used on the client side and does not need to implemented for a server cache.

```
Module: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.

```
Module: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.

```
Module: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.