

Secure Socket Layer

Copyright © 1999-2016 Ericsson AB. All Rights Reserved.

Secure Socket Layer 8.0.1

December 6, 2016

Copyright © 1999-2016 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..

December 6, 2016

1 SSL User's Guide

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

1.1 Introduction

1.1.1 Purpose

Transport Layer Security (TLS) and its predecessor, the Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communications security over a computer network. The protocols use 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).

1.1.2 Prerequisites

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

1.2 TLS and its Predecessor, SSL

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

By default SSL/TLS is run over the TCP/IP protocol even though you can plug in any other reliable transport protocol with the same Application Programming Interface (API) as the `gen_tcp` module in Kernel.

If a client and a server wants to use an upgrade mechanism, such as defined by RFC 2817, to upgrade a regular TCP/IP connection to an SSL 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.

1.2.1 Security Overview

To achieve authentication and privacy, the client and server perform a TLS 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 handshake.

The TLS handshake protocol and data transfer is run on top of the TLS 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

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

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

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

1.3 Using SSL API

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

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

1.3.1 Setting up Connections

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

Minimal Example

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

To set up client/server connections:

Step 1: Start the server side:

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

Step 2: Create an SSL listen socket:

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

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

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

Step 4: Start the client side:

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

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

Step 5: Do the SSL handshake:

```
4 server> ok = ssl:ssl_accept(Socket).
ok
```

Step 6: Send a message over SSL:

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

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

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

Upgrade Example

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

To upgrade to an SSL connection:

Step 1: Start the server side:

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

Step 2: Create a normal TCP listen socket:

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

Step 3: Accept client connection:

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

Step 4: Start the client side:

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

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

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

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

Step 6: Do the SSL handshake:

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

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

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

```
{ok,{sslsocket,[...]}}
```

Step 8: Send a message over SSL:

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

Step 9: Set `active true` on the SSL socket:

```
4 server> ssl:setopts(SSLSocket, [{active, true}]).
ok
```

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

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

1.4 Using SSL for Erlang Distribution

This section describes how the Erlang distribution can use SSL 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 exra distribution module, `inet_tls_dist`, can be used as an alternative. All distribution connections will use SSL and all participating Erlang nodes in a distributed system must use this distribution module.

The security level depends on the parameters provided to the SSL 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 SSL:

- **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 SSL.

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 SSL 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 SSL 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 SSL parameters are supplied (see the next section).

1.4.3 Specifying SSL Options

For SSL to work, at least a public key and a certificate must be specified for the server side. In the following example, the PEM-files consist of two entries, the server certificate and its private key.

On the `erl` command line you can specify options that the SSL distribution adds when creating a socket.

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

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

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

The server can also take the options `dhfile` and `fail_if_no_peer_cert` (also prefixed).

`client_`-prefixed options are used when the distribution initiates a connection to another node. `server_`-prefixed options are used when accepting a connection from a remote node.

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

The command-line argument for specifying the SSL 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 can now look as follows (line breaks in the command are for readability, and are 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>
```

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

1.4.4 Setting up Environment to Always Use SSL

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

1.4.5 Using SSL distribution over IPv6

It is possible to use SSL 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_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>
```

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

2 Reference Manual

ssl

Application

The ssl application is an implementation of the SSL/TLS protocol in Erlang.

- Supported SSL/TLS-versions are SSL-3.0, TLS-1.0, TLS-1.1, and TLS-1.2.
- For security reasons SSL-2.0 is not supported.
- For security reasons SSL-3.0 is no longer supported by default, but can be configured.
- For security reasons DES cipher suites are no longer supported by default, but can be configured.
- 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 latest TLS specification so it is not motivated to implement them.
- Compression is not supported.
- 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.

DEPENDENCIES

The SSL application uses the `public_key` 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:protocol()*`<optional>`

Protocol supported by started clients and servers. If this option is not set, it defaults to all protocols currently supported 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>`

Name of the session cache callback module that implements the `ssl_session_cache_api` behavior. Defaults to `ssl_session_cache`.

`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.

`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.

`ssl_pem_cache_clean = integer() <optional>`

Number of milliseconds between PEM cache validations. Defaults to 2 minutes.

ssl:clear_pem_cache/0

`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.

ERROR LOGGER AND EVENT HANDLERS

The SSL application uses the default *OTP error logger* to log unexpected errors and TLS alerts. The logging of TLS alerts may be turned off with the `log_alert` option.

SEE ALSO

application(3)

ssl

Erlang module

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

DATA TYPES

The following data types are used in the functions for SSL:

```
boolean() =
    true | false
option() =
    socketoption() | ssl_option() | transport_option()
socketoption() =
    proplists:property()
```

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

For valid options, see the *inet(3)* and *gen_tcp(3)* manual pages in Kernel.

```
ssl_option() =
    {verify, verify_type()}
    | {verify_fun, {fun(), term()}}
    | {fail_if_no_peer_cert, boolean()}
    | {depth, integer()}
    | {cert, public_key:der_encoded()}
    | {certfile, path()}
    |   {key,   {'RSAPrivateKey'|   'DSAPrivateKey'   |   'ECPrivateKey'   |
    'PrivateKeyInfo', public_key:der_encoded()}}
    | {keyfile, path()}
    | {password, string()}
    | {cacerts, [public_key:der_encoded()]}
    | {cacertfile, path()}
    | {dh, public_key:der_encoded()}
    | {dhfile, path()}
    | {ciphers, ciphers()}
    |   {user_lookup_fun,   {fun(),   term()}},   {psk_identity,   string()},
    {srp_identity, {string(), string()}}
    | {reuse_sessions, boolean()}
    | {reuse_session, fun()} {next_protocols_advertised, [binary()]}
    | {client_preferred_next_protocols, {client | server, [binary()]} | {client
    | server, [binary()], binary()}}
```

```
    | {log_alert, boolean()}
    | {server_name_indication, hostname() | disable}
    | {sni_hosts, [{hostname(), [ssl_option()]}]}
    | {sni_fun, SNIfun::fun()}
transport_option() =
    {cb_info, {CallbackModule::atom(), DataTag::atom(), ClosedTag::atom(),
    ErrTag:atom()}}
```

Defaults to `{gen_tcp, tcp, tcp_closed, tcp_error}`. Can be used to customize the transport layer. 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.

`CallbackModule =`

`atom()`

`DataTag =`

`atom()`

Used in socket data message.

`ClosedTag =`

`atom()`

Used in socket close message.

```
verify_type() =
    verify_none | verify_peer
path() =
    string()
```

Represents a file path.

```
public_key:der_encoded() =
    binary()
```

ASN.1 DER-encoded entity as an Erlang binary.

```
host() =
    hostname() | ipaddress()
hostname() =
    string()
ip_address() =
    {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6
sslsocket() =
```

opaque()

```
protocol() =
    sslv3 | tlsv1 | 'tlsv1.1' | 'tlsv1.2'
```

```
ciphers() =
```

```
   = [ciphersuite()] | string()
```

According to old API.

```
ciphersuite() =
```

```
   {key_exchange(),  cipher(),  MAC::hash()}  |  {key_exchange(),  cipher(),
   MAC::hash(), PRF::hash()}
```

```
key_exchange()=
```

```
   rsa | dhe_dss | dhe_rsa | dh_anon | psk | dhe_psk | rsa_psk | srp_anon
   | srp_dss | srp_rsa | ecdh_anon | ecdh_ecdsa | ecdhe_ecdsa | ecdh_rsa |
   ecdhe_rsa
```

```
cipher() =
```

```
   rc4_128 | des_cbc | '3des_ede_cbc' | aes_128_cbc | aes_256_cbc | aes_128_gcm
   | aes_256_gcm
```

```
hash() =
```

```
   md5 | sha | sha224 | sha256 | sha348 | sha512
```

```
prf_random() =
```

```
   client_random | server_random
```

```
srp_param_type() =
```

```
   srp_1024 | srp_1536 | srp_2048 | srp_3072 | srp_4096 | srp_6144 | srp_8192
```

```
SNIfun::fun()
```

```
   = fun(ServerName :: string()) -> [ssl_option()]
```

SSL OPTION DESCRIPTIONS - COMMON for SERVER and CLIENT

The following options have the same meaning in the client and the server:

`{cert, public_key:der_encoded()}`

The DER-encoded users certificate. If this option is supplied, it overrides option `certfile`.

`{certfile, path()}`

Path to a file containing the user certificate.

`{key, {'RSAPrivateKey'| 'DSAPrivateKey' | 'ECPrivateKey' |'PrivateKeyInfo', public_key:der_encoded()}}`

The DER-encoded user's private key. If this option is supplied, it overrides option `keyfile`.

`{keyfile, path()}`

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`.

`{password, string()}`

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

`{ciphers, ciphers()}`

Supported cipher suites. The function `cipher_suites/0` can be used to find all ciphers that are supported by default. `cipher_suites(all)` can be called to find all available cipher suites. Pre-Shared Key (**RFC 4279** and **RFC 5487**), Secure Remote Password (**RFC 5054**), RC4 cipher suites, and anonymous cipher suites only

work if explicitly enabled by this option; they are supported/enabled by the peer also. Anonymous cipher suites are supported for testing purposes only and are not be used when security matters.

`{secure_renegotiate, boolean()}`

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

`{depth, integer()}`

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

`{verify_fun, {Verifyfun :: fun(), InitialUserState :: term()}}`

The verification fun is to be defined as follows:

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

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

- If the verify callback fun returns `{fail, Reason}`, the verification process is immediately stopped, an alert is sent to the peer, and the TLS/SSL 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/SSL 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 can not 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, {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 in*X509 certificate extensions*. Requires the OTP inets application.

`ssl_crl_hash_dir`

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

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

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

The following argument is required:

`{dir, string()}`

Specifies the directory in which the CRLs can be found.

`{partial_chain, fun(Chain::[DerCert]) -> {trusted_ca, DerCert} | 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.

`{versions, [protocol()]}`

TLS protocol versions supported by started clients and servers. This option overrides the application environment option `protocol_version`. If the environment option is not set, it defaults to all versions, except SSL-3.0, supported by the SSL application. See also *ssl(6)*.

`{hibernate_after, integer()|undefined}`

When an integer-value is specified, `ssl_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.

`{user_lookup_fun, {Lookupfun :: fun(), UserState :: term()}}`

The lookup fun is to defined as follows:

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

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

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

`{padding_check, boolean()}`

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

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

`{beast_mitigation, one_n_minus_one | zero_n | disabled}`

Affects SSL-3.0 and TLS-1.0 connections only. Used to change the BEAST mitigation strategy to interoperate with legacy software. Defaults to `one_n_minus_one`.

`one_n_minus_one` - Perform 1/n-1 BEAST mitigation.

`zero_n` - Perform 0/n BEAST mitigation.

`disabled` - Disable BEAST mitigation.

> Warning:
>
> Using `{beast_mitigation, disabled}` makes SSL or TLS vulnerable to the BEAST attack.

SSL OPTION DESCRIPTIONS - CLIENT SIDE

The following options are client-specific or have a slightly different meaning in the client than in the server:

`{verify, verify_type()}`

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

`{reuse_sessions, boolean()}`

Specifies if the client is to try to reuse sessions when possible.

`{cacerts, [public_key:der_encoded()]}`

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

`{cacertfile, path()}`

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

`{alpn_advertised_protocols, [binary()]}`

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

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.

`{psk_identity, string()}`

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

`{srp_identity, {Username :: string(), Password :: string()}`

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

`{server_name_indication, hostname()}`

Can be specified when upgrading a TCP socket to a TLS socket to use the TLS Server Name Indication extension.

`{server_name_indication, disable}`

When starting a TLS connection without upgrade, the Server Name Indication extension is sent if possible. This option can be used to disable that behavior.

`{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', 'sslv3']}])
> ssl:connect(Host, Port, [...{versions, [tlsv1.1', 'tlsv1', 'sslv3']}, {fallback, true}])
> ssl:connect(Host, Port, [...{versions, ['tlsv1', 'sslv3']}, {fallback, true}])
> ssl:connect(Host, Port, [...{versions, ['sslv3']}, {fallback, true}])
> ```
>
> may use it to avoid undesired TLS version downgrade. Note that TLS_FALLBACK_SCSV must also be supported by the server for the prevention to work.

`{signature_algs, [{hash(), ecdsa | rsa | dsa}]}`

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

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

```
{sha, rsa},
{sha, dsa},
]
```

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

SSL OPTION DESCRIPTIONS - SERVER SIDE

The following options are server-specific or have a slightly different meaning in the server than in the client:

`{cacerts, [public_key:der_encoded()]}`

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

`{cacertfile, path()}`

Path to a file containing PEM-encoded CA certificates. The CA certificates are used to build the server certificate chain and for client authentication. The CAs are also used in the list of acceptable client CAs passed to the client when a certificate is requested. Can be omitted if there is no need to verify the client and if there are no intermediate CAs for the server certificate.

`{dh, public_key:der_encoded()}`

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

`{dhfile, path()}`

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.

`{verify, verify_type()}`

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

`{fail_if_no_peer_cert, boolean()}`

Used together with `{verify, verify_peer}` by an SSL 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.

`{reuse_sessions, boolean()}`

Specifies if the server is to agree to reuse sessions when requested by the clients. See also option `reuse_session`.

`{reuse_session, fun(SuggestedSessionId, PeerCert, Compression, CipherSuite) -> boolean()}`

Enables the SSL 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()`.

`{alpn_preferred_protocols, [binary()]}`

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.

`{next_protocols_advertised, Protocols :: [binary()]}`

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.

`{psk_identity, string()}`

Specifies the server identity hint, which the server presents to the client.

`{log_alert, boolean()}`

If set to `false`, error reports are not displayed.

`{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(), [ssl_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, SNIfun::fun()}`

If the server receives a SNI (Server Name Indication) from the client, the given function will be called to retrieve `[ssl_option()]` for the indicated server. These options will be merged into predefined `[ssl_option()]`. The function should be defined as: `fun(ServerName :: string()) -> [ssl_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.

`{signature_algs, [{hash(), ecdsa | rsa | dsa}]}`

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

`{v2_hello_compatible, boolean()}`

If true, the server accepts clients that send hello messages on SSL-2.0 format but offers supported SSL/TLS versions. Defaults to false, that is the server will not interoperate with clients that offers SSL-2.0.

General

When an SSL socket is in active mode (the default), data from the socket is delivered to the owner of the socket in the form of messages:

- `{ssl, Socket, Data}`
- `{ssl_closed, Socket}`
- `{ssl_error, Socket, Reason}`

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

Exports

`cipher_suites() ->`
`cipher_suites(Type) -> ciphers()`

Types:

`Type = erlang | openssl | all`

Returns a list of supported cipher suites. `cipher_suites()` is equivalent to `cipher_suites(erlang)`. Type `openssl` is provided for backwards compatibility with the old SSL, which used OpenSSL. `cipher_suites(all)` returns all available cipher suites. The cipher suites not present in `cipher_suites(erlang)` but included in `cipher_suites(all)` are not used unless explicitly configured by the user.

`clear_pem_cache() -> ok`

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

`connect(Socket, SslOptions) ->`
`connect(Socket, SslOptions, Timeout) -> {ok, SslSocket} | {error, Reason}`

Types:

`Socket = socket()`
`SslOptions = [ssl_option()]`
`Timeout = integer() | infinity`
`SslSocket = sslsocket()`
`Reason = term()`

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

`connect(Host, Port, Options) ->`
`connect(Host, Port, Options, Timeout) -> {ok, SslSocket} | {error, Reason}`

Types:

`Host = host()`
`Port = integer()`
`Options = [option()]`
`Timeout = integer() | infinity`
`SslSocket = sslsocket()`
`Reason = term()`

Opens an SSL connection to `Host`, `Port`.

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

Types:

`SslSocket = sslsocket()`
`Reason = term()`

Closes an SSL connection.

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

Types:

SslSocket = sslsocket()

How = timeout() | {NewController::pid(), timeout()}

Reason = term()

Closes or downgrades an SSL 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}]`

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

Types:

SslSocket = sslsocket()

NewOwner = pid()

Reason = term()

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:

Item = protocol | cipher_suite | sni_hostname | atom()

Meaningful atoms, not specified above, are the ssl option names.

Result = [{Item::atom(), Value::term()}]

Reason = term()

Returns all relevant information about the connection, ssl options that are undefined will be filtered out.

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

Types:

Items = [Item]

Item = protocol | cipher_suite | sni_hostname | atom()

Meaningful atoms, not specified above, are the ssl option names.

Result = [{Item::atom(), Value::term()}]

Reason = term()

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

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

format_error(Reason) -> string()

Types:

Reason = term()

Presents the error returned by an SSL function as a printable string.

getopts(Socket, OptionNames) -> {ok, [socketoption()]} | {error, Reason}

Types:

Socket = sslsocket()

OptionNames = [atom()]

Gets the values of the specified socket options.

getstat(Socket) -> {ok, OptionValues} | {error, inet:posix()}

getstat(Socket, OptionNames) -> {ok, OptionValues} | {error, inet:posix()}

Types:

Socket = sslsocket()

OptionNames = [atom()]

OptionValues = [{inet:stat_option(), integer()}]

Gets one or more statistic options for the underlying TCP socket.

See inet:getstat/2 for statistic options description.

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

Types:

Port = integer()

Options = options()

ListenSocket = sslsocket()

Creates an SSL listen socket.

negotiated_protocol(Socket) -> {ok, Protocol} | {error, protocol_not_negotiated}

Types:

Socket = sslsocket()

Protocol = binary()

Returns the protocol negotiated through ALPN or NPN extensions.

peercert(Socket) -> {ok, Cert} | {error, Reason}

Types:

Socket = sslsocket()

Cert = binary()

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

peername(Socket) -> {ok, {Address, Port}} | {error, Reason}

Types:

Socket = sslsocket()

Address = ipaddress()

Port = integer()

Returns the address and port number of the peer.

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

Types:

```
Socket = sslsocket()
Secret = binary() | master_secret
Label = binary()
Seed = [binary() | prf_random()]
WantedLength = non_neg_integer()
```

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.

Can only be used with TLS connections; `{error, undefined}` is returned for SSLv3 connections.

recv(Socket, Length) ->
recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}

Types:

```
Socket = sslsocket()
Length = integer()
Timeout = integer()
Data = [char()] | binary()
```

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(Socket) -> ok | {error, Reason}

Types:

```
Socket = 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.

send(Socket, Data) -> ok | {error, Reason}

Types:

```
Socket = sslsocket()
Data = iodata()
```

Writes `Data` to `Socket`.

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

setopts(Socket, Options) -> ok | {error, Reason}

Types:

```
Socket = sslsocket()
Options = [socketoption]()
```

Sets options according to `Options` for socket `Socket`.

`shutdown(Socket, How) -> ok | {error, Reason}`

Types:

```
Socket = sslsocket()
How = read | write | read_write
Reason = reason()
```

Immediately closes a socket in one or two directions.

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

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

`ssl_accept(Socket) ->`
`ssl_accept(Socket, Timeout) -> ok | {error, Reason}`

Types:

```
Socket = sslsocket()
Timeout = integer()
Reason = term()
```

Performs the SSL/TLS server-side handshake.

`Socket` is a socket as returned by *ssl:transport_accept/[1,2]*

`ssl_accept(Socket, SslOptions) ->`
`ssl_accept(Socket, SslOptions, Timeout) -> {ok, Socket} | ok | {error, Reason}`

Types:

```
Socket = socket() | sslsocket()
SslOptions = [ssl_option()]
Timeout = integer()
Reason = term()
```

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

> Warning:
>
> The listen socket is to be in mode `{active, false}` before telling the client that the server is ready to upgrade by calling this function, else the upgrade succeeds or does not succeed depending on timing.

If `Socket` is an `sslsocket()`: provides extra SSL/TLS options to those specified in *ssl:listen/2* and then performs the SSL/TLS handshake.

`sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}`

Types:

```
Socket = sslsocket()
```

```
Address = ipaddress()
Port = integer()
```

Returns the local address and port number of socket `Socket`.

```
start() ->
start(Type) -> ok | {error, Reason}
```

Types:

```
Type = permanent | transient | temporary
```

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

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

Stops the SSL application.

```
transport_accept(ListenSocket) ->
transport_accept(ListenSocket, Timeout) -> {ok, NewSocket} | {error, Reason}
```

Types:

```
ListenSocket = NewSocket = sslsocket()
Timeout = integer()
Reason = reason()
```

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

> Warning:
>
> The socket returned can only be used with *ssl:ssl_accept[2,3]*. No traffic can be sent or received before that call.

The accepted socket inherits the options set for `ListenSocket` in *ssl: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() -> [versions_info()]
```

Types:

```
versions_info() = {app_vsn, string()} | {supported | available,
[protocol()]
```

Returns version information relevant for the SSL application.

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

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

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

SEE ALSO

inet(3) and *gen_tcp(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.

Exports

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

Types:

Entries = *http_uri:uri()* | {file, string()} | {der, [*public_key:der_encoded()*]}

Reason = term()

Delete CRLs from the ssl applications local cache.

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

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

Types:

CRLSrc = {file, string()} | {der, [*public_key:der_encoded()*]}

URI = *http_uri:uri()*

Reason = term()

Insert CRLs into the ssl applications local cache.

ssl_crl_cache_api

Erlang module

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

DATA TYPES

The following data types are used in the functions below:

`cache_ref() =`
opaque()
`dist_point() =`
#'DistributionPoint'{} see *X509 certificates records*

Exports

`fresh_crl(DistributionPoint, CRL) -> FreshCRL`

Types:

`DistributionPoint = dist_point()`
`CRL = [public_key:der_encoded()]`
`FreshCRL = [public_key:der_encoded()]`

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

`lookup(DistributionPoint, Issuer, DbHandle) -> not_available | CRLs`
`lookup(DistributionPoint, DbHandle) -> not_available | CRLs`

Types:

`DistributionPoint = dist_point()`
`Issuer = public_key:issuer_name()`
`DbHandle = cache_ref()`
`CRLs = [public_key:der_encoded()]`

Lookup the CRLs belonging to the distribution point `Distributionpoint`. This function may choose to only look in the cache or to follow distribution point links depending on how the cache is administrated.

The `Issuer` argument contains the issuer name of the certificate to be checked. Normally the returned CRL should be issued by this issuer, except if the `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.

`select(Issuer, DbHandle) -> CRLs`

Types:

`Issuer = public_key:issuer_name()`

```
DbHandle = cache_ref()
```

Select the CRLs in the cache that are issued by `Issuer`

ssl_session_cache_api

Erlang module

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

DATA TYPES

The following data types are used in the functions for `ssl_session_cache_api`:

```
cache_ref() =
    opaque()
key() =
    {partialkey(), session_id()}
partialkey() =
    opaque()
session_id() =
    binary()
session() =
    opaque()
```

Exports

`delete(Cache, Key) -> _`

Types:

`Cache = cache_ref()`
`Key = key()`

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

`foldl(Fun, Acc0, Cache) -> Acc`

Types:

Calls `Fun(Elem, AccIn)` on successive elements of the cache, starting with `AccIn == Acc0`. `Fun/2` must return a new accumulator, which is passed to the next call. The function returns the final value of the accumulator. `Acc0` is returned if the cache is empty.

`init(Args) -> opaque()`

Types:

`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.

`lookup(Cache, Key) -> Entry`

Types:

```
Cache = cache_ref()
Key = key()
Entry = session() | undefined
```

Looks up a cache entry. Is to be callable from any process.

`select_session(Cache, PartialKey) -> [session()]`

Types:

```
Cache = cache_ref()
PartialKey = partialkey()
Session = session()
```

Selects sessions that can be reused. Is to be callable from any process.

`terminate(Cache) -> _`

Types:

```
Cache = term() - as returned by init/0
```

Takes care of possible cleanup that is needed when the cache handling process terminates.

`update(Cache, Key, Session) -> _`

Types:

```
Cache = cache_ref()
Key = key()
Session = session()
```

Caches a new session or updates an already cached one. Is only called from the cache handling process.