Secure Socket Layer

Secure Socket Layer 5.3.1
November 21, 2013

November 21, 2013

1 SSL User's Guide

The *SSL* application provides secure communication over sockets.

1.1 Transport Layer Security (TLS) and its predecessor, Secure Socket Layer (SSL)

The erlang SSL application currently implements the protocol SSL/TLS for currently supported versions see *ssl(3)*

By default erlang SSL is run over the TCP/IP protocol even though you could plug in any other reliable transport protocol with the same API as gen_tcp.

If a client and server wants to use an upgrade mechanism, such as defined by RFC2817, to upgrade a regular TCP/IP connection to an SSL connection the erlang SSL API supports this. This can be useful for things such as supporting HTTP and HTTPS on the same port and implementing virtual hosting.

1.1.1 Security overview

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

1.1.2 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, a public and a private one) 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 that uses a keyed-hash MAC (Message Authenticity Code), or HMAC, to protect the message's 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."

1.1.3 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 build by having the issuer in its turn being certified by an other certificate and so on until you reach the so called root certificate that is self signed i.e. issued by itself.

Certificates are issued by *certification authorities* (*CA*s) only. There are a handful of top CAs in the world that issue root certificates. You can examine the certificates of several of them by clicking through the menus of your web browser.

1.1.4 Authentication of Sender

Authentication of the sender is done by public key path validation as defined in RFC 3280. Simplified that means that each certificate in the certificate chain is issued by the one before, the certificates attributes are valid ones, and the root cert is a trusted cert that is present in the trusted certs database kept by the peer.

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

The client may choose to accept some path evaluation errors for instance a web browser may ask the user if they want to accept an unknown CA root certificate. The server, if it request a certificate, will on the other hand not accept any path validation errors. It is configurable if the server should accept or reject an "empty" certificate as response to a certificate request.

1.1.5 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 will be lost at application restart or takeover. Users may define their own callback module to handle session data storage if persistent data storage is required. Session data will also be invalidated after 24 hours from it was saved, for security reasons. It is of course possible to configure the amount of time the session data should be saved.

SSL clients will by default try to reuse an available session, SSL servers will by default agree to reuse sessions when clients ask to do so.

1.2 Using the SSL API

1.2.1 General information

To see relevant version information for ssl you can call ssl:versions/0

To see all supported cipher suites call ssl:cipher_suites/0. Note that available cipher suites for a connection will depend on your certificate. It is also possible to specify a specific cipher suite(s) that you want your connection to use. Default is to use the strongest available.

1.2.2 Setting up connections

Here follows some small example of how to set up client/server connections using the erlang shell. The returned value of the sslsocket has been abbreviated with `[...]` as it can be fairly large and is opaque.

Minmal example

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

Start server side

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

Create an ssl listen socket

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

Do a transport accept on the ssl listen socket

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

Start client side

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

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

Do the ssl handshake

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

Send a messag over ssl

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

Flush the shell message queue to see that we got the message 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 have to aggre to do so. Agreement may be accompliced by using a protocol such the one used by HTTP specified in RFC 2817.

Start server side

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

Create a normal tcp listen socket

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

Accept client connection

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

Start client side

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

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

Make sure active is set to false before trying to upgrade a connection to an ssl connection, otherwhise ssl handshake messages may be deliverd to the wrong process.

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

Do the ssl handshake.

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

Upgrade to an ssl connection. Note that the client and server must agree upon the upgrade and 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,[...]}}
```

Send a messag over ssl

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

Set active true on the ssl socket

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

Flush the shell message queue to see that we got the message sent on the client side

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

1.3 Using SSL for Erlang Distribution

This chapter describes how the Erlang distribution can use SSL to get additional verification and security.

1.3.1 Introduction

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

In the SSL application there is an additional distribution module, `inet_tls_dist` which can be used as an alternative. All distribution connections will be using 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.

Setting up Erlang distribution over SSL involves some simple but necessary steps:

- Building boot scripts including the SSL application
- Specifying the distribution module for net_kernel
- Specifying security options and other SSL options

The rest of this chapter describes the above mentioned steps in more detail.

1.3.2 Building boot scripts including the SSL application

Boot scripts are built using the `systools` utility in the SASL application. Refer to the SASL documentations for more information on systools. 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 Erlang distributions bin directory. The source for the script can be found under the Erlang installation top directory under `releases/<OTP version>/start_clean.rel`. Copy that script to another location (and preferably another name) and add the applications crypto, public_key and SSL with their current version numbers after the STDLIB application.

An example .rel file with SSL added may look like this:

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

Note that the version numbers surely will differ in your system. Whenever one of the applications included in the script is upgraded, the script has to be changed.

Assuming the above .rel file is stored in a file `start_ssl.rel` in the current directory, a boot script can be built like this:

```
1> systools:make_script("start_ssl",[]).
```

There will now be a file `start_ssl.boot` in the current directory. To test the boot script, start Erlang with the `-boot` command line parameter specifying this boot script (with its full path but without the `.boot` suffix), in Unix it could look like this:

```
$ 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 really started.

As an alternative to building a bootscript, one can explicitly add the path to the SSL `ebin` directory on the command line. This is done with the 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 applications code can be reached, the distribution will start. The `-pa` method is only recommended for testing purposes.

> **Note:**
>
> Note that the clone of the SSL application is necessary to enable the use of the SSL code in such an early bootstage as needed to setup the distribution, however this will make it impossible to soft upgrade the SSL application.

1.3.3 Specifying distribution module for net_kernel

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

Extending the command line from above gives us the following:

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

For the distribution to actually be started, we need to 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>
```

Note however that a node started in this way will refuse to talk to other nodes, as no ssl parameters are supplied (see below).

1.3.4 Specifying SSL options

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

On the `erl` command line one can specify options that the SSL distribution will add when creating a socket.

One can specify the simpler SSL options certfile, keyfile, password, cacertfile, verify, reuse_sessions, secure_renegotiate, depth, hibernate_after and ciphers (use old string format) by adding the prefix server_ or client_ to the option name. The server can also take the options dhfile and fail_if_no_peer_cert (also prefixed). `client_`-prfixed options are used when the distribution initiates a connection to another node and the `server_`-prefixed options are used when accepting a connection from a remote node.

More complex options such as verify_fun are not available at the moment but a mechanism to handle such options may be added in a future release.

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 should be followed by pairs of SSL options and their values. The `-ssl_dist_opt` argument can be repeated any number of times.

An example command line would now look something like this (line breaks in the command are for readability, they should 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 will be fully functional, using SSL as the distribution protocol.

1.3.5 Setting up environment to always use SSL

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

In a Unix (Bourne) shell it could look like this (line breaks for readability, they should not 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.

2 Reference Manual

The *SSL* application provides secure communication over sockets.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see *Licenses*.

ssl

Application

DEPENDENCIES

The ssl application uses the Erlang applications public_key and crypto to handle public keys and encryption, hence these applications needs to be loaded for the ssl application to work. In an embedded environment that means they need to be started with application:start/[1,2] before the ssl application is started.

ENVIRONMENT

The following application environment configuration parameters are defined for the SSL application. See *application(3)*for more information about configuration parameters.

Note that the environment parameters can be set on the command line, for instance,

```
erl ... -ssl protocol_version '[sslv3, tlsv1]' ....
```

`protocol_version = [sslv3|tlsv1] <optional>`.

Protocol that will be supported by started clients and servers. If this option is not set it will default to all protocols currently supported by the erlang ssl application. Note that this option may be overridden by the version option to ssl:connect/[2,3] and ssl:listen/2.

`session_lifetime = integer() <optional>`

The lifetime of session data in seconds.

`session_cb = atom() <optional>`

Name of session cache callback module that implements the ssl_session_cache_api behavior, defaults to ssl_session_cache.erl.

`session_cb_init_args = list() <optional>`

List of arguments to the init function in session cache callback module, defaults to [].

SEE ALSO

application(3)

ssl

Erlang module

This module contains interface functions to the Secure Socket Layer.

SSL

- ssl requires the crypto and public_key applications.
- Supported SSL/TLS-versions are SSL-3.0, TLS-1.0, TLS-1.1 and TLS-1.2.
- For security reasons sslv2 is not supported.
- Ephemeral Diffie-Hellman cipher suites are supported but not Diffie Hellman Certificates cipher suites.
- Elliptic Curve cipher suites are supported if crypto 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 spec so there is not any real motivation to implement them.
- CRL and policy certificate extensions are not supported yet. However CRL verification is supported by public_key, only not integrated in ssl yet.

COMMON DATA TYPES

The following data types are used in the functions below:

```
boolean() = true | false
```

```
option() = socketoption() | ssloption() | transportoption()
```

```
socketoption() = proplists:property() - The default socket options are [{mode,list},{packet, 0},{header, 0},{active, true}].
```

For valid options see *inet(3)* and *gen_tcp(3)*.

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

```
transportoption() = {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 and behave as gen_tcp and in addition 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 will call inet directly.
```

```
CallbackModule = atom()
```

```
DataTag = atom() - tag used in socket data message.
```

```
ClosedTag = atom() - tag used in socket close message.
```

```
verify_type() = verify_none | verify_peer

path() = string() - representing a file path.

der_encoded() = binary() -Asn1 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 to the user.

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

ciphers() = [ciphersuite()] | string() (according to old API)

ciphersuite() = {key_exchange(), cipher(), 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

hash() = md5 | sha

prf_random() = client_random | server_random

srp_param_type() = srp_1024 | srp_1536 | srp_2048 | srp_3072 | srp_4096 |
srp_6144 | srp_8192
```

SSL OPTION DESCRIPTIONS - COMMON for SERVER and CLIENT

Options described here are options that are have the same meaning in the client and the server.

{cert, der_encoded()}
: The DER encoded users certificate. If this option is supplied it will override the certfile option.

{certfile, path()}
: Path to a file containing the user's certificate.

{key, {'RSAPrivateKey'| 'DSAPrivateKey' | 'ECPrivateKey' |'PrivateKeyInfo', der_encoded()}}
: The DER encoded users private key. If this option is supplied it will override the keyfile option.

{keyfile, path()}
: Path to file containing user's private PEM encoded key. As PEM-files may contain several entries this option defaults to the same file as given by certfile option.

{password, string()}
: String containing the user's password. Only used if the private keyfile is password protected.

{cacerts, [der_encoded()]}
: The DER encoded trusted certificates. If this option is supplied it will override the cacertfile option.

{cacertfile, path()}
: Path to file containing PEM encoded CA certificates (trusted certificates used for verifying a peer certificate). May be omitted if you do not want to verify the peer.

{ciphers, ciphers()}
: The cipher suites that should be supported. The function `cipher_suites/0` can be used to find all ciphers that are supported by default. `cipher_suites(all)` may be called to find all available cipher suites. Pre-Shared Key (**RFC 4279** and **RFC 5487**), Secure Remote Password (**RFC 5054**) and anonymous cipher suites only work if explicitly enabled by this option and they are supported/enabled by the peer also. Note that anonymous cipher suites are supported for testing purposes only and should not be used when security matters.

{ssl_imp, new | old}
: No longer has any meaning as the old implementation has been removed, it will be ignored.

{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 i.e. secure renegotiation will be used if possible but it will fallback to unsecure renegotiation if the peer does not support **RFC 5746**.

{depth, integer()}

The depth is the maximum number of non-self-issued intermediate certificates that may 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 it is 2 PEER, CA, CA, ROOT-CA and so on. The default value is 1.

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

The verification fun should be defined as:

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

The verify fun will be called during the X509-path validation when an error or an extension unknown to the ssl application is encountered. Additionally it will be called when a certificate is considered valid by the path validation to allow access to each certificate in the path to the user application. Note that it will differentiate between the peer certificate and CA certificates by using valid_peer or valid as the second argument to the verify 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 and an alert is sent to the peer and the TLS/SSL handshake is terminated. If the verify callback fun returns {valid, UserState}, the verification process is continued. If the verify callback fun always returns {valid, UserState}, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. If called with an extension unknown to the user application the return value {unknown, UserState} should be used.

The default verify_fun option in verify_peer mode:

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

The default verify_fun option in verify_none mode:

```
{fun(_,{bad_cert, _}, UserState) ->
  {valid, UserState};
    (_,{extension, _}, UserState) ->
  {unknown, UserState};
    (_, valid, UserState) ->
  {valid, UserState};
    (_, valid_peer, UserState) ->
         {valid, UserState}
```

```
end, []}
```

Possible path validation errors:

{bad_cert, cert_expired}, {bad_cert, invalid_issuer}, {bad_cert, invalid_signature}, {bad_cert, unknown_ca}, {bad_cert, selfsigned_peer}, {bad_cert, name_not_permitted}, {bad_cert, missing_basic_constraint}, {bad_cert, invalid_key_usage}

{versions, [protocol()]}

TLS protocol versions that will be 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 supported by the SSL application. See also *ssl(6)*

{hibernate_after, integer()|undefined}

When an integer-value is specified, the `ssl_connection` will go 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 will never go into hibernation.

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

The lookup fun should be defined as:

```
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 will be called by the client and server to determine the shared secret. When called by the client, PSKIdentity will be set to the hint presented by the server or undefined. When called by the server, PSKIdentity is the identity presented by the client.

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

SSL OPTION DESCRIPTIONS - CLIENT SIDE

Options described here are client specific or has a slightly different meaning in the client than in the server.

{verify, verify_type()}

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

{reuse_sessions, boolean()}

Specifies if client should try to reuse sessions when possible.

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

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

Indicates the client will try to perform Next Protocol Negotiation.

If precedence is server the negotiated protocol will be the first protocol that appears on the server advertised list that is also on the client preference list.

If precedence is client the negotiated protocol will be the first protocol that appears on the client preference list that 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 will fallback to the first protocol in its list or if a default is supplied it will fallback to that instead. If the server does not support Next Protocol Negotiation the connection will be aborted 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 the user_look_fun.

{srp_identity, {Username :: string(), Password :: string()}
Specifies the Username and Password to use to authenticate to the server.

SSL OPTION DESCRIPTIONS - SERVER SIDE

Options described here are server specific or has a slightly different meaning in the server than in the client.

{dh, der_encoded()}
The DER encoded Diffie Hellman parameters. If this option is supplied it will override the dhfile option.

{dhfile, path()}
Path to file containing PEM encoded Diffie Hellman parameters, for the server to use if a cipher suite using Diffie Hellman key exchange is negotiated. If not specified default parameters will be used.

{verify, verify_type()}
Servers only do the x509-path validation in verify_peer mode, as it then will send a certificate request to the client (this message is not sent if the verify option is verify_none) and you may then also want to specify the 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 will fail if the client does not have a certificate to send, i.e. sends a empty certificate, if set to false it will only fail if the client sends an invalid certificate (an empty certificate is considered valid).

{reuse_sessions, boolean()}
Specifies if the server should agree to reuse sessions when the clients request to do so. See also the reuse_session option.

{reuse_session, fun(SuggestedSessionId, PeerCert, Compression, CipherSuite) -> boolean()}
Enables the ssl server to have a local policy for deciding if a session should be reused or not, only meaningful 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().

{next_protocols_advertised, Protocols :: [binary()]}
The list of protocols to send to the client if the client indicates it supports the Next Protocol extension. The client may 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 `negotiated_next_protocol/1` method.

{psk_identity, string()}
Specifies the server identity hint the server presents to the client.

{log_alert, boolean()}
If false, error reports will not be displayed.

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 timeout in milliseconds. The default value for a `Timeout` argument 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 old ssl that used openssl. cipher_suites(all) returns all available cipher suites. The cipher suites not present in cipher_suites(erlang) but in included in cipher_suites(all) will not be used unless explicitly configured by the user.

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

Types:

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

Upgrades a gen_tcp, or equivalent, connected socket to an ssl socket i.e. 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()

Close an ssl connection.

```
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_info(SslSocket) -> {ok, {ProtocolVersion, CipherSuite}} | {error,
Reason}
```

Types:

CipherSuite = ciphersuite()
ProtocolVersion = protocol()

Returns the negotiated protocol version and cipher suite.

```
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()]

Get the value of the specified socket options.

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

Types:

Port = integer()
Options = options()
ListenSocket = sslsocket()

Creates an ssl listen socket.

```
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.

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

Types:

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

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

The `Length` argument is only meaningful when the socket is in `raw` mode 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.

The optional `Timeout` parameter specifies a timeout in milliseconds. The default value is `infinity`.

```
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()

Use the pseudo random function (PRF) of a TLS session to generate additional key material. It either takes user generated values for `Secret` and `Seed` or atoms directing it use a specific value from the session security parameters.

This function can only be used with TLS connections, `{error, undefined}` is returned for SSLv3 connections.

```
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 the socket `Socket`.

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

Types:

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

Immediately close 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, the `{exit_on_close, false}` option is useful.

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

Types:

ListenSocket = sslsocket()
Timeout = integer()
Reason = term()

The `ssl_accept` function establish the SSL connection on the server side. It should be called directly after `transport_accept`, in the spawned server-loop.

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

Types:

ListenSocket = socket()
SslOptions = ssloptions()
Timeout = integer()
Reason = term()

Upgrades a gen_tcp, or equivalent, socket to an ssl socket i.e. performs the ssl server-side handshake.

> ### Warning:
>
> Note that the listen socket should be in {active, false} mode before telling the client that the server is ready to upgrade and calling this function, otherwise the upgrade may or may not succeed depending on timing.

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

Types:

Socket = sslsocket()
Address = ipaddress()
Port = integer()

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

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

Types:

`Type = permanent | transient | temporary`

Starts the Ssl application. Default type is temporary. *application(3)*

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

Stops the Ssl application. *application(3)*

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

Types:

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

Accepts an incoming connection request on a listen socket. `ListenSocket` must be a socket returned from `listen/2`. The socket returned should be passed to `ssl_accept` to complete ssl handshaking and establishing the connection.

> Warning:
>
> The socket returned can only be used with `ssl_accept`, no traffic can be sent or received before that call.

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() -> [{SslAppVer, SupportedSslVer, AvailableSslVsn}]
```

Types:

`SslAppVer = string()`
`SupportedSslVer = [protocol()]`
`AvailableSslVsn = [protocol()]`

Returns version information relevant for the ssl application.

```
negotiated_next_protocol(Socket) -> {ok, Protocol} | {error,
next_protocol_not_negotiated}
```

Types:

`Socket = sslsocket()`
`Protocol = binary()`

Returns the Next Protocol negotiated.

SEE ALSO

inet(3) and *gen_tcp(3)*

ssl_session_cache_api

Erlang module

Common Data Types

The following data types are used in the functions below:

```
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. Will only be 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() -> opaque()
```

Types:

Performs possible initializations of the cache and returns a reference to it that will be used as parameter to the other api functions. Will be called by the cache handling processes init function, hence putting the same requirements on it as a normal process init function.

```
lookup(Cache, Key) -> Entry
```

Types:

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

Looks up a cache entry. Should be callable from any process.

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

Types:

Cache = cache_ref()

PartialKey = partialkey()
Session = session()

Selects sessions that could be reused. Should 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 a already cached one. Will only be called from the cache handling process.