Secure Socket Layer

version 2.3

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

1 SSL User's Guide 1

1.1 Erlang Distribution Using SSL 1

1.1.1 Introduction 1

1.1.2 Building boot scripts including the SSL application 2

1.1.3 Specifying distribution module for net_kernel 3

1.1.4 Specifying security options and other SSL options 3

1.1.5 Setting up environment to always use SSL 4

1.2 SSL Release Notes 4

1.2.1 SSL 2.3.4 5

1.2.2 SSL 2.3.3 5

1.2.3 SSL 2.3.2 5

1.2.4 SSL 2.3.1 5

1.2.5 SSL 2.3 6

1.2.6 SSL 2.2.1 6

1.2.7 SSL 2.2 6

1.2.8 SSL 2.1 6

1.2.9 SSL 2.0 7

2 SSL Reference Manual 9

2.1 ssl 11

2.2 ssl 13

2.3 ssl_socket 19

Chapter 1

SSL User's Guide

The *SSL* application provides secure communication over sockets.

1.1 Erlang Distribution Using SSL

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

1.1.1 Introduction

The Erlang distribution can in theory use almost any connection based protocol as bearer. A module that implements the protocol specific parts of 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_ssl_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 depends on how the connections are set up, one can use key files or certificates to just get a crypted connection. One can also make the SSL package verify the certificates of other nodes to get additional security. 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.

1.1.2 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 SSL application with its current version number after the STDLIB application.

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

```
{release, {"OTP  APN 181 01","P7A"}, {erts, "5.0"},
 [{kernel,"2.5"},
  {stdlib,"1.8.1"},
  {ssl,"2.2.1"}]}.
```

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_server).
<0.32.0>
```

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 really need not be started for the distribution to come up, a primitive version of the ssl server is started by the distribution module itself, so as long as the primitive code server can reach the code, the distribution will start. The `-pa` method is only recommended for testing purpouses.

1.1.3 Specifying distribution module for net_kernel

The distribution module for SSL is named `inet_ssl_dist` and is specified on the command line whit 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_ssl` on the command line.

Extending the command line from above gives us the following:

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

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_ssl -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 certificates or key files are supplied (see below).

When the SSL distribution starts, the OTP system is in its early boot stage, why neither `application` nor `code` are usable. As SSL needs to start a port program in this early stage, it tries to determine the path to that program from the primitive code loaders code path. If this fails, one need to specify the directory where the port program resides. This can be done either with an environment variable `ERL_SSL_PORTPROGRAM_DIR` or with the command line option `-ssl_portprogram_dir`. The value should be the directory where the `ssl_esock` port program is located. Note that this option is never needed in a normal Erlang installation.

1.1.4 Specifying security options and other SSL options

For SSL to work, you either need certificate files or a key file. Certificate files can be specified both when working as client and as server (connecting or accepting).

On the `erl` command line one can specify options that the ssl distribution will add when creation a socket. It is mandatory to specify at least a key file or client and server certificates. One can specify any *SSL option* on the command line, but must not specify any socket options (like packet size and such). The SSL options are listed in the Reference Manual. The only difference between the options in the reference manual and the ones that can be specified to the distribution on the command line is that `certfile` can (and usually needs to) be specified as `client_certfile` and `server_certfile`. The `client_certfile` is used when the distribution initiates a connection to another node and the `server_cerfile` is used when accepting a connection from a remote node.

The command line argument for specifying the SSL options is named `-ssl_dist_opt` and should be followed by an even number of SSL options/option 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_ssl
  -ssl_dist_opt client_certfile "/home/me/ssl/erlclient.pem"
  -ssl_dist_opt server_certfile "/home/me/ssl/erlserver.pem"
  -ssl_dist_opt verify 1 depth 1
  -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.

1.1.5 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):

```
$ ERL_FLAGS="-boot \"/home/me/ssl/start_ssl\" -proto_dist inet_ssl
  -ssl_dist_opt client_certfile \"/home/me/ssl/erlclient.pem\"
  -ssl_dist_opt server_certfile \"/home/me/ssl/erlserver.pem\"
  -ssl_dist_opt verify 1 -ssl_dist_opt depth 1"
$ 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_ssl"]},
 {ssl_dist_opt,["client_certfile","/home/me/ssl/erlclient.pem"]},
 {ssl_dist_opt,["server_certfile","/home/me/ssl/erlserver.pem"]},
 {ssl_dist_opt,["verify","1"]},
 {ssl_dist_opt,["depth","1"]},
 {home,["/home/me"]}]
```

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

1.2 SSL Release Notes

This document describes the changes made to the SSL application.

1.2.1 SSL 2.3.4

Improvements and New Features

- All TCP options allowed in gen_tcp, are now also allowed in SSL, except the option {`reuseaddr`, `Boolean`}. A new function `getopts` has been added to the SSL interface module `ssl`.
  OwnId: OTP-4305, OTP-4159

1.2.2 SSL 2.3.3

Fixed Bugs and Malfunctions

- The roles of the SSLeay and OpenSSL packages has been clarified in the ssl(6) application manual page. Also the URLs from which to download SSLeay has been updated.
  OwnId: OTP-4002
  Aux Id: seq5269
- A call to `ssl:listen(Port, Options)` with `Options = []` resulted in the cryptic {`error`, `ebadf`} return value. The return value has been changed to {`error, enooptions`}, and the behaviour has been documented in the `listen/2` function.
  OwnId: OTP-4016
  Aux Id: seq7006
- Use of the option {`nodelay, boolean()`} crashed the `ssl_server`.
  OwnId: OTP-4070
  Aux Id:
- A bug caused the Erlang distribution over ssl to fail. This bug has now been fixed.
  OwnId: OTP-4072
  Aux Id:
- On Windows when the SSL port program encountered an error code not anticipated it crashed.
  OwnId: OTP-4132
  Aux Id:

1.2.3 SSL 2.3.2

Fixed Bugs and Malfunctions

- The `ssl:accept/1-2` function sometimes returned {`error, {What, Where}`} instead of {`error, What`}, where `What` is an atom.
  OwnId: OTP-3775
  Aux Id: seq4991

1.2.4 SSL 2.3.1

Fixed Bugs and Malfunctions

- Sometimes the SSL portprogram would loop in an accept loop, without terminating even when the SSL application was stopped..
  OwnId: OTP-3691

1.2.5 SSL 2.3

Functions have been added to SSL to experimentally support Erlang distribution.

1.2.6 SSL 2.2.1

The 2.2.1 version of SSL provides code replacement in runtime by upgrading from, or downgrading to, versions 2.1 and 2.2.

1.2.7 SSL 2.2

Improvements and New Features

- The restriction that only the creator of an SSL socket can read from and write to the socket has been lifted.
  OwnId: OTP-3301
- The option {`packet, cdr`} for SSL sockets has been added, which means that SSL sockets also supports CDR encoded packets.
  OwnId: OTP-3302

Known Bugs and Problems

- Setting of a CA certificate file with the `cacertfile` option (in calls to `ssl:accept/1/2` or `ssl:connect/3/4`) does not work due to weaknesses in the SSLeay package.
  A work-around is to set the OS environment variable `SSL_CERT_FILE` before SSL is started. However, then the CA certificate file will be global for all connections.
  OwnId: OTP-3146
- When changing controlling process of an SSL socket, a temporary process is started, which is not gen_server compliant.
  OwnId: OTP-3146
- Although there is a `cache` timeout option, it is silently ignored.
  OwnId: OTP-3146
- There is currently no way to restrict the cipher sizes.
  OwnId: OTP-3146

1.2.8 SSL 2.1

Improvements and New Features

- The set of possible error reasons has been extended to contain diagnostics on erronous certificates and failures to verify certificates.
  OwnId: OTP-3145
- The maximum number of simultaneous SSL connections on Windows has been increased from 31 to 127.
  OwnId: OTP-3145

Fixed Bugs and Malfunctions

- A dead-lock occuring when write queues are not empty has been removed.
  OwnId: OTP-3145
- Error reasons have been unified and changed.
  (** POTENTIAL INCOMPATIBILITY **)
  OwnId: OTP-3145
- On Windows a check of the existence of the environment variable `ERLSRV_SERVICE_NAME` has been added. If that variable is defined, the port program of the SSL application will not terminated when a user logs off.
  OwnId: OTP-3145
- An error in the setting of the `nodelay` option has been corrected.
  OwnId: OTP-3145
- The confounded notions of verify mode and verify depth has been corrected. The option `verifydepth` has been removed, and the two separate options `verify` and `depth` has been added.
  (** POTENTIAL INCOMPATIBILITY **)
  OwnId: OTP-3145

Known Bugs and Problems

- Setting of a CA certificate file with the `cacertfile` option (in calls to `ssl:accept/1/2` or `ssl:connect/3/4`) does not work due to weaknesses in the SSLeay package.
  A work-around is to set the OS environment variable `SSL_CERT_FILE` before SSL is started. However, then the CA certificate file will be global for all connections.
  OwnId: OTP-3146
- When changing controlling process of an SSL socket, a temporary process is started, which is not gen_server compliant.
  OwnId: OTP-3146
- Although there is a `cache` timeout option, it is silently ignored.
  OwnId: OTP-3146
- There is currently no way to restrict the cipher sizes.
  OwnId: OTP-3146

1.2.9 SSL 2.0

A complete new version of SSL with separate I/O channels for all connections with non-blocking I/O multiplexing.

SSL Reference Manual

Short Summaries

- Application ssl [page 11] – The SSL Application
- Erlang Module ssl [page 13] – Interface Functions for Secure Socket Layer
- Erlang Module ssl_socket [page 19] – Old interface to Secure Socket Layer

ssl

No functions are exported.

ssl

The following functions are exported:

- `accept(ListenSocket) -> {ok, Socket} | {error, Reason}`
  [page 15] Accept an incoming connection request.
- `accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}`
  [page 15] Accept an incoming connection request.
- `close(Socket) -> ok | {error, Reason}`
  [page 15] Close a socket returned by `accept/1/2`, `connect/3/4`, or `listen/2`.
- `connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}`
  [page 15] Connect to `Port` at `Address`.
- `connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}`
  [page 15] Connect to `Port` at `Address`.
- `controlling_process(Socket, NewOwner) -> ok | {error, Reason}`
  [page 15] Assign a new controlling process to the socket.
- `format_error(ErrorCode) -> string()`
  [page 15] Return an error string.
- `getopts(Socket, OptionsTags) -> {ok, Options} | {error, Reason}`
  [page 16] Get options set for socket
- `listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}`
  [page 16] Set up a socket to listen on a port on the local host.
- `peername(Socket) -> {ok, {Address, Port}} | {error, Reason}`
  [page 16] Return peer address and port.

- `pid(Socket) -> pid()`
  [page 16] Return the pid of the socket process.
- `port(Socket) -> {ok, Port}`
  [page 16] Return local port number of socket.
- `recv(Socket, Length) -> {ok, Data} | {error, Reason}`
  [page 17] Receive data on socket.
- `recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}`
  [page 17] Receive data on socket.
- `send(Socket, Data) -> ok | {error, Reason}`
  [page 17] Write data to a socket.
- `setopts(Socket, Options) -> ok | {error, Reason}`
  [page 17] Set socket options.
- `sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}`
  [page 17] Return the local address and port.

ssl_socket

The following functions are exported:

- `listen(Protocol, Family, Address, Mode)`
  [page 19] Set up a server listening to `Address`
- `accept(ListenSocket, SSLFlags)`
  [page 20] Accept an incoming connection
- `client(Protocol, Family, Address, Mode, SSLFlags)`
  [page 23] Set up a SSL client connection
- `controlling_process(Socket, Pid)`
  [page 24] Switch controlling process for a socket
- `peername(Socket)`
  [page 24] Return the name of the other end of a socket
- `resolve()`
  [page 24] Return the official name of the current host.
- `resolve(IPAddress)`
  [page 24] Return the official name of the host with a certain address
- `close(Socket)`
  [page 24] Close a socket
- `start()`
  [page 25] Start the socket server
- `stop()`
  [page 25] Stop the socket server

ssl

Application

The Secure Socket Layer (SSL) application provides secure socket communication over TCP/IP.

Environment

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

`debug = true | false <optional>` Causes debug information to be written to standard output. Default is `false`.

`debugdir = path() | false <optional>` Causes debug information output controlled by `debug` and `msgdebug` to be printed to a file named `ssl_esock.<pid>.log` in the directory specified by `debugdir`, where `<pid>` is the operating system specific textual representation of the process indentifier of the external port program of the SSL application. Default is `false`, i.e. no log file is produced.

`msgdebug = true | false <optional>` Sets `debug = true` and causes also the contents of low level messages to be printed to standard output. Default is `false`.

`port_program = string() | false <optional>` Name of port program. The default is `ssl_esock`.

`pproxylsport = integer() | false <optional>` Define the port number of the listen port of the SSL port program. Almost never is this option needed.

`pproxylsbacklog = integer() | false <optional>` Set the listen queue size of the listen port of the SSL port program. The default is 5.

SSL libraries

The current implementation of the SSL application is based on the *SSLeay* package version 0.9.0. It can be downloaded from several of the mirror sites listed at the site http://www.openssl.org[1]. For the relation between SSLeay and OpenSSL, see below.

The user has to fetch the SSLeay package, compile and install the libraries `libcrypto.so` and `libssl.so` (UNIX), or the libraries `libeay32.dll` and `ssleay32.dll` (WIN32). The WIN32 libraries must be compiled and linked with WinSock2.

In order to build SSLeay-0.9.0 for WinSock2 on Windows NT 4.0 do as follows:

[1] URL: `http://www.openssl.org`

1. In `crypto/bio/b_sock.c:int BIO_sock_init()` remove the call to `WSACancelBlockingCall()`.
2. In `crypto/bn/bn.h` replace `#define BN_ULLONG unsigned _int64` by `#define BN_ULLONG unsigned __int64`.
3. In `crypto/bn/bn_mulw.c:bn_add_words()` replace `return(ll&BN_MASK2);` by `return (BN_ULONG)(ll&BN_MASK2);`.
4. In `apps/s_socket.c:sock_cleanup()` remove call to `WSACancelBlockingCall()`.
5. In Configure replace `"VC-WIN32","cl:::BN_LLONG RC4_INDEX ".$x86_gcc_opts.":::"` by `"VC-WIN32","cl:::RC4_INDEX ".$x86_gcc_opts.":::"`.
6. In `mf-ddl.nt` replace `wsock32.lib` by `ws2_32.lib`.

The `ssl_esock` port program has to be built by linking object files and libraries. An example `Makefile` is provided in the `ssl-X.Y/priv/obj` directory, where also the object files are found.

SSLeay and OpenSSL

The last version of the SSLeay package was 0.9.0b. It was continued by the open source project OpenSSL, and its first release was 0.9.1c.

There should be no problems in using an OpenSSL release instead of the SSLeay 0.9.0 release on Unix (that has however not been tested). For WIN32 there are problems (even if you follow the procedure above). The OpenSSL support for WIN32 seems not to be whole-hearted; in particular the implimenation still relies on the now obsolete Winsock 1.1 interface.

Other SSL packages

There are also commercially available SSL libraries, e.g. *C/SSL* from Baltimore Technologies Ltd[2], and *SSL-C* from RSA Data Security Australia Pty Ltd[3], which may be supported by the SSL application in the future.

Restrictions

Users must be aware of export restrictions and patent rights concerning cryptographic software.

SEE ALSO

application(3)

[2] URL: `http://www.baltimoretechnologies.com/`
[3] URL: `http://www.rsasecurity.com.au/`

ssl

Erlang Module

This module contains interface functions to the Secure Socket Layer. New implementations shall use this module, and not the old `ssl_socket` module, which is obsolete.

Common data types

The following datatypes are used in the functions below:

- `options() = [option()]`
- `option() = socketoption() | ssloption()`
- `socketoption() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {nodelay, boolean()} | {active, activetype()} | {backlog, integer()} | {ip, ipaddress()}`
- `ssloption() = {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()} | {cachetimeout, integer()}`
- `packettype()` (see inet(3))
- `activetype()` (see inet(3))
- `reason() = atom() | {atom(), string()}`
- `bytes() = [byte()]`
- `string() = [byte()]`
- `byte() = 0 | 1 | 2 | ... | 255`
- `code() = 0 | 1 | 2`
- `depth() = byte()`
- `address() = hostname() | ipstring() | ipaddress()`
- `ipaddress() = ipstring() | iptuple()`
- `hostname() = string()`
- `ipstring() = string()`
- `iptuple() = {byte(), byte(), byte(), byte()}`
- `sslsocket()`
-

The socket options {backlog, integer()} and {ip, ipaddress} are for listen/2 only.

The following socket options are set by default: {mode, list}, {packet, 0}, {header, 0}, {nodelay, false}, {active, true}, {backlog, 5}, and {ip, {0,0,0,0}}.

Note that the options {mode, binary} and binary are equivalent. Similarly {mode, list and the absence of option binary are equivalent.

The ssl options are for setting specific SSL parameters as follows:

- {verify, code()} Specifies type of verification: 0 = do not verify peer; 1 = verify peer, verify client once, 2 = verify peer, verify client once, fail if no peer certificate. The default value is 0.
- {depth, depth()} Specifies verification depth, i.e. how far in a chain of certificates the verification process shall proceed before the verification is considered successful. The default value is 1.
- {certfile, path()} Path to a file containing a chain of PEM encoded certificates.
- {keyfile, path()} Path to file containing user's private PEM encoded key.
- {password, string()} String containing the user's password. Only used if the private keyfile is password protected.
- {cacertfile, path()} Path to file containing PEM encoded CA certificates.
- {ciphers, string()} String of ciphers as a colon separated list of ciphers.
- {cachetimeout, integer()} Session cache timeout in seconds.

The type sslsocket() is opaque to the user.

The owner of a socket is the one that created it by a call to accept/1, connect/3/4/, or listen/2.

When a 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.

Functions listed below may return the value {error, closed}, which only indicates that the SSL socket is considered closed for the operation in question. It is for instance possible to have {error, closed} returned from an call to send/2, and a subsequent call to recv/3 returning {ok, Data}.

Hence a return value of {error, closed} must not be interpreted as if the socket was completely closed. On the contrary, in order to free all resources occupied by an SSL socket, close/1 must be called, or else the process owning the socket has to terminate.

For each SSL socket there is an Erlang process representing the socket. When a socket is opened, that process links to the calling client process. Implementations that want to detect abnormal exits from the socket process by receiving {'EXIT', Pid, Reason} messages, should use the function pid/1 to retreive the process identifier from the socket, in order to be able to match exit messages properly.

Exports

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

Types:

- ListenSocket = Socket = sslsocket()
- Timeout = integer()

Accepts an incoming connection request on a listen socket. `ListenSocket` must be a socket returned from `listen/2`.

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.

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

Types:

- Socket = sslsocket()

Closes a socket returned by `accept/1/2`, `connect/3/4`, or `listen/2`

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

Types:

- Address = address()
- Port = integer()
- Options = [connect_option()]
- connect_option() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {nodelay, boolean()} | {active, activetype()} | {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()} | {cachetimeout, integer()}
- Timeout = integer()
- Socket = sslsocket()

Connects to `Port` at `Address`. If the optional `Timeout` argument is specified, and a connection could not be established within the given time, `{error, timeout}` is returned. The default value for `Timeout` is `infinity`.

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

Types:

- Socket = sslsocket()
- NewOwner = pid()

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

```
format_error(ErrorCode) -> string()
```

Types:

- ErrorCode = term()

Returns a diagnostic string describing an error.

`getopts(Socket, OptionsTags) -> {ok, Options} | {error, Reason}`

Types:

- Socket = sslsocket()
- OptionTags = [optiontag()]()

Returns the options the tags of which are `OptionTags` for for the socket `Socket`.

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

Types:

- Port = integer()
- Options = [listen_option()]
- listen_option() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {active, activetype()} | {backlog, integer()} | {ip, ipaddress()} | {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()} | {cachetimeout, integer()}
- ListenSocket = sslsocket()

Sets up a socket to listen on port `Port` at the local host. If `Port` is zero, `listen/2` picks an available port number (use `port/1` to retreive it).

The listen queue size defaults to 5. If a different value is wanted, the option {`backlog`, `Size`} should be added to the list of options.

An empty `Options` list is considered an error, and {`error`, `enooptions`} is returned.

The returned `ListenSocket` can only be used in calls to `accept/1/2`.

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

Types:

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

Returns the address and port number of the peer.

`pid(Socket) -> pid()`

Types:

- Socket = sslsocket()

Returns the pid of the socket process. The returned pid should only be used for receiving exit messages.

`port(Socket) -> {ok, Port}`

Types:

- Socket = sslsocket()
- Port = integer()

Returns the local port number of socket Socket.

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

Types:

- Socket = sslsocket()
- Length = integer() >= 0
- Timeout = integer()
- Data = bytes() | binary()

Receives data on socket Socket when the socket is in passive mode, i.e. when the option {active, false} has been specified.

A notable return value is {error, closed} which indicates that the socket is closed.

A positive value of the Length argument is only valid when the socket is in raw mode (option {packet, 0} is set, and the option binary is *not* set); otherwise it should be set to 0, whence all available bytes are returned.

If the optional Timeout parameter is specified, and no data was available within the given time, {error, timeout} is returned. The default value for Timeout is infinity.

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

Types:

- Socket = sslsocket()
- Data = iolist() | binary()

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.

```
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.

ERRORS

The possible error reasons and the corresponding diagnostic strings returned by `format_error/1` are either the same as those defined in the `inet(3)` reference manual, or as follows:

`closed` Connection closed for the operation in question.

`ebadsocket` Connection not found (internal error).

`ebadstate` Connection not in connect state (internal error).

`ebrokertype` Wrong broker type (internal error).

`ecacertfile` Own CA certificate file is invalid.

`ecertfile` Own certificate file is invalid.

`echaintoolong` The chain of certificates provided by peer is too long.

`ecipher` Own list of specified ciphers is invalid.

`ekeyfile` Own private key file is invalid.

`ekeymismatch` Own private key does not match own certificate.

`enoissuercert` Cannot find certificate of issuer of certificate provided by peer.

`enoservercert` Attempt to do accept without having set own certificate.

`enotlistener` Attempt to accept on a non-listening socket.

`enoproxysocket` No proxy socket found (internal error).

`enooptions` The list of options is empty.

`eoptions` Invalid list of options.

`epeercert` Certificate provided by peer is in error.

`epeercertexpired` Certificate provided by peer has expired.

`epeercertinvalid` Certificate provided by peer is invalid.

`eselfsignedcert` Certificate provided by peer is self signed.

`esslaccept` Server SSL handshake procedure between client and server failed.

`esslconnect` Client SSL handshake procedure between client and server failed.

`esslerrssl` SSL protocol failure. Typically because of a fatal alert from peer.

`ewantconnect` Protocol wants to connect, which is not supported in this version of the SSL application.

`ex509lookup` Protocol wants X.509 lookup, which is not supported in this version of the SSL application.

{`badcall, Call`} Call not recognized for current mode (active or passive) and state of socket.

{`badcast, Cast`} Call not recognized for current mode (active or passive) and state of socket.

{`badinfo, Info`} Call not recognized for current mode (active or passive) and state of socket.

SEE ALSO

gen_tcp(3), inet(3)

ssl_socket

Erlang Module

This manual describes the old interface to Secure Socket Layer. It should not be used for new development.

The information in this manual is not up-to-date, and will not be updated in the future. However, the following applies for the SSL 2.0 version: Windows and UNIX are supported; the "-log " option in SSLFlags is not supported anymore.

SSL Sockets are the secure BSD UNIX interface to communication protocols based on SSLeay library written by Eric Young (eay@mincom.oz.au).

Users of the SSL sockets must be aware of the patent rights and export restrictions of cryprographic algorithms in Europe and USA. Please see the Requirements [page 26]*section and the SSLeay documentations on the legal aspects on algorithm use.*

Only the AF_INET protocol family and the STREAM protocols are supported.

A socket is a full duplex communications channel between two UNIX processes, either over a network to a remote machine, or locally between processes running on the same machine. A socket connects two parties, the initiator and the connector. The initiator is the UNIX process which first opens the socket. It issues a series of system calls to set up the socket and then waits for another process to create a connection to the socket. When the connector starts, it also issues a series of system calls to set up the socket. Then both processes continue running and the communications channel is bound to a file descriptor which both processes use for reading and writing.

Exports

`listen(Protocol, Family, Address, Mode)`

Sets up a socket listening to Address. It also binds the name specified by Address to the socket. Protocol must be the atom STREAM (connection-oriented). Family must be AF_INET.

The UNIX process that is to connect to the socket can run on any other accessible machine on the Internet. The Address is an integer specifying what port number is to be listened to. This port number uniquely identifies the socket on the machine. If port number 0 is chosen, a free port number is automatically chosen by the UNIX kernel. Note: These port numbers are not to be confused with Erlang ports; they are UNIX-socket ports. Socket ports are used with a host name to create an end point for a socket connection. listen/4 with Protocol=STREAM returns the tuple {Filedescriptor, Portnumber}. Filedescriptor is an integer specifying the file descriptor assigned to the socket which is being listened to. Portnumber is an integer specifying the port number assigned to the socket. If Address is not zero in the call to listen, the returned port number is equal to Address.

Mode must be one of:

```
{packet, N}
{binary_packet, N}
raw       == {packet, 0}
onebyte   == {packet, 1}
twobytes  == {packet, 2}
fourbytes == {packet, 4}
asn1
```

where valid values for N are 0, 1, 2 and 4. This parameter specifies the way to read or write to the socket. If Mode is {packet, N}, then each series of bytes written to the socket will be prepended with N bytes indicating the length of the string. These N bytes are in binary format, with the most significant byte first. In this way it can be checked that all bytes that were written also are read. For this reason no partitioned messages will ever be delivered.

If Mode is {binary_packet, N}, the socket is in binary mode, and binary data will be prepended with a bytes header of N. When data is delivered to a socket in binary mode, the data will be delivered as a binary (instead of being unpacked as a byte list.) If N is 0, nothing will be prepended. If Mode is asn1, the receiving side of the connection will assume that BER-coded ASN.1 messages are sent on the socket. The header of the ASN.1 message will then be checked to find out the total length of the ASN.1 message. That number of bytes will then be read from the socket and only one message at a time delivered to the Erlang runtime system. *Note!* the asn1 mode will only work if all BER encoded data uses the definite length form. If the indefinite length form is used (the sender's decision), only the tag and length bytes will be received and then the connection will be broken. If the indefinite length form can occur (received by the Erlang runtime system) the raw or {packet,0} mode should be used.

For this reason if the options {packet, N}, {binary_packet, N} (N > 0) or asn1 are set on the socket, all that is written at the sender side will be read (in one chunk) on the reader side. This can be very convenient as this is not guaranteed in TCP. In TCP the messages may be divided partition in unpredictable ways. With TCP a STREAM of bytes is delivered; it is not a datagram protocol.

Example:

```
ListenSocket = ssl_socket:listen('STREAM', 'AF_INET', 3000,
                                 {packet, 2}).
```

ListenSocket may be bound to {3, 3000}, where 3 is a file descriptor and 3000 is the port listened to. If not successful the process evaluating listen evaluates exit({listen, syncerror}). This happens if, for example, Portnumber is set to a number which is already occupied on the machine.

`accept(ListenSocket, SSLFlags)`

After a listen, the incoming requests to connect for a connection oriented (STREAM) socket may be accepted. This is done with the call accept. The parameter ListenSocket is the tuple returned from the previous call to listen. The call to accept suspends the caller until a connection has been established from outside. A process identifier is returned to the caller. This process is located between the user and the actual socket. All communication with the socket is through this process, which understands a series of messages and also sends a series of messages to the process that initiated the call to accept.

SSLFlags is an ASCII list which contains a combination of the following options separated by space/s:

-cert ARG specify the certificate file to use. File should be in PEM format. Server must always have a certificate.

-key ARG specify the private key file to use. File should be in PEM format. If certificate file contains private key then there is no need to specify private key file.

-cipher ARG specify the list of ciphers to use, list of the following: NULL-MD5 RC4-MD5 EXP-RC4-MD5 IDEA-CBC-MD5 RC2-CBC-MD5 EXP-RC2-CBC-MD DES-CBC-MD5 DES-CBC-SHA DES-CBC3-MD5 DES-CBC3-SHA DES-CFB-M1, separated by ':'. If this option is not specified then the value of environment variable SSL_CIPHER will be used.

-verify ARG specify the certificate verification level. ARG could be one of: 0 - server does not ask for a client certificate; client does not check the server certificate but uses it for establishing a SSL connection 1 - server asks for client certificate; both do a certificate check; if it fails because of unknown issuer certificate the connection still gets established 2 - server asks for client certificate; both do a certificate check; SSL connection gets established only if the certificate check is successful. Note: default level of verification is 0.

-log ARG specify the log file

Example:

```
Socket = ssl_socket:accept(ListenSocket,
                           "-cert server_cert.pem -key server_key.pem")
```

After the statement above it is possible to communicate with the socket. The messages, which may be sent to the socket are:

```
Socket ! {self(), {deliver, ByteList}}.
```

or

```
Socket ! {self(), {deliver, Binary}}.
```

Causes Binary/ByteList to be written to the socket.

```
Socket ! {self(), close}.
```

Closes the socket down in an orderly way. If the socket is not closed in this way, it will be automatically closed when the process terminates. The messages that can be received from the socket are best explained by an example:

```
receive
    {Socket, {socket_closed, normal}} ->
        ok;   %% socket closed by foreign host
    {Socket, {socket_closed, Error}} ->
        notok; %% something has happened to the socket
    {Socket, {fromsocket, Bytes}} ->
        {bytes, Bytes}
end.
```

Two messages may be sent to the socket, i.e. deliver and close. The socket can send three messages back: two error messages and one message indicating the arrival of new data. All of these are shown below.

Input to the socket:

- {self(), {deliver, ByteList}}
- {self(), {deliver, Binary}}
- {self(), close}

Output from the socket:

- {Socket, {socket_closed, normal}}
- {Socket, {socket_closed, Error}}
- {Socket, {fromsocket, ByteList}}
- {Socket, {fromsocket, Binary}}

It may sometimes be convenient to listen to several sockets at the same time. This is most easily achieved by having one Erlang process for each port number for listening.

Another common situation in network programming is when a server is listening to one or more ports waiting for a connect message from the network. Once it arrives, a separate process is spawned to specifically handle the connection. It returns and continues waiting for new connections from the network.

The code for this could be similar to the following:

```
top(Port) ->
    Listen = ssl_socket:listen('STREAM', 'AF_INET', Port,
                               {packet, 2}),
    loop(Listen).

loop(Listen) ->
    Pid = spawn(mymod, connection, [Listen, self()]),
    receive
        {Pid, ok} ->
            loop(Listen)
    end.

connection(Listen, Father) ->
    Socket = ssl_socket:accept(Listen, "-cert ssl_server.pem"),
    Father ! {self(), ok},
    Socket ! {self(), {deliver, "Hello there"}},
    .....
    ....
```

This code first spawns a process, and lets the new process be suspended while waiting for the connection from the network. Once the new process is connected, the original process is informed about it by the {self(), ok} message. That process then spawns another, etc.

If there is a listening function to a port and accept/2 has been evaluated, the process is suspended and cannot be aborted. In order to stop accepting input, the process making the call receives an EXIT signal. The accept call will then terminate and no more connections will be accepted until a new accept call is made to the same ListenSocket. To achieve this, loop(Listen) can be modified in the following way:

```
loop(Listen) ->
    Pid = spawn(mymod, connection, [Listen, self()]),
    loop(Pid, Listen).

loop(Pid, Listen) ->
    receive
        {Pid, ok} ->
            loop(Listen);
        stop ->
            exit(Pid, abort),
            exit(normal)
    end.
```

After the code above has received the `stop` message and exited, there is no error in the `Listen` socket. It is still intact and can be used again in a new call to `loop/1`.

Another common situation in socket programming is wanting to listen to an address for connections, and then having all the connections handled by a single special process (that reads and writes several sockets simultaneously). The code for that would be similar to the following example:

```
my_accept(ListenFd, User) ->
    S = ssl_socket:accept(ListenFd, "-cert ssl_server.pem"),
    ssl_socket:controlling_process(S, User),
    my_accept(ListenFd, User).
```

The process User runs code that is similar to the following:

```
run(Sockets) when list(Sockets) ->
    receive
        {From, {fromsocket, Bytes}} ->
            case lists:member(From, Sockets) of
                true ->  %% old socket
                    handle_input(Bytes),
                    run(Sockets);
                false ->  %% new connection
                    handle_input(Bytes),
                    run([From|Sockets])
            end;
        ..........   etc.
```

`client(Protocol, Family, Address, Mode, SSLFlags)`

If another UNIX process is already listening to a socket, the socket on the client side may be opened with this call. As before, `Protocol` must be the atom `STREAM` and `Family` must be `AF_INET`. `Address` must be a tuple of the type {`IPAddress`, `Portnumber`}. It may be argued that users should not have to know port numbers, only names of services as in the BSD library routine `getservbyname()`. However, this idea has not been implemented in this package, so when a client is to be connected to a socket over the Internet, the port number has to be specified. Examples:

```
Socket1 =
  ssl_socket:client('STREAM', 'AF_INET',
                    {'gin.eua.ericsson.se', 1000}, raw,
                    "-cert client_cert.pem -cert client_key.pem"),
Socket2 =
  ssl_socket:client('STREAM', 'AF_INET',
                    {'134.138.99.53', 1002}, asn1,
                    "-cert ssl_client.pem"),
Socket3 =
  ssl_socket:client('STREAM', 'AF_INET',
                    {'gin', 1003}, {binary_packet, 4}, ""),
```

As can be seen in the examples above, several formats are allowed for `Address`. The `Mode` variable in the call to `client` is the same as in the calls to `listen`. The `SSLFlags` variable is the same as in the calls to `accept`, with one exception it is recommended for client to have a certificate but it is not necessary.

`client` returns a process identifier of a process with the same characteristics as the process described for the `accept` call above.

`controlling_process(Socket, Pid)`

When a value has been returned from the call to `accept` or the call to `client`, the Pid of the process which performed the initiation is known by the socket. All output from the socket is sent to this process. All input to the socket must also be wrapped with the Pid of the original process.

If the controlling process is to be changed, the socket must be informed. This is similar to the way an Erlang port needs to know the Pid of the process which opened it. The socket (and the port) must know where to send messages. The function above assigns a new controlling process to the socket. Thus, this function ensures that all output from the socket is sent to a process other than the process which created the socket. It also ensures that no messages from the socket are lost while the switch takes place.

`peername(Socket)`

Returns the name of the peer to `Socket`.

If AF_UNIX is used `peername` returns the filename used as address of a string. If AF_INET is used `peername` returns the tuple {`Portnumber, IPAddress`}.

`resolve()`

Returns the official name of the current host.

`resolve(IPAddress)`

Returns the official name of the host with the address `IPAddress`.

`close(Socket)`

Closes the socket. This is equivalent to sending a {`self(), close`} message to the process controlling the socket. It also operates on sockets returned by the `listen` call. This is the method to stop the listening to a socket.

`start()`

Starts the socket server.

`stop()`

Stops the socket server, and closes all open sockets.

FEATURES

Even if a socket is opened in {`packet, N`} mode, it is possible to write binaries to it. The receiving part of the socket determines if data from the socket is to be unpacked as a byte list or not. i.e. a sender may be in binary mode ({`binary_packet, N`}) and the receiver in byte list mode ({`packet, N`}) or vice versa. The only restriction is that the packet sizes must match.

The modes `raw` and `twobytes` are kept for backwards compatibility, and the modes `onebyte` and `fourbytes` have been added for forward compatibility.

In order to be able to use this module it is required to generate a key and a certificate.

For test purposes a private key and a certificate can be generated by using:

```
req -new -x509 -nodes -out test.pem -keyout test.pem
ln -sf test.pem `x509 -noout -hash < test.pem`.0
```

Certificate signing request can be generated by using:

```
req -new -out csr.pem -keyout key.pem -days XXX
```

A certificate signing request (csr.pem) is then could be send to a Certificate Authority (CA) for the purpose of of CA signing the request.

Some of Certification Authorities:

`http://www.verisign.com`[4] - Verisign
`http://www.thawte.com/certs/`[5] - Thawte Consulting
`http://www.eurosign.com`[6] - EuroSign
`http://www.cost.se`[7] - COST

Environment variables `SSL_CERT_DIR` and `SSL_CERT_FILE` could be used to set the location of the certificate of the trusted certifying authority. This is used during the certificate verification process.

[4] URL: `http://www.verisign.com`
[5] URL: `http://www.thawte.com/certs/`
[6] URL: `http://eurosign.com`
[7] URL: `http://www.cost.se`

REQUIREMENTS

When using this module, both client and server must be SSL-enabled. A SSL-server will hang if a non-SSL client tries to connect to it. If a SSL-client tries to connect to a non-SSL-server, the connection will fail.

SSL sockets need the SSLeay version 0.6.6 package installed in shared library form. You can get it from ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL[8] or you can find other mirrored locations at http://www.psy.uq.oz.au/~ftp/Crypto/[9].

The SSLeay package implements several well known cryptographic algorithms. Some of these are protected by software patents in some countries. The package can be configured to exclude algorithms at installation. Below follows a summary on software patents and restrictions for algorithms in SSLeay, see the SSLeay documentation for details:

The use of the RSA algorithm must be licensed in the USA due to US software patents. This includes any products sold to the USA that use the SSLeay RSA package. Export from the USA is restricted for software containing cryptographic algorithms.

The IDEA algorithm is protected by a patent in Europe and must be licensed.

General use of cryptography is prohibited in France.

BUGS

At this stage it is not possible to establish connection between a server and a client residing on the same Erlang node due to blocking of SSL_connect().

Please note that at this stage it is not possible to use private key encrypted with a pass phrase. To remove pass phrase do:

```
rsa -in key-protected -out key-unprotected.pem
```

The result of this restriction is that the secury of the private key relies on the file system security mechanism. Keep the private key and the certificate in separate files.

[8] URL: `ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL`
[9] URL: `http://www.psy.uq.oz.au/~ftp/Crypto/`

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.

`accept/1`
ssl , 15

`accept/2`
ssl , 15
ssl_socket , 20

`client/5`
ssl_socket , 23

`close/1`
ssl , 15
ssl_socket , 24

`connect/3`
ssl , 15

`connect/4`
ssl , 15

`controlling_process/2`
ssl , 15
ssl_socket , 24

`format_error/1`
ssl , 15

`getopts/2`
ssl , 16

`listen/2`
ssl , 16

`listen/4`
ssl_socket , 19

`peername/1`
ssl , 16
ssl_socket , 24

`pid/1`
ssl , 16

`port/1`
ssl , 16

`recv/2`
ssl , 17

`recv/3`
ssl , 17

`resolve/0`
ssl_socket , 24

`resolve/1`
ssl_socket , 24

`send/2`
ssl , 17

`setopts/2`
ssl , 17

`sockname/1`
ssl , 17

ssl
`accept/1`, 15
`accept/2`, 15
`close/1`, 15
`connect/3`, 15
`connect/4`, 15
`controlling_process/2`, 15
`format_error/1`, 15
`getopts/2`, 16
`listen/2`, 16
`peername/1`, 16
`pid/1`, 16
`port/1`, 16
`recv/2`, 17
`recv/3`, 17
`send/2`, 17
`setopts/2`, 17
`sockname/1`, 17

ssl_socket
`accept/2`, 20

client/5, 23
close/1, 24
controlling_process/2, 24
listen/4, 19
peername/1, 24
resolve/0, 24
resolve/1, 24
start/0, 25
stop/0, 25

start/0
ssl_socket , 25

stop/0
ssl_socket , 25