1 SSH User's Guide

The Erlang Secure Shell (SSH) application, ssh, implements the SSH Transport Layer Protocol and provides SSH File Transfer Protocol (SFTP) clients and servers.

1.1 Introduction

SSH is a protocol for secure remote logon and other secure network services over an insecure network.

1.1.1 Scope and Purpose

SSH provides a single, full-duplex, and byte-oriented connection between client and server. The protocol also provides privacy, integrity, server authentication, and man-in-the-middle protection.

The ssh application is an implementation of the SSH Transport, Connection and Authentication Layer Protocols in Erlang. It provides the following:

- API functions to write customized SSH clients and servers applications
- The Erlang shell available over SSH
- An SFTP client (ssh_sftp) and server (ssh_sftpd)

1.1.2 Prerequisites

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

1.1.3 SSH Protocol Overview

Conceptually, the SSH protocol can be partitioned into four layers:

<table>
<thead>
<tr>
<th>SSH Client/Server Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection Protocol</td>
</tr>
<tr>
<td>Authentication Protocol</td>
</tr>
<tr>
<td>Transport Protocol</td>
</tr>
<tr>
<td>TCP/IP Stack</td>
</tr>
</tbody>
</table>

Figure 1.1: SSH Protocol Architecture
1.1 Introduction

Transport Protocol

The SSH Transport Protocol is a secure, low-level transport. It provides strong encryption, cryptographic host authentication, and integrity protection. A minimum of Message Authentication Code (MAC) and encryption algorithms are supported. For details, see the ssh(3) manual page in ssh.

Authentication Protocol

The SSH Authentication Protocol is a general-purpose user authentication protocol run over the SSH Transport Layer Protocol. The ssh application supports user authentication as follows:

- Using public key technology. RSA and DSA, X509-certificates are not supported.
- Using keyboard-interactive authentication. This is suitable for interactive authentication methods that do not need any special software support on the client side. Instead, all authentication data is entered from the keyboard.
- Using a pure password-based authentication scheme. Here, the plain text password is encrypted before sent over the network.

Several configuration options for authentication handling are available in ssh:connect[3,4] and ssh:daemon[2,3].

The public key handling can be customized by implementing the following behaviours from ssh:

- Module ssh_client_key_api.
- Module ssh_server_key_api.

Connection Protocol

The SSH Connection Protocol provides application-support services over the transport pipe, for example, channel multiplexing, flow control, remote program execution, signal propagation, and connection forwarding. Functions for handling the SSH Connection Protocol can be found in the module ssh_connection in ssh.

Channels

All terminal sessions, forwarded connections, and so on, are channels. Multiple channels are multiplexed into a single connection. All channels are flow-controlled. This means that no data is sent to a channel peer until a message is received to indicate that window space is available. The initial window size specifies how many bytes of channel data that can be sent to the channel peer without adjusting the window. Typically, an SSH client opens a channel, sends data (commands), receives data (control information), and then closes the channel. The ssh_client_channel behaviour handles generic parts of SSH channel management. This makes it easy to write your own SSH client/server processes that use flow-control and thus opens for more focus on the application logic.

Channels come in the following three flavors:

- **Subsystem** - Named services that can be run as part of an SSH server, such as SFTP (ssh_sftpd), that is built into the SSH daemon (server) by default, but it can be disabled. The Erlang ssh daemon can be configured to run any Erlang-implemented SSH subsystem.
- **Shell** - Interactive shell. By default the Erlang daemon runs the Erlang shell. The shell can be customized by providing your own read-eval-print loop. You can also provide your own Command-Line Interface (CLI) implementation, but that is much more work.
- **Exec** - One-time remote execution of commands. See function ssh_connection:exec/4 for more information.

1.1.4 Where to Find More Information

For detailed information about the SSH protocol, refer to the following Request for Comments (RFCs):

- **RFC 4250** - Protocol Assigned Numbers
- **RFC 4251** - Protocol Architecture
- **RFC 4252** - Authentication Protocol
1.2 Getting Started

1.2.1 General Information

The following examples use the utility function `ssh:start/0` to start all needed applications (crypto, public_key, and ssh). All examples are run in an Erlang shell, or in a bash shell, using `openssh` to illustrate how the `ssh` application can be used. The examples are run as the user `otptest` on a local network where the user is authorized to log in over `ssh` to the host `tarlop`.

If nothing else is stated, it is presumed that the `otptest` user has an entry in the `authorized_keys` file of `tarlop` (allowed to log in over `ssh` without entering a password). Also, `tarlop` is a known host in the `known_hosts` file of the user `otptest`. This means that host-verification can be done without user-interaction.

1.2.2 Using the Erlang ssh Terminal Client

The user `otptest`, which has bash as default shell, uses the `ssh:shell/1` client to connect to the `openssh` daemon running on a host called `tarlop`:

```
1> ssh:start().
ok
2> {ok, S} = ssh:shell("tarlop").
otptest@tarlop:> pwd
/home/otptest
otptest@tarlop:> exit
logout
```

1.2.3 Running an Erlang ssh Daemon

The `system_dir` option must be a directory containing a host key file and it defaults to `/etc/ssh`. For details, see Section Configuration Files in `ssh(6)`.

**Note:**

Normally, the `/etc/ssh` directory is only readable by root.

The option `user_dir` defaults to directory `users ~/.ssh`.

**Step 1.** To run the example without root privileges, generate new keys and host keys:

```
$bash> ssh-keygen -t rsa -f /tmp/ssh_daemon/ssh_host_rsa_key
[...]
$bash> ssh-keygen -t rsa -f /tmp/otptest_user/.ssh/id_rsa
[...]
```

**Step 2.** Create the file `/tmp/otptest_user/.ssh/authorized_keys` and add the content of `/tmp/otptest_user/.ssh/id_rsa.pub`.

**Step 3.** Start the Erlang `ssh` daemon:
1.2 Getting Started

Step 4. Use the `openssh` client from a shell to connect to the Erlang `ssh` daemon:

```
$bash> ssh tarlop -p 8989 -i /tmp/otptest_user/.ssh/id_rsa -o UserKnownHostsFile=/tmp/otptest_user/.ssh/known_hosts
The authenticity of host 'tarlop' can't be established.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'tarlop' (RSA) to the list of known hosts.
Eshell V5.10 (abort with ^G)
```

There are two ways of shutting down an `ssh` daemon, see **Step 5a** and **Step 5b**.

**Step 5a.** Shut down the Erlang `ssh` daemon so that it stops the listener but leaves existing connections, started by the listener, operational:

```
3> ssh:stop_listener(Sshd).
ok
4>
```

**Step 5b.** Shut down the Erlang `ssh` daemon so that it stops the listener and all connections started by the listener:

```
3> ssh:stop_daemon(Sshd).
ok
4>
```

### 1.2.4 One-Time Execution

**Erlang client contacting OS standard ssh server**

In the following example, the Erlang shell is the client process that receives the channel replies as Erlang messages. Do an one-time execution of a remote OS command ("pwd") over `ssh` to the ssh server of the OS at the host "tarlop":

```
1> ssh:start().
ok
2> {ok, ConnectionRef} = ssh:connect("tarlop", 22, []).
   {ok,<0.57.0>}
3> {ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
   {ok,0}
4> success = ssh_connection:exec(ConnectionRef, ChannelId, "pwd", infinity).
5> flush(). % Get all pending messages. NOTE: ordering may vary!
   Shell got {ssh_cm,<0.57.0>,{data,0,0,<<"/home/otptest
">>}}
   Shell got {ssh_cm,<0.57.0>,{eof,0}}
   Shell got {ssh_cm,<0.57.0>,{exit_status,0,0}}
   Shell got {ssh_cm,<0.57.0>,{closed,0}}
   ok
6> ssh:connection_info(ConnectionRef, channels).
   {channels,[]}
7>
```
1.2 Getting Started

See `ssh_connection` and `ssh_connection:exec/4` for finding documentation of the channel messages.

To collect the channel messages in a program, use `receive...end` instead of `flush/1`:

```erl
  5> receive
  5>     {ssh_cm, ConnectionRef, {data, ChannelId, Type, Result}} when Type == 0 ->
  5>         {ok, Result}
  5>     {ssh_cm, ConnectionRef, {data, ChannelId, Type, Result}} when Type == 1 ->
  5>         {error, Result}
  5> end.
  {ok, <<"/home/otptest\n">>}
```

Note that only the exec channel is closed after the one-time execution. The connection is still up and can handle previously opened channels. It is also possible to open a new channel:

```erl
  7> {ok, NewChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
  8> {ok,1}
```

To close the connection, call the function `ssh:close(ConnectionRef)`. As an alternative, set the option `{idle_time, 1}` when opening the connection. This will cause the connection to be closed automatically when there are no channels open for the specified time period, in this case 1 ms.

**OS standard client and Erlang daemon (server)**

An Erlang SSH daemon could be called for one-time execution of a "command". The "command" must be as if entered into the erlang shell, that is a sequence of Erlang expressions ended by a period (.). Variables bound in that sequence will keep their bindings throughout the expression sequence. The bindings are disposed when the result is returned.

Here is an example of a suitable expression sequence:

```erl
  A=1, B=2, 3 == (A + B).
```

It evaluates to `true` if submitted to the Erlang daemon started in Step 3 above:

```bash
$ bash> ssh tarlop -p 8989 "A=1, B=2, 3 == (A + B)."
true
$ bash>
```

The same example but now using the Erlang ssh client to contact the Erlang server:

```erl
  1> {ok, ConnectionRef} = ssh:connect("tarlop", 8989, []).
  2> {ok, <0.216.0>}
  3> {ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
  4> {ok,0}
  5> success = ssh_connection:exec(ConnectionRef, ChannelId,
  6>       "A=1, B=2, 3 == (A + B).",
  7>       infinity).
  8> success
  9> flush().
```

```
Shell got {ssh_cm,<0.216.0>,{data,0,0,<<"true">>}}
Shell got {ssh_cm,<0.216.0>,{exit_status,0,0}}
Shell got {ssh_cm,<0.216.0>,{eof,0}}
Shell got {ssh_cm,<0.216.0>,{closed,0}}
oke
5>
```
Note that Erlang shell specific functions and control sequences like for example \h()\ are not supported.

I/O from a function called in an Erlang ssh daemon

Output to stdout on the server side is also displayed as well as the resulting term from the function call:

```
$bash> ssh tarlop -p 8989 'io:format("Hello!~n~nHow are ~p?~n",[you]).'
Hello!
How are you?
ok
$bash>
```

And similar for reading from stdin. As an example we use `io:read/1` which displays the argument as a prompt on stdout, reads a term from stdin and returns it in an ok-tuple:

```
$bash> ssh tarlop -p 8989 'io:read("write something: ").'
write something: [a,b,c].
{ok,[a,b,c]}
$bash>
```

The same example but using the Erlang ssh client:

```
Eshell V10.5.2 (abort with ^G)
1> ssh:start().
ok
2> {ok, ConnectionRef} = ssh:connect(loopback, 8989, []).
{ok,<0.92.0>}
3> {ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
{ok,0}
4> success = ssh_connection:exec(ConnectionRef, ChannelId, 
    "io:read("write something: ").", 
    infinity).
success
5> flush().
Shell got {ssh_cm,<0.92.0>,{data,0,0,<<"write something: ">>}}
ok
% All data is sent as binaries with string contents:
6> ok = ssh_connection:send(ConnectionRef, ChannelId, <<"[a,b,c].">>).
ok
7> flush().
ok
% Nothing is received, because the io:read/1
% requires the input line to end with a newline.
% Send a newline (it could have been included in the last send):
8> ssh_connection:send(ConnectionRef, ChannelId, <<"\n>>).
ok
9> flush().
Shell got {ssh_cm,<0.92.0>,{data,0,0,<<"{ok,\[a,b,c\]}">>}}
Shell got {ssh_cm,<0.92.0>,{exit status,0,0}}
Shell got {ssh_cm,<0.92.0>,{eof,0}}
Shell got {ssh_cm,<0.92.0>,{closed,0}}
ok
10>
```

Configuring the server's (daemon's) command execution

Every time a daemon is started, it enables one-time execution of commands as described in the previous section unless explicitly disabled.
There is often a need to configure some other exec evaluator to tailor the input language or restrict the possible functions to call. There are two ways of doing this which will be shown with examples below. See \texttt{ssh:daemon/2,3} and \texttt{execdaemon_option()} for details.

Examples of the two ways to configure the exec evaluator:

- **Disable one-time execution.**
  To modify the daemon start example above to reject one-time execution requests, we change Step 3 by adding the option \{exec, disabled\} to:

  ```erlang
  1> ssh:start().
  ok
  2> \{ok, Sshd\} = ssh:daemon(8989, [[system_dir, "/tmp/ssh_daemon"],
    \{user_dir, "/tmp/otptest_user/.ssh"},
    \{exec, disabled\}]).
  \{ok, \langle0.54.0\rangle\}
  3>
  
  A call to that daemon will return the text "Prohibited." on stderr (depending on the client and OS), and the exit status 255:

  ```bash
  $bash> ssh tarlop -p 8989 "test."  
  Prohibited.  
  $bash> echo $?  
  255  
  $bash>
  ```

  And the Erlang client library also returns the text "Prohibited." on data type 1 instead of the normal 0 and exit status 255:

  ```erlang
  2> \{ok, ConnectionRef\} = ssh:connect(loopback, 8989, []).  
  \{ok, \langle0.92.8\rangle\}
  3> \{ok, ChannelId\} = ssh_connection:session_channel(ConnectionRef, infinity).
  \{ok, \langle\rangle\}
  4> success = ssh_connection:exec(ConnectionRef, ChannelId, "test.").
  success
  5> flush().
  Shell got \{ssh_cm, \langle0.106.0\rangle, \langledata, 0, \langle\rangle, <<"Prohibited."angle\rangle\}
  Shell got \{ssh_cm, \langle0.106.0\rangle, \langleexit_status, 0, 255\rangle\}
  Shell got \{ssh_cm, \langle0.106.0\rangle, \langleeof, \rangle\rangle\}
  Shell got \{ssh_cm, \langle0.106.0\rangle, \langleclosed, \rangle\rangle\}
  ok
  6>
  ```

- **Install an alternative evaluator.**
  Start the daemon with a reference to a \texttt{fun()} that handles the evaluation:
1> ssh:start().
ok
2> MyEvaluator = fun("1") -> {ok, some_value};
   ("2") -> {ok, some_other_value};
   ("3") -> {ok, V} = io:read("input erlang term>> "),
            {ok, V};
   (Err) -> {error,{bad_input,Err}}
end.
3> {ok, Sshd} = ssh:daemon(1234, [{system_dir, "/tmp/ssh_daemon"},
                   {user_dir, "/tmp/otptest_user/.ssh"},
                   {exec, {direct,MyEvaluator}}]).
   {ok,<0.275.0>}
4>
and call it:

```
$bash> ssh localhost -p 1234 1
some_value
$bash> ssh localhost -p 1234 2
some_other_value
# I/O works:
$bash> ssh localhost -p 1234 3
input erlang term>> abc.
   abc
# Check that Erlang evaluation is disabled:
$bash> ssh localhost -p 1234 1+ 2.
   **Error** {bad_input,"1+ 2."}
$bash>
```

Note that spaces are preserved and that no point (.) is needed at the end - that was required by the default evaluator.

The error return in the Erlang client (The text as data type 1 and exit_status 255):

```
2> {ok, ConnectionRef} = ssh:connect(loopback, 1234, []).
   {ok,<0.92.0>}
3> {ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
   {ok,0}
4> success = ssh_connection:exec(ConnectionRef, ChannelId, "1+ 2."
   success
5> flush().
   Shell got {ssh_cm,<0.106.0>,[data,0,1,<<"**Error** {bad_input,\"1+ 2.\"}>>]}
   Shell got {ssh_cm,<0.106.0>,[exit_status,0,255]}
   Shell got {ssh_cm,<0.106.0>,[eof,0]}
   Shell got {ssh_cm,<0.106.0>,[closed,0]}
   ok
6>
```

The fun() in the exec option could take up to three arguments (Cmd, User and ClientAddress). See the exec_daemon_option() for the details.

Note:

An old, discouraged and undocumented way of installing an alternative evaluator exists.

It still works, but lacks for example I/O possibility. It is because of that compatibility we need the {direct,...} construction.
1.2.5 SFTP Server

Start the Erlang ssh daemon with the SFTP subsystem:

```
1> ssh:start().
  ok
2> ssh:daemon(8989, [{system_dir, "/tmp/ssh_daemon"},
   {user_dir, "/tmp/otptest_user/.ssh"},
   {subsystems, [ssh_sftpd:subsystem_spec(
     [{cwd, "/tmp/sftp/example"}])}]).
  {ok,<0.54.0>}
```

Run the OpenSSH SFTP client:

```
$bash> sftp -oPort=8989 -o IdentityFile=/tmp/otptest_user/.ssh/id_rsa -o UserKnownHostsFile=/tmp/otptest_user/.ssh/known_hosts tarlop
```

1.2.6 SFTP Client

Fetch a file with the Erlang SFTP client:

```
1> ssh:start().
  ok
2> {ok, ChannelPid, Connection} = ssh_sftp:start_channel("tarlop", []).
  {ok,<0.57.0>,<0.51.0>}
3> ssh_sftp:read_file(ChannelPid, "/home/otptest/test.txt").
  {ok,<<"This is a test file
">>}
```

1.2.7 SFTP Client with TAR Compression

Basic example

This is an example of writing and then reading a tar file:

```
{ok,HandleWrite} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [write]),
ok = erl_tar:add(HandleWrite, ..... ),
ok = erl_tar:add(HandleWrite, ..... ),
...
ok = erl_tar:close(HandleWrite),
% And for reading
{ok,HandleRead} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [read]),
{ok,NameValueList} = erl_tar:extract(HandleRead,[memory]),
ok = erl_tar:close(HandleRead),
```

Example with encryption

The previous Basic example can be extended with encryption and decryption as follows:
First three parameters depending on which crypto type we select:
Key = <<"This is a 256 bit key. abcdefgh">>,
Ivec0 = crypto:strong_rand_bytes(16),
DataSize = 1024, % DataSize rem 16 = 0 for aes_cbc

Initialization of the CryptoState, in this case it is the Ivector.
InitFun = fun() -> {ok, Ivec0, DataSize} end,

How to encrypt:
EncryptFun =
  fun(PlainBin, Ivec) ->
    EncryptedBin = crypto:block_encrypt(aes_cbc256, Key, Ivec, PlainBin),
    {ok, EncryptedBin, crypto:next_iv(aes_cbc, EncryptedBin)}
  end,

What to do with the very last block:
CloseFun =
  fun(PlainBin, Ivec) ->
    EncryptedBin = crypto:block_encrypt(aes_cbc256, Key, Ivec, pad(16, PlainBin) %% Last chunk ),
    {ok, EncryptedBin}
  end,

Cw = [InitFun, EncryptFun, CloseFun],
{ok, HandleWrite} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [write, {crypto, Cw}]),
ok = erl_tar:add(HandleWrite, .... ),
ok = erl_tar:add(HandleWrite, .... ),
...
ok = erl_tar:add(HandleWrite, .... ),
ok = erl_tar:close(HandleWrite),

And for decryption (in this crypto example we could use the same InitFun
% as for encryption):
DecryptFun =
  fun(EncryptedBin, Ivec) ->
    PlainBin = crypto:block_decrypt(aes_cbc256, Key, Ivec, EncryptedBin),
    {ok, PlainBin, crypto:next_iv(aes_cbc, EncryptedBin)}
  end,

Cr = [InitFun, DecryptFun],
{ok, HandleRead} = ssh_sftp:open_tar(ChannelPid, ?tar_file_name, [read, {crypto, Cr}]),
{ok, NameValueList} = erl_tar:extract(HandleRead, [memory]),
ok = erl_tar:close(HandleRead),

1.2.8 Creating a Subsystem
A small ssh subsystem that echoes N bytes can be implemented as shown in the following example:
1.2 Getting Started

-module(ssh_echo_server).
-behaviour(ssh_server_channel). % replaces ssh_daemon_channel
-record(state, {
    n, id, cm }).
-export([init/1, handle_msg/2, handle_ssh_msg/2, terminate/2]).

init([N]) ->
    {ok, #state{n = N}}.

handle_msg({ssh_channel_up, ChannelId, ConnectionManager}, State) ->
    {ok, State#state{id = ChannelId, cm = ConnectionManager}}.

handle_ssh_msg({ssh_cm, CM, {data, ChannelId, 0, Data}}, #state{n = N} = State) ->
    M = N - size(Data),
    case M > 0 of
        true ->
            ssh_connection:send(CM, ChannelId, Data),
            {ok, State#state{n = M}};
        false ->
            <<SendData:N/binary, _/binary>> = Data,
            ssh_connection:send(CM, ChannelId, SendData),
            ssh_connection:send_eof(CM, ChannelId),
            {stop, ChannelId, State}
    end;
handle_ssh_msg({ssh_cm, _ConnectionManager, 
    {data, ChannelId, 1, Data}}, State) ->
    error_logger:format(standard_error, " ~p~n", [binary_to_list(Data)]),
    {ok, State};

handle_ssh_msg({ssh_cm, _, {signal, _, _}}, State) ->
    % Ignore signals according to RFC 4254 section 6.9.
    {ok, State};

handle_ssh_msg({ssh_cm, _, {exit_signal, ChannelId, _, _Error, _}}, State) ->
    {stop, ChannelId, State};

handle_ssh_msg({ssh_cm, _, {exit_status, ChannelId, _Status}}, State) ->
    {stop, ChannelId, State}.

terminate(_Reason, _State) ->
    ok.

The subsystem can be run on the host tarlop with the generated keys, as described in Section Running an Erlang ssh Daemon:

1> ssh:start().
ok
2> ssh:daemon(8989, [{system_dir, "/tmp/ssh_daemon"},
    {user_dir, "/tmp/otptest_user/.ssh"}
    {subsystems, [{"echo_n", {ssh_echo_server, [10]}}]})].
{ok,<0.54.0>}
3>
1> ssh:start().
ok
2> {ok, ConnectionRef} = ssh:connect("tarlop", 8989,
   [{user_dir, "/tmp/otptest_user/.ssh"}]).
   {ok,<0.57.0>}
3> {ok, ChannelId} = ssh_connection:session_channel(ConnectionRef, infinity).
4> success = ssh_connection:subsystem(ConnectionRef, ChannelId, "echo n", infinity).
5> ok = ssh_connection:send(ConnectionRef, ChannelId, "0123456789", infinity).
6> flush().
   {ssh_msg, <0.57.0>, {data, 0, 1, "0123456789"}}
   {ssh_msg, <0.57.0>, {eof, 0}}
   {ssh_msg, <0.57.0>, {closed, 0}}
7> {error, closed} = ssh_connection:send(ConnectionRef, ChannelId, "10", infinity).

See also ssh_client_channel(3) (replaces ssh_channel(3)).

1.3 Terminology

1.3.1 General Information

In the following terms that may cause confusion are explained.

1.3.2 The term "user"

A "user" is a term that everyone understands intuitively. However, the understandings may differ which can cause confusion.

The term is used differently in OpenSSH and SSH in Erlang/OTP. The reason is the different environments and use cases that are not immediately obvious.

This chapter aims at explaining the differences and giving a rationale for why Erlang/OTP handles "user" as it does.

In OpenSSH

Many have been in contact with the command 'ssh' on a Linux machine (or similar) to remotely log in on another machine. One type

```ssh host```

to log in on the machine named host. The command prompts for your password on the remote host and then you can read, write and execute as your user name has rights on the remote host. There are stronger variants with pre-distributed keys or certificates, but that are for now just details in the authentication process.

You could log in as the user anotheruser with

```ssh anotheruser@host```

and you will then be enabled to act as anotheruser on the host if authorized correctly.

So what does "your user name has rights" mean? In a UNIX/Linux/etc context it is exactly as that context: The user could read, write and execute programs according to the OS rules. In addition, the user has a home directory ($HOME) and there is a $HOME/ .ssh/ directory with ssh-specific files.

SSH password authentication

When SSH tries to log in to a host, the ssh protocol communicates the user name (as a string) and a password. The remote ssh server checks that there is such a user defined and that the provided password is acceptable.

If so, the user is authorized.
1.3 Terminology

SSH public key authentication

This is a stronger method where the ssh protocol brings the user name, the user's public key and some cryptographic information which we could ignore here.

The ssh server on the remote host checks:
- That the user has a home directory,
- that home directory contains a .ssh/ directory and
- the .ssh/ directory contains the public key just received in the authorized_keys file

if so, the user is authorized.

The SSH server on UNIX/Linux/etc after a succesful authentication

After a succesful incoming authentication, a new process runs as the just authenticated user.

Next step is to start a service according to the ssh request. In case of a request of a shell, a new one is started which handles the OS-commands that arrives from the client (that's "you").

In case of a sftp request, an sftp server is started in with the user's rights. So it could read, write or delete files if allowed for that user.

In Erlang/OTP SSH

For the Erlang/OTP SSH server the situation is different. The server executes in an Erlang process in the Erlang emulator which in turn executes in an OS process. The emulator does not try to change its user when authenticated over the SSH protocol. So the remote user name is only for authentication purposes in the Erlang/OTP SSH application.

Password authentication in Erlang SSH

The Erlang/OTP SSH server checks the user name and password in the following order:
- If a pwdfun is defined, that one is called and the returned boolean is the authentication result.
- Else, if the user_passwords option is defined and the username and the password matches, the authentication is a success.
- Else, if the option password is defined and matches the password the authentication is a success. Note that the use of this option is not recommended in non-test code.

Public key authentication in Erlang SSH

The user name, public key and cryptographic data (a signature) that is sent by the client, are used as follows (some steps left out for clarity):
- A callback module is selected using the options key_cb.
- The callback module is used to check that the provided public key is one of the user's pre-stored. In case of the default callback module, the files authorized_keys and authorized_keys2 are searched in a directory found in the following order:
  - If the option user_dir_fun is defined, that fun is called and the returned directory is used,
  - Else, If the option user_dir is defined, that directory is used,
  - Else the subdirectory .ssh in the home directory of the user executing the OS process of the Erlang emulator is used.

If the provided public key is not found, the authentication fails.
- Finally, if the provided public key is found, the signature provided by the client is checked with the public key.

The Erlang/OTP SSH server after a succesful authentication

After a successful authentication an Erlang process is handling the service request from the remote ssh client. The rights of that process are those of the user of the OS process running the Erlang emulator.
If a shell service request arrives to the server, an *Erlang shell* is opened in the server's emulator. The rights in that shell is independent of the just authenticated user.

In case of an sftp request, an sftp server is started with the rights of the user of the Erlang emulator's OS process. So with sftp the authenticated user does not influence the rights.

So after an authentication, the user name is not used anymore and has no influence.

### 1.4 Configuring algorithms in SSH

#### 1.4.1 Introduction

To fully understand how to configure the algorithms, it is essential to have a basic understanding of the SSH protocol and how OTP SSH app handles the corresponding items.

The first subsection will give a short background of the SSH protocol while later sections describes the implementation and provides some examples.

**Basics of the ssh protocol's algorithms handling**

SSH uses different sets of algorithms in different phases of a session. Which algorithms to use is negotiated by the client and the server at the beginning of a session. See [RFC 4253](https://tools.ietf.org/html/rfc4253), "The Secure Shell (SSH) Transport Layer Protocol" for details.

The negotiation is simple: both peers sends their list of supported algorithms to the other part. The first algorithm on the client's list that also in on the server's list is selected. So it is the client's ordering of the list that gives the priority for the algorithms.

There are five lists exchanged in the connection setup. Three of them are also divided in two directions, to and from the server.

The lists are (named as in the SSH application's options):

- **kex**
  - Key exchange.
  - An algorithm is selected for computing a secret encryption key. Among examples are: the old nowadays weak 'diffie-hellman-group-exchange-sha1' and the very strong and modern 'ecdh-sha2-nistp512'.

- **public_key**
  - Server host key
  - The asymmetric encryption algorithm used in the server's private-public host key pair. Examples include the well-known RSA 'ssh-rsa' and elliptic curve 'ecdsa-sha2-nistp521'.

- **cipher**
  - Symmetric cipher algorithm used for the payload encryption. This algorithm will use the key calculated in the kex phase (together with other info) to generate the actual key used. Examples are triple-DES '3des-cbc' and one of many AES variants 'aes192-ctr'.
  - This list is actually two - one for each direction server-to-client and client-to-server. Therefore it is possible but rare to have different algorithms in the two directions in one connection.

- **mac**
  - Message authentication code
  - "Check sum" of each message sent between the peers. Examples are SHA 'hmac-shal' and SHA2 'hmac-sha2-512'.
### 1.4 Configuring algorithms in SSH

This list is also divided into two for the both directions compression

If and how to compress the message. Examples are none, that is, no compression and zlib.

This list is also divided into two for the both directions

#### The SSH app’s mechanism

The set of algorithms that the SSH app uses by default depends on the algorithms supported by the:

- crypto app,
- The cryptolib OTP is linked with, usually the ones the OS uses, probably OpenSSL,
- and finally what the SSH app implements

Due to this, it is impossible to list in documentation what algorithms that are available in a certain installation.

There is an important command to list the actual algorithms and their ordering: `ssh:default_algorithms/0`.

```plaintext
0> ssh:default_algorithms().
{[kex,["ecdh-sha2-nistp384","ecdh-sha2-nistp521",
   "ecdh-sha2-nistp256","diffie-hellman-group-exchange-sha256",
   "diffie-hellman-group16-sha512",
   "diffie-hellman-group18-sha512",
   "diffie-hellman-group14-sha256",
   "diffie-hellman-group14-sha1",
   "diffie-hellman-group-exchange-shal"],
   [public_key,"ecdsa-sha2-nistp384","ecdsa-sha2-nistp521",
   "ecdsa-sha2-nistp256","ssh-rsa","rsa-sha2-256",
   "rsa-sha2-512","ssh-dss"],
   [cipher,[[client2server,"aes256-gcm@openssh.com",
   "aes256-ctr","aes192-ctr","aes128-gcm@openssh.com",
   "aes128-ctr","aes128-cbc","3des-cbc"],
   [server2client,"aes256-gcm@openssh.com","aes256-ctr",
   "aes192-ctr","aes128-gcm@openssh.com","aes128-ctr",
   "aes128-cbc","3des-cbc"]],
   [mac,[[client2server,"hmac-sha2-256","hmac-sha2-512",
   "hmac-shal"],
   [server2client,"hmac-sha2-256","hmac-sha2-512",
   "hmac-shal"]]],
   [compression,[[client2server,["none","zlib@openssh.com","zlib"],
   [server2client,["none","zlib@openssh.com","zlib"]]]]}
```

To change the algorithm list, there are two options which can be used in `ssh:connect/2,3,4` and `ssh:daemon/2,3,4`. The options could of course be used in all other functions that initiate connections.

The options are preferred_algorithms and modify_algorithms. The first one replaces the default set, while the latter modifies the default set.

#### 1.4.2 Replacing the default set: preferred_algorithms

See the Reference Manual for details

Here follows a series of examples ranging from simple to more complex.

To foresee the effect of an option there is an experimental function `ssh:chk_algos_opts(Opts)`. It mangles the options preferred_algorithms and modify_algorithms in the same way as `ssh:daemon`, `ssh:connect` and their friends does.

**Example 1**

Replace the kex algorithms list with the single algorithm 'diffie-hellman-group14-sha256':

```plaintext`
Example 2

In the lists that are divided in two for the two directions (c.f. \texttt{cipher}) it is possible to change both directions at once:

```plaintext
2> ssh:chk_algos_opts(
    [{preferred_algorithms,
      [{kex, ['diffie-hellman-group14-sha256']}
    ]
  }],
  [{kex,['diffie-hellman-group14-sha256']},
   {public_key,['ecdh-sha2-nistp384','ecdsa-sha2-nistp521',
               'ecdh-sha2-nistp256','ssh-rsa','rsa-sha2-256',
               'rsa-sha2-512','ssh-dss']},
   {cipher,[[client2server,['aes256-gcm@openssh.com',
                   'aes256-ctr','aes192-ctr','aes128-gcm@openssh.com',
                   'aes128-ctr','aes128-cbc','3des-cbc']]},
   {server2client,['aes256-gcm@openssh.com','aes256-ctr',
                   'aes192-ctr','aes128-gcm@openssh.com','aes128-ctr',
                   'aes128-cbc','3des-cbc']}]])
```

Note that both lists in \texttt{cipher} has been changed to the provided value ('aes128-ctr').

Example 3

In the lists that are divided in two for the two directions (c.f. \texttt{cipher}) it is possible to change only one of the directions:
1.4 Configuring algorithms in SSH

Example 4

It is of course possible to change more than one list:

Note that the ordering of the tuples in the lists didn’t matter.

1.4.3 Modifying the default set: modify_algorithms

A situation where it might be useful to add an algorithm is when one need to use a supported but disabled one. An example is the 'diffie-hellman-group1-sha1' which nowadays is very unsecure and therefore disabled. It is however still supported and might be used.

The option preferred_algorithms may be complicated to use for adding or removing single algorithms. First one has to list them with ssh:default_algorithms() and then do changes in the lists.
To facilitate addition or removal of algorithms the option `modify_algorithms` is available. See the `Reference Manual` for details.

The option takes a list with instructions to append, prepend or remove algorithms:

```
{modify_algorithms, [{append, ...},
  {prepend, ...},
  {rm, ...}]
}
```

Each of the ... can be a `algs_list()` as the argument to the `preferred_algorithms` option.

**Example 5**

As an example let's add the Diffie-Hellman Group1 first in the kex list. It is supported according to `Supported algorithms`.

5> ssh:chk_algos_opts(
    [{modify_algorithms,
      [{prepend,
        [{kex, ['diffie-hellman-group1-sha1']}]}
      ]}
    ]).

```
[{kex, 'diffie-hellman-group1-sha1', 'ecdh-sha2-nistp384',
  'ecdh-sha2-nistp521', 'ecdh-sha2-nistp256',
  'diffie-hellman-group-exchange-sha256',
  'diffie-hellman-group16-sha512',
  'diffie-hellman-group18-sha512',
  'diffie-hellman-group14-sha256',
  'diffie-hellman-group14-sha1',
  'diffie-hellman-group-exchange-sha1']},
{public_key, ['ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521',
  'ecdsa-sha2-nistp256', 'ssh-rsa', 'rsa-sha2-256',
  'rsa-sha2-512', 'ssh-dss']},
{cipher, [{client2server, ['aes256-gcm@openssh.com',
    'aes256-ctr', 'aes192-ctr', 'aes128-gcm@openssh.com',
    'aes128-ctr', 'aes128-cbc', '3des-cbc']},
  {server2client, ['aes256-gcm@openssh.com', 'aes256-ctr',
    'aes192-ctr', 'aes128-gcm@openssh.com', 'aes128-ctr',
    'aes128-cbc', '3des-cbc']]),
{mac, [{client2server, ['hmac-sha2-256', 'hmac-sha2-512',
    'hmac-sha1']},
  {server2client, ['hmac-sha2-256', 'hmac-sha2-512',
    'hmac-sha1']})],
{compression, [{client2server, [none, 'zlib@openssh.com', 'zlib']},
  {server2client, [none, 'zlib@openssh.com', 'zlib']})]
```

And the result shows that the Diffie-Hellman Group1 is added at the head of the kex list

**Example 6**

In this example, we in put the 'diffie-hellman-group1-sha1' first and also move the 'ecdh-sha2-nistp521' to the end in the kex list, that is, append it.
1.4 Configuring algorithms in SSH

Note that the appended algorithm is removed from its original place and then appended to the same list.

Example 7

In this example, we use both options (preferred_algorithms and modify_algorithms) and also try to prepend an unsupported algorithm. Any unsupported algorithm is quietly removed.

It is of course questionable why anyone would like to use the both these options together, but it is possible if an unforeseen need should arise.
2 Reference Manual

The ssh application is an Erlang implementation of the Secure Shell Protocol (SSH) as defined by RFC 4250 - 4254.
SSH

Application

The ssh application is an implementation of the SSH protocol in Erlang. ssh offers API functions to write customized SSH clients and servers as well as making the Erlang shell available over SSH. An SFTP client, ssh_sftp, and server, ssh_sftpd, are also included.

DEPENDENCIES

The ssh application uses the applications public_key and crypto to handle public keys and encryption. Hence, these applications must be loaded for the ssh application to work. In an embedded environment this means that they must be started with application:start/1,2 before the ssh application is started.

CONFIGURATION

The ssh application does not have an application-specific configuration file, as described in application(3). However, by default it use the following configuration files from OpenSSH:

- known_hosts
- authorized_keys
- authorized_keys2
- id_dsa
- id_rsa
- id_ecdsa
- ssh_host_dsa_key
- ssh_host_rsa_key
- ssh_host_ecdsa_key

By default, ssh looks for id_dsa, id_rsa, id_ecdsa_key, known_hosts, and authorized_keys in ~/.ssh, and for the host key files in /etc/ssh. These locations can be changed by the options user_dir and system_dir.

Public key handling can also be customized through a callback module that implements the behaviors ssh_client_key_api and ssh_server_key_api.

See also the default callback module documentation in ssh_file.

Public Keys

id_dsa, id_rsa and id_ecdsa are the users private key files. Notice that the public key is part of the private key so the ssh application does not use the id_<*>.pub files. These are for the user's convenience when it is needed to convey the user's public key.

Known Hosts

The known_hosts file contains a list of approved servers and their public keys. Once a server is listed, it can be verified without user interaction.

Authorized Keys

The authorized_key file keeps track of the user's authorized public keys. The most common use of this file is to let users log in without entering their password, which is supported by the Erlang ssh daemon.
**Host Keys**

RSA, DSA and ECDSA host keys are supported and are expected to be found in files named `ssh_host_rsa_key`, `ssh_host_dsa_key` and `ssh_host_ecdsa_key`.

**ERROR LOGGER AND EVENT HANDLERS**

The `ssh` application uses the default *OTP error logger* to log unexpected errors or print information about special events.

**SUPPORTED SPECIFICATIONS AND STANDARDS**

The supported SSH version is 2.0.

**Algorithms**

The actual set of algorithms may vary depending on which OpenSSL crypto library that is installed on the machine. For the list on a particular installation, use the command `ssh:default_algorithms/0`. The user may override the default algorithm configuration both on the server side and the client side. See the options `preferred_algorithms` and `modify_algorithms` in the `ssh:daemon/1,2,3` and `ssh:connect/3,4` functions.

Supported algorithms are (in the default order):

Key exchange algorithms:
- `ecdh-sha2-nistp384`
- `ecdh-sha2-nistp521`
- `ecdh-sha2-nistp256`
- `diffie-hellman-group-exchange-sha256`
- `diffie-hellman-group16-sha512`
- `diffie-hellman-group18-sha512`
- `diffie-hellman-group14-sha256`
- `curve25519-sha256`
- `curve25519-sha256@libssh.org`
- `curve448-sha512`
- `diffie-hellman-group14-sha1`
- `diffie-hellman-group-exchange-shal`
- `(diffie-hellman-group1-shal, retired: It can be enabled with the `preferred_algorithms` or `modify_algorithms` options. Use for example the Option value `{modify_algorithms, [{kex, ['diffie-hellman-group1-shal']}]}`)

Public key algorithms:
- `ecdsa-sha2-nistp384`
- `ecdsa-sha2-nistp521`
- `ecdsa-sha2-nistp256`
- `ssh-ed25519`
- `ssh-ed448`
- `ssh-rsa`
- `rsa-sha2-256`
- `rsa-sha2-512`
- `ssh-dss`
MAC algorithms
- hmac-sha2-256
- hmac-sha2-512
- hmac-sha1
- (hmac-sha1-96) It can be enabled with the **preferred_algorithms** or **modify_algorithms** options. Use for example the Option value `{modify_algorithms, [[append, [[mac,['hmac-sha1-96']]]]]}`

Encryption algorithms (ciphers)
- chacha20-poly1305@openssh.com
- aes256-gcm@openssh.com
- aes256-ctr
- aes192-ctr
- aes128-gcm@openssh.com
- aes128-ctr
- aes256-cbc
- aes192-cbc
- aes128-cbc
- 3des-cbc
- (AEAD_AES_128_GCM, not enabled per default)
- (AEAD_AES_256_GCM, not enabled per default)

See the text at the description of the **rfc 5647 further down** for more information regarding AEAD_AES_*_GCM.

Following the internet de-facto standard, the cipher and mac algorithm AEAD_AES_128_GCM is selected when the cipher aes128-gcm@openssh.com is negotiated. The cipher and mac algorithm AEAD_AES_256_GCM is selected when the cipher aes256-gcm@openssh.com is negotiated.

Compression algorithms
- none
- zlib@openssh.com
- zlib

Unicode support
Unicode filenames are supported if the emulator and the underlaying OS support it. See section DESCRIPTION in the file manual page in Kernel for information about this subject.
The shell and the cli both support unicode.

Rfcs
The following rfc:s are supported:
  Except
  - 9.4.6 Host-Based Authentication
  - 9.5.2 Proxy Forwarding
  - 9.5.3 X11 Forwarding
Except
• 9. Host-Based Authentication: "hostbased"

• **RFC 4253**, The Secure Shell (SSH) Transport Layer Protocol.
  Except
  • 8.1. diffie-hellman-group1-sha1. Disabled by default, can be enabled with the `preferred_algorithms` or `modify_algorithms` options.

• **RFC 4254**, The Secure Shell (SSH) Connection Protocol.
  Except
  • 6.3. X11 Forwarding
  • 7. TCP/IP Port Forwarding

  Except
  • `num-prompts > 1`
  • password changing
  • other identification methods than userid-password


• **RFC 4716**, The Secure Shell (SSH) Public Key File Format.

• **RFC 5647**, AES Galois Counter Mode for the Secure Shell Transport Layer Protocol.

  There is an ambiguity in the synchronized selection of cipher and mac algorithm. This is resolved by OpenSSH in the ciphers aes128-gcm@openssh.com and aes256-gcm@openssh.com which are implemented. If the explicit ciphers and macs `AEAD_AES_128_GCM` or `AEAD_AES_256_GCM` are needed, they could be enabled with the options `preferred_algorithms` or `modify_algorithms`.

  **Warning:**
  If the client or the server is not Erlang/OTP, it is the users responsibility to check that other implementation has the same interpretation of `AEAD_AES_*_GCM` as the Erlang/OTP SSH before enabling them. The `aes*`-gcm@openssh.com variants are always safe to use since they lack the ambiguity.

The second paragraph in section 5.1 is resolved as:
• If the negotiated cipher is `AEAD_AES_128_GCM`, the mac algorithm is set to `AEAD_AES_128_GCM`.
• If the negotiated cipher is `AEAD_AES_256_GCM`, the mac algorithm is set to `AEAD_AES_256_GCM`.
• If the mac algorithm is `AEAD_AES_128_GCM`, the cipher is set to `AEAD_AES_128_GCM`.
• If the mac algorithm is `AEAD_AES_256_GCM`, the cipher is set to `AEAD_AES_256_GCM`.

  The first rule that matches when read in order from the top is applied

• **RFC 5656**, Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer.
  Except
  • 5. ECMQV Key Exchange
  • 6.4. ECMQV Key Exchange and Verification Method Name
  • 7.2. ECMQV Message Numbers
  • 10.2. Recommended Curves
• **RFC 6668**, SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol
  Comment: Defines hmac-sha2-256 and hmac-sha2-512
• **Draft-ietf-curdle-ssh-kex-sha2 (work in progress)**, Key Exchange (KEX) Method Updates and Recommendations for Secure Shell (SSH).
  Deviations:
  • The `diffie-hellman-group1-sha1` is not enabled by default, but is still supported and can be enabled with the options `preferred_algorithms` or `modify_algorithms`.
  • The questionable sha1-based algorithms `diffie-hellman-group-exchange-sha1` and `diffie-hellman-group14-sha1` are still enabled by default for compatibility with ancient clients and servers. They can be disabled with the options `preferred_algorithms` or `modify_algorithms`. They will be disabled by default when the draft is turned into an RFC.
  Implemented are:
  • The Extension Negotiation Mechanism
  • The extension `server-sig-algs`
• **Secure Shell (SSH) Key Exchange Method using Curve25519 and Curve448 (work in progress)**
• **Ed25519 and Ed448 public key algorithms for the Secure Shell (SSH) protocol (work in progress)**

**SEE ALSO**

`application(3)`
This is the interface module for the SSH application. The Secure Shell (SSH) Protocol is a protocol for secure remote login and other secure network services over an insecure network. See ssh(6) for details of supported RFCs, versions, algorithms and unicode handling.

With the SSH application it is possible to start clients and to start daemons (servers).

Clients are started with `connect/2`, `connect/3` or `connect/4`. They open an encrypted connection on top of TCP/IP. In that encrypted connection one or more channels could be opened with `ssh_connection:session_channel/2,4`.

Each channel is an isolated "pipe" between a client-side process and a server-side process. Those process pairs could handle for example file transfers (sftp) or remote command execution (shell, exec and/or cli). If a custom shell is implemented, the user of the client could execute the special commands remotely. Note that the user is not necessarily a human but probably a system interfacing the SSH app.

A server-side subsystem (channel) server is requested by the client with `ssh_connection:subsystem/4`.

A server (daemon) is started with `daemon/1`, `daemon/2` or `daemon/3`. Possible channel handlers (subsystems) are declared with the `subsystem` option when the daemon is started.

To just run a shell on a remote machine, there are functions that bundles the needed three steps needed into one: `shell/1,2,3`. Similarly, to just open an sftp (file transfer) connection to a remote machine, the simplest way is to use `ssh_sftp:start_channel/1,2,3`.

To write your own client channel handler, use the behaviour `ssh_client_channel`. For server channel handlers use `ssh_server_channel` behaviour (replaces `ssh_daemon_channel`).

Both clients and daemons accepts options that controls the exact behaviour. Some options are common to both. The three sets are called `Client Options`, `Daemon Options` and `Common Options`.

The descriptions of the options uses the Erlang Type Language with explaining text.

### Keys and files

A number of objects must be present for the SSH application to work. Thoose objects are per default stored in files.

The default names, paths and file formats are the same as for OpenSSH. Keys could be generated with the `ssh-keygen` program from OpenSSH. See the User's Guide.

The paths could easily be changed by options: `user_dir` and `system_dir`.

A completly different storage could be interfaced by writing call-back modules using the behaviours `ssh_client_key_api` and/or `ssh_server_key_api`. A callback module is installed with the option `key_cb` to the client and/or the daemon.

### Daemons

The keys are by default stored in files:

- Mandatory: one or more Host key(s), both private and public. Default is to store them in the directory `/etc/ssh` in the files
  - `ssh_host_dsa_key` and `ssh_host_dsa_key.pub`
• ssh_host_rsa_key and ssh_host_rsa_key.pub
• ssh_host_ecdsa_key and ssh_host_ecdsa_key.pub

The host keys directory could be changed with the option system_dir.

• Optional: one or more User's public key in case of publickey authorization. Default is to store them concatenated in the file .ssh/authorized_keys in the user's home directory.

The user keys directory could be changed with the option user_dir.

Clients

The keys and some other data are by default stored in files in the directory .ssh in the user's home directory. The directory could be changed with the option user_dir.

• Optional: a list of Host public key(s) for previously connected hosts. This list is handled by the SSH application without any need of user assistance. The default is to store them in the file known_hosts.

The host_accepting_client_options() are associated with this list of keys.

• Optional: one or more User's private key(s) in case of publickey authorization. The default files are
  • id_dsa and id_dsa.pub
  • id_rsa and id_rsa.pub
  • id_ecdsa and id_ecdsa.pub

Data Types

Client Options

client_options() = [client_option()]

client_option() =
  ssh_file:pubkey_passphrase_client_options() |
  host_accepting_client_options() |
  authentication_client_options() |
  diffie_hellman_group_exchange_client_option() |
  connect_timeout_client_option() |
  recv_ext_info_client_option() |
  opaque_client_options() |
  gen_tcp:connect_option() |
  common_option()

Options for clients. The individual options are further explained below or by following the hyperlinks.

host_accepting_client_options() =
  {silently_accept_hosts, accept_hosts()} |
  {user_interaction, boolean()} |
  {save_accepted_host, boolean()} |
  {quiet_mode, boolean()}

accept_hosts() =
  boolean() |
  accept_callback() |
  {HashAlgoSpec :: fp_digest_alg(), accept_callback()}

fp_digest_alg() = md5 | crypto:sha1() | crypto:sha2()

accept_callback() =
fun((PeerName :: string(), fingerprint()) -> boolean())
fingerprint() = string() | [string()]
silently_accept_hosts

This option guides the `connect` function on how to act when the connected server presents a Host Key that the client has not seen before. The default is to ask the user with a question on stdio of whether to accept or reject the new Host Key. See the option `user_dir` for specifying the path to the file `known_hosts` where previously accepted Host Keys are recorded. See also the option `key_cb` for the general way to handle keys.

The option can be given in three different forms as seen above:

- The value is a boolean(). The value `true` will make the client accept any unknown Host Key without any user interaction. The value `false` preserves the default behaviour of asking the user on stdio.
- An accept_callback() will be called and the boolean return value `true` will make the client accept the Host Key. A return value of `false` will make the client to reject the Host Key and as a result the connection will be closed. The arguments to the fun are:
  - PeerName - a string with the name or address of the remote host.
  - FingerPrint - the fingerprint of the Host Key as `public_key:ssh_hostkey_fingerprint/1` calculates it.
- A tuple `{HashAlgoSpec, accept_callback}`). The HashAlgoSpec specifies which hash algorithm shall be used to calculate the fingerprint used in the call of the accept_callback(). The HashAlgoSpec is either an atom or a list of atoms as the first argument in `public_key:ssh_hostkey_fingerprint/2`. If it is a list of hash algorithm names, the FingerPrint argument in the accept_callback() will be a list of fingerprints in the same order as the corresponding name in the HashAlgoSpec list.

user_interaction

If `false`, disables the client to connect to the server if any user interaction is needed, such as accepting the server to be added to the `known_hosts` file, or supplying a password.

Even if user interaction is allowed it can be suppressed by other options, such as `silently_accept_hosts` and `password`. However, those options are not always desirable to use from a security point of view.

Defaults to `true`.

save_accepted_host

If `true`, the client saves an accepted host key to avoid the accept question the next time the same host is connected. If the option `key_cb` is not present, the key is saved in the file "known_hosts". See option `user_dir` for the location of that file.

If `false`, the key is not saved and the key will still be unknown at the next access of the same host.

Defaults to `true`.

quiet_mode

If `true`, the client does not print anything on authorization.

Defaults to `false`.

authentication_client_options() =
  {user, string()} | {password, string()}

user

Provides the username. If this option is not given, `ssh` reads from the environment (LOGNAME or USER on UNIX, USERNAME on Windows).
password

Provides a password for password authentication. If this option is not given, the user is asked for a password, if the password authentication method is attempted.

diffie_hellman_group_exchange_client_option() =
{dh_gex_limits,
{Min :: integer() => 1,
I :: integer() => 1,
Max :: integer() => 1}}

Sets the three diffie-hellman-group-exchange parameters that guides the connected server in choosing a group. See RFC 4419 for the details. The default value is \{1024, 6144, 8192\}.

connect_timeout_client_option() = \{connect_timeout, timeout()\}

Sets a timeout on the transport layer connect time. For gen_tcp the time is in milli-seconds and the default value is \texttt{infinity}.

See the parameter \texttt{Timeout} in \texttt{connect/4} for a timeout of the negotiation phase.

recv_ext_info_client_option() = \{recv_ext_info, boolean()\}

Make the client tell the server that the client accepts extension negotiation, that is, include \texttt{ext-info-c} in the kexinit message sent. See RFC 8308 for details and \texttt{ssh(6)} for a list of currently implemented extensions.

Default value is \texttt{true} which is compatible with other implementations not supporting ext-info.

Daemon Options (Server Options)

daoen_options() = \[\texttt{daemon\_option()}\]

\texttt{daemon\_option()} =

\texttt{subsystem\_daemon\_option()} \mid
\texttt{shell\_daemon\_option()} \mid
\texttt{exec\_daemon\_option()} \mid
\texttt{ssh\_cli\_daemon\_option()} \mid
\texttt{authentication\_daemon\_options()} \mid
\texttt{diffie\_hellman\_group\_exchange\_daemon\_option()} \mid
\texttt{negotiation\_timeout\_daemon\_option()} \mid
\texttt{hardening\_daemon\_options()} \mid
\texttt{callbacks\_daemon\_options()} \mid
\texttt{send\_ext\_info\_daemon\_option()} \mid
\texttt{opaque\_daemon\_options()} \mid
\texttt{gen\_tcp:listen\_option()} \mid
\texttt{common\_option()}

Options for \texttt{daemons}. The individual options are further explained below or by following the hyperlinks.

subsystem_daemon_option() = \{subsystems, subsystem_specs()\}

subsystem_specs() = \[subsystem_spec()\]

subsystem_spec() = \{Name :: string(), mod_args()\}

Defines a subsystem in the daemon.

The \texttt{subsystem\_name} is the name that a client requests to start with for example \texttt{ssh\_connection:subsystem/4}.

The \texttt{channel\_callback} is the module that implements the \texttt{ssh\_server\_channel} (replaces \texttt{ssh\_daemon\_channel}) behaviour in the daemon. See the section \texttt{Creating a Subsystem} in the User’s Guide for more information and an example.
If the subsystems option is not present, the value of ssh_sftpd:subsystem_spec([]) is used. This enables the sftp subsystem by default. The option can be set to the empty list if you do not want the daemon to run any subsystems.

```erlang
shell_daemon_option() = {shell, shell_spec()}
shell_spec() = mod_fun_args() | shell_fun() | disabled
shell_fun() = 'shell_fun/1'() | 'shell_fun/2'()
'shell_fun/1'() = fun((User :: string()) -> pid())
'shell_fun/2'() = fun((User :: string(), PeerAddr :: inet:ip_address()) -> pid())
```

Defines the read-eval-print loop used in a daemon when a shell is requested by the client. The default is to use the Erlang shell: `{shell, start, []}`

See the option exec-option for a description of how the daemon executes shell-requests and exec-requests depending on the shell- and exec-options.

```erlang
eexec_daemon_option() = {exec, exec_spec()}
eexec_spec() =
  {direct, exec_fun()} | disabled | deprecated_exec_opt()
eexec_fun() = 'exec_fun/1'() | 'exec_fun/2'() | 'exec_fun/3'()
'exec_fun/1'() = fun((Cmd :: string()) -> exec_result())
'exec_fun/2'() = fun((Cmd :: string(), User :: string()) -> exec_result())
'exec_fun/3'() = fun((Cmd :: string(), User :: string(), ClientAddr :: ip_port()) -> exec_result())
eexec_result() =
  {ok, Result :: term()} | {error, Reason :: term()}
```

This option changes how the daemon executes exec-requests from clients. The term in the return value is formatted to a string if it is a non-string type. No trailing newline is added in the ok-case.

See the User's Guide section on One-Time Execution for examples.

Error texts are returned on channel-type 1 which usually is piped to stderr on e.g Linux systems. Texts from successful execution are returned on channel-type 0 and will in similar manner be piped to stdout. The exit-status code is set to 0 for success and 255 for errors. The exact results presented on the client side depends on the client and the client's operating system.

In case of the {direct, exec_fun()} variant or no exec-option at all, all reads from standard_input will be from the received data-events of type 0. Those are sent by the client. Similarly all writes to standard_output will be sent as data-events to the client. An OS shell client like the command 'ssh' will usually use stdin and stdout for the user interface.

The option cooperates with the daemon-option shell in the following way:

1. If neither the exec-option nor the shell-option is present:

   The default Erlang evaluator is used both for exec and shell requests. The result is returned to the client.

2. If the exec_spec's value is disabled (the shell-option may or may not be present):

   No exec-requests are executed but shell-requests are not affected, they follow the shell_spec's value.
3. If the `exec-option` is present and the `exec_spec` value =/= `disabled` (the `shell-option` may or may not be present):

   The `exec_spec` fun() is called with the same number of parameters as the arity of the fun, and the result is returned to the client. Shell-requests are not affected, they follow the `shell_spec`'s value.

4. If the `exec-option` is present and the `shell-option` is present with the default Erlang shell as the `shell_spec`'s value:

   The default Erlang evaluator is used both for exec and shell requests. The result is returned to the client.

5. If the `exec-option` is absent, and the `shell-option` is present with a value that is neither the default Erlang shell nor the value `disabled`:

   The exec-request is not evaluated and an error message is returned to the client. Shell-requests are executed according to the value of the `shell_spec`.

6. If the `exec-option` is absent, and the `shell_spec`'s value is `disabled`:

   Exec requests are executed by the default shell, but shell-requests are not executed.

If a custom CLI is installed (see the option `ssh_cli`) the rules above are replaced by those implied by the custom CLI.

**Note:**

The `exec-option` has existed for a long time but has not previously been documented. The old definition and behaviour are retained but obey the rules 1-6 above if conflicting. The old and undocumented style should not be used in new programs.

deprecated_exec_opt() = function() | `mod_fun_args()`

Old-style exec specification that are kept for compatibility, but should not be used in new programs

`ssh_cli_daemon_option()` = `{ssh_cli, `mod_args()` | no_cli}

Provides your own CLI implementation in a daemon.

It is a channel callback module that implements a shell and command execution. The shell's read-eval-print loop can be customized, using the option `shell`. This means less work than implementing an own CLI channel. If `ssh_cli` is set to `no_cli`, the CLI channels like `shell` and `exec` are disabled and only subsystem channels are allowed.

authentication_daemon_options() =

   `ssh_file:system_dir_daemon_option()` |
   {auth_method_kb_interactive_data, prompt_texts()}
   | {user_passwords, [{UserName :: string(), Pwd :: string()}]} |
   {password, string()}
   | {pwdfun, pwdfun_2() | pwdfun_4()}

prompt_texts() = `kb_int_tuple()` | `kb_int_fun_3()`

`kb_int_tuple()` =

   {Name :: string(),
    Instruction :: string(),
    Prompt :: string(),
    Echo :: boolean()}

`kb_int_fun_3()` =

   fun((Peer :: `ip_port()`, User :: string(), Service :: string()) ->
      `kb_int_tuple()`)  

pwdfun_2() =
fun((User :: string(), Password :: string()) -> boolean())

pwdfun_4() = fun((User :: string(), Password :: string(), PeerAddress :: ip_port(), State :: any()) -> boolean() | disconnect | {boolean(), NewState :: any()})

auth_method_kb_interactive_data

Sets the text strings that the daemon sends to the client for presentation to the user when using keyboard-interactive authentication.

If the fun/3 is used, it is called when the actual authentication occurs and may therefore return dynamic data like time, remote ip etc.

The parameter Echo guides the client about need to hide the password.

The default value is: {auth_method_kb_interactive_data, ("SSH server", "Enter password for ">

user_passwords

Provides passwords for password authentication. The passwords are used when someone tries to connect to the server and public key user-authentication fails. The option provides a list of valid usernames and the corresponding passwords.

corresponding passwords.

password

Provides a global password that authenticates any user.

Warning:

Intended to facilitate testing.

From a security perspective this option makes the server very vulnerable.

pwdfun with pwdfun_4()

Provides a function for password validation. This could used for calling an external system or handling passwords stored as hash values.

This fun can also be used to make delays in authentication tries for example by calling timer:sleep/1.

To facilitate for instance counting of failed tries, the State variable could be used. This state is per connection only. The first time the pwdfun is called for a connection, the State variable has the value undefined.

The fun should return:
• true if the user and password is valid
• false if the user or password is invalid
• disconnect if a SSH_MSG_DISCONNECT message should be sent immediately. It will be followed by a close of the underlying tcp connection.
• {true, NewState: any()} if the user and password is valid
• {false, NewState: any()} if the user or password is invalid

A third usage is to block login attempts from a missbehaving peer. The State described above can be used for this. The return value disconnect is useful for this.
pwdfun with pwdfun_2()

Provides a function for password validation. This function is called with user and password as strings, and returns:

• true if the user and password is valid
• false if the user or password is invalid

This variant is kept for compatibility.

diffie_hellman_group_exchange_daemon_option() =
{dh_gex_groups,
  [explicit_group(),] |
  explicit_group_file() |
  ssh_moduli_file()} |
{dh_gex_limits, {Min :: integer() >= 1, Max :: integer() >= 1}}

explicit_group() =
{Size :: integer() >= 1,
 G :: integer() >= 1,
 P :: integer() >= 1}

explicit_group_file() = {file, string()}

ssh_moduli_file() = {ssh_moduli_file, string()}

dh_gex_groups

Defines the groups the server may choose among when diffie-hellman-group-exchange is negotiated. See RFC 4419 for details. The three variants of this option are:

{Size=integer(),G=integer(),P=integer()}

The groups are given explicitly in this list. There may be several elements with the same Size. In such a case, the server will choose one randomly in the negotiated Size.

{file, filename()}

The file must have one or more three-tuples \{Size=integer(),G=integer(),P=integer()\} terminated by a dot. The file is read when the daemon starts.

{ssh_moduli_file, filename()}

The file must be in ssh-keygen moduli file format. The file is read when the daemon starts.

The default list is fetched from the public_key application.

dh_gex_limits

Limits what a client can ask for in diffie-hellman-group-exchange. The limits will be \{MaxUsed = \min\{MaxClient,Max\}, MinUsed = \max\{MinClient,Min\}\} where MaxClient and MinClient are the values proposed by a connecting client.

The default value is \{0,infinity\}.

If MaxUsed < MinUsed in a key exchange, it will fail with a disconnect.

See RFC 4419 for the function of the Max and Min values.

negotiation_timeout_daemon_option() =
{negotiation_timeout, timeout()}

Maximum time in milliseconds for the authentication negotiation. Defaults to 120000 ms (2 minutes). If the client fails to log in within this time, the connection is closed.

hardening_daemon_options() =
{max_sessions, integer() >= 1} |
{max_channels, integer() >= 1} |
{parallel_login, boolean()} |
{minimal_remote_max_packet_size, integer() >= 1}

max_sessions

The maximum number of simultaneous sessions that are accepted at any time for this daemon. This includes sessions that are being authorized. Thus, if set to N, and N clients have connected but not started the login process, connection attempt N+1 is aborted. If N connections are authenticated and still logged in, no more logins are accepted until one of the existing ones log out.

The counter is per listening port. Thus, if two daemons are started, one with \{max_sessions,N\} and the other with \{max_sessions,M\}, in total N+M connections are accepted for the whole ssh application.

Notice that if parallel_login is false, only one client at a time can be in the authentication phase.

By default, this option is not set. This means that the number is not limited.

max_channels

The maximum number of channels with active remote subsystem that are accepted for each connection to this daemon

By default, this option is not set. This means that the number is not limited.

parallel_login

If set to false (the default value), only one login is handled at a time. If set to true, an unlimited number of login attempts are allowed simultaneously.

If the max_sessions option is set to N and parallel_login is set to true, the maximum number of simultaneous login attempts at any time is limited to N-K, where K is the number of authenticated connections present at this daemon.

Warning:

Do not enable parallel_logins without protecting the server by other means, for example, by the max_sessions option or a firewall configuration. If set to true, there is no protection against DOS attacks.

minimal_remote_max_packet_size

The least maximum packet size that the daemon will accept in channel open requests from the client. The default value is 0.

callbacks_daemon_options() =
{failfun,
 fun((User :: string(),
      PeerAddress :: inet:ip_address(),
      Reason :: term()) ->
      term()) |
{connectfun,
 fun((User :: string(),
      PeerAddress :: inet:ip_address(),
      Method :: string()) ->
      term())}

connectfun

Provides a fun to implement your own logging when a user authenticates to the server.

failfun

Provides a fun to implement your own logging when a user fails to authenticate.
ssh

send_ext_info_daemon_option() = {send_ext_info, boolean()}

Make the server (daemon) tell the client that the server accepts extension negotiation, that is, include ext-info-s in the kexinit message sent. See RFC 8308 for details and ssh(6) for a list of currently implemented extensions.

Default value is true which is compatible with other implementations not supporting ext-info.

Options common to clients and daemons

common_options() = [common_option()]

common_option() = ssh_file:user_dir_common_option() | profile_common_option() | max_idle_time_common_option() | key_cb_common_option() | disconnectfun_common_option() | unexpectedfun_common_option() | ssh_msg_debug_fun_common_option() | rekey_limit_common_option() | id_string_common_option() | pref_public_key_algs_common_option() | preferred_algorithms_common_option() | modify_algorithms_common_option() | auth_methods_common_option() | inet_common_option() | fd_common_option()

The options above can be used both in clients and in daemons (servers). They are further explained below.

profile_common_option() = {profile, atom()}

Used together with ip-address and port to uniquely identify a ssh daemon. This can be useful in a virtualized environment, where there can be more that one server that has the same ip-address and port. If this property is not explicitly set, it is assumed that the ip-address and port uniquely identifies the SSH daemon.

max_idle_time_common_option() = {idle_time, timeout()}

Sets a time-out on a connection when no channels are open. Defaults to infinity. The unit is milliseconds.

The timeout is not active until channels are started, so it does not limit the time from the connection creation to the first channel opening.

rekey_limit_common_option() = {rekey_limit,
 Bytes ::
    limit_bytes() | 
    {Minutes :: limit_time(), Bytes :: limit_bytes()}}

limit_bytes() = integer() >= 0 | infinity
limit_time() = integer() >= 1 | infinity

Sets the limit when rekeying is to be initiated. Both the max time and max amount of data could be configured:

• (Minutes, Bytes) initiate rekeying when any of the limits are reached.
• Bytes initiate rekeying when Bytes number of bytes are transferred, or at latest after one hour.

When a rekeying is done, both the timer and the byte counter are restarted. Defaults to one hour and one GByte.

If Minutes is set to infinity, no rekeying will ever occur due to that max time has passed. Setting Bytes to infinity will inhibit rekeying after a certain amount of data has been transferred. If the option value is set
to \((\text{infinity, infinity})\), no rekeying will be initiated. Note that rekeying initiated by the peer will still be performed.

\[
\text{key\_cb\_common\_option}() = \\
\{\text{key\_cb,} \\
\quad \text{Module : atom()} | \{\text{Module : atom(), Opts : [term()]}\} \}
\]

Module implementing the behaviour ssh\_client\_key\_api and/or ssh\_server\_key\_api. Can be used to customize the handling of public keys. If callback options are provided along with the module name, they are made available to the callback module via the options passed to it under the key 'key\_cb\_private'.

The Opts defaults to [] when only the Module is specified.

The default value of this option is \((\text{ssh\_file, []})\). See also the manpage of ssh\_file.

A call to the call-back function \(F\) will be

\[
\text{Module:F}(..., \{(\text{key\_cb\_private, Opts})|\text{UserOptions}\})
\]

where \(...\) are arguments to \(F\) as in ssh\_client\_key\_api and/or ssh\_server\_key\_api. The UserOptions are the options given to ssh\:connect, ssh\:shell or ssh\:daemon.

\[
\text{pref\_public\_key\_algs\_common\_option}() = \\
\{\text{pref\_public\_key\_algs, [pubkey\_alg()]} \}
\]

List of user (client) public key algorithms to try to use.

The default value is the public\_key entry in the list returned by ssh\:default\_algorithms/0.

If there is no public key of a specified type available, the corresponding entry is ignored. Note that the available set is dependent on the underlying cryptolib and current user’s public keys.

See also the option user\_dir for specifying the path to the user’s keys.

\[
\text{disconnectfun\_common\_option}() = \\
\{\text{disconnectfun, fun((Reason :: term()) -> void | any())} \}
\]

Provides a fun to implement your own logging when the peer disconnects.

\[
\text{unexpectedfun\_common\_option}() = \\
\{\text{unexpectedfun, fun((Message :: term(), \{Host :: term(), Port :: term()\}) -> report | skip}) \}
\]

Provides a fun to implement your own logging or other action when an unexpected message arrives. If the fun returns report the usual info report is issued but if skip is returned no report is generated.

\[
\text{ssh\_msg\_debug\_fun\_common\_option}() = \\
\{\text{ssh\_msg\_debug\_fun,} \\
\quad \text{fun((ssh:connection\_ref(),} \\
\quad \quad \text{AlwaysDisplay :: boolean(),} \\
\quad \quad \text{Msg :: binary(),} \\
\quad \quad \text{LanguageTag :: binary()} -> \}
\quad \text{any())} \}
\]

Provide a fun to implement your own logging of the SSH message \text{SSH\_MSG\_DEBUG}. The last three parameters are from the message, see RFC 4253, section 11.3. The \text{connection\_ref}() is the reference to the connection on which the message arrived. The return value from the fun is not checked.

The default behaviour is ignore the message. To get a printout for each message with AlwaysDisplay = true, use for example \{ssh\_msg\_debug\_fun, fun(_,true,M,_)-> io:format("DEBUG: ~p~n", [M]) end\}

Ericsson AB. All Rights Reserved.: SSH | 37
id_string_common_option() =
    {id_string,
     string() |
     random |
     {random, Nmin :: integer() >= 1, Nmax :: integer() >= 1}}

The string the daemon will present to a connecting peer initially. The default value is "Erlang/VSN" where VSN is the ssh application version number.

The value random will cause a random string to be created at each connection attempt. This is to make it a bit more difficult for a malicious peer to find the ssh software brand and version.

The value \{random, Nmin, Nmax\} will make a random string with at least Nmin characters and at most Nmax characters.

preferred_algorithms_common_option() =
    {preferred_algorithms, algs_list()}

algs_list() = [alg_entry()]

alg_entry() =
    {kex, [kex_alg()]} |
    {public_key, [pubkey_alg()]}
    {cipher, double_algs(cipher_alg())} |
    {mac, double_algs(mac_alg())} |
    {compression, double_algs(compression_alg())}

kex_alg() =
    'diffie-hellman-group-exchange-sha1' |
    'diffie-hellman-group-exchange-sha256' |
    'diffie-hellman-group1-sha1' | 'diffie-hellman-group14-sha1' |
    'diffie-hellman-group14-sha256' |
    'diffie-hellman-group16-sha512' |
    'diffie-hellman-group18-sha512' | 'curve25519-sha256' |
    'curve25519-sha256@libssh.org' | 'curve448-sha512' |
    'ecdh-sha2-nistp256' | 'ecdh-sha2-nistp384' |
    'ecdh-sha2-nistp521'

pubkey_alg() =
    'ecdsa-sha2-nistp256' | 'ecdsa-sha2-nistp384' |
    'ecdsa-sha2-nistp521' | 'ssh-ed25519' | 'ssh-ed448' |
    'rsa-sha2-256' | 'rsa-sha2-512' | 'ssh-dss' | 'ssh-rsa'

cipher_alg() =
    '3des-cbc' | 'AES/AES_128_GCM' | 'AES/AES_256_GCM' |
    'aes128-cbc' | 'aes128-ctr' | 'aes128-gcm@openssh.com' |
    'aes192-ctr' | 'aes192-cbc' | 'aes256-cbc' | 'aes256-ctr' |
    'aes256-gcm@openssh.com' | 'chacha20-poly1305@openssh.com'

mac_alg() =
    'AES/AES_128_GCM' | 'AES/AES_256_GCM' | 'hmac-shal' |
    'hmac-sha1-96' | 'hmac-sha2-256' | 'hmac-sha2-512'

compression_alg() = none | zlib | 'zlib@openssh.com'
double_algs(AlgType) =
    [{client2server, [AlgType]} | {server2client, [AlgType]}] |
    [AlgType]

List of algorithms to use in the algorithm negotiation. The default algs_list() can be obtained from default_algorithms/0.
If an alg_entry() is missing in the algs_list(), the default value is used for that entry.

Here is an example of this option:

```plaintext
{preferred_algorithms,
 [public_key,['ssh-rsa','ssh-dss']],&
 [cipher,[client2server,['aes128-ctr']],&
 [server2client,['aes128-cbc','3des-cbc']]),
 [mac,['hmac-sha2-256','hmac-sha1']],&
 [compression,[none,zlib]]
 }
```

The example specifies different algorithms in the two directions (client2server and server2client), for cipher but specifies the same algorithms for mac and compression in both directions. The kex (key exchange) is implicit but public_key is set explicitly.

For background and more examples see the User's Guide.

If an algorithm name occurs more than once in a list, the behaviour is undefined. The tags in the property lists are also assumed to occur at most one time.

**Warning:**

Changing the values can make a connection less secure. Do not change unless you know exactly what you are doing. If you do not understand the values then you are not supposed to change them.

```plaintext
modify_algorithms_common_option() =
 {modify_algorithms, modify_algs_list()}
```

```plaintext
modify_algs_list() =
 [{append, algs_list()} |
  {prepend, algs_list()} |
  {rm, algs_list()}]
```

Modifies the list of algorithms to use in the algorithm negotiation. The modifications are applied after the option preferred_algorithms (if existing) is applied.

The algorithm for modifications works like this:

• Input is the modify_algs_list() and a set of algorithms A obtained from the preferred_algorithms option if existing, or else from the ssh:default_algorithms/0.

• The head of the modify_algs_list() modifies A giving the result A'.

  The possible modifications are:

  • Append or prepend supported but not enabled algorithm(s) to the list of algorithms. If the wanted algorithms already are in A they will first be removed and then appended or prepended.

  • Remove (rm) one or more algorithms from A.

  • Repeat the modification step with the tail of modify_algs_list() and the resulting A'.

If an unsupported algorithm is in the modify_algs_list(), it will be silently ignored.

If there are more than one modify_algorithms options, the result is undefined.

Here is an example of this option:
The example specifies that:

- the old key exchange algorithm 'diffie-hellman-group1-sha1' should be the main alternative. It will be the main alternative since it is prepended to the list.
- The compression algorithm none (= no compression) is removed so compression is enforced.

For background and more examples see the User's Guide.

**inet_common_option() = {inet, inet | inet6}**

IP version to use when the host address is specified as any.

**auth_methods_common_option() = {auth_methods, string()}**

Comma-separated string that determines which authentication methods that the client shall support and in which order they are tried. Defaults to "publickey, keyboard-interactive, password"

Note that the client is free to use any order and to exclude methods.

**fd_common_option() = {fd, gen_tcp:socket()}**

Allows an existing file-descriptor to be used (passed on to the transport protocol).

**Other data types**

- **host() = string() | inet:ip_address() | loopback**
- **ip_port() = {inet:ip_address(), inet:port_number()}**
- **mod_args() = {Module :: atom(), Args :: list()}**
- **mod_fun_args() =
  
  {Module :: atom(), Function :: atom(), Args :: list()}**
- **open_socket() = gen_tcp:socket()**

The socket is supposed to be result of a gen_tcp:connect or a gen_tcp:accept. The socket must be in passive mode (that is, opened with the option {active,false}).

**daemon_ref()**

Opaque data type representing a daemon.

Returned by the functions daemon/1,2,3.

**connection_ref()**

Opaque data type representing a connection between a client and a server (daemon).

Returned by the functions connect/2,3,4 and ssh_sftp:start_channel/2,3.

**channel_id()**

Opaque data type representing a channel inside a connection.

Returned by the functions ssh_connection:session_channel/2,4.

**connection_info_tuple() =

  {client_version, version()} |
  {server_version, version()} |
  {user, string()} |
  {peer, {inet:hostname(), ip_port()}} |
{sockname, ip_port()} | 
{options, client_options()} | 
{algorithms, conn_info_algs()} | 
{channels, conn_info_channels()}

version() = {protocol_version(), software_version()}

protocol_version() = 
{Major :: integer() >= 1, Minor :: integer() >= 0}

software_version() = string()

conn_info_algs() =

[#{kex, kex_alg()} |
 {hkey, pubkey_alg()} |
 {encrypt, cipher_alg()} |
 {decrypt, cipher_alg()} |
 {send_mac, mac_alg()} |
 {recv_mac, mac_alg()} |
 {compress, compression_alg()} |
 {decompress, compression_alg()} |
 {send_ext_info, boolean()} |
 {recv_ext_info, boolean()}]

conn_info_channels() = [proplists:proplist()]

Return values from the connection_info/1 and connection_info/2 functions.

In the option info tuple are only the options included that differs from the default values.

daemon_info_tuple() =

{port, inet:port_number()} |
 {ip, inet:ip_address()} |
 {profile, atom()} |
 {options, daemon_options()}

Return values from the daemon_info/1 and daemon_info/2 functions.

In the option info tuple are only the options included that differs from the default values.

opaque_client_optionsopaque_daemon_optionsopaque_common_options

Opaque types that define experimental options that are not to be used in products.

Exports

close(ConnectionRef) -> ok | {error, term()}

Types:
   ConnectionRef = connection_ref()

Closes an SSH connection.

connect(Host, Port, Options) -> Result
connect(Host, Port, Options, NegotiationTimeout) -> Result
connect(TcpSocket, Options) -> Result
connect(TcpSocket, Options, NegotiationTimeout) -> Result

Types:
   Host = host()
Port = inet:port_number()
Options = client_options()
TcpSocket = open_socket()
NegotiationTimeout = timeout()
Result = {ok, connection_ref()} | {error, term()}

Connects to an SSH server at the Host on Port.
As an alternative, an already open TCP socket could be passed to the function in TcpSocket. The SSH initiation and negotiation will be initiated on that one with the SSH that should be at the other end.

No channel is started. This is done by calling ssh_connection:session_channel/[2, 4].
The NegotiationTimeout is in milli-seconds. The default value is infinity. For connection timeout, use the option connect_timeout.

connection_info(ConnectionRef) -> InfoTupleList
connection_info(ConnectionRef, Key ::ItemList | Item) ->
        InfoTupleList | InfoTuple

Types:
    ConnectionRef = connection_ref()
    ItemList = [Item]
    Item =
            client_version | server_version | user | peer | sockname |
                    options | algorithms | sockname
    InfoTupleList = [InfoTuple]
    InfoTuple = connection_info_tuple()

Returns information about a connection intended for e.g debugging or logging.
When the Key is a single Item, the result is a single InfoTuple.

set_sock_opts(ConnectionRef, SocketOptions) ->
        ok | {error, inet:posix()}

Types:
    ConnectionRef = connection_ref()
    SocketOptions = [gen_tcp:option()]

Sets tcp socket options on the tcp-socket below an ssh connection.
This function calls the inet:setopts/2, read that documentation and for gen_tcp:option(). All gen_tcp socket options except active, deliver, mode and packet are allowed. The excluded options are reserved by the SSH application.

Warning:
This is an extremely dangerous function. You use it on your own risk.
Some options are OS and OS version dependent. Do not use it unless you know what effect your option values will have on an TCP stream.
Some values may destroy the functionality of the SSH protocol.

get_sock_opts(ConnectionRef, SocketGetOptions) ->
ok | {error, inet:posix()}

Types:
  ConnectionRef = connection_ref()
  SocketGetOptions = [gen_tcp:option_name()]

Get tcp socket option values of the tcp-socket below an ssh connection.
This function calls the inet:getopts/2, read that documentation.

daemon(Port | TcpSocket) -> Result
daemon(Port | TcpSocket, Options) -> Result
daemon(HostAddress, Port, Options) -> Result

Types:
  Port = integer()
  TcpSocket = open_socket()
  Options = daemon_options()
  HostAddress = host() | any
  Result = {ok, daemon_ref()} | {error, atom()}

Starts a server listening for SSH connections on the given port. If the Port is 0, a random free port is selected. See daemon_info/1 about how to find the selected port number.

As an alternative, an already open TCP socket could be passed to the function in TcpSocket. The SSH initiation and negotiation will be initiated on that one when an SSH starts at the other end of the TCP socket.

For a description of the options, see Daemon Options.

Please note that by historical reasons both the HostAddress argument and the gen_tcp connect_option() {ip,Address} set the listening address. This is a source of possible inconsistent settings.

The rules for handling the two address passing options are:

• if HostAddress is an IP-address, that IP-address is the listening address. An 'ip'-option will be discarded if present.
• if HostAddress is the atom loopback, the listening address is loopback and an loopback address will be chosen by the underlying layers. An 'ip'-option will be discarded if present.
• if HostAddress is the atom any and no 'ip'-option is present, the listening address is any and the socket will listen to all addresses
• if HostAddress is any and an 'ip'-option is present, the listening address is set to the value of the 'ip'-option

daemon_info(DaemonRef) ->
  {ok, InfoTupleList} | {error, bad_daemon_ref}

daemon_info(DaemonRef, Key :: ItemList | Item) ->
  InfoTupleList | InfoTuple | {error, bad_daemon_ref}

Types:
  DaemonRef = daemon_ref()
  ItemList = [Item]
  Item = ip | port | profile | options
  InfoTupleList = [InfoTuple]
  InfoTuple = daemon_info_tuple()

Returns information about a daemon intended for e.g debugging or logging.
When the Key is a single Item, the result is a single InfoTuple.

Note that daemon_info/1 and daemon_info/2 returns different types due to compatibility reasons.

default_algorithms() -> algs_list()

Returns a key-value list, where the keys are the different types of algorithms and the values are the algorithms themselves.

See the User's Guide for an example.

shell(Host | TcpSocket) -> Result
shell(Host, Port, Options) -> Result
shell(Host, Port, Options, Options) -> Result

Types:
   Host = host()
   TcpSocket = open_socket()
   Port = inet:port_number()
   Options = client_options()
   Result = ok | {error, Reason::term()}

Connects to an SSH server at Host and Port (defaults to 22) and starts an interactive shell on that remote host.

As an alternative, an already open TCP socket could be passed to the function in TcpSocket. The SSH initiation and negotiation will be initiated on that one and finally a shell will be started on the host at the other end of the TCP socket.

For a description of the options, see Client Options.

The function waits for user input, and does not return until the remote shell is ended (that is, exit from the shell).

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

Types:
   Type = permanent | transient | temporary

Utility function that starts the applications crypto, public_key, and ssh. Default type is temporary. For more information, see the application(3) manual page in Kernel.

stop() -> ok | {error, term()}

Stops the ssh application. For more information, see the application(3) manual page in Kernel.

stop_daemon(DaemonRef :: daemon_ref()) -> ok
stop_daemon(Address :: inet:ip_address(),
  Port :: inet:port_number()) -> ok
stop_daemon(Address :: any | inet:ip_address(),
  Port :: inet:port_number(),
  Profile :: atom()) -> ok

Stops the listener and all connections started by the listener.
stop_listener(SysSup :: daemon_ref()) -> ok
stop_listener(Address :: inet:ip_address(),
               Port :: inet:port_number()) -> ok
stop_listener(Address :: any | inet:ip_address(),
               Port :: inet:port_number(),
               Profile :: term()) -> ok

Stops the listener, but leaves existing connections started by the listener operational.
SSH services (clients and servers) are implemented as channels that are multiplexed over an SSH connection and communicates over the **SSH Connection Protocol**. This module provides a callback API that takes care of generic channel aspects for clients, such as flow control and close messages. It lets the callback functions take care of the service (application) specific parts. This behavior also ensures that the channel process honors the principal of an OTP-process so that it can be part of a supervisor tree. This is a requirement of channel processes implementing a subsystem that will be added to the ssh applications supervisor tree.

When implementing a ssh subsystem for daemons, use `-behaviour(ssh_server_channel)` (Replaces ssh_daemon_channel) instead.

Functions in this module are not supposed to be called outside a module implementing this behaviour!

### Exports

```erlang
call(ChannelRef, Msg) ->
call(ChannelRef, Msg, Timeout) -> Reply | {error, Reason}
```

**Types:**

- `ChannelRef = pid()`
- `Msg = term()`
- `Timeout = timeout()`
- `Reply = term()`
- `Reason = closed | timeout`

Makes a synchronous call to the channel process by sending a message and waiting until a reply arrives, or a time-out occurs. The channel calls `Module:handle_call/3` to handle the message. If the channel process does not exist, `{error, closed}` is returned.

```erlang
cast(ChannelRef, Msg) -> ok
```

**Types:**

- `ChannelRef = pid()`
- As returned by `start_link/4`
**Msg = term()**

Sends an asynchronous message to the channel process and returns ok immediately, ignoring if the destination node
or channel process does not exist. The channel calls `Module:handle_cast/2` to handle the message.

```erlang
enter_loop(State) -> _
```

Types:

- `State` = `term()`
  - as returned by `init/1`

Makes an existing process an `ssh_client_channel` (replaces `ssh_channel`) process. Does not return, instead
the calling process enters the `ssh_client_channel` (replaces `ssh_channel`) process receive loop and become an
`ssh_client_channel` process. The process must have been started using one of the start functions in `proc_lib`,
see the `proc_lib(3)` manual page in STDLIB. The user is responsible for any initialization of the process and must
call `init/1`.

```erlang
init(Options) -> {ok, State} | {ok, State, Timeout} | {stop, Reason}
```

Types:

- `Options` = `[{Option, Value}]`
- `State` = `term()`
- `Timeout` = `timeout()`
- `Reason` = `term()`

The following options must be present:

- `(channel_cb, atom())`
  - The module that implements the channel behaviour.
- `(init_args(), list())`
  - The list of arguments to the `init` function of the callback module.
- `(cm, ssh:connection_ref())`
  - Reference to the `ssh` connection as returned by `ssh:connect/3`.
- `(channel_id, ssh:channel_id())`
  - Id of the `ssh` channel as returned by `ssh_connection:session_channel/2,4`.

**Note:**

This function is normally not called by the user. The user only needs to call if the channel process needs to be
started with help of `proc_lib` instead of calling `start/4` or `start_link/4`.

```erlang
reply(Client, Reply) -> _
```

Types:

- `Client` = `opaque()`
- `Reply` = `term()`

This function can be used by a channel to send a reply to a client that called `call/[2,3]` when the reply cannot be
declared in the return value of `Module:handle_call/3`.

- `Client` must be the `From` argument provided to the callback function `handle_call/3`. `Reply` is an arbitrary
term, which is given back to the client as the return value of `call/[2,3]`.
ssh_client_channel

start(SshConnection, ChannelId, ChannelCb, CbInitArgs) ->
start_link(SshConnection, ChannelId, ChannelCb, CbInitArgs) -> {ok,
ChannelRef} | {error, Reason}

Types:
  SshConnection = ssh:connection_ref()
  As returned by ssh:connect/3

  ChannelId = ssh:channel_id()
  As returned by ssh_connection:session_channel/[2,4].

  ChannelCb = atom()
  Name of the module implementing the service-specific parts of the channel.

  CbInitArgs = [term()]
  Argument list for the init function in the callback module.

  ChannelRef = pid()

Starts a process that handles an SSH channel. It is called internally, by the ssh daemon, or explicitly by the ssh client implementations. The behavior sets the trap_exit flag to true.

Callback Functions

The following functions are to be exported from a ssh_client_channel callback module.

Callback timeouts

The timeout values that can be returned by the callback functions have the same semantics as in a gen_server. If the time-out occurs, handle_msg/2 is called as handle_msg(timeout, State).

Exports

Module:code_change(OldVsn, State, Extra) -> {ok, NewState}

Types:
  OldVsn = term()
  In the case of an upgrade, OldVsn is Vsn, and in the case of a downgrade, OldVsn is {down, Vsn}. Vsn is defined by the vsn attribute(s) of the old version of the callback module Module. If no such attribute is defined, the version is the checksum of the BEAM file.

  State = term()
  Internal state of the channel.

  Extra = term()
  Passed "as-is" from the {advanced, Extra} part of the update instruction.

Converts process state when code is changed.

This function is called by a client-side channel when it is to update its internal state during a release upgrade or downgrade, that is, when the instruction \{update, Module, Change,...\}, where Change=\{advanced, Extra\}, is given in the appup file. For more information, refer to Section 9.11.6 Release Handling Instructions in the System Documentation.
Note:

Soft upgrade according to the OTP release concept is not straight forward for the server side, as subsystem channel processes are spawned by the ssh application and hence added to its supervisor tree. The subsystem channels can be upgraded when upgrading the user application, if the callback functions can handle two versions of the state, but this function cannot be used in the normal way.

Module:init(Args) -> {ok, State} | {ok, State, timeout()} | {stop, Reason}
Types:
  Args = term()
  Last argument to start_link/4.
  State = term()
  Reason = term()

Makes necessary initializations and returns the initial channel state if the initializations succeed.
For more detailed information on time-outs, see Section Callback timeouts.

Module:handle_call(Msg, From, State) -> Result
Types:
  Msg = term()
  From = opaque()
  Is to be used as argument to reply/2
  State = term()
  Result = {reply, Reply, NewState} | {reply, Reply, NewState, timeout()} |
           {noreply, NewState} | {noreply, NewState, timeout()} | {stop, Reason, 
           Reply, NewState} | {stop, Reason, NewState}
  Reply = term()
  Will be the return value of call/[2,3]
  NewState = term()
  Reason = term()

Handles messages sent by calling call/[2,3]
For more detailed information on time-outs, see Section Callback timeouts.

Module:handle_cast(Msg, State) -> Result
Types:
  Msg = term()
  State = term()
  Result = {noreply, NewState} | {noreply, NewState, timeout()} | {stop, 
           Reason, NewState}
  NewState = term()
  Reason = term()

Handles messages sent by calling cast/2.
For more detailed information on time-outs, see Section Callback timeouts.
ssh_client_channel

Module:handle_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}

Types:
  Msg = timeout | term()
  ChannelId = ssh:channel_id()
  State = term()

Handles other messages than SSH Connection Protocol, call, or cast messages sent to the channel.
Possible Erlang 'EXIT' messages is to be handled by this function and all channels are to handle the following message.

{ssh_channel_up, ssh:channel_id(), ssh:connection_ref()}

This is the first message that the channel receives. It is sent just before the init/1 function returns successfully. This is especially useful if the server wants to send a message to the client without first receiving a message from it. If the message is not useful for your particular scenario, ignore it by immediately returning {ok, State}.

Module:handle_ssh_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}

Types:
  Msg = ssh_connection:event()
  ChannelId = ssh:channel_id()
  State = term()

Handles SSH Connection Protocol messages that may need service-specific attention. For details, see ssh_connection:event().
The following message is taken care of by the ssh_client_channel behavior.

{closed, ssh:channel_id()}

The channel behavior sends a close message to the other side, if such a message has not already been sent. Then it terminates the channel with reason normal.

Module:terminate(Reason, State) -> _

Types:
  Reason = term()
  State = term()

This function is called by a channel process when it is about to terminate. Before this function is called, ssh_connection:close/2 is called, if it has not been called earlier. This function does any necessary cleaning up. When it returns, the channel process terminates with reason Reason. The return value is ignored.
ssh_server_channel

Erlang module

Note:

This module replaces ssh_daemon_channel.
The old module is still available for compatibility, but should not be used for new programs. The old module will not be maintained except for some error corrections.

SSH services (clients and servers) are implemented as channels that are multiplexed over an SSH connection and communicates over the SSH Connection Protocol. This module provides a callback API that takes care of generic channel aspects for daemons, such as flow control and close messages. It lets the callback functions take care of the service (application) specific parts. This behavior also ensures that the channel process honors the principal of an OTP-process so that it can be part of a supervisor tree. This is a requirement of channel processes implementing a subsystem that will be added to the ssh applications supervisor tree.

Note:

When implementing a client subsystem handler, use -behaviour(ssh_client_channel) instead.

Callback Functions

The following functions are to be exported from a ssh_server_channel callback module.

Exports

Module:init(Args) -> {ok, State} | {ok, State, timeout()} | {stop, Reason}
Types:

   Args = term()
   Last argument to start_link/4.
   State = term()
   Reason = term()

Makes necessary initializations and returns the initial channel state if the initializations succeed.
The time-out values that can be returned have the same semantics as in a gen_server. If the time-out occurs, handle_msg/2 is called as handle_msg(timeout, State).

Module:handle_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}
Types:

   Msg = timeout | term()
   ChannelId = ssh:channel_id()
   State = term()

Handles other messages than SSH Connection Protocol, call, or cast messages sent to the channel.
Possible Erlang 'EXIT' messages is to be handled by this function and all channels are to handle the following message.
ssh_server_channel

{ssh_channel_up, ssh:channel_id(), ssh:connection_ref()}

This is the first message that the channel receives. This is especially useful if the server wants to send a message to the client without first receiving a message from it. If the message is not useful for your particular scenario, ignore it by immediately returning {ok, State}.

Module:handle_ssh_msg(Msg, State) -> {ok, State} | {stop, ChannelId, State}
Types:
  Msg = ssh_connection:event()
  ChannelId = ssh:channel_id()
  State = term()
Handles SSH Connection Protocol messages that may need service-specific attention. For details, see ssh_connection:event().

The following message is taken care of by the ssh_server_channel behavior.

{closed, ssh:channel_id()}

The channel behavior sends a close message to the other side, if such a message has not already been sent. Then it terminates the channel with reason normal.

Module:terminate(Reason, State) -> _
Types:
  Reason = term()
  State = term()
This function is called by a channel process when it is about to terminate. Before this function is called, ssh_connection:close/2 is called, if it has not been called earlier. This function does any necessary cleaning up. When it returns, the channel process terminates with reason Reason. The return value is ignored.
ssh_connection

Erlang module

The SSH Connection Protocol is used by clients and servers, that is, SSH channels, to communicate over the SSH connection. The API functions in this module send SSH Connection Protocol events, which are received as messages by the remote channel handling the remote channel. The Erlang format of those messages is (see also below):

\{ssh_cm, ssh:connection_ref(), channel_msg()\}

If the ssh_client_channel behavior is used to implement the channel process, these messages are handled by handle_ssh_msg/2.

Data Types

ssh_data_type_code() = integer() >= 0

The valid values are 0 ("normal") and 1 ("stderr"), see RFC 4254, Section 5.2.

result() = req_status() | {error, reason()}
reason() = closed | timeout

The result of a call.

If the request reached the peer, was handled and the response reached the requesting node the req_status() is the status reported from the peer.

If not, the reason() indicates what went wrong:

closed  indicates that the channel or connection was closed when trying to send the request
timeout indicates that the operation exceeded a time limit

req_status() = success | failure

The status of a request. Coresponds to the SSH_MSG_CHANNEL_SUCCESS and SSH_MSG_CHANNEL_FAILURE values in RFC 4254, Section 5.4.

SSH Connection Protocol: General

event() = {ssh_cm, ssh:connection_ref(), channel_msg()}

channel_msg() =
  data_ch_msg() |
  eof_ch_msg() |
  closed_ch_msg() |
  pty_ch_msg() |
  env_ch_msg() |
  shell_ch_msg() |
  exec_ch_msg() |
  signal_ch_msg() |
  window_change_ch_msg() |
  exit_status_ch_msg() |
  exit_signal_ch_msg()

As mentioned in the introduction, the SSH Connection Protocol events are handled as messages. When writing a channel handling process without using the support by the ssh_client_channel behavior the process must handle those messages.
want_reply() = boolean()

Messages that include a WantReply expect the channel handling process to call ssh_connection:reply_request/4 with the boolean value of WantReply as the second argument.

Data Transfer (RFC 4254, section 5.2)

data_ch_msg() =
  {data,
   ssh:channel_id(),
   ssh_data_type_code(),
   Data :: binary()}

Data has arrived on the channel. This event is sent as a result of calling ssh_connection:send/3,4,5.

Closing a Channel (RFC 4254, section 5.3)

eof_ch_msg() = {eof, ssh:channel_id()}

Indicates that the other side sends no more data. This event is sent as a result of calling ssh_connection:send_eof/2.

closed_ch_msg() = {closed, ssh:channel_id()}

This event is sent as a result of calling ssh_connection:close/2. Both the handling of this event and sending it are taken care of by the ssh_client_channel behavior.

Requesting a Pseudo-Terminal (RFC 4254, section 6.2)

pty_ch_msg() =
  {pty,
   ssh:channel_id(),
   want_reply(),
   {Terminal :: string(),
    CharWidth :: integer() >= 0,
    RowHeight :: integer() >= 0,
    PixelWidth :: integer() >= 0,
    PixelHeight :: integer() >= 0,
    TerminalModes :: [term_mode()]}}

term_mode() =
  {Opcode :: atom() | byte(), Value :: integer() >= 0}

A pseudo-terminal has been requested for the session. Terminal is the value of the TERM environment variable value, that is, vt100. Zero dimension parameters must be ignored. The character/row dimensions override the pixel dimensions (when non-zero). Pixel dimensions refer to the drawable area of the window. Opcode in the TerminalModes list is the mnemonic name, represented as a lowercase Erlang atom, defined in RFC 4254, Section 8. It can also be an Opcode if the mnemonic name is not listed in the RFC. Example: OP code: 53, mnemonic name ECHO erlang atom: echo. This event is sent as a result of calling ssh_connection:ptty_alloc/4.

Environment Variable Passing (RFC 4254, section 6.4)

env_ch_msg() =
  {env,
   ssh:channel_id(),
   want_reply(),
   Var :: string(),
   Value :: string()}

Environment variables can be passed to the shell/command to be started later. This event is sent as a result of calling ssh_connection:setenv/5.
Starting a Shell or Command (RFC 4254, section 6.5)

shell_ch_msg() = {shell, ssh:channel_id(), want_reply()}

This message requests that the user default shell is started at the other end. This event is sent as a result of calling ssh_connection:shell/2.

exec_ch_msg() =
    {exec, ssh:channel_id(), want_reply(), Command :: string()}

This message requests that the server starts execution of the given command. This event is sent as a result of calling ssh_connection:exec/4.

Window Dimension Change Message (RFC 4254, section 6.7)

window_change_ch_msg() =
    {window_change, ssh:channel_id(),
     CharWidth :: integer() >= 0,
     RowHeight :: integer() >= 0,
     PixelWidth :: integer() >= 0,
     PixelHeight :: integer() >= 0}

When the window (terminal) size changes on the client side, it can send a message to the server side to inform it of the new dimensions. No API function generates this event.

Signals (RFC 4254, section 6.9)

signal_ch_msg() =
    {signal, ssh:channel_id(), SignalName :: string()}

A signal can be delivered to the remote process/service using the following message. Some systems do not support signals, in which case they are to ignore this message. There is currently no function to generate this event as the signals referred to are on OS-level and not something generated by an Erlang program.

Returning Exit Status (RFC 4254, section 6.10)

exit_status_ch_msg() =
    {exit_status, ssh:channel_id(),
     ExitStatus :: integer() >= 0}

When the command running at the other end terminates, the following message can be sent to return the exit status of the command. A zero exit_status usually means that the command terminated successfully. This event is sent as a result of calling ssh_connection:exit_status/3.

exit_signal_ch_msg() =
    {exit_signal, ssh:channel_id(),
     ExitSignal :: string(),
     ErrorMsg :: string(),
     LanguageString :: string()}

A remote execution can terminate violently because of a signal. Then this message can be received. For details on valid string values, see RFC 4254 Section 6.10, which shows a special case of these signals.
Exports

adjust_window(ConnectionRef, ChannelId, NumOfBytes) -> ok

Types:
    ConnectionRef = ssh:connection_ref()
    ChannelId = ssh:channel_id()
    NumOfBytes = integer()

Adjusts the SSH flow control window. This is to be done by both the client- and server-side channel processes.

Note:
Channels implemented with the ssh_client_channel behavior do not normally need to call this function as flow control is handled by the behavior. The behavior adjusts the window every time the callback handle_ssh_msg/2 returns after processing channel data.

close(ConnectionRef, ChannelId) -> ok

Types:
    ConnectionRef = ssh:connection_ref()
    ChannelId = ssh:channel_id()

A server- or client-channel process can choose to close their session by sending a close event.

Note:
This function is called by the ssh_client_channel behavior when the channel is terminated, see ssh_client_channel(3). Thus, channels implemented with the behavior are not to call this function explicitly.

exec(ConnectionRef, ChannelId, Command, Timeout) -> result()

Types:
    ConnectionRef = ssh:connection_ref()
    ChannelId = ssh:channel_id()
    Command = string()
    Timeout = timeout()

Is to be called by a client-channel process to request that the server starts executing the given command. The result is several messages according to the following pattern. The last message is a channel close message, as the exec request is a one-time execution that closes the channel when it is done.

N x data message(s)
    The result of executing the command can be only one line or thousands of lines depending on the command.

0 or 1 x eof message
    Indicates that no more data is to be sent.

0 or 1 x exit signal message
    Not all systems send signals. For details on valid string values, see RFC 4254, Section 6.10

0 or 1 x exit status message
    It is recommended by the SSH Connection Protocol to send this message, but that is not always the case.
1 x closed status message

Indicates that the ssh_client_channel started for the execution of the command has now been shut down. See the User's Guide section on One-Time Execution for examples.

\[ \text{exit_status(ConnectionRef, ChannelId, Status) -> ok} \]

Types:
- ConnectionRef = \text{ssh:connection_ref()}
- ChannelId = \text{ssh:channel_id()}
- Status = integer()

Is to be called by a server-channel process to send the exit status of a command to the client.

\[ \text{ptty_alloc(ConnectionRef, ChannelId, Options) -> result()} \]
\[ \text{ptty_alloc(ConnectionRef, ChannelId, Options, Timeout) -> result()} \]

Types:
- ConnectionRef = \text{ssh:connection_ref()}
- ChannelId = \text{ssh:channel_id()}
- Options = \text{proplists:proplist()}
- Timeout = \text{timeout()}\]

Sends an SSH Connection Protocol \text{pty_req}, to allocate a pseudo-terminal. Is to be called by an SSH client process.

Options:
- \{term, string\}
  Defaults to \text{os:getenv("TERM")} or \text{vt100} if it is undefined.
- \{width, integer\}
  Defaults to 80 if \text{pixel_width} is not defined.
- \{height, integer\}
  Defaults to 24 if \text{pixel_height} is not defined.
- \{pixel_width, integer\}
  Is disregarded if \text{width} is defined.
- \{pixel_height, integer\}
  Is disregarded if \text{height} is defined.
- \{pty_opts, [\{posix_atom(), integer\}\]}\]
  Option can be an empty list. Otherwise, see possible POSIX names in Section 8 in RFC 4254.

\[ \text{reply_request(ConnectionRef, WantReply, Status, ChannelId) -> ok} \]

Types:
ssh_connection

ConnectionRef = ssh:connection_ref()
WantReply = boolean()
Status = req_status()
ChannelId = ssh:channel_id()

Sends status replies to requests where the requester has stated that it wants a status report, that is, WantReply = true. If WantReply is false, calling this function becomes a "noop". Is to be called while handling an SSH Connection Protocol message containing a WantReply boolean value.

send(ConnectionRef, ChannelId, Data) ->
send(ConnectionRef, ChannelId, Data, Timeout) ->
send(ConnectionRef, ChannelId, Type, Data) ->
send(ConnectionRef, ChannelId, Type, Data, TimeOut) -> ok | Error

Types:
  ConnectionRef = ssh:connection_ref()
  ChannelId = ssh:channel_id()
  Data = iodata()
  Type = ssh_data_type_code()
  Timeout = timeout()
  Error = {error, reason()}

Is to be called by client- and server-channel processes to send data to each other.

The function subsystem/4 and subsequent calls of send/3, 4, 5 must be executed in the same process.

send_eof(ConnectionRef, ChannelId) -> ok | {error, closed}

Types:
  ConnectionRef = ssh:connection_ref()
  ChannelId = ssh:channel_id()

Sends EOF on channel ChannelId.

session_channel(ConnectionRef, Timeout) -> Result
session_channel(ConnectionRef, InitialWindowSize, MaxPacketSize, Timeout) -> Result

Types:
  ConnectionRef = ssh:connection_ref()
  InitialWindowSize = MaxPacketSize = integer() >= 1
  Timeout = timeout()
  Result = {ok, ssh:channel_id()} | {error, reason()}

Opens a channel for an SSH session. The channel id returned from this function is the id used as input to the other functions in this module.

setenv(ConnectionRef, ChannelId, Var, Value, Timeout) -> result()

Types:
Environment variables can be passed before starting the shell/command. Is to be called by a client channel processes.

**shell**\(\text{(ConnectionRef, ChannelId)} \rightarrow \text{Result}\)  

Types:  
- ConnectionRef = \text{ssh:connection_ref()}  
- ChannelId = \text{ssh:channel_id()}  
- Result = \text{ok} | \text{success} | \text{failure} | \{\text{error, timeout}\}

Is to be called by a client channel process to request that the user default shell (typically defined in /etc/passwd in Unix systems) is executed at the server end.  

Note: the return value is \text{ok} instead of \text{success} unlike in other functions in this module. This is a fault that was introduced so long ago that any change would break a large number of existing software.

**subsystem**\(\text{(ConnectionRef, ChannelId, Subsystem, Timeout)} \rightarrow \text{result()}\)  

Types:  
- ConnectionRef = \text{ssh:connection_ref()}  
- ChannelId = \text{ssh:channel_id()}  
- Subsystem = \text{string()}  
- Timeout = \text{timeout()}  

Is to be called by a client-channel process for requesting to execute a predefined subsystem on the server.  

The function \text{subsystem/4} and subsequent calls of \text{send/3,4,5} must be executed in the same process.
ssh_client_key_api

Erlang module

Behavior describing the API for public key handling of an SSH client. By implementing the callbacks defined in this behavior, the public key handling of an SSH client can be customized. By default the ssh application implements this behavior with help of the standard OpenSSH files, see the ssh(6) application manual.

Data Types

client_key_cb_options() = [{key_cb_private, term()} | ssh:client_option()]

Options provided to ssh:connect/[3,4].
The option list given in the key_cb option is available with the key key_cb_private.

Exports

Module:add_host_key(HostNames, PublicHostKey, ConnectOptions) -> ok | {error, Reason}

Types:

    HostNames = string()  
    Description of the host that owns the PublicHostKey.

    PublicHostKey = public_key:public_key()  
    Of ECDSA keys, only the Normally an RSA, DSA or ECDSA public key, but handling of other public keys can be added.

    ConnectOptions = client_key_cb_options()  
    Adds a host key to the set of trusted host keys.

Module:is_host_key(Key, Host, Algorithm, ConnectOptions) -> Result

Types:

    Key = public_key:public_key()  
    Normally an RSA, DSA or ECDSA public key, but handling of other public keys can be added.

    Host = string()  
    Description of the host.

    Algorithm = ssh:pubkey_alg()  
    Host key algorithm.

    ConnectOptions = client_key_cb_options()  
    Result = boolean()

Checks if a host key is trusted.

Module:user_key(Algorithm, ConnectOptions) -> {ok, PrivateKey} | {error, Reason}

Types:

    Algorithm = ssh:pubkey_alg()  
    Host key algorithm.
ConnectOptions = `client_key_cb_options()`

`PrivateKey = public_key:private_key()`

Private key of the user matching the Algorithm.

`Reason = term()`

Fetched the user's public key matching the Algorithm.

**Note:**

The private key contains the public key.
ssh_server_key_api

Erlang module

Behaviour describing the API for public key handling of an SSH server. By implementing the callbacks defined in this behavior, the public key handling of an SSH server can be customized. By default the SSH application implements this behavior with help of the standard OpenSSH files, see the ssh(6) application manual.

Data Types

daoemon_key_cb_options() = [{key_cb_private, term()} | ssh:daemon_option()]

Options provided to ssh:daemon/2,3.
The option list given in the key_cb option is available with the key key_cb_private.

Exports

Module:host_key(Algorithm, DaemonOptions) -> {ok, Key} | {error, Reason}

Types:

  Algorithm = ssh:pubkey_alg()
  Host key algorithm.

  DaemonOptions = daemon_key_cb_options()

  PrivateKey = public_key:private_key() | crypto:engine_key_ref()
  Private key of the host matching the Algorithm. It may be a reference to a 'ssh-rsa', rsa-sha2-* or 'ssh-dss' (NOT ecdsa) key stored in a loaded Engine.

  Reason = term()

Fetches the private key of the host.

Module:is_auth_key(PublicUserKey, User, DaemonOptions) -> Result

Types:

  PublicUserKey = public_key:public_key()
  Normally an RSA, DSA or ECDSA public key, but handling of other public keys can be added

  User = string()
  User owning the public key.

  DaemonOptions = daemon_key_cb_options()

  Result = boolean()

Checks if the user key is authorized.
This module is the default callback handler for the client’s and the server’s user and host “database” operations. All data, for instance key pairs, are stored in files in the normal file system. This page documents the files, where they are stored and configuration options for this callback module.

The intention is to be compatible with the OpenSSH storage in files. Therefore it mimics directories and filenames of OpenSSH.

Ssh_file implements the ssh_server_key_api and the ssh_client_key_api. This enables the user to make an own interface using for example a database handler.

Such another callback module could be used by setting the option key_cb when starting a client or a server (with for example ssh:connect, ssh:daemon of ssh:shell).

Note:
The functions are Callbacks for the SSH app. They are not intended to be called from the user's code!

Files, directories and who uses them

Daemons
Daemons uses all files stored in the SYSDIR directory.

Optionally, in case of publickey authorization, one or more of the remote user’s public keys in the USERDIR directory are used. See the files USRDIR/authorized_keys and USERDIR/authorized_keys2.

Clients
Clients uses all files stored in the USERDIR directory.

Directory contents

LOCALUSER
The user name of the OS process running the Erlang virtual machine (emulator).

SYSDIR
This is the directory holding the server's files:

- ssh_host_dsa_key - private dss host key (optional)
- ssh_host_rsa_key - private rsa host key (optional)
- ssh_host_ecdsa_key - private edcsa host key (optional)
- ssh_host_ed25519_key - private eddsa host key for curve 25519 (optional)
- ssh_host_ed448_key - private eddsa host key for curve 448 (optional)

At least one host key must be defined. The default value of SYSDIR is /etc/ssh.

For security reasons, this directory is normally accessible only to the root user.

To change the SYSDIR, see the system_dir option.

USERDIR
This is the directory holding the files:
**Options for the default ssh_file callback module**

- `user_dir_common_option() = {user_dir, string()}`
  
  Sets the user directory.

- `user_dir_fun_common_option() = {user_dir_fun, user2dir()}`
  
  ```erlang
  user2dir() =
  fun((RemoteUserName :: string()) -> UserDir :: string())
  ```

  Sets the user directory dynamically by evaluating the `user2dir` function.

- `system_dir_daemon_option() = {system_dir, string()}`
  
  Sets the system directory.

- `pubkey_passphrase_client_options() =`  
  
  ```erlang
  {dsa_pass_phrase, string()} |
  {rsa_pass_phrase, string()} |
  {ecdsa_pass_phrase, string()}
  ```

  If the user's DSA, RSA or ECDSA key is protected by a passphrase, it can be supplied with these options. Note that EdDSA passphrases (Curves 25519 and 448) are not implemented.

**Exports**

- `host_key(Algorithm, DaemonOptions) -> {ok, Key} | {error, Reason}`

**Types and description**

See the api description in `ssh_server_key_api, Module:host_key/2`.

**Options**

- `system_dir`

**Files**

- `SYSDIR/ssh_host_rsa_key`
- `SYSDIR/ssh_host_dsa_key`
- `SYSDIR/ssh_host_ecdsa_key`
- `SYSDIR/ssh_host_ed25519_key`
- `SYSDIR/ssh_host_ed448_key`
is_auth_key(PublicUserKey, User, DaemonOptions) -> Result

Types and description
See the api description in ssh_server_key_api: Module:is_auth_key/3.

Options
• user_dir_fun
• user_dir

Files
• USERDIR/authorized_keys
• USERDIR/authorized_keys2

add_host_key(HostNames, PublicHostKey, ConnectOptions) -> ok | {error, Reason}

Types and description
See the api description in ssh_client_key_api, Module:add_host_key/3.

Option
• user_dir

File
• USERDIR/known_hosts

is_host_key(Key, Host, Algorithm, ConnectOptions) -> Result

Types and description
See the api description in ssh_client_key_api, Module:is_host_key/4.

Option
• user_dir

File
• USERDIR/known_hosts

user_key(Algorithm, ConnectOptions) -> {ok, PrivateKey} | {error, Reason}

Types and description
See the api description in ssh_client_key_api, Module:user_key/2.

Options
• user_dir
• dsa_pass_phrase
• rsa_pass_phrase
• ecdsa_pass_phrase

Note that EdDSA passphrases (Curves 25519 and 448) are not implemented.

Files
• USERDIR/id_dsa
• USERDIR/id_rsa
• USERDIR/id_ecdsa
• USERDIR/id_ed25519
• USERDIR/id_ed448
ssh_sftp

Erlang module

This module implements an SSH FTP (SFTP) client. SFTP is a secure, encrypted file transfer service available for SSH.

Data Types

sftp_option() =
  {timeout, timeout()} |
  {sftp_vsn, integer() >= 1} |
  {window_size, integer() >= 1} |
  {packet_size, integer() >= 1}

Error cause

reason() = atom() | string() | tuple()

A description of the reason why an operation failed.

The atom() value is formed from the sftp error codes in the protocol-level responses as defined in draft-ietf-secsh-filexfer-13 section 9.1. The codes are named as SSH_FX_* which are transformed into lowercase of the star-part. E.g. the error code SSH_FX_NO_SUCH_FILE will cause the reason() to be no_such_file.

The string() reason is the error information from the server in case of an exit-signal. If that information is empty, the reason is the exit signal name.

The tuple() reason are other errors like for example {exit_status,1}.

Crypto operations for open_tar

tar_crypto_spec() = encrypt_spec() | decrypt_spec()

encrypt_spec() = {init_fun(), crypto_fun(), final_fun()}

decrypt_spec() = {init_fun(), crypto_fun()}

Specifies the encryption or decryption applied to tar files when using open_tar/3 or open_tar/4.

The encryption or decryption is applied to the generated stream of bytes prior to sending the resulting stream to the SFTP server.

For code examples see Section Example with encryption in the ssh Users Guide.

init_fun() =
  fun(() -> {ok, crypto_state()}) |
  fun(() -> {ok, crypto_state(), chunk_size()})

chunk_size() = undefined | integer() >= 1

crypto_state() = any()

The init_fun() in the tar_crypto_spec is applied once prior to any other crypto operation. The intention is that this function initiates the encryption or decryption for example by calling crypto:crypto_init/4 or similar. The crypto_state() is the state such a function may return.

If the selected cipher needs to have the input data partitioned into blocks of a certain size, the init_fun() should return the second form of return value with the chunk_size() set to the block size. If the chunk_size() is undefined, the size of the PlainBins varies, because this is intended for stream crypto, whereas a fixed chunk_size() is intended for block crypto. A chunk_size() can be changed in the return from the crypto_fun(). The value can be changed between pos_integer() and undefined.

crypto_fun() =
fun((TextIn :: binary(), crypto_state()) -> crypto_result())
crypto_result() =
  {ok, TextOut :: binary(), crypto_state()} |
  {ok, TextOut :: binary(), crypto_state(), chunk_size()}

The initial crypto_state() returned from the init_fun() is folded into repeated applications of the crypto_fun() in the tar_crypto_spec. The binary returned from that fun is sent to the remote SFTP server and the new crypto_state() is used in the next call of the crypto_fun().

If the crypto_fun() returns a chunk_size(), that value is as block size for further blocks in calls to crypto_fun().

final_fun() =
  fun((FinalTextIn :: binary(), crypto_state()) ->
     {ok, FinalTextOut :: binary()})

If doing encryption, the final_fun() in the tar_crypto_spec is applied to the last piece of data. The final_fun() is responsible for padding (if needed) and encryption of that last piece.

Exports

apread(ChannelPid, Handle, Position, Len) -> {async, N} | Error
Types:
  ChannelPid = pid()
  Handle = term()
  Position = Len = integer()
  Error = {error, reason()}
  N = term()

The apread/4 function reads from a specified position, combining the position/3 and aread/3 functions.

apwrite(ChannelPid, Handle, Position, Data) -> {async, N} | Error
Types:
  ChannelPid = pid()
  Handle = term()
  Position = integer()
  Data = binary()
  Error = {error, reason()}
  N = term()

The apwrite/4 function writes to a specified position, combining the position/3 and awrite/3 functions.

aread(ChannelPid, Handle, Len) -> {async, N} | Error
Types:
ChannelPid = pid()
Handle = term()
Len = integer()
Error = {error, reason()}
N = term()

Reads from an open file, without waiting for the result. If the handle is valid, the function returns \(\{\text{async}, \ N\}\), where \(N\) is a term guaranteed to be unique between calls of \texttt{aread}. The actual data is sent as a message to the calling process. This message has the form \(\{\text{async_reply}, \ N, \ \text{Result}\}\), where \text{Result} is the result from the read, either \(\{\text{ok}, \ \text{Data}\}\), \text{eof}, or \(\{\text{error}, \ \text{reason()}\}\).

\texttt{awrite(ChannelPid, Handle, Data) -> \{async, N\} | Error}

Types:
- ChannelPid = pid()
- Handle = term()
- Data = binary()
- Error = {error, reason()}
- N = term()

Writes to an open file, without waiting for the result. If the handle is valid, the function returns \(\{\text{async}, \ N\}\), where \(N\) is a term guaranteed to be unique between calls of \texttt{awrite}. The result of the write operation is sent as a message to the calling process. This message has the form \(\{\text{async_reply}, \ N, \ \text{Result}\}\), where \text{Result} is the result from the write, either \text{ok}, or \(\{\text{error}, \ \text{reason()}\}\).

\texttt{close(ChannelPid, Handle) -> ok | Error}
\texttt{close(ChannelPid, Handle, Timeout) -> ok | Error}

Types:
- ChannelPid = pid()
- Handle = term()
- Timeout = timeout()
- Error = {error, reason()}

Closes a handle to an open file or directory on the server.

\texttt{delete(ChannelPid, Name) -> ok | Error}
\texttt{delete(ChannelPid, Name, Timeout) -> ok | Error}

Types:
- ChannelPid = pid()
- Name = string()
- Timeout = timeout()
- Error = {error, reason()}

Deletes the file specified by \text{Name}.

\texttt{del_dir(ChannelPid, Name) -> ok | Error}
\texttt{del_dir(ChannelPid, Name, Timeout) -> ok | Error}

Types:
ChannelPid = pid()
Name = string()
Timeout = timeout()
Error = {error, reason()}

Deletes a directory specified by Name. The directory must be empty before it can be successfully deleted.

list_dir(ChannelPid, Path) -> {ok, FileNames} | Error
list_dir(ChannelPid, Path, Timeout) -> {ok, FileNames} | Error

Types:
  ChannelPid = pid()
  Path = string()
  Timeout = timeout()
  FileNames = [FileName]
  FileName = string()
  Error = {error, reason()}

Lists the given directory on the server, returning the filenames as a list of strings.

make_dir(ChannelPid, Name) -> ok | Error
make_dir(ChannelPid, Name, Timeout) -> ok | Error

Types:
  ChannelPid = pid()
  Name = string()
  Timeout = timeout()
  Error = {error, reason()}

Creates a directory specified by Name. Name must be a full path to a new directory. The directory can only be created in an existing directory.

make_symlink(ChannelPid, Name, Target) -> ok | Error
make_symlink(ChannelPid, Name, Target, Timeout) -> ok | Error

Types:
  ChannelPid = pid()
  Name = Target = string()
  Timeout = timeout()
  Error = {error, reason()}

Creates a symbolic link pointing to Target with the name Name.

open(ChannelPid, Name, Mode) -> {ok, Handle} | Error
open(ChannelPid, Name, Mode, Timeout) -> {ok, Handle} | Error

Types:
ChannelPid = pid()
Name = string()
Mode = [read | write | append | binary | raw]
Timeout = timeout()
Handle = term()
Error = {error, reason()}

Opens a file on the server and returns a handle, which can be used for reading or writing.

opendir(ChannelPid, Path) -> {ok, Handle} | Error
opendir(ChannelPid, Path, Timeout) -> {ok, Handle} | Error

Types:
ChannelPid = pid()
Path = string()
Timeout = timeout()
Handle = term()
Error = {error, reason()}

Opens a handle to a directory on the server. The handle can be used for reading directory contents.

open_tar(ChannelPid, Path, Mode) -> {ok, Handle} | Error
open_tar(ChannelPid, Path, Mode, Timeout) -> {ok, Handle} | Error

Types:
ChannelPid = pid()
Path = string()
Mode = [read | write | {crypto, tar_crypto_spec()}
Timeout = timeout()
Handle = term()
Error = {error, reason()}

Opens a handle to a tar file on the server, associated with ChannelPid. The handle can be used for remote tar creation and extraction. The actual writing and reading is performed by calls to erl_tar:add/3,4 and erl_tar:extract/2. Note: The erl_tar:init/3 function should not be called, that one is called by this open_tar function.

For code examples see Section SFTP Client with TAR Compression in the ssh Users Guide.
The crypto mode option is explained in the data types section above, see Crypto operations for open_tar. Encryption is assumed if the Mode contains write, and decryption if the Mode contains read.

position(ChannelPid, Handle, Location) ->
   {ok, NewPosition} | Error
position(ChannelPid, Handle, Location, Timeout) ->
   {ok, NewPosition} | Error

Types:
ChannelPid = pid()
Handle = term()
Location =
   Offset |
   {bof, Offset} |
ssh_sftp

{cur, Offset} | {eof, Offset} | 
bof | cur | eof

Timeout = timeout()
Offset = NewPosition = integer()
Error = {error, reason()}

Sets the file position of the file referenced by Handle. Returns \{ok, NewPosition\} (as an absolute offset) if successful, otherwise \{error, reason()\}. Location is one of the following:

Offset

The same as \{bof, Offset\}.
{bof, Offset}
Absolute offset.
{cur, Offset}
Offset from the current position.
{eof, Offset}
Offset from the end of file.
bof | cur | eof
The same as earlier with Offset 0, that is, \{bof, 0\} | \{cur, 0\} | \{eof, 0\}.

pread(ChannelPid, Handle, Position, Len) ->
{oak, Data} | eof | Error

pread(ChannelPid, Handle, Position, Len, Timeout) ->
{oak, Data} | eof | Error

Types:

ChannelPid = pid()
Handle = term()
Position = Len = integer()
Timeout = timeout()
Data = string() | binary()
Error = {error, reason()}

The pread/3,4 function reads from a specified position, combining the position/3 and read/3,4 functions.

pwrite(ChannelPid, Handle, Position, Data) -> ok | Error
pwrite(ChannelPid, Handle, Position, Data, Timeout) -> ok | Error

Types:

ChannelPid = pid()
Handle = term()
Position = integer()
Data = iolist()
Timeout = timeout()
Error = {error, reason()}

The pwrite/3,4 function writes to a specified position, combining the position/3 and write/3,4 functions.
read(ChannelPid, Handle, Len) -> {ok, Data} | eof | Error
read(ChannelPid, Handle, Len, Timeout) -> {ok, Data} | eof | Error

Types:
  ChannelPid = pid()
  Handle = term()
  Len = integer()
  Timeout = timeout()
  Data = string() | binary()
  Error = {error, reason()}

Reads Len bytes from the file referenced by Handle. Returns \{ok, Data\}, \{eof\}, or \{error, reason()\}. If the file is opened with \texttt{binary}, \texttt{Data} is a binary, otherwise it is a string.

If the file is read past \texttt{eof}, only the remaining bytes are read and returned. If no bytes are read, \texttt{eof} is returned.

read file(ChannelPid, File) -> {ok, Data} | Error
read file(ChannelPid, File, Timeout) -> {ok, Data} | Error

Types:
  ChannelPid = pid()
  File = string()
  Data = binary()
  Timeout = timeout()
  Error = {error, reason()}

Reads a file from the server, and returns the data in a binary.

read file info(ChannelPid, Name) -> {ok, FileInfo} | Error
read file info(ChannelPid, Name, Timeout) -> {ok, FileInfo} | Error

Types:
  ChannelPid = pid()
  Name = string()
  Timeout = timeout()
  FileInfo = file:file_info()
  Error = {error, reason()}

Returns a \texttt{file_info} record from the file system object specified by Name or Handle. See \texttt{file:read_file_info/2} for information about the record.

Depending on the underlying OS:es links might be followed and info on the final file, directory etc is returned. See \texttt{read_link_info/2} on how to get information on links instead.

read link(ChannelPid, Name) -> {ok, Target} | Error
read link(ChannelPid, Name, Timeout) -> {ok, Target} | Error

Types:
ChannelPid = pid()
Name = Target = string()
Timeout = timeout()
Error = {error, reason()}

Reads the link target from the symbolic link specified by name.

read_link_info(ChannelPid, Name) -> {ok, FileInfo} | Error
read_link_info(ChannelPid, Name, Timeout) ->
{ok, FileInfo} | Error

Types:
ChannelPid = pid()
Name = string()
FileInfo = file:file_info()
Timeout = timeout()
Error = {error, reason()}

Returns a file_info record from the symbolic link specified by Name or Handle. See file:read_link_info/2 for information about the record.

rename(ChannelPid, OldName, NewName) -> ok | Error
rename(ChannelPid, OldName, NewName, Timeout) -> ok | Error

Types:
ChannelPid = pid()
OldName = NewName = string()
Timeout = timeout()
Error = {error, reason()}

Renames a file named OldName and gives it the name NewName.

start_channel(ConnectionRef) ->
start_channel(ConnectionRef, SftpOptions) -> {ok, ChannelPid} | Error
start_channel(Host) ->
start_channel(Host, Options) ->
start_channel(TcpSocket) ->
start_channel(TcpSocket, Options) -> {ok, ChannelPid, ConnectionRef} | Error

Types:
Host = ssh:host()
Port = inet:port_number()
TcpSocket = ssh:open_socket()
Options = [ sftp_option() | ssh:client_option() ]
SftpOptions = [ sftp_option() ]
ChannelPid = pid()
ConnectionRef = ssh:connection_ref()
Error = {error, reason()}
If no connection reference is provided, a connection is set up, and the new connection is returned. An SSH channel process is started to handle the communication with the SFTP server. The returned pid for this process is to be used as input to all other API functions in this module.

Options:

\{(timeout, timeout())\}

There are two ways to set a timeout for the underlying ssh connection:

- If the connection timeout option `connect_timeout` is set, that value is used also for the negotiation timeout and this option (timeout) is ignored.
- Otherwise, this option (timeout) is used as the negotiation timeout only and there is no connection timeout set

The value defaults to `infinity`.

\{(sftp_vsn, integer())\}

Desired SFTP protocol version. The actual version is the minimum of the desired version and the maximum supported versions by the SFTP server.

All other options are directly passed to `ssh:connect/3` or ignored if a connection is already provided.

```
stop_channel(ChannelPid) -> ok
```

Types:

`ChannelPid = pid()`

Stops an SFTP channel. Does not close the SSH connection. Use `ssh:close/1` to close it.

```
write(ChannelPid, Handle, Data) -> ok | Error
write(ChannelPid, Handle, Data, Timeout) -> ok | Error
```

Types:

`ChannelPid = pid()`

`Handle = term()`

`Data = iodata()`

`Timeout = timeout()`

`Error = {error, reason()}`

Writes data to the file referenced by `Handle`. The file is to be opened with `write` or `append` flag. Returns ok if successful or `{error, reason()}` otherwise.

```
write_file(ChannelPid, File, Data) -> ok | Error
write_file(ChannelPid, File, Data, Timeout) -> ok | Error
```

Types:

`ChannelPid = pid()`

`File = string()`

`Data = iodata()`

`Timeout = timeout()`

`Error = {error, reason()}`

Writes a file to the server. The file is created if it does not exist but overwritten if it exists.
write_file_info(Channels, Name, FileInfo) -> ok | Error
write_file_info(Channels, Name, FileInfo, Timeout) -> ok | Error
Types:
   Channels = pid()
   Name = string()
   FileInfo = file:file_info()
   Timeout = timeout()
   Error = {error, reason()}
Writes file information from a file_info record to the file specified by Name. See file:write_file_info/[2,3] for information about the record.
ssh_sftpd

Erlang module

Specifies a channel process to handle an SFTP subsystem.

Exports

subsystem_spec(Options) -> Spec

Types:

Options =
    [{cwd, string()} |
     {file_handler, CallbackModule :: string()} |
     {max_files, integer()} |
     {root, string()} |
     {sftpd_vsn, integer()}]

Spec = {Name, {CbMod, Options}}

Name = string()

CbMod = atom()

Is to be used together with ssh:daemon/[1,2,3]

The Name is "sftp" and CbMod is the name of the Erlang module implementing the subsystem using the
ssh_server_channel (replaces ssh_daemon_channel) behaviour.

Options:

cwd

Sets the initial current working directory for the server.

file_handler

Determines which module to call for accessing the file server. The default value is ssh_sftpd_file, which
uses the file and filelib APIs to access the standard OTP file server. This option can be used to plug in other
file servers.

max_files

The default value is 0, which means that there is no upper limit. If supplied, the number of filenames returned to
the SFTP client per READDIR request is limited to at most the given value.

root

Sets the SFTP root directory. Then the user cannot see any files above this root. If, for example, the root directory
is set to /tmp, then the user sees this directory as /. If the user then writes cd /etc, the user moves to /
tmp/etc.

sftpd_vsn

Sets the SFTP version to use. Defaults to 5. Version 6 is under development and limited.