NAME

ssh — OpenSSH SSH client (remote login program)

SYNOPSIS

ssh [**-1246AaCfgKkMNnqsTtVvXxY**] [**-b** *bind_address*] [**-c** *cipher_spec*] [**-D** [*bind_address*:]*port*] [**-e** *escape_char*] [**-F** *configfile*] [**-i** *identity_file*] [**-L** [*bind_address*:]*port*:*host*:*hostport*] [**-l** *login_name*] [**-m** *mac_spec*] [**-O** *ctl_cmd*] [**-o** *option*] [**-p** *port*] [**-R** [*bind_address*:]*port*:*host*:*hostport*] [**-S** *ctl_path*] [**-w** *local_tun*[:*remote_tun*]] [*user*@]*hostname* [*command*]

DESCRIPTION

ssh (SSH client) is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP ports can also be forwarded over the secure channel.

ssh connects and logs into the specified *hostname* (with optional *user* name). The user must prove his/her identity to the remote machine using one of several methods depending on the protocol version used (see below).

If *command* is specified, it is executed on the remote host instead of a login shell.

The options are as follows:

-1 Forces **ssh** to try protocol version 1 only.

-2 Forces **ssh** to try protocol version 2 only.

-4 Forces **ssh** to use IPv4 addresses only.

-6 Forces **ssh** to use IPv6 addresses only.

-A Enables forwarding of the authentication agent connection. This can also be specified on a per-host basis in a configuration file.

Agent forwarding should be enabled with caution. Users with the ability to bypass file permissions on the remote host (for the agent's Unix-domain socket) can access the local agent through the forwarded connection. An attacker cannot obtain key material from the agent, however they can perform operations on the keys that enable them to authenticate using the identities loaded into the agent.

-a Disables forwarding of the authentication agent connection.

-b *bind_address*
Use *bind_address* on the local machine as the source address of the connection. Only useful on systems with more than one address.

-C Requests compression of all data (including stdin, stdout, stderr, and data for forwarded X11 and TCP connections). The compression algorithm is the same used by **gzip**(1), and the "level" can be controlled by the **CompressionLevel** option for protocol version 1. Compression is desirable on modem lines and other slow connections, but will only slow down things on fast networks. The default value can be set on a host-by-host basis in the configuration files; see the **Compression** option.

-c *cipher_spec*
Selects the cipher specification for encrypting the session.

Protocol version 1 allows specification of a single cipher. The supported values are "3des", "blowfish", and "des". *3des* (triple-des) is an encrypt-decrypt-encrypt triple with three different keys. It is believed to be secure. *blowfish* is a fast block cipher; it appears very secure and is much faster than *3des*. *des* is only supported in the **ssh** client for interoperability with legacy protocol 1 implementations that do not support the *3des* cipher. Its use is strongly discouraged due to cryptographic weaknesses. The default is "3des".

For protocol version 2, *cipher_spec* is a comma-separated list of ciphers listed in order of preference. The supported ciphers are: 3des-cbc, aes128-cbc, aes192-cbc, aes256-cbc, aes128-ctr, aes192-ctr, aes256-ctr, arcfour128, arcfour256, arcfour, blowfish-cbc, and cast128-cbc. The default is:

```
aes128-cbc,3des-cbc,blowfish-cbc,cast128-cbc,arcfour128,
arcfour256,arcfour,aes192-cbc,aes256-cbc,aes128-ctr,
aes192-ctr,aes256-ctr
```

-D [*bind_address*:]*port*
: Specifies a local "dynamic" application-level port forwarding. This works by allocating a socket to listen to *port* on the local side, optionally bound to the specified *bind_address*. Whenever a connection is made to this port, the connection is forwarded over the secure channel, and the application protocol is then used to determine where to connect to from the remote machine. Currently the SOCKS4 and SOCKS5 protocols are supported, and **ssh** will act as a SOCKS server. Only root can forward privileged ports. Dynamic port forwardings can also be specified in the configuration file.

  IPv6 addresses can be specified with an alternative syntax: [*bind_address*/]*port* or by enclosing the address in square brackets. Only the superuser can forward privileged ports. By default, the local port is bound in accordance with the **GatewayPorts** setting. However, an explicit *bind_address* may be used to bind the connection to a specific address. The *bind_address* of "localhost" indicates that the listening port be bound for local use only, while an empty address or '*' indicates that the port should be available from all interfaces.

-e *escape_char*
: Sets the escape character for sessions with a pty (default: '~'). The escape character is only recognized at the beginning of a line. The escape character followed by a dot ('.') closes the connection; followed by control-Z suspends the connection; and followed by itself sends the escape character once. Setting the character to "none" disables any escapes and makes the session fully transparent.

-F *configfile*
: Specifies an alternative per-user configuration file. If a configuration file is given on the command line, the system-wide configuration file (**/etc/ssh/ssh_config**) will be ignored. The default for the per-user configuration file is **~/.ssh/config**.

-f
: Requests **ssh** to go to background just before command execution. This is useful if **ssh** is going to ask for passwords or passphrases, but the user wants it in the background. This implies **-n**. The recommended way to start X11 programs at a remote site is with something like **ssh -f host xterm**.

-g
: Allows remote hosts to connect to local forwarded ports.

-I *smartcard_device*
: Specify the device **ssh** should use to communicate with a smartcard used for storing the user's private RSA key. This option is only available if support for smartcard devices is compiled in (default is no support).

`-i` *identity_file*
: Selects a file from which the identity (private key) for RSA or DSA authentication is read. The default is `~/.ssh/identity` for protocol version 1, and `~/.ssh/id_rsa` and `~/.ssh/id_dsa` for protocol version 2. Identity files may also be specified on a per-host basis in the configuration file. It is possible to have multiple `-i` options (and multiple identities specified in configuration files).

`-K`
: Enables forwarding (delegation) of GSSAPI credentials to the server.

`-k`
: Disables forwarding (delegation) of GSSAPI credentials to the server.

`-L` [*bind_address*:]*port*:*host*:*hostport*
: Specifies that the given port on the local (client) host is to be forwarded to the given host and port on the remote side. This works by allocating a socket to listen to *port* on the local side, optionally bound to the specified *bind_address*. Whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to *host* port *hostport* from the remote machine. Port forwardings can also be specified in the configuration file. IPv6 addresses can be specified with an alternative syntax: [*bind_address*/]*port*/*host*/*hostport* or by enclosing the address in square brackets. Only the superuser can forward privileged ports. By default, the local port is bound in accordance with the `GatewayPorts` setting. However, an explicit *bind_address* may be used to bind the connection to a specific address. The *bind_address* of "localhost" indicates that the listening port be bound for local use only, while an empty address or '*' indicates that the port should be available from all interfaces.

`-l` *login_name*
: Specifies the user to log in as on the remote machine. This also may be specified on a per-host basis in the configuration file.

`-M`
: Places the `ssh` client into "master" mode for connection sharing. Multiple `-M` options places `ssh` into "master" mode with confirmation required before slave connections are accepted. Refer to the description of `ControlMaster` in `ssh_config`(5) for details.

`-m` *mac_spec*
: Additionally, for protocol version 2 a comma-separated list of MAC (message authentication code) algorithms can be specified in order of preference. See the `MACs` keyword for more information.

`-N`
: Do not execute a remote command. This is useful for just forwarding ports (protocol version 2 only).

`-n`
: Redirects stdin from `/dev/null` (actually, prevents reading from stdin). This must be used when `ssh` is run in the background. A common trick is to use this to run X11 programs on a remote machine. For example, `ssh -n shadows.cs.hut.fi emacs &` will start an emacs on shadows.cs.hut.fi, and the X11 connection will be automatically forwarded over an encrypted channel. The `ssh` program will be put in the background. (This does not work if `ssh` needs to ask for a password or passphrase; see also the `-f` option.)

`-O` *ctl_cmd*
: Control an active connection multiplexing master process. When the `-O` option is specified, the *ctl_cmd* argument is interpreted and passed to the master process. Valid commands are: "check" (check that the master process is running) and "exit" (request the master to exit).

`-o` *option*
: Can be used to give options in the format used in the configuration file. This is useful for specifying options for which there is no separate command-line flag. For full details of the

options listed below, and their possible values, see `ssh_config`(5).

AddressFamily
BatchMode
BindAddress
ChallengeResponseAuthentication
CheckHostIP
Cipher
Ciphers
ClearAllForwardings
Compression
CompressionLevel
ConnectionAttempts
ConnectTimeout
ControlMaster
ControlPath
DynamicForward
EscapeChar
ExitOnForwardFailure
ForwardAgent
ForwardX11
ForwardX11Trusted
GatewayPorts
GlobalKnownHostsFile
GSSAPIAuthentication
GSSAPIDelegateCredentials
HashKnownHosts
Host
HostbasedAuthentication
HostKeyAlgorithms
HostKeyAlias
HostName
IdentityFile
IdentitiesOnly
KbdInteractiveDevices
LocalCommand
LocalForward
LogLevel
MACs
NoHostAuthenticationForLocalhost
NumberOfPasswordPrompts
PasswordAuthentication
PermitLocalCommand
Port
PreferredAuthentications
Protocol
ProxyCommand
PubkeyAuthentication
RekeyLimit
RemoteForward

RhostsRSAAuthentication
RSAAuthentication
SendEnv
ServerAliveInterval
ServerAliveCountMax
SmartcardDevice
StrictHostKeyChecking
TCPKeepAlive
Tunnel
TunnelDevice
UsePrivilegedPort
User
UserKnownHostsFile
VerifyHostKeyDNS
XAuthLocation

`-p` *port*
: Port to connect to on the remote host. This can be specified on a per-host basis in the configuration file.

`-q`
: Quiet mode. Causes all warning and diagnostic messages to be suppressed. Only fatal errors are displayed. If a second `-q` is given then even fatal errors are suppressed, except for those produced due solely to bad arguments.

`-R` [*bind_address*:]*port*:*host*:*hostport*
: Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side. This works by allocating a socket to listen to *port* on the remote side, and whenever a connection is made to this port, the connection is forwarded over the secure channel, and a connection is made to *host* port *hostport* from the local machine.

  Port forwardings can also be specified in the configuration file. Privileged ports can be forwarded only when logging in as root on the remote machine. IPv6 addresses can be specified by enclosing the address in square braces or using an alternative syntax: [*bind_address*/]*host*/*port*/*hostport*.

  By default, the listening socket on the server will be bound to the loopback interface only. This may be overriden by specifying a *bind_address*. An empty *bind_address*, or the address '*', indicates that the remote socket should listen on all interfaces. Specifying a remote *bind_address* will only succeed if the server's `GatewayPorts` option is enabled (see `sshd_config`(5)).

`-S` *ctl_path*
: Specifies the location of a control socket for connection sharing. Refer to the description of `ControlPath` and `ControlMaster` in `ssh_config`(5) for details.

`-s`
: May be used to request invocation of a subsystem on the remote system. Subsystems are a feature of the SSH2 protocol which facilitate the use of SSH as a secure transport for other applications (eg. `sftp`(1)). The subsystem is specified as the remote command.

`-T`
: Disable pseudo-tty allocation.

`-t`
: Force pseudo-tty allocation. This can be used to execute arbitrary screen-based programs on a remote machine, which can be very useful, e.g. when implementing menu services. Multiple `-t` options force tty allocation, even if `ssh` has no local tty.

`-V` Display the version number and exit.

`-v` Verbose mode. Causes `ssh` to print debugging messages about its progress. This is helpful in debugging connection, authentication, and configuration problems. Multiple `-v` options increase the verbosity. The maximum is 3.

`-w` *local_tun*[:*remote_tun*]
Requests tunnel device forwarding with the specified `tun`(4) devices between the client (*local_tun*) and the server (*remote_tun*).

The devices may be specified by numerical ID or the keyword "any", which uses the next available tunnel device. If *remote_tun* is not specified, it defaults to "any". See also the `Tunnel` and `TunnelDevice` directives in `ssh_config`(5). If the `Tunnel` directive is unset, it is set to the default tunnel mode, which is "point-to-point".

`-X` Enables X11 forwarding. This can also be specified on a per-host basis in a configuration file.

X11 forwarding should be enabled with caution. Users with the ability to bypass file permissions on the remote host (for the user's X authorization database) can access the local X11 display through the forwarded connection. An attacker may then be able to perform activities such as keystroke monitoring.

For this reason, X11 forwarding is subjected to X11 SECURITY extension restrictions by default. Please refer to the `ssh -Y` option and the `ForwardX11Trusted` directive in `ssh_config`(5) for more information.

`-x` Disables X11 forwarding.

`-Y` Enables trusted X11 forwarding. Trusted X11 forwardings are not subjected to the X11 SECURITY extension controls.

`ssh` may additionally obtain configuration data from a per-user configuration file and a system-wide configuration file. The file format and configuration options are described in `ssh_config`(5).

`ssh` exits with the exit status of the remote command or with 255 if an error occurred.

AUTHENTICATION

The OpenSSH SSH client supports SSH protocols 1 and 2. Protocol 2 is the default, with `ssh` falling back to protocol 1 if it detects protocol 2 is unsupported. These settings may be altered using the `Protocol` option in `ssh_config`(5), or enforced using the `-1` and `-2` options (see above). Both protocols support similar authentication methods, but protocol 2 is preferred since it provides additional mechanisms for confidentiality (the traffic is encrypted using AES, 3DES, Blowfish, CAST128, or Arcfour) and integrity (hmac-md5, hmac-sha1, hmac-ripemd160). Protocol 1 lacks a strong mechanism for ensuring the integrity of the connection.

The methods available for authentication are: GSSAPI-based authentication, host-based authentication, public key authentication, challenge-response authentication, and password authentication. Authentication methods are tried in the order specified above, though protocol 2 has a configuration option to change the default order: `PreferredAuthentications`.

Host-based authentication works as follows: If the machine the user logs in from is listed in `/etc/hosts.equiv` or `/etc/ssh/shosts.equiv` on the remote machine, and the user names are the same on both sides, or if the files `~/.rhosts` or `~/.shosts` exist in the user's home directory on the remote machine and contain a line containing the name of the client machine and the name of the user on that machine, the user is considered for login. Additionally, the server *must* be able to verify the client's host key (see the description of `/etc/ssh/ssh_known_hosts` and `~/.ssh/known_hosts`, below) for login to be permitted. This authentication method closes security holes due to IP spoof-

ing, DNS spoofing, and routing spoofing. [Note to the administrator: `/etc/hosts.equiv`, `~/.rhosts`, and the rlogin/rsh protocol in general, are inherently insecure and should be disabled if security is desired.]

Public key authentication works as follows: The scheme is based on public-key cryptography, using cryptosystems where encryption and decryption are done using separate keys, and it is unfeasible to derive the decryption key from the encryption key. The idea is that each user creates a public/private key pair for authentication purposes. The server knows the public key, and only the user knows the private key. `ssh` implements public key authentication protocol automatically, using either the RSA or DSA algorithms. Protocol 1 is restricted to using only RSA keys, but protocol 2 may use either. The **HISTORY** section of `ssl`(8) contains a brief discussion of the two algorithms.

The file `~/.ssh/authorized_keys` lists the public keys that are permitted for logging in. When the user logs in, the `ssh` program tells the server which key pair it would like to use for authentication. The client proves that it has access to the private key and the server checks that the corresponding public key is authorized to accept the account.

The user creates his/her key pair by running `ssh-keygen`(1). This stores the private key in `~/.ssh/identity` (protocol 1), `~/.ssh/id_dsa` (protocol 2 DSA), or `~/.ssh/id_rsa` (protocol 2 RSA) and stores the public key in `~/.ssh/identity.pub` (protocol 1), `~/.ssh/id_dsa.pub` (protocol 2 DSA), or `~/.ssh/id_rsa.pub` (protocol 2 RSA) in the user's home directory. The user should then copy the public key to `~/.ssh/authorized_keys` in his/her home directory on the remote machine. The `authorized_keys` file corresponds to the conventional `~/.rhosts` file, and has one key per line, though the lines can be very long. After this, the user can log in without giving the password.

The most convenient way to use public key authentication may be with an authentication agent. See `ssh-agent`(1) for more information.

Challenge-response authentication works as follows: The server sends an arbitrary "challenge" text, and prompts for a response. Protocol 2 allows multiple challenges and responses; protocol 1 is restricted to just one challenge/response. Examples of challenge-response authentication include BSD Authentication (see `login.conf`(5)) and PAM (some non-OpenBSD systems).

Finally, if other authentication methods fail, `ssh` prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

`ssh` automatically maintains and checks a database containing identification for all hosts it has ever been used with. Host keys are stored in `~/.ssh/known_hosts` in the user's home directory. Additionally, the file `/etc/ssh/ssh_known_hosts` is automatically checked for known hosts. Any new hosts are automatically added to the user's file. If a host's identification ever changes, `ssh` warns about this and disables password authentication to prevent server spoofing or man-in-the-middle attacks, which could otherwise be used to circumvent the encryption. The `StrictHostKeyChecking` option can be used to control logins to machines whose host key is not known or has changed.

When the user's identity has been accepted by the server, the server either executes the given command, or logs into the machine and gives the user a normal shell on the remote machine. All communication with the remote command or shell will be automatically encrypted.

If a pseudo-terminal has been allocated (normal login session), the user may use the escape characters noted below.

If no pseudo-tty has been allocated, the session is transparent and can be used to reliably transfer binary data. On most systems, setting the escape character to “none” will also make the session transparent even if a tty is used.

The session terminates when the command or shell on the remote machine exits and all X11 and TCP connections have been closed.

ESCAPE CHARACTERS

When a pseudo-terminal has been requested, **ssh** supports a number of functions through the use of an escape character.

A single tilde character can be sent as **~~** or by following the tilde by a character other than those described below. The escape character must always follow a newline to be interpreted as special. The escape character can be changed in configuration files using the **EscapeChar** configuration directive or on the command line by the **-e** option.

The supported escapes (assuming the default '~') are:

~. Disconnect.

~^Z Background **ssh**.

~# List forwarded connections.

~& Background **ssh** at logout when waiting for forwarded connection / X11 sessions to terminate.

~? Display a list of escape characters.

~B Send a BREAK to the remote system (only useful for SSH protocol version 2 and if the peer supports it).

~C Open command line. Currently this allows the addition of port forwardings using the **-L** and **-R** options (see above). It also allows the cancellation of existing remote port-forwardings using **-KR**[*bind_address*:]*port*. **!***command* allows the user to execute a local command if the **PermitLocalCommand** option is enabled in **ssh_config**(5). Basic help is available, using the **-h** option.

~R Request rekeying of the connection (only useful for SSH protocol version 2 and if the peer supports it).

TCP FORWARDING

Forwarding of arbitrary TCP connections over the secure channel can be specified either on the command line or in a configuration file. One possible application of TCP forwarding is a secure connection to a mail server; another is going through firewalls.

In the example below, we look at encrypting communication between an IRC client and server, even though the IRC server does not directly support encrypted communications. This works as follows: the user connects to the remote host using **ssh**, specifying a port to be used to forward connections to the remote server. After that it is possible to start the service which is to be encrypted on the client machine, connecting to the same local port, and **ssh** will encrypt and forward the connection.

The following example tunnels an IRC session from client machine "127.0.0.1" (localhost) to remote server "server.example.com":

```
$ ssh -f -L 1234:localhost:6667 server.example.com sleep 10
$ irc -c '#users' -p 1234 pinky 127.0.0.1
```

This tunnels a connection to IRC server "server.example.com", joining channel "#users", nickname "pinky", using port 1234. It doesn't matter which port is used, as long as it's greater than 1023 (remember, only root can open sockets on privileged ports) and doesn't conflict with any ports already in use. The connection is forwarded to port 6667 on the remote server, since that's the standard port for IRC services.

The `-f` option backgrounds `ssh` and the remote command "sleep 10" is specified to allow an amount of time (10 seconds, in the example) to start the service which is to be tunnelled. If no connections are made within the time specified, `ssh` will exit.

X11 FORWARDING

If the `ForwardX11` variable is set to "yes" (or see the description of the `-X`, `-x`, and `-Y` options above) and the user is using X11 (the `DISPLAY` environment variable is set), the connection to the X11 display is automatically forwarded to the remote side in such a way that any X11 programs started from the shell (or command) will go through the encrypted channel, and the connection to the real X server will be made from the local machine. The user should not manually set `DISPLAY`. Forwarding of X11 connections can be configured on the command line or in configuration files.

The `DISPLAY` value set by `ssh` will point to the server machine, but with a display number greater than zero. This is normal, and happens because `ssh` creates a "proxy" X server on the server machine for forwarding the connections over the encrypted channel.

`ssh` will also automatically set up Xauthority data on the server machine. For this purpose, it will generate a random authorization cookie, store it in Xauthority on the server, and verify that any forwarded connections carry this cookie and replace it by the real cookie when the connection is opened. The real authentication cookie is never sent to the server machine (and no cookies are sent in the plain).

If the `ForwardAgent` variable is set to "yes" (or see the description of the `-A` and `-a` options above) and the user is using an authentication agent, the connection to the agent is automatically forwarded to the remote side.

VERIFYING HOST KEYS

When connecting to a server for the first time, a fingerprint of the server's public key is presented to the user (unless the option `StrictHostKeyChecking` has been disabled). Fingerprints can be determined using `ssh-keygen`(1):

```
$ ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key
```

If the fingerprint is already known, it can be matched and verified, and the key can be accepted. If the fingerprint is unknown, an alternative method of verification is available: SSH fingerprints verified by DNS. An additional resource record (RR), SSHFP, is added to a zonefile and the connecting client is able to match the fingerprint with that of the key presented.

In this example, we are connecting a client to a server, "host.example.com". The SSHFP resource records should first be added to the zonefile for host.example.com:

```
$ ssh-keygen -r host.example.com.
```

The output lines will have to be added to the zonefile. To check that the zone is answering fingerprint queries:

```
$ dig -t SSHFP host.example.com
```

Finally the client connects:

```
$ ssh -o "VerifyHostKeyDNS ask" host.example.com
[...]
Matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)?
```

See the `VerifyHostKeyDNS` option in `ssh_config`(5) for more information.

SSH-BASED VIRTUAL PRIVATE NETWORKS

`ssh` contains support for Virtual Private Network (VPN) tunnelling using the `tun`(4) network pseudo-device, allowing two networks to be joined securely. The `sshd_config`(5) configuration option `PermitTunnel` controls whether the server supports this, and at what level (layer 2 or 3 traffic).

The following example would connect client network 10.0.50.0/24 with remote network 10.0.99.0/24 using a point-to-point connection from 10.1.1.1 to 10.1.1.2, provided that the SSH server running on the gateway to the remote network, at 192.168.1.15, allows it.

On the client:

```
# ssh -f -w 0:1 192.168.1.15 true
# ifconfig tun0 10.1.1.1 10.1.1.2 netmask 255.255.255.252
# route add 10.0.99.0/24 10.1.1.2
```

On the server:

```
# ifconfig tun1 10.1.1.2 10.1.1.1 netmask 255.255.255.252
# route add 10.0.50.0/24 10.1.1.1
```

Client access may be more finely tuned via the `/root/.ssh/authorized_keys` file (see below) and the `PermitRootLogin` server option. The following entry would permit connections on `tun`(4) device 1 from user "jane" and on tun device 2 from user "john", if `PermitRootLogin` is set to "forced-commands-only":

```
tunnel="1",command="sh /etc/netstart tun1" ssh-rsa ... jane
tunnel="2",command="sh /etc/netstart tun2" ssh-rsa ... john
```

Since an SSH-based setup entails a fair amount of overhead, it may be more suited to temporary setups, such as for wireless VPNs. More permanent VPNs are better provided by tools such as `ipsecctl`(8) and `isakmpd`(8).

ENVIRONMENT

`ssh` will normally set the following environment variables:

`DISPLAY` — The `DISPLAY` variable indicates the location of the X11 server. It is automatically set by `ssh` to point to a value of the form "hostname:n", where "hostname" indicates the host where the shell runs, and 'n' is an integer $\geq$ 1. `ssh` uses this special value to forward X11 connections over the secure channel. The user should normally not set `DISPLAY` explicitly, as that will render the X11 connection insecure (and will require the user to manually copy any required authorization cookies).

`HOME` — Set to the path of the user's home directory.

`LOGNAME` — Synonym for `USER`; set for compatibility with systems that use this variable.

`MAIL` — Set to the path of the user's mailbox.

`PATH` — Set to the default `PATH`, as specified when compiling `ssh`.

`SSH_ASKPASS` — If `ssh` needs a passphrase, it will read the passphrase from the current terminal if it was run from a terminal. If `ssh` does not have a terminal associated with it but `DISPLAY` and `SSH_ASKPASS` are set, it will execute the program specified by `SSH_ASKPASS` and open an X11 window to read the passphrase. This is particularly useful when calling `ssh` from a `.xsession` or related script. (Note that on some machines it may be necessary to redi-

rect the input from `/dev/null` to make this work.)

`SSH_AUTH_SOCK` Identifies the path of a UNIX-domain socket used to communicate with the agent.

`SSH_CONNECTION` Identifies the client and server ends of the connection. The variable contains four space-separated values: client IP address, client port number, server IP address, and server port number.

`SSH_ORIGINAL_COMMAND` This variable contains the original command line if a forced command is executed. It can be used to extract the original arguments.

`SSH_TTY` This is set to the name of the tty (path to the device) associated with the current shell or command. If the current session has no tty, this variable is not set.

`TZ` This variable is set to indicate the present time zone if it was set when the daemon was started (i.e. the daemon passes the value on to new connections).

`USER` Set to the name of the user logging in.

Additionally, `ssh` reads `~/.ssh/environment`, and adds lines of the format "VARNAME=value" to the environment if the file exists and users are allowed to change their environment. For more information, see the `PermitUserEnvironment` option in `sshd_config`(5).

FILES

~/.rhosts
This file is used for host-based authentication (see above). On some machines this file may need to be world-readable if the user's home directory is on an NFS partition, because `sshd`(8) reads it as root. Additionally, this file must be owned by the user, and must not have write permissions for anyone else. The recommended permission for most machines is read/write for the user, and not accessible by others.

~/.shosts
This file is used in exactly the same way as `.rhosts`, but allows host-based authentication without permitting login with rlogin/rsh.

~/.ssh/authorized_keys
Lists the public keys (RSA/DSA) that can be used for logging in as this user. The format of this file is described in the `sshd`(8) manual page. This file is not highly sensitive, but the recommended permissions are read/write for the user, and not accessible by others.

~/.ssh/config
This is the per-user configuration file. The file format and configuration options are described in `ssh_config`(5). Because of the potential for abuse, this file must have strict permissions: read/write for the user, and not accessible by others. It may be group-writable provided that the group in question contains only the user.

~/.ssh/environment
Contains additional definitions for environment variables; see **ENVIRONMENT**, above.

~/.ssh/identity
~/.ssh/id_dsa
~/.ssh/id_rsa
Contains the private key for authentication. These files contain sensitive data and should be readable by the user but not accessible by others (read/write/execute). `ssh` will simply ignore a private key file if it is accessible by others. It is possible to specify a passphrase

when generating the key which will be used to encrypt the sensitive part of this file using 3DES.

~/.ssh/identity.pub
~/.ssh/id_dsa.pub
~/.ssh/id_rsa.pub
: Contains the public key for authentication. These files are not sensitive and can (but need not) be readable by anyone.

~/.ssh/known_hosts
: Contains a list of host keys for all hosts the user has logged into that are not already in the systemwide list of known host keys. See `sshd`(8) for further details of the format of this file.

~/.ssh/rc
: Commands in this file are executed by `ssh` when the user logs in, just before the user's shell (or command) is started. See the `sshd`(8) manual page for more information.

/etc/hosts.equiv
: This file is for host-based authentication (see above). It should only be writable by root.

/etc/ssh/shosts.equiv
: This file is used in exactly the same way as `hosts.equiv`, but allows host-based authentication without permitting login with rlogin/rsh.

`/etc/ssh/ssh_config`
: Systemwide configuration file. The file format and configuration options are described in `ssh_config`(5).

/etc/ssh/ssh_host_key
/etc/ssh/ssh_host_dsa_key
/etc/ssh/ssh_host_rsa_key
: These three files contain the private parts of the host keys and are used for host-based authentication. If protocol version 1 is used, `ssh` must be setuid root, since the host key is readable only by root. For protocol version 2, `ssh` uses `ssh-keysign`(8) to access the host keys, eliminating the requirement that `ssh` be setuid root when host-based authentication is used. By default `ssh` is not setuid root.

/etc/ssh/ssh_known_hosts
: Systemwide list of known host keys. This file should be prepared by the system administrator to contain the public host keys of all machines in the organization. It should be world-readable. See `sshd`(8) for further details of the format of this file.

/etc/ssh/sshrc
: Commands in this file are executed by `ssh` when the user logs in, just before the user's shell (or command) is started. See the `sshd`(8) manual page for more information.

SEE ALSO

`scp`(1), `sftp`(1), `ssh-add`(1), `ssh-agent`(1), `ssh-argv0`(1), `ssh-keygen`(1), `ssh-keyscan`(1), `tun`(4), `hosts.equiv`(5), `ssh_config`(5), `ssh-keysign`(8), `sshd`(8)

The Secure Shell (SSH) Protocol Assigned Numbers, RFC 4250, 2006.

The Secure Shell (SSH) Protocol Architecture, RFC 4251, 2006.

The Secure Shell (SSH) Authentication Protocol, RFC 4252, 2006.

The Secure Shell (SSH) Transport Layer Protocol, RFC 4253, 2006.

The Secure Shell (SSH) Connection Protocol, RFC 4254, 2006.

Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints, RFC 4255, 2006.

Generic Message Exchange Authentication for the Secure Shell Protocol (SSH), RFC 4256, 2006.

The Secure Shell (SSH) Session Channel Break Extension, RFC 4335, 2006.

The Secure Shell (SSH) Transport Layer Encryption Modes, RFC 4344, 2006.

Improved Arcfour Modes for the Secure Shell (SSH) Transport Layer Protocol, RFC 4345, 2006.

Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol, RFC 4419, 2006.

The Secure Shell (SSH) Public Key File Format, RFC 4716, 2006.

AUTHORS

OpenSSH is a derivative of the original and free ssh 1.2.12 release by Tatu Ylonen. Aaron Campbell, Bob Beck, Markus Friedl, Niels Provos, Theo de Raadt and Dug Song removed many bugs, re-added newer features and created OpenSSH. Markus Friedl contributed the support for SSH protocol versions 1.5 and 2.0.