AUUG Winter 2002
SSH tips, tricks & protocol tutorial

Damien Miller (djm@mindrot.org)

August 2002

Contents

1 About this document

1.1 Copyright

1.2 Disclaimer

This document is offered in good faith. No responsibility is accepted by the author for any loss or damage caused in any way to any person or equipment, as a direct or indirect consequence of use or misuse of the information contained herein.

1.3 Audience

This document is intended for users and administrators of Unix-like operating systems. It assumes a moderate level of familiarity with the Unix command-line and a basic working knowledge of TCP/IP networking.

1.4 A note on the examples

All the examples contained herein were written for OpenSSH 3.4. They should work relatively unchanged on more or less recent versions of OpenSSH. They are unlikely to work on other SSH implementations without adjustment.

1.5 Revision

This is the initial revision.

2 Introduction

2.1 What is SSH

SSH (Secure SHell) is a network protocol which provides a replacement for insecure remote login and command execution facilities, such as telnet, rlogin and rsh. SSH encrypts traffic in both directions, preventing traffic sniffing and password theft. SSH also offers several additional useful features:

- Compression: traffic may be optionally compressed at the stream level.
- Public key authentication: optionally replacing password authentication.
- Authentication of the server: making "man-in-the-middle" attack more difficult
- Port forwarding: arbitrary TCP sessions can be forwarded over an SSH connection.
- X11 forwarding: SSH can forward your X11 sessions too.
- File transfer: the SSH protocol family includes two file transfer protocols.

2.2 History

SSH was created by Tatu Ylönen in 1995 and was at first released under an open-source license. Later versions were to bear increasing restrictive licenses, though they generally remained free for non-commercial use. He went on to form SSH Communications security which sells commercial SSH implementations to this day. The earlier versions of his code implement what is now referred to as *SSH protocol v.1*.

In 1997 a process began to make the SSH protocols Internet standards under the auspices of the IETF. This lead to the development of *version 2* of the SSH protocol. In the rewrite, the protocol was split into a transport layer, and connection and authentication protocols. Several security issues were also addressed as part of this process.

In 1999, the OpenBSD[1] team discovered (by way of OSSH[2]) the early free versions for Tatu Ylönen's original code and set about cleaning them up to modern standards. The result was named "OpenSSH" and debuted in the OpenBSD 2.6 release of December 1999. OpenSSH was extended by Markus Friedl to support SSH protocol v.2 in early 2000. OpenSSH remains the most popular, complete and portable free SSH implementation and has been included in many OS releases. The full history of OpenSSH is documented here[3].

[1] http://www.openbsd.org/
[2] ftp://ftp.pdc.kth.se/pub/krypto/ossh/
[3] http://www.openbsd.org/history.html

3 Basic SSH usage

3.1 Remote login

The basic syntax to log into a remote host is:

```
ssh hostname
```

If you want to specify a username, you may do it using an rlogin-compatible format:

```
ssh -l user hostname
```

or a slightly more simple syntax:

```
ssh user@hostname
```

If you are running your sshd on a non-standard port, you may also specify that on the command-line:

```
ssh -p 2222 user@hostname
```

3.2 Initial server key discovery

The first time your client connects to a ssh server, it asks you to verify the server's key.

```
[djm@roku djm]$ ssh root@hachi.mindrot.org
The authenticity of host 'hachi.mindrot.org (203.36.198.102)' can't be established.
RSA key fingerprint is cd:41:70:30:48:07:16:81:e5:30:34:66:f1:56:ef:db.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
root@hachi.mindrot.org's password:
Last login: Tue Aug 27 10:56:25 2002
[root@hachi root]#
```

This is done to prevent an attacker impersonating a server, which would give them the opportunity to capture your password or the contents of your session. Once you have verified the server's key, it is recorded by the client in `~/.ssh/known_hosts` so it can be automatically checked upon each connection. If the server's key changes, the client raises a warning:

```
[djm@roku djm]$ ssh hachi
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
cd:41:70:30:48:07:16:81:e5:30:34:66:f1:56:ef:db.
Please contact your system administrator.
Add correct host key in /home/djm/.ssh/known_hosts to get rid of this message.
Offending key in /home/djm/.ssh/known_hosts:24
RSA host key for localhost has changed and you have requested strict checking.
Host key verification failed.
```

3.3 Executing commands remotely

SSH also supports remote command execution. When you log in, a pseudo-terminal is assigned to your session and your session will remain open until you explicitly log out or is killed from the server end. In remote command execution mode, SSH will execute your specified command with the remote user's shell and then exit as soon as it finished:

```
[djm@roku djm]$ ssh root@hachi.mindrot.org "ls -C /bin"
root@hachi.mindrot.org's password:
[          cpio       echo       ksh        mv         rm         sleep
cat        csh        ed         ln         pax        rmail      stty
chgrp      date       eject      ls         ps         rmd160     sync
chio       dd         expr       md5        pwd        rmdir      tar
chmod      df         hostname   mkdir      rcp        sh         test
cp         domainname kill       mt         rksh       sha1
```

Note that this won't work for programs which need a terminal to operate (e.g. text editors such as vi). To use programs like this, you need to force SSH to allocate a pseudo-terminal using the `-t` flag:

```
ssh -t user@hostname vi /tmp/foo
```

3.3.1 Redirecting commands' input and output

You may also redirect standard file descriptors (stdin, stdout & stderr) as usual when using SSH. This makes for some very useful tricks:

```
[djm@roku djm]$ ssh root@hachi.mindrot.org "ls /bin | grep -i rm"
root@hachi.mindrot.org's password:
rm
rmail
rmd160
rmdir
```

In this example, the *grep* command is executed on the remote machine. One could achieve the same output running the *grep* command on the local machine instead:

```
ssh root@hachi.mindrot.org "ls /bin" | grep -i rm
```

Redirection of stdio is very useful for shuffling data between machines. This example loads a hypothetical SQL file onto a remote machine and massages the output:

```
ssh hachi "psql billing" < billing.sql | grep -v ^INFO
```

Warning: a common error when redirecting output from an SSH process is to have commands which produce output in initialisation scripts which are executed every time the shell is run (e.g. .tcshrc, .kshrc, .bashrc, etc) rather than in login scripts (e.g. .profile, .login, .bash_profile). If output-producing commands are in shell init scripts, their output will be included along with the output of your commands. They also break file transfer using SSH.

3.4 File transfer

SSH offers a number of ways to transfer files between machines. Most of these take advantage of the aforementioned input/output redirection features of SSH.

3.4.1 scp

scp is the original SSH file transfer mechanism. It is modeled on BSD rcp, a protocol with a 15+ year history which has no RFC. Its syntax is very simple:

```
scp [user@]host:/path/to/source/file /path/to/destination/file
```

Will copy a remote file to a local destination. To copy a local file to a remote destination, one uses the opposite syntax:

```
scp /path/to/source/file [user@]host:/path/to/destination/file
```

In either of these cases, the source file may be a wild-card matching multiple files. If a patch is left off the destination file specification, the remote user's home directory is assumed. E.g.:

```
scp /home/djm/*.diff hachi:
```

scp does not support copying between two remote destinations very well. It is possible using the following syntax:

```
scp [user@]host1:/path [user@]host2:/path
```

For this to work, host1 must be configured for password less access to host2 (see section 4). Also little feedback is given to the user on whether the operation succeeded.

scp can also copy files recursively:

```
scp -r source-path [user@]host:/destination-path
```

```
scp -r [user@]host:/source-path /destination-path
```

While it is useful for simple file transfer tasks, it has a number of limitations. The most annoying of these is poor handling of file which contain characters which may be interpreted by the shell (e.g. spaces). For example:

```
[djm@roku djm]$ scp "hachi:/mp3/J.S Bach/Matthaus Passion 0101.ogg" /tmp
cp: cannot stat ‘/mp3/J.S.’: No such file or directory
cp: cannot stat ‘Bach/Matthaus’: No such file or directory
cp: cannot stat ‘Passion’: No such file or directory
cp: cannot stat ‘0101.ogg’: No such file or directory
```

In these cases you need to double-escape the characters in question:

```
scp "hachi:/mp3/J.S.\ Bach/Matthaus\ Passion\ 0101.ogg" /tmp
```

Another problem inherent to scp is that it needs to be able to find a scp binary at the remote end. Usually such commands are correctly installed in the remote systems $PATH, but if they are not then transfers will fail:

```
[djm@roku djm]$ scp hachi:/tmp/foo /tmp
bash: scp: command not found
```

3.4.2 draft-secsh-filexfer (a.k.a sftp)

Many of the shortcomings of the scp protocol have been addressed in the IETF working group. The result of this is the protocol described in the *draft-secsh-filexfer-** set of Internet-drafts. This protocol, better known as *sftp*, is a generic file transfer protocol which is designed to be run over any secure transport.

sftp looks very much like the Unix block API, with equivalents to open(), read(), write(), lseek() as well as readdir() and friends. This similarity has led some to consider it more closely related to NFS than "file transfer" protocols such as FTP.

OpenSSH includes an interactive sftp client:

```
[djm@roku ssh-tutorial]$ sftp hachi
Connecting to hachi...
sftp> cd /usr/share/games
sftp> ls
drwxr-xr-x    8 root     wheel         512 Aug 21 19:01 .
drwxr-xr-x   22 root     wheel         512 Apr 30  2001 ..
drwxr-xr-x    2 root     wheel         512 Aug 21 19:01 atc
drwxr-xr-x    2 root     wheel         512 Aug 21 19:01 boggle
drwxr-xr-x    2 root     wheel         512 Apr 30  2001 ching
drwxr-xr-x    2 root     wheel         512 Aug 21 19:01 fortune
drwxr-xr-x    2 root     wheel         512 Aug 21 19:01 larn
drwxr-xr-x    2 root     wheel        1024 Aug 21 19:01 quiz.db
-r--r--r--    1 root     games        2030 Aug 21 19:01 cards.pck
-r--r--r--    1 root     games       10087 Aug 21 19:01 cribbage.instr
-r--r--r--    1 root     games        1565 Aug 21 19:01 fish.instr
-r--r--r--    1 root     games        1941 Aug 21 19:01 wump.info
sftp> lcd /tmp
sftp> get c*
Fetching /usr/share/games/cards.pck to cards.pck
Fetching /usr/share/games/ching to ching
Cannot download a directory: /usr/share/games/ching
Fetching /usr/share/games/cribbage.instr to cribbage.instr
sftp> quit
```

3.4.3 tar-over-ssh

As mentioned in section 3.3.1, ssh can be used as transport to redirect input and output between hosts. This ability makes it easy to transfer files using standard unix archiving utilities like *tar* and *cpio*. These have advantages when you need to transfer a large numbers of file, preserve file attributes exactly and copy hard or symbolic links.

The following example will copy all files and directories from `/usr/share/games` on host `hachi` to `/tmp` on the local machine. Note that this will preserve the directory structure and attributes including utimes, owner, group and permission information.

```
[root@roku root]# ssh hachi "cd /usr/share/games ; tar cf - ./a*" | \
> (cd /tmp ; tar xpvf -)
./atc
./atc/Atlantis
./atc/Game_List
./atc/Killer
./atc/OHare
./atc/Tic-Tac-Toe
```

To copy local files to a remote destination, a symmetrical command may be used:

```
(cd /tmp ; tar cf - ./xyz*) | ssh hachi "cd /tmp ; tar xcvf -"
```

A slight modification to the above example makes it easy to obtain a local tar file of a remote set of files (note the extra compression step):

```
ssh hachi "cd /tmp ; tar cvf - ./* | bzip2 -9" > tmp.tar.bz2
```

This technique is very useful for simple unattended backups, once password-less authentication has been configured (section 4).

3.4.4 rsync

Rsync[4] is a package and algorithm to two sets of files into synchronisation. Rsync just sends the differences between the two sets of files over the network instead of sending their entire contents. Rsync is often used as a very powerful mirroring process or as a replacement for the scp/rcp command. Rsync includes support for ssh with a single command-line option.

Rsync can be used to simple list files on the remote machine, in a particular directory:

```
rsync -e ssh djm@hachi:/tmp/
```

To synchronise/copy a remote set of files to a local set:

```
rsync -ve ssh djm@hachi:/bin/c* /tmp
```

To synchronise/copy a local set of files with a local set:

```
rsync -ve ssh djm@hachi:/bin/c* /tmp
```

Rsync has many more options and features, these are best described in its excellent man page.

[4] http://rsync.samba.org/rsync/

4 Public key authentication

SSH includes an ability to authenticate users using public keys. Instead of authenticating the user with a password, the server will verify a challenge signed by the user's private key against its copy of the user's public key.

Setting up public key authentication requires you to generate a public/private key pair and install the public portion on the server. It is also possible to restrict what a given key is able to do and what addresses they are allowed to log in from.

4.1 Generating public keys

To generate a public key, use the `ssh-keygen` utility. `ssh-keygen` can generate three types of keys: *rsa*, *dsa* and *rsa1*. *rsa1* keys are used for authentication by the legacy SSH protocol v.1, the other two types may be used for SSH protocol v.2 public key authentication. Select the type of key that you wish to generate by passing the `-t` option to `ssh-keygen`. Normally you will want to use *rsa* keys as they are somewhat faster to authenticate than *dsa* keys.

```
[djm@roku ssh-tutorial]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/djm/.ssh/id_rsa):
Created directory '/home/djm/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/djm/.ssh/id_rsa.
Your public key has been saved in /home/djm/.ssh/id_rsa.pub.
The key fingerprint is:
3c:7e:41:2c:d2:51:f8:0b:ef:78:e7:e3:22:eb:af:6a djm@roku.mindrot.org
```

You may also generate keys without passphrases, which are useful when used with key restrictions (section 4.4):

```
ssh-keygen -t dsa -N '' -f ~/.ssh/id_dsa_example
```

4.2 Public key authentication

Once you have generated a key pair, you must now install the public key on the server that you wish to log into. The public portion is stored in the file with the extension `.pub` in an ASCII encoding:

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAoosorAF8t6k6cmNXiPdP4eE63YFLr/3SjA
GLzCKAJ4cWyAPIrIdeaud1e+y5rj+1E6qEYM3Nl6Sju2dL21+ia+toqA2SQCtUrZTBYVyX
2D4f1x31oK4pTIlWrYzGuj+k3h3tmbr5AdUOk5kxki/xiLRx91gIuWC60qCsYJYVV10H9h
2LRNaSh2YRPptf7aJk+4QcwUuu6QB9g4WBznWWpwj7YeT7n57f38kTbSvatr5hrPWTRFYB
qT4LJqvalkrxQNX143uWOmfTMKV2pUBcMWroVR7Xo2d4Gh6VS2rpKxnq+CNjjj12TunVHR
qbbdkua5Ml/HbpHubmta/dGkoFrQ== Laptop key
```

NB. The above is really a single, unbroken line of text.

To enable public key authentication on a server, you need to append the public portion of the key to the `~/.ssh/authorized_keys` file. This may be accomplished with the following command-line:

```
ssh hachi "umask 077; cat >> .ssh/authorized_keys" < ~/.ssh/id_rsa.pub
```

The restrictive umask is required because the server will refuse to read `~/.ssh/authorized_keys` files which have loose permissions. Once the public key is installed on the server, you should now be able to authenticate using your private key:

```
[djm@roku ssh-tutorial]$ ssh djm@localhost
Enter passphrase for key '/home/djm/.ssh/id_rsa':
Last login: Thu Aug 29 11:08:29 2002
[djm@roku ssh-tutorial]$
```

Notice we are asked for the *private key's* passphrase instead of the user's password on the server.

4.3 Using ssh-agent

So far the use of public key authentication may not seem to have much benefit - we have only traded the need to type your server's password with the need to enter a (potentially longer) private key passphrase.

The solution to this inconvenience is `ssh-agent`, a small program which you run once per login (or X11) session and load your key(s) into. Once `ssh-agent` has your key(s) loaded, it will automatically provide them to the ssh client.

To start up `ssh-agent`, you need something like the following line in your `.profile` (or equivalent):

```
test -z "$SSH_AUTH_SOCK" && eval `ssh-agent -s`
```

When executed, `ssh-agent` will emit a couple of environment variables to standard output. The *eval* directive ensures they are imported into your environment. The *test* directive at the start of the line ensures that you don't end up running excess copies of `ssh-agent`.

Once `ssh-agent` is running, you need to load your private keys into it. This may be done using the `ssh-add` program. Running ssh-add with no arguments will attempt to load the three default key files (protocol v1 RSA, protocol v.2 RSA & DSA) into your agent:

```
[djm@roku ssh-tutorial]$ ssh-add
Enter passphrase for /home/djm/.ssh/id_rsa:
Identity added: /home/djm/.ssh/id_rsa (/home/djm/.ssh/id_rsa)
```

Once keys are in the agent, you can log in without the need to re-enter your passphrase:

```
[djm@roku ssh-tutorial]$ ssh djm@hachi
Last login: Thu Aug 29 12:40:18 2002 from localhost.localdomain
```

You also use `ssh-add` can check which keys are loaded into the agent:

```
[djm@roku ssh-tutorial]$ ssh-add -l
2048 40:a6:0a:59:e9:15:c0:d6:85:87:ec:63:5d:cc:06:ab /home/djm/.ssh/id_rsa (RSA)
2048 39:9f:9c:47:a9:be:94:f6:1e:e6:a5:97:2d:b0:74:c3 /home/djm/.ssh-old/id_rsa (RSA)
```

`ssh-add` also provides the ability to delete keys from the agent:

```
[djm@roku ssh-tutorial]$ ssh-add -D
All identities removed.
```

4.4 Public key restrictions

Public keys may have restrictions placed on them at the server end. The most common restriction is the so-called *forced command*. This forces a given key to always execute a specified command, regardless of what was requested by the client. This is done using the following syntax:

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/authorized_keys
command="/bin/ls -l /tmp" ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAoosorAF
8t6k6cmNXiPdP4eE63YFLr/3SjAGLzCKAJ4cWyAPIrIdeaud1e+y5rj+1E6qEYM3Nl6Sju
2dL21+ia+toqA2SQCtUrZTBYVyX2D4f1x31oK4pTIlWrYzGuj+k3h3tmbr5AdUOk5kxki/
xiLRx91gIuWC60qCsYJYVV10H9h2LRNaSh2YRPptf7aJk+4QcwUuu6QB9g4WBznWWpwj7Y
eT7n57f38kTbSvatr5hrPWTRFYBqT4LJqvalkrxQNX143uWOmfTMKV2pUBcMWroVR7Xo2d
4Gh6VS2rpKxnq+CNjjj12TunVHRqbbdkua5Ml/HbpHubmta/dGkoFrQ== Laptop key
```

The example forces the use of the specified key to run `/bin/ls -l /tmp` at login:

```
[djm@roku ssh-tutorial]$ ssh djm@hachi netstat
arch        date       gunzip mv         sleep
ash        dd       gzip netstat         sort
...
```

Notice how the command specified on the command-line was ignored. The same thing would have happened if I had not specified a command. When a forced command is applied, the original command that the client requested (if any) is stored in the `$SSH_ORIGINAL_COMMAND` environment variable. This may be useful in scripts which restrict access to one of a set of predefined allowed commands.

Another useful restriction is the `from=""` clause. This permits access using the specified key from hosts listed within, but denies access to everyone else. Note that this denial does not prevent the user from authenticating via another means, e.g. password. Basic wild card support is allowed in `from=""` restrictions:

```
from="192.168.*" ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAoosorAF ...
```

The same restriction mechanism may also be used to set environment variables:

```
environment="FREEDOM=SLAVERY" ssh-rsa AAAAB3NzaC1yc2EAAAABIw ...
```

There are a number of other restrictions relating to channel forwarding (explained in section 5) and pseudo-terminal requests. These are important if you wish to fully restrict a key:

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/authorized_keys
from="192.168.*",command="cvs server",no-pseudo-terminal,no-agent-forwarding,
no-X11-forwarding,no-port-forwarding ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQ
...
```

This key is only allowed to connect from 192.168.0.0/16, is not allowed to request a pseudo-terminal, is not allowed to set up any forwarding and is forced to use the command `cvs server`. This, incidentally, is an excellent way to provide CVS only access for remote developers.

Highly restricted, password-less keys are very useful for automated tasks such as remote backup:

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/authorized_keys
command="cd /var/cvs ; tar cvf - ./* | bzip2 -9 | gpg --encrypt -r djm@mindrot.org",
no-pty,no-agent-forwarding,from="192.168.*",no-X11-forwarding,no-port-forwarding
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQ
...
```

Connecting with the specified key to this host would yield an compressed, OpenPGP encrypted backup of `/var/cvs`. This could be utilised with a password-less private key out of *cron*:

```
[djm@roku ssh-tutorial]$ sudo crontab -lu root
0 0 * * * ssh -i .ssh/id_rsa_backup hachi > /var/backup/cvs-‘date +%Y%m%d‘.tar.bz2.gpg
```

5 SSH Forwarding

The SSH protocol has the ability to multiplex various connections over a SSH channel. These "forwardings" allow you to transport TCP/IP, X11 and `ssh-agent` sessions over a SSH session.

5.1 Authentication agent forwarding

The most simple example of a forwarding is "agent forwarding". This allows you to forward a connection to a local `ssh-agent` (see section 4.3) over a SSH connection, so you can continue to use it on the remote machine.

This is not switched on by default as it may lead to security problems if you forward your agent (containing your private keys) to an untrusted host. Between trusted hosts, agent forwarding is very useful.

Agent forwarding may be enabled on the command-line:

```
ssh -A trustedhost
```

Or in the client configuration file on a per-host basis:

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/config
Host trustedhost
ForwardAgent yes
```

Once activated, it is just like having an agent running on the remote machine:

```
[djm@roku ssh-tutorial]$ ssh -A hachi
Last login: Thu Aug 29 12:58:01 2002 from localhost.localdomain
[djm@argon djm]$ ssh-add -l
2048 40:a6:0a:59:e9:15:c0:d6:85:87:ec:63:5d:cc:06:ab /home/djm/.ssh/id_rsa (RSA)
2048 39:9f:9c:47:a9:be:94:f6:1e:e6:a5:97:2d:b0:74:c3 /home/djm/.ssh-old/id_rsa (RSA)
```

5.2 X11 forwarding

The SSH protocol can also forward X11 connections, allowing you to securely display remote X11 apps locally. Again, this option is not on by default for security reasons. It also requires that the server end have an `xauth` binary accessible to set up the MIT-MAGIC-COOKIE-1 authentication for your X server.

X11 forwarding may be enabled from the command-line or the client configuration file:

```
ssh -X hachi xclock
```

```
[djm@roku ssh-tutorial]$ cat ~/.ssh/config
Host trustedhost
ForwardX11 yes
```

5.3 Port forwarding

One of the most flexible uses of SSH is port forwarding, which allows SSH to forward arbitrary TCP sessions. Since these connections are carried over the SSH channel, they are fully encrypted. This makes port-forwarding useful as a way to add security to traditionally insecure protocols.

SSH supports port-forwarding from server to client (a.k.a local) and from client to server (a.k.a remote).

Local port forwarding allows you to forward a port on the client machine through a SSH connection to a host and port which the remove SSH server will connect to. Local port forwardings may be specified using the `-L localport:remotehost:remoteport` command-line option. For example this command will make the local port 8000 connect to the remote host 10.88.45.12 port 80:

```
ssh -L8000:10.88.45.12:80 somehost
```

It is also possible to enter these into the client configuration file:

```
Host fw.somedomain.com.au
        LocalForward 8000 somehost.int.somesomain.com.au:80
```

This is very useful for administering machines which live behind firewalls. Using port-forwarding over a connection to the firewall, you can gain access to all the TCP services of the protected machines as though you are connecting from the firewall itself. Another useful trick is establishing a port-forward to a remote proxy server to circumvent a local web-filter.

When using local port-forwarding the default behavior is to only allow connections from localhost to the forwarded ports (this is done for security reasons). To allow other addresses to connect to the forwarded port you need to specify the *GatewayPorts* option. This may be done on the command-line as `-ogatewayports=yes` or in the client configuration file.

Remote port forwarding is the opposite: it connects a port on the server end to a host and port on the client side. The syntax is similar: `-R remoteport:localhost:localport`. The following example will cause connections to port 2500 on the remote end to connect to 10.34.54.12 port 25 on the local end:

```
ssh -R2500:10.34.54.12:25 somehost
```

5.4 Dynamic port forwarding

OpenSSH also supports a mode which allows "dynamic" port-forwarding. In this configuration, OpenSSH acts as a SOCKS4[5] proxy on a specified port. Clients connecting to this port can specify a remote address and port they wish to connect to using the SOCKS4 protocol. This mode is useful for "burrowing" through firewalls (if you have clients which support SOCKS4). Dynamic port forwarding is setup using the `-D port` flag, where *port* is the port that the ssh client will listen for SOCKS4 requests on.

[5] http://www.socks.nec.com/protocol/socks4.protocol

6 SSH Implementations

6.1 OpenSSH

http://www.openssh.com/

OpenSSH is the most popular of the SSH implementations[6]. OpenSSH support both SSH protocols (v.1 & v.2) and is distributed under an open-source BSD license. OpenSSH runs on *BSD, Linux, Solaris, Windows (via CygWin), HP/UX, Irix, Mac OS X, AIX, SCO, Tru64 and many other platforms.

OpenSSH is now included in most free operating system distributions (Linux, *BSD) as well as several commercial ones (including Mac OS X and IBM AIX). It also forms the basis of the Solaris 9 SSH implementation.

6.2 SSH Communications Corporation

http://www.ssh.com/

SSH communications corporation was founded by Tatu Ylönen, the originator of SSH). They provide commercial implementations of the SSH protocols v.1 and v.2.

They also offer non-commercial use of their v.1 implementation and a restricted version of their v.2 implementation. Their products are supported under Windows, Linux and several flavors of Unix.

6.3 Unix

- OSSH

  ftp://ftp.pdc.kth.se/pub/krypto/ossh/

  A SSH protocol v.1 only implementation by Björn Grönvall, based on old code from Tatu Ylönen which was released under an open-source license. OSSH formed the basis for early versions of OpenSSH.

- FreSSH

  http://www.fressh.org/

  A free SSH protocol v.1 only implementation by Eric Haszlakiewicz, Thor Lancelot Simon and Jason R. Thorpe.

- LSH/Psst

  http://www.net.lut.ac.uk/psst/

  A GPL SSH protocol v.2 implementation by Niels Möller. LSH is interesting for its support of SPKI[7].

6.4 Windows

- PuTTY

  http://www.chiark.greenend.org.uk/ sgtatham/putty/

  A high-quality telnet and SSH protocol v.1 & v.2 implementation from Simon Tatham. It includes a ssh-agent (pagent) as well scp and sftp support.

[6] Source: http://www.openssh.com/usage/index.html

[7] Simple Public Key Infrastructure - http://www.syntelos.org/spki/

- TTSSH

  http://www.zip.com.au/ roca/ttssh.html

  A free plugin providing SSH protocol v.1 capability for Tera Term Pro[8], a free terminal emulator. Written by Robert O'Callahan.

6.5 Macintosh

- NiftyTelnet SSH

  http://www.lysator.liu.se/ jonasw/freeware/niftyssh/

  An enhanced version of NiftyTelnet[9], which provides SSH protocol v.1 support.

- MacSSH

  http://www.macssh.com/

  A free SSH protocol v.2 implementation based on BetterTelnet[10] and LSH.

6.6 Other

- Top Gun SSH

  http://www.ai/ iang/TGssh/

  A SSH protocol v.1 implementation for palmtops running PalmOS. It requires PilotSSLeay[11].

- MindTerm

  http://www.appgate.org/

  Mindterm is a Java implementation of the SSH protocols. Mindterm 1 is SSH protocol v.1 only and is open-source. More recent versions support protocol v.2 and are commercial.

- Java-SSH

  http://www.cl.cam.ac.uk/ fapp2/software/java-ssh/

  A SSH protocol v.1 client with a confusing license.

- SSH Plugin

  http://www.mud.de/se/jta/doc/plugins/SSH.html

  Another java SSH protocol v.1 client. Open-source (GPL) license.

[8] http://hp.vector.co.jp/authors/VA002416/teraterm.html
[9] http://andrew2.andrew.cmu.edu/dist/niftytelnet.html
[10] http://www.cstone.net/ rbraun/mac/telnet/
[11] http://www.isaac.cs.berkeley.edu/pilot/