

The Secure Shell and OpenSSH

SECURE ACCESS

As anyone who has ever left a vital file behind will appreciate, the ability to remotely connect to a system is immensely useful. Derek Clifford explains how to do this securely with SSH



Derek Clifford – Director of Micro Logic Consultants, a consultancy specialising in configuration management, design and rollout of standard configurations, and general Windows and unix support

With more and more users permanently connected to the Internet, it can be useful, when away from your home or office, to be able to connect to your own server or network. In most cases (I hope) this will have been made virtually impossible by the firewall software or hardware installed as of necessity these days. Simply opening up the firewall to allow FTP, telnet and other communication would be madness, and apart from the vulnerability would further compromise the systems because these programs transmit unencrypted passwords. The secure shell (SSH) offers a solution to this problem, by both controlling access in a secure way, and by using public key encryption to secure communication.

History

SSH was originally written by Tatu Yl nen, and the first release was Freely available. However further developments of the original program were issued under more restrictive licences, which severely limited its commercial use. In 1999 Bj rn Gr nvall took the original Free release and produced a more reliable product called OSSH. When this became known to the developers of the OpenBSD system, they took this version and produced OpenSSH, which contained no proprietary or patented software or algorithms, such components being used from external libraries. The OpenBSD group continued to develop OpenSSH, but found that porting to other Unix systems was complicated, and required many changes for system dependencies. Thus the OpenBSD group now produce the core developments of OpenSSH for OpenBSD, and other groups port this code to produce a portable version.

Legal problems

Like Phil Zimmerman's PGP there were both legal

and commercial problems with the product. The ban on the export of strong encryption from the USA was overcome by sending a non-US developer to Canada to develop the first version of OpenSSH. The RSA patent on the asymmetric encryption algorithm made legal commercial use difficult, but this problem disappeared with the expiry of the patent.

Protocols

The concept of public key cryptography in which a pair of keys are used, one remaining secret, the other freely publishable to all was mooted by Diffie and Hellman in 1976. Up to this time the major cryptographic algorithms relied on a single key being kept secret and accessed only by the sender and recipient of a message. In 1977 a practical implementation of the public/private key system was developed by Rivest, Shamir and Adleman (RSA). The RSA algorithm and other further developments of the technique are the most popular and most secure methods of encryption available. OpenSSH offers the choice of RSA and DSA algorithms for the identification of users and hosts

The original SSH1 protocol has two variants: 1.3 and 1.5. These used the public key/private key RSA (RSA public key encryption) algorithms for authorisation, and simpler 3DES (DES encryption algorithm) and Blowfish (Blowfish cipher) systems for encoding data. Problems with the RSA patent made commercial use of SSH difficult, but the US patent expired in September 2000, so there is no longer a problem. SSH1 uses a cyclic redundancy check to maintain data integrity, but this has been found to be crackable.

SSH2 was introduced to overcome the RSA patent issue, and to improve data integrity. The DSA (Digital Signature Algorithm) and DH (Diffie-Hellman key agreement) encryption algorithms are used for

authentication, with which there are no patent problems. The CRC problem is solved by using a HMAC algorithm.

OpenSSH supports all of these variants, but there is little point in using anything but SSH2, unless a system does not have suitable clients available.



Getting OpenSSH

The latest version of OpenSSH is 3.2.3, and was released on 22 May 2002. The portable software for non-BSD systems is designated with version numbers such as 3.2.3p1. rpms for Red Hat distributions and a source rpm are also available. The current portable download is *openssh-3.2.3p1.tar.gz*, and a suitable download mirror site (there is a very extensive set of mirrors) can be located at <http://www.openssh.com/portable.html>. The software requires two other packages to be installed, Zlib (a compression library) and OpenSSL (Secure Socket Layer) 0.9.6 or later.

SSH Components

The secure shell system comprises a server daemon *sshd*, several clients: *ssh* and *slogin* (secure equivalents to *rsh*, the remote shell, and *rlogin*) *scp* (secure remote copy), *sftp* (secure ftp) and utilities for generating and using identification keys. The daemon needs to be started automatically on the remote machine through one of the startup scripts, and the clients and utilities need to be installed on the client machine. In practice the easiest option is just to install the software on both client and server, as it is necessary to generate a host key for each machine, which the installation software does automatically.

Installation

For the majority of Linux and other Unix systems it will be necessary to compile the source. Having expanded the tarball, the sequence:

```
./configure
make
make install
```

will compile the system, install it and generate the host keys. The latest version installs by default to */usr/local/sbin/ssh*, and its configuration files to */usr/local/etc* which may not be where an earlier version exists in your distribution. These can be overridden with the switches:

```
./configure --prefix=/usr --sysconfdir=/etc/ssh
```

which will install to */usr/sbin/ssh* with configuration files in */etc/ssh*.

The system is controlled by the configuration files */etc/ssh_config*, which controls the client programs and *sshd_config*, which controls the server daemon. A user can override these global settings through settings in the local *~/.ssh/config* file. Options in *ssh_config* are applied to a specific host, or group of hosts selected by wildcards, and control the overall parameters to be used when communicating with that host. Settings are applied once only, so host specific parameters must be set in the file before system-wide defaults. The order of precedence in selecting the parameters is first any command-line options given to *ssh*, followed by user-defined configuration files and finally the system-wide default file.

Many of the default settings will be suitable for the normal user and are described in the manpages, but there are one or two parameters which are worth looking at. On the client side the parameter *FallBackToRsh* can take the values yes or no, and setting it to yes will cause *ssh* to revert to the standard Unix remote shell *rsh* if *ssh* is not running on the target host. Although a warning is issued this could lead to passwords being revealed. Fortunately the default for this parameter is no. If Xwindows sessions are to be used over the secure shell, the parameter *ForwardX11* and *ForwardAgent* must be set to yes (default is no). This will allow X11 traffic, and automatically set the remote shell's DISPLAY variable to direct the output of the X server correctly. Systems behind firewalls may have difficulty with the fact that *ssh* uses low-numbered ports to make connections. If this is a problem the parameter *UsePrivilegedPort* can be set to no, to cause ports above 1024 to be used. Port 22 will have to be opened to allow the SSH server to function. The SSH daemon configuration file also contains a setting which is required to be enabled if X11 is to be used. The parameter *X11Forwarding* must be set to yes.

Basic use of ssh

Having set up the system and started *sshd* (probably by modifying one of the startup .rc files) the simplest



```

administrator@heisenberg:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/administrator/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/administrator/.ssh/id_rsa.
Your public key has been saved in /home/administrator/.ssh/id_rsa.pub.
The key fingerprint is:
fc:24:23:dc:06:3f:5c:f4:0b:6d:9b:c5:1e:52:74:b2 administrator@heisenberg
administrator@heisenberg:~$

```

Figure 1: Setting up public and private keys with ssh-keygen

option is to start a session on a remote host with the command:

```
bohr# ssh hostname
```

The first time this command is executed the system will report that the identity of *hostname* cannot be confirmed, as the public key of *hostname* is not yet known on the local machine. The identity of the machine should really be verified, but it may not be practical to do so. The message does report the beginning of the remote host's public key, so this may be checked to give some confidence that the correct machine has been reached. On proceeding the system will add the remote host's public key to the list of known hosts, and will in future verify the identity of the host.

Because the user is not yet known to the remote host, the password for the user on the remote machine will be required. The need to type a password each time may be removed if the user sets his public key in the *.ssh/authorized_keys* of the target user's home directory on the remote machine.

Having entered the password, the user is running a shell on the remote host, no password has been sent in readable form over the network, and all subsequent communication between the machines is encrypted.

Setting up a key pair

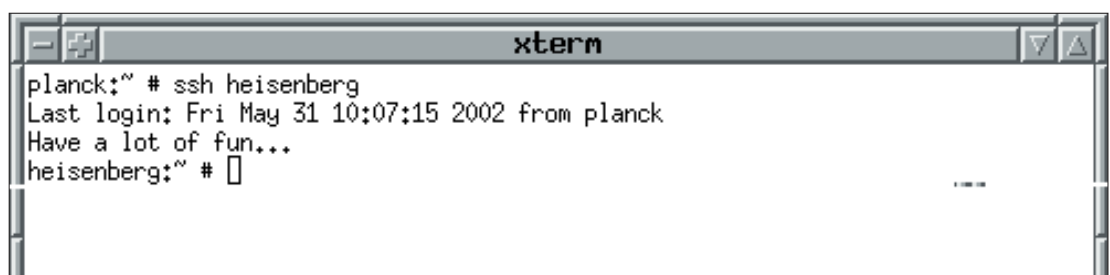
To remove the need to type in a password for the remote user account, a public and private key pair is generated. The utility to perform this task is *ssh-*

keygen. Most of the default settings are suitable, but it is necessary to specify the type of key to be generated. the switch *-t* controls this, and the allowed values are *rsa* and *dsa* for the SSH2 protocol, or *rsa1* for the SSH1 protocol. The key length can range from 512 bits to 2048 bits, with a default of 1024 (*-b* switch). The user is asked where to store the key, but the default is usually appropriate, and a passphrase is input and verified. The passphrase cannot be recovered from the key, so if it is lost new keys will need to be generated and distributed. Use of the utility is shown in Figure 1. The output of the utility is two keyfiles, in the case of RSA encryption: *id_rsa* and *id_rsa.pub*. The public key may be widely distributed (*.pub*) but the private key must never be revealed. In order to use the keys, the public key must be installed in the *authorized_keys* file in the *\$HOME/.ssh* directory of the user account to be made accessible on the remote host.

The authentication agent

Simply adding the user's public key to the *authorized_keys* file merely replaces the request for a password with a request for the key's passphrase. The trick to allow secure but friendly access to the remote host is to have the key available in memory, and for this the authentication agent *ssh-agent* is used. The agent is given a command, and all children of the agent inherit the keys added. For example the command:

```
bohr # ssh-agent $SHELL
```



```

planck:~ # ssh heisenberg
Last login: Fri May 31 10:07:15 2002 from planck
Have a lot of fun...
heisenberg:~ #

```

Figure 2: Passwordless but secure access with Xwindows started through ssh-agent

spawns a shell. Keys may now be added and will be available to all sessions started in the shell. Adding the current user's key is the default action of *ssh-add*, while other keys may be added by specifying the user's keyfile:

```
ssh-add /home/user/.ssh/id_rsa
```

For each key to be added the passphrase will be requested, but this will be required only once, any remote sessions being started will automatically supply the key and the user will be logged on without a dialogue. The *-l* switch to *ssh-add* lists the keys available in memory. Obviously to gain the best use of the authorisation agent it should be started as the parent of all subsequent shells in the user's initialisation files.

scp and sftp

Apart from the fact that there are additional switches for selecting encryption types, and if interactive authentication is used the programs will request passwords or passphrases, these programs behave in exactly the same way as *rcp* and *ftp*.

Xwindows

It is necessary to set the X11 switches in the configuration files to 'yes' in order to pass X11 traffic, and to set the DISPLAY variable. Obviously it would be very tedious to have to type the passphrase or password in every Xterm opened, so the preferred method of starting the Xwindows system is with *ssh-agent*. This will ensure that the agent makes the security keys available for every window opened (Figure 2).

Windows and Mac clients

If you are stuck with only a Windows or Mac system to access your server, there are some free products available. For Windows PuTTY provides a client which supports SSH (Figure 3), together with *scp* and *sftp* clients, plus the ability to generate key pairs. TTSSH is also a free Windows client

Info

- OpenSSH <http://www.openssh.com/>
- OpenSSL <http://www.openssl.org/>
- Zlib <http://www.gzip.org/zlib/>
- PuTTY <http://www.chiark.greenend.org.uk/~sgtatham/putty/>
- TTSSH <http://www.zip.com.au/~roca/ttssh.html>
- TeraTerm Pro <http://download.com.com/3000-2155-890547.html?legacy=cnet>
- Nifty Telnet
- <http://www.lysator.liu.se/~jonasw/freeware/niftyssh/>
- MacSSH <http://pro.wanadoo.fr/chombier/>

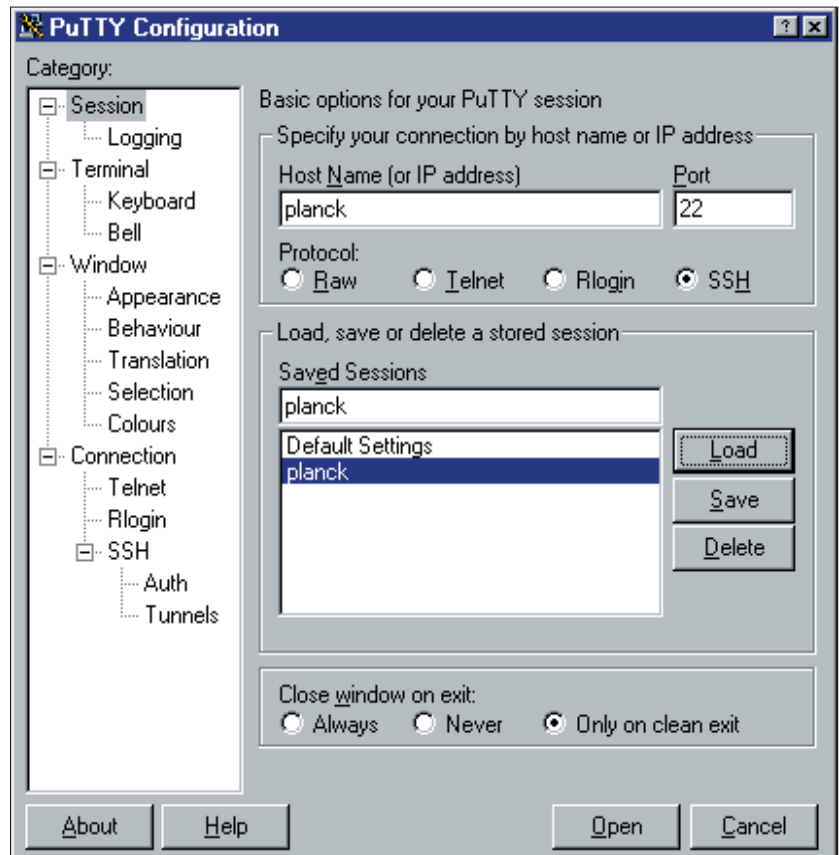


Figure 3: The Windows PuTTY client supports SSH

which is an extension to TeraTerm Pro, but only supports the SSH1 protocol, and does not provide key generation or *scp* and *sftp* utilities. The Macintosh is catered for by Nifty Telnet (figure 4) (which only supports the SSH1 protocol) and MacSSH (which only supports SSH2).

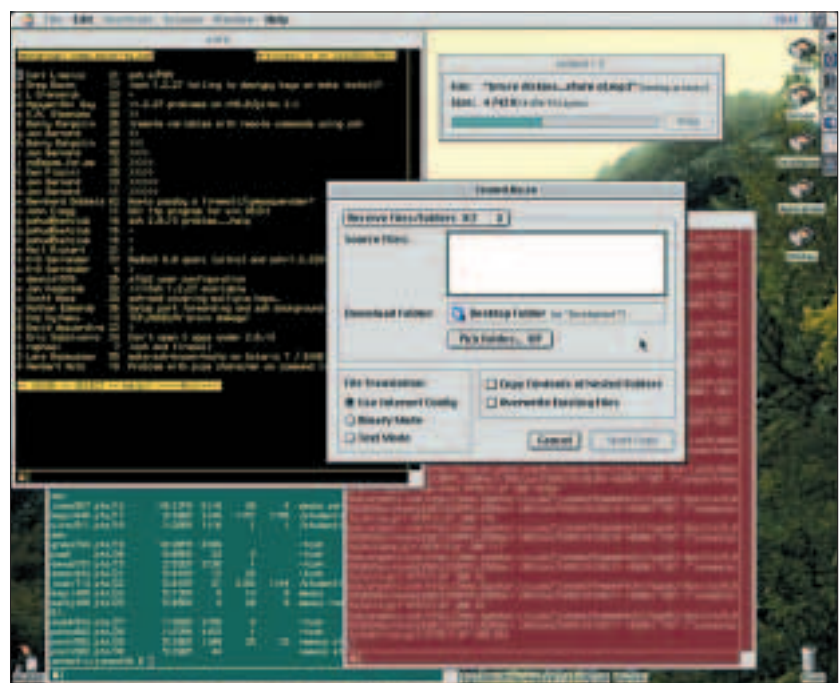


Figure 4: Macintosh support