Aurelio Scetta, 123rf

Secure connections with SSH

TUNNEL BUILDER

Whether you need an encrypted tunnel between multiple PCs or graphical applications over a LAN, the all-purpose SSH tool leaves little to be desired. **BY JÖRG HARMUTH**

Telnet is probably the best known solution for providing users with console access to remote machines. However convenient this dinosaur of network communication might be, it has a major disadvantage: All the data are sent in plaintext over the wire. If an attacker sniffs the connection, he or she will quickly learn the administrative password for the server. Admittedly, it probably isn't quite that easy, but the danger is there all the same. For this reason, all popular Linux distributions install the Secure Shell (SSH) as a safer alternative.

SSH's configuration files are located in */etc/ssh*, where you will find one file for the server (*sshd_config*) and another for the client (*ssh_config*). The files contain a huge number of options, which are explained in detail in the man pages. Users don't typically need to make any major changes. The defaults used by openSUSE 11.0 are user friendly but still secure enough not to make additional configuration worthwhile.

How It Works

The SSH client/server architecture is based on TCP/IP. The SSH server (*sshd*) runs on one machine, where it listens for incoming connections on TCP port 22. The client simply uses this port to connect to the server.

When a connection is established, quite a few things happen in the background. First, the server and client negotiate the SSH protocol version to use for the communications. Currently, SSH 1 and SSH 2 are available, but SSH 2 is standard today because of its better security. Details – including details of encryption – are given in the "SSH Protocol Versions" box.

Second, the server and client negotiate the algorithm, followed by the key that

```
debian:~# ssh sector
The authenticity of host 'sector (192.168.10.100)' can't be established.
RSA key fingerprint is 81:00:6e:dc:49:e1:5b:1d:76:86:8c:a4:55:91:0d:29.
Are you sure you want to continue connecting (yes/no)? y
Please type 'yes' or 'no': yes
Warning: Permanently added 'sector,192.168.10.100' (RSA) to the list of known host
Password:
Last login: Tue Sep 27 14:45:53 2005 from 192.168.10.254
Loading /usr/share/keymaps/i386/qwertz/de-latin1-nodeadkeys.kmap.gz
SECTOR:~$
```

Figure 1: On initial login, SSH imports the host key from the remote machine.

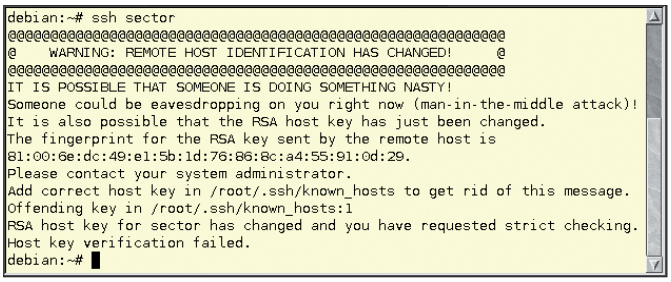

Figure 2: If the host key changes, the SSH client will refuse to connect.

both will use for the data transfer. The key is used once only for the current communication session, and both ends destroy it when the connection is broken. For extended sessions, the key will change at regular intervals with one hour being the default.

Initial Login

The easiest approach is to log in using the classic username/password method. The SSH client will automatically use your username as the login name on the remote machine. The first time you log in, the client will not know the server's host key and will prompt you to confirm that you really do want to set up a connection with the remote machine. The program waits for you to confirm before generating the fingerprint (Figure 1).

If you want to check the key fingerprint, contact the administrator of the remote machine. This prevents man-in-the-middle attacks, wherein an attacker reroutes network traffic to his own machine while spoofing a genuine login to your machine.

If you confirm the security prompt and enter your password in such a case, the attacker will then own your password; thus, some caution is recommended. If the host key changes, the client will refuse to connect when you log in later. Figure 2 shows the output from the SSH client.

The only thing that will help here is to remove the offending fingerprint from your *$HOME/.ssh./known_hosts* file and then accept the new key after contacting the administrator on the remote machine. To configure this behavior, use the *StrictHostKeyChecking* variable in *ssh_config*.

If you do not want to use your current account name to log in to the remote machine but have a different account name, the *-l login_name* option can help you. For example, the command line *ssh -l tuppes sector* will log you into the remote machine as the user *tuppes*. SSH also accepts the following syntax: *ssh tuppes@sector*. To run a single command on the remote machine, you simply append it to the command line (Listing 1).

If you get tired of typing your password, public key authentication gives you an alternative approach. This uses encryption methods such as those used by GnuPG. Before you can use the public key approach, you first need to run *ssh-keygen* to generate a pair of keys: *ssh-keygen -b 1024 -t rsa*. The software will tell you that it has created a keypair with a public key and a private key on the basis of the RSA approach. When prompted to enter a password, press Enter twice. The program will then tell you where it has stored the data and will display the fingerprint for the new key.

In the example here, the software generated an RSA keypair (the *-t rsa* option) with a length of 1024 bits (the *-b 1024* option). An RSA key is fine for use with both protocol versions. For security reasons, the key length should not be less than 1024 bits. To be absolutely safe you can use a key length of 2048 bits: 2048-bit keys are regarded as safe until the year 2020 based on the current state of the art. The key length has no influence on the data transfer speed because the program does not use this key to encrypt the data.

The next step is to copy the public key to the *$HOME/.ssh/ authorized_keys* file on the remote machine from, for example, a floppy disk:

```
mount /media/floppy
cat /media/floppy/id.rsa.pub >> ➥
   $HOME/.ssh/authorized_keys
umount /media/floppy
```

Certainly you should avoid transferring the key by an insecure method, such as email or FTP. Figure 3 shows the fairly unspectacular login with the new key.

Passwords protect keys for interactive sessions; otherwise, anybody with physical access to your computer could use your keys to log in to the remote machine. Key-based, password-free logins are often used to automate copying of files to remote machines.

For example, if you back up your data every evening and would like to automatically copy your data to a remote machine, keys without passwords are a useful approach. If the key was password protected, you would need to enter the password for the SSH key to copy the data – so much for automated copying.

Useful Freebies

The SSH package includes two more useful programs: Secure Copy (*scp*) and Secure FTP (*sftp*). As the names suggest, these programs are used to copy and transfer files by FTP via SSH. The basic syntax for the two programs is similar.

For example, the following command copies a file named *test.txt* from your home directory on the remote machine to your current working directory:

```
scp RemoteComputer:test.txt .
```

Depending on your authentication method, you might need to enter your password to do this; however, the colon is mandatory in all cases. It separates the name of the remote machine from the pathname. Also, you need to specify the local path. The easiest case is your current working directory, which is represented by the dot at the end of the line. If you want to copy multiple files, just type a blank-delimited list of the file names:

Listing 1: Running Commands on the Remote Machine

```
01 jha@scotti:~$ ssh sector "ls -l"
02 Password:
03 insgesamt 52
04 Drwxr-xr-x 3 tuppes users 4096 2005-08-26 12:38 .
05 Drwxr-xr-x 16 root root 4096 2005-09-07 13:47 ..
06 -rw-rw-r-- 1 tuppes users 266 2005-04-12 12:00 .alias
```

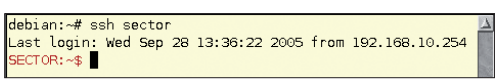

Figure 3: Public key authentication makes the login more user friendly by removing the password prompt.

```
debian:~# netstat -tlpn | grep 23 | grep ssh
tcp        0      0 127.0.0.1:23            0.0.0.0:*               LISTEN      3311/ssh
tcp6       0      0 :::22                   :::*                    LISTEN      2364/sshd
tcp6       0      0 ::1:23                  :::*                    LISTEN      3311/ssh
debian:~# netstat -tpn | grep ssh
tcp        0      0 192.168.10.254:32790    192.168.10.100:22       ESTABLISHED3311/ssh
debian:~#
```

Figure 4: The netstat program showing an existing SSH tunnel.

```
scp RemoteComputerA:test1.txt ➥
   RemoteComputerB:test2.txt .
```

If you use the standard login approach, the client will prompt you to enter your password for each file you copy. If you use the public key method discussed previously, there is no need to type a password. The command *scp RemoteComputerA:test.txt RemoteComputerB:* copies the file from remote computer A to remote computer B. To copy a file as the user *tuppes* from */home/tuppes/files* to your local directory, type:

```
scp ➥
 tuppes@RemoteComputer:files/test.txt .
```

Unlike SSH, you do not specify the *-l username* option here. If you are copying in the other direction – from local to remote – the procedure is just as easy:

```
scp ./test.txt tuppes@➥
   RemoteComputer:/files/
```

scp copies the *test.txt* file from your current working directory to */home/tuppes/files* on the remote machine. Again, watch out for the closing colon.

Sftp uses the same command structure as *scp* but has two operating modes: an interactive mode, like the one you might be familiar with from FTP, and a batch mode. To use *sftp* to retrieve the sample file from the remote machine in batch mode, type the following:

```
sftp RemoteComputer:test.txt .
```

If you type *sftp RemoteComputer:test.txt remote_test.txt*, the program will rename the local copy of the file to *remote_test.txt*. Typing *sftp RemoteComputer* opens an interactive, encrypted FTP session on the remote machine, and the server will accept FTP commands such as *GET* or *PUT* in the session.

Building Tunnels

SSH also lets you encapsulate other protocols. For example, you can run the telnet protocol over an encrypted SSH connection – and do it transparently for users. The technical term for encapsulating one protocol inside another is tunneling.

The standard specifies that programs must be running on the same machine to use the tunnel. If you want to let other machines on the network use the tunnel, you must specify *-o GatewayPorts = yes* when setting up the tunnel. The alternative approach is to set the option in the *ssh_config* configuration file.

This setup is similar to a VPN (Virtual Private Network) connection but is easier to implement. The SSH variant has the disadvantage that you can only forward a single TCP port. Thus, you need an SSH tunnel for each port you want to forward. If you want to encrypt all communications between two machines, a VPN is probably a better choice.

Any user can set up a tunnel, although tunnels for privileged ports (i.e., ports below 1024) are reserved for root. To open a tunnel to a remote machine encapsulating the telnet protocol (port 23), enter the following:

```
ssh -c blowfish -L ➥
   23:RemoteBox:23 RemoteBox
```

The command uses the *-L* option to open a tunnel from local port 23 on the local machine (the first *23*) to port 23 on the remote machine. The fast Blowfish method is used for encryption. If you type two remote machine names, you can take advantage of another of SSH's features: the ability to build a tunnel that opens a tunnel from the first machine, via the second, to a third. The following command

```
ssh -L 23:192.168.1.1:23 192.168.20.5
```

starts the tunnel on the local machine, and routes it by way of an intermediate station (192.168.1.1) to its endpoint. The generic syntax for opening a tunnel from the local machine to the remote computer is thus: *ssh -L LocalPort:Remote-*

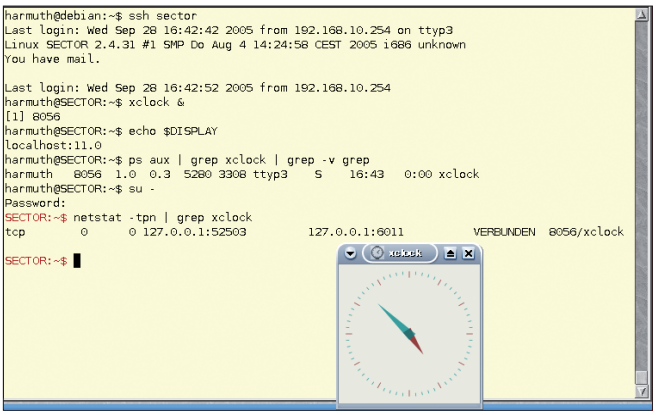

Figure 5: A forwarded X11 connection – the Xclock is running on a remote machine.

ComputerA:RemotePort RemoteComputerB. For a direct tunnel, the two host designations are identical.

Tunnel Tricks

In Figure 4, the *netstat* command demonstrates that I really have set up a telnet connection via SSH. The first *netstat* command tells me that an SSH process with a process ID of 3311 is listening on port 23. The second command shows that a connection to port 22 with precisely this PID (3311) exists.

If you were to look more closely at the syntax used to open a tunnel, you might be led to assume that the local and remote ports do not need to be identical – and this is true. Assuming the remote machine is running a proxy configured for transparent proxying on port 3128, you could redirect all HTTP requests in the following way:

```
ssh -o GatewayPorts=yes -L 80:RemoteComputer:3128 RemoteComputer
```

This process of redirecting one port to another is known as port forwarding. In order for other computers on the network to use the tunnel, the use of the *-o GatewayPorts = yes* parameter is required.

In a similar fashion, tunneling works in the reverse direction. The following syntax allows you to set up a return tunnel from the remote machine to your local computer:

```
ssh -R RemotePort:LocalComputer:LocalPort RemoteComputer
```

In my proxy example, this would be:

```
ssh -o GatewayPorts=yes -R 3128:LocalComputer:80 RemoteComputer
```

Graphical Tunnels

The X Window System is natively network-capable, but almost nobody uses this ability because communications are again unencrypted over the wire. Tunneling with SSH makes this a far more attractive proposal.

To tunnel X11, the SSH daemon (*sshd*) emulates an X server and occupies a display (number 11 by default). When you log in to the server, the server sets the *DISPLAY* environmental variable to this value, or to *localhost:11.0* to be more precise. The idea is to avoid collisions with the X server running locally. Information sent by a computer to this display is encrypted and sent to your machine.

OpenSUSE 11.0 enables X11 forwarding (the technical term for the process I just described) by default. If needed, you can disable X11 forwarding on the machine configured for forwarding by setting the *X11Forwarding* variable to *no* in *etc/ssh/sshd_config*. The *X11DisplayOffset* variable with a default value of *10* defines the distance between the virtual display and the physical display; you should keep the default here.

If the machine on which you want to display tunneled X11 is an openSUSE 11.0 machine, the *etc/ssh/ssh_config* file will already have the *ForwardX11Trusted* variable set to *yes*. This completes the configuration work.

Next, log in to the remote machine and launch, for example, the *Xclock* program. Figure 5 shows the display (*localhost:11.0*), the process, and the matching network connections.

Conclusions

The SSH package includes a collection of important programs that make working on networks far more secure. The feature scope covers anything from basic encrypted connections, through tunneling and port forwarding, to X11 forwarding, leaving very little to be desired in daily use. ■

SSH Protocol Versions

The current versions of SSH are 1.3, 1.5, and 2. Compared with version 2, the capabilities offered by version 1 are fairly limited – especially the choice of encryption algorithms. The version 1 releases use the insecure DES or the secure, but fairly slow, Triple DES (3DES). The Blowfish algorithm provides a fast and – so far – secure encryption technology. Version 2 includes the AES algorithm and others.

Another issue that affects version 1 is that vulnerabilities in the protocol make it theoretically possible to hack the encryption. Compared with version 1, version 2 is slower, which is noticeable if you need to transfer larger volumes of data. Version 1 relies on a key generated client side when negotiating the key. The server sends its public key to the client, which then generates a 256-bit random number, encrypts the number with the server's public key, and returns the results to the server.

The data stream is then encrypted with the random number generated by this method. If a sniffer is listening in on this communication, he or she will then own the (encrypted) key. Brute force will then give the attacker the plaintext key, although this typically takes several years.

Protocol version 2 relies on a Diffie-Hellman exchange that never transmits the key over the wire. The server and client simply exchange data that put them in a position to generate the same key independently of one another.

It won't help an attackers to sniff this data because they will not have the values they need to calculate the key. This approach to generating the key is far more secure because neither of the two communication partners defines the key.

Other enhancements to version 2 include the software's ability to check the data integrity using cryptographic hashes (the Message Authentication Code method) rather than the unreliable CRC (Cyclic Redundancy Check) method. Support for multiplexing is also improved. Both ends seem to transfer data simultaneously.

All of the examples in this article use SSH 2, although some of them will also work with version 1.