AN INTRODUCTION TO SSH

Get to grips with SSH and lord it over remote machines…

If you use more than one computer then you will probably, at some point, want to do something on one that isn't in front of you.

SSH (the Secure Shell) isn't another command-line shell like Bash – it's a networking protocol that you can use to connect securely to a remote computer across an insecure network like the internet. It establishes an encrypted connection to a remote computer, executes a command there and redirects its input and output across the connection. In SSH, the shell is like a wrapper surrounding a path through an insecure network that encrypts everything sent through it.

The way most people use SSH is as a command-line to enter commands on a remote machine, which you can then work on as if it were there in front of you. This remote login is what SSH does if it isn't given a specific command.

Using SSH requires a client on the local computer and a server on the remote one. The implementation found on most Linux distributions is called OpenSSH, and both the client and server packages should be in your distro's repository; they may even be installed by default. On Debian-based systems, the client (**openssh-client**) is installed by default, but you may need to install the server on machines that you want to connect to:

```
sudo apt-get install openssh-server
```

This installs and starts the SSH server. Once you have set up a remote host you can connect like this:

```
ssh remotehost
```

where **remotehost** is the hostname of the remote computer (or you can use its IP address). You will be prompted for your password on that remote machine. SSH assumes your local and remote user names are the same unless you tell it to use a different one

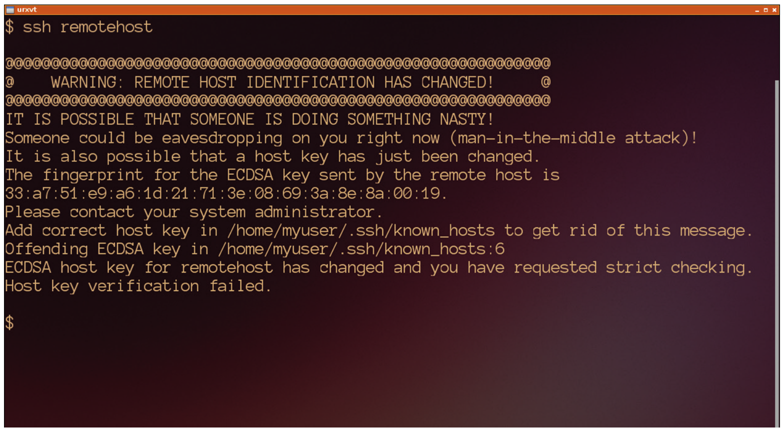
```
ssh user@remotehost
```

You will see a serious-looking warning when you connect to a remote host for the first time, but assuming you're happy that what you've connected to is what you asked for, you can enter **yes** to continue the connection.

```
The authenticity of host 'remotehost (192.168.1.15)' can't be established.
ECDSA key fingerprint is 6d:c4:cf:43:75:a5:79:e0:74:a0:b7:22:b7:da:e1:25.
Are you sure you want to continue connecting (yes/no)?
```

This happens because SSH uses public-key cryptography to authenticate any server you connect to. The server responds with its public key as confirmation of its identity but your SSH client doesn't recognise it. Your client tries to compare keys it receives with copies kept in a file at **~/.ssh/known_hosts**. When you connect for the first time, it has no copy to compare against, so it displays the warning message instead. When you respond **yes** to continue connecting, you tell your client to trust the server and it adds the received key to the **known_hosts** file.

On subsequent connections, the client compares the key sent by the server with the one previously recorded and aborts the connection if they don't match. It provides some useful information to help you investigate and resolve the problem. The connection is authenticated if the client and server keys match.



```
$ ssh remotehost

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    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 a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
33:a7:51:e9:a6:1d:21:71:3e:08:69:3a:8e:8a:00:19.
Please contact your system administrator.
Add correct host key in /home/myuser/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/myuser/.ssh/known_hosts:6
ECDSA host key for remotehost has changed and you have requested strict checking.
Host key verification failed.

$
```

SSH gives a dark and foreboding warning when host keys don't match.

The keymaster

SSH requires that each server has a unique key that consists of the public key it sends to connecting clients and a corresponding private key that it keeps secret. Most distributions automatically generate these server host keys when SSH is installed or started for the first time.

What has happened so far is that the client and server have established a secure communications channel but you have yet to authenticate yourself as a user. The simplest way to do this is by entering a password but you can (and, arguably, should) use a key exchange instead. Before you can do this you need to generate your own key:

```
ssh-keygen -t rsa
```

which, by default, saves keys in **~/.ssh** with the private key in a file called **id_rsa** and the public key in **id_rsa.pub**. You can choose whether to enter a passphrase to protect your private key. If you do, you will need to enter the passphrase whenever you use the private key. An unprotected private key is only as secure as the file it's in.

You need to copy your public key (not your private key) to any remote server that you want to connect to, and there is an easy way to do this:

```
ssh-copy-id remotehost
```

You will need to authenticate before the key can be copied, which will require that you enter your password for the remote machine. If the remote SSH server has disabled password authentication then this will not work and you will need to ask the remote server's administrator to copy your public key onto the server for you. Your public key is stored on the remote server in **~/.ssh/authorized_keys**.

Beyond the command-line

Many people use SSH just to get a command prompt on a remote server, but you can do more than that. If you enable X forwarding on the server then you can launch GUI applications and have them display on your local desktop. The **-X** parameter enables this mode. If you wanted, for example, to run a Firefox browser on the remote host, you could do:

```
ssh -X remotehost firefox
```

Forwarding the X protocol over SSH is an example of 'tunnelling', and you can use SSH as a tunnel for any network traffic. The example below forwards email SMTP traffic to a mail server on the remote network:

```
ssh -N -f remotehost -L 25:remotemailhost:25
```

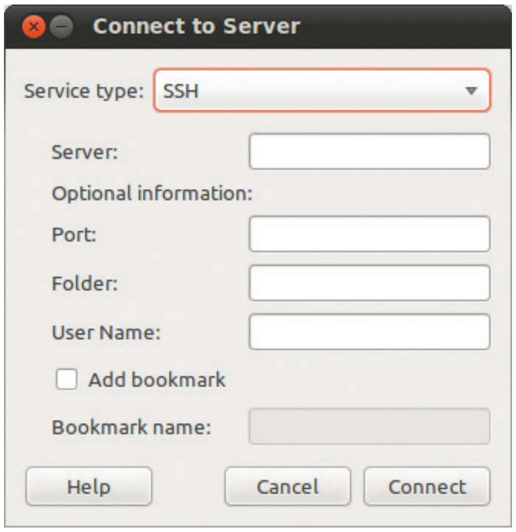
The parameters run SSH as a daemon (**-f**) without executing a remote command (**-N**) because we're tunnelling instead. The **-L 25:remotehost:25** tells SSH to listen on a local port 25 and forward any traffic it receives across the SSH connection to **remotehost** and onwards to **remotemailhost port 25**.

Another use for SSH is to copy files, and there are a few ways to do this. The **scp** (secure copy) command enables you to copy files between your local filesystem and a remote SSH server. You use it similarly to the regular **cp** command, except that the remote path is prefixed with the remote host name (and, optionally, username):

```
scp /local/path user@remotehost:/remote/path
```

You can also copy files with **sftp**, which looks and feels like a basic FTP client. It's also handled by the SSH server, so nothing additional is required to use it.

```
sftp remotehost
```



Gnome has the Gnome Virtual Filesystem (GVFS), which natively supports SSH mounts.

Server configuration

The SSH server has a configuration file, usually located at **/etc/ssh/sshd_config**. Some settings you should review are listed below.

- Set **PasswordAuthentication** to **No** to disable password authentication. Users will need to supply their public key before they can connect.
- Set **PermitRootLogin** to **No** to prevent remote logins as the root user.
- Set **X11Forwarding** to **Yes** to allow use of X applications over SSH.
- Use **AllowUsers** or **DenyUsers** if you need to restrict the users who can connect.
- Use **Banner** to display a message when a connection is established. Specify a file, usually **/etc/issue** containing the message text.

After changing the server configuration, reload it with

```
sudo killall -HUP sshd
```

If you ever need to re-generate a server's keys:

```
rm /etc/ssh/ssh_host_* && ssh-keygen -A
```

If you need constant access to many remote files, you may prefer to mount a remote filesystem and use it as if it were a local one. You can use **sshfs** to do this – it's a userspace filesystem built on top of Fuse.

```
sudo apt-get install sshfs
sudo adduser myuser fuse
```

This installs it and adds your user ID to the **fuse** group, enabling you to mount a directory on a remote SSH server:

```
mkdir ~.mountpoint
sshfs remotehost:/remote/path ~/mountpoint
```

You can then interact with the remote files as if they were local. When you are ready to unmount the file system, use this command:

```
fusermount -u ~/mountpoint
```

Securing SSH

SSH is a secure protocol, but there are steps that you can take to increase its security. The first thing most admins will do is insist that users supply their public key and disable password authentication.

You may also restrict what connecting users can do. This is another use for the **~/.ssh/authorized_keys** file. By prefixing a user's key with a command, any connection will run that command regardless of what the user requested.

```
command="/usr/bin/something args..." ssh-rsa AAAAB3Nza-
C1yc...
```

You might expose an SSH server on the internet so that you can connect to a server in a distant location. If you do this, you can change the SSH port from its default of 22 to a more obscure port of your choice.

John Lane is a technology consultant with a penchant for Linux. He helps new business start-ups make the most of open source.

LV PRO TIP

With the 'ssh agent' you only need to enter your passphrase the first time it's needed.