Logging in with the Secure Shell

by Andy Pepperdine

It is possible to set up a computer so that users of other computers can connect to it as though they were logging in directly. The most common method of doing so is using the so-called Secure Shell, ssh, which encrypts the connection between the two machines to prevent anyone else from eavesdropping on the network traffic.

The encryption is integral to the process and is essential if the connection is across a public network, such as the Internet when connecting to administer a remote web server.

There are applications for Windows that will allow you to log in to a Linux server across ssh, but this paper does not address them. See the references for one source of information.

Introduction

When you log in to your machine, whether a desktop, laptop, tablet or whatever it is, then that machine will execute your commands and applications for you, and deliver the result. All the operations are performed locally on that machine.

However, what if you want to operate on another machine?

There are two distinct cases to consider. If all you want to do is access files, then that can be done with file sharing software, and is not the subject here.

The other method is to use the computing power of the remote machine to perform your calculations and deliver the results to you locally. This may be because the remote machine has applications that will not run locally. Or it may be that you need more power to produce an answer quickly, or perhaps it is because there are security implications so that only operations on the remote machine can access files on that machine.

Whatever, the reason, the secure shell, ssh, was designed to allow you to log in to a distant computer, safely across a public network, if necessary.

Terminology

In these notes, the terms server and client will be used to identify the two machines. It is very important that you do the setup on each appropriately, and I will try to be very clear about which machine is being used at any stage.

The **client** machine is the one you type on – the one with a keyboard and display and is local to you. It can be any of the usual devices that you operate.

The **client user** is the user under which you are working when you log in there.

The **server** machine is the one you log in to remotely and will actually do all the heavy work, while your client merely passes a small amount of data along the wires to tell it what to do and to display the returned information. It could reside on the same desk as you, in the next room, or anywhere in the world.

So long as there is a network connecting the client and server, they can talk to one another.

The **server user** is the name of the user when you log in to the server. Note that is not necessarily the case that the server user and client user are the same. Client user *fred* could be used to log in to server user *george*, for example.

Typically, the common case is that logging on will only give you a command line interface. This is adequate for administering a server. It may not be good enough for home working where a GUI would be good. Both can be arranged.

Methods of access

The shell, ssh, provides two very different methods of making the connection.

The first, and older of the two, is to access the server with your usual login password. This is NOT now recommended. It could be sensible if you are on a local network that you believe has no external interference. But it definitely not safe across a public network, since the login data is not encrypted across the network, and an eavesdropper could see it I clear text.

The other method is by using a public and private key. This method is now the recommended one for all connections. Instead of sending your password across, what happens is that the server sends a challenge to the client using the client's public key. Only the client with the correct private key can answer that challenge correctly, and so be allowed access. After this authorisation exchange has taken place, all data sent across the network will be encrypted, including any passwords that you may need to convey to perform any operation on the server.

I will concentrate almost exclusively on this second method.

Outline of demonstration

There are several important configuration options that can affect how ssh is to be set up. The ones I think you may have to take account of will be introduced after the basic outline of the process has been established.

If a home directory is encrypted, then there will extra things you must do before remote log in is possible. This situation will be considered later.

Preparing the server

Everything in this section is done on the server.

The first thing to do is make sure the server can accept logins over the network. You have to ensure that the ssh daemon is available and running. On Ubuntu or Debian based systems, that is achieved by installing the package *openssh-server*.

For each user identity that may be used when logging in, there will be a home directory on the server for that user, usually named /home/*user*/ where *user* is the name of the user. Authorisation information for that user is held in a directory in the home directory. So login as that user and create the directory (if not already present):

```
cd ~
mkdir ~/.ssh
```

The recommendation is to make sure that this authorisation data cannot be read by anyone else on the server. In fact, it is a good idea to remember that the server may be used by other people as well with their own users and home directories. To prevent peeping by change the permissions on the directory:

```
chmod 700 ~/.ssh
```

Using the default setup, you should now be able to login from a client.

If you wish to use more than one user on the server, then create that directory for each of the server users.

Prepare the client

Preparation of a client is even easier as the client part of ssh is installed by almost all Linux distributions. However, you will still need to create the ssh data area in the home directory of any user wishing to log in to a server. The data is stored in a similar place. So do on the client:

```
cd ~
mkdir ~/.ssh
chmod 700 ~/ssh
```

You are now ready to test logging in.

Machine addressing

In order to access the server, the client needs a way of addressing it, identifying which machine to log in to. This is done by the same means as internet addresses, or local network addresses are done, viz. either by IP address or by URL. For instance, the name

```
zeus@mountolympus.net
```

would refer to the user named *zeus* on the server found at location *mountolympus.net*.

Test logging in (if safe)

If you feel it is safe to log in with your password, then you should be able to make the connection with the command:

```
ssh user@server
```

where user and server are described in the preceding section. On a local network, you may only have IP addresses, but across the Internet, you should use full URL addressing to resolve the domain names.

If you would rather not try this, then skip this section and continue with the key creation steps.

The ssh command will put you into a command line shell running on the server, as you wil see by the fact that the prompt will have changed.

To log out use the command:

```
exit
```

which is the usual way of exiting from a command line prompt.

Configure the server for key authentication

The configuration file for the server side of ssh resides at /etc/ssh/sshd_config and contains all the parameter values to control the ssh daemon. (Note the name of the file: there's a 'd' in it.) The first step will be to save the original configuration file somewhere so if there is a problem you can get back to the starting point again.

Since this a file owned by the administrator user, root, you cannot simply edit it. I would recommend copying it to your local area, edit it there, and then copy back using sudo to switch to root for just the copy.

```
cp /etc/ssh/sshd_config save_area/sshd_config
cp /etc/ssh/sshd_config temp/sshd_config
# use text edit to change the config file
sudo cp temp/sshd_config /etc/ssh/sshd_config
```

To ensure that authentication is by means of keys only, and not passwords, then look for the line:

```
PasswordAuthentication yes
```

and change yes to no. You may wish to postpone this until you are certain keys are working as expected.

The line

```
#AuthorizedKeysFile   %h/.ssh/authorized_keys
```

is commented out in Linux Mint, but specifies where the server will look for keys to use to challenge a user trying to log in. The default location is in the home directory of the server user.

When you are satisfied that you have made the relevant changes to the configuration file, copy it back to the /etc/ssh directory, and then restart the ssh daemon:

```
sudo service ssh restart
```

Creating your keys

Now when a client tries to log on, the server will look for the appropriate public key saved in the authorisation file of the server user. Using this public key, it will challenge the client to decrypt a

message it sends to the client. The client will then use its private key to decrypt and reply to the challenge.

The client must generate this key pair, and then give the server the public part, keeping the private part to itself. This private part will identify the client user and device to the server. each user/device combination should have a unique pair of keys.

To create a key, on the client device logged in as the client user, issue the command:

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

It will prompt you for a location to store the file, and the default is in the home directory at ~/.ssh/ with filename id_*rsa.pub* for the public part, and *id_rsa* for the private part, which is the right place by default for the configuration, so leave it.

It will also ask for a password to unlock the private key. This password will replace the login password when you log in, since it will be needed in order to answer the authentication challenge. If you just hit return at that point, it will not ask for a password to access the private key. In this way you would get access without a password at all.

When this has been generated, look at the directory ~/.ssh and make sure that all the files there have permissions only for the owning user. If in doubt do this:

```
cd ~/.ssh
chmod 600 *
```

That will prevent other users being able to read those files and so usurp your logins to the server.

Copying your public key to the server

Having got your key pair for the client, you will have to transfer the public part to right place on the server. By default, the right place is in the ~/.ssh directory on the server, where there should be a file named authorized_keys (note American spelling). This will contain all the keys to be used to log in as that server user, one after the other. So you should append the public key you have generated for your device to this file.

NOTE: the name of the file you have to transfer should end with *.pub* as it is the public part you need. Do not copy the private key over.

There are several ways to do this, but I'll take the easy way and transfer it on a memory stick. Since only the public key is being transferred, it does not matter who sees it. The private key is the important part.

The way of appending one file to another is, when logged in as the server user on the server:

```
cat /path/to/transferred/public/key >> ~/.ssh/authorized_keys
```

where the path to the transferred key could be directly from the memory stick, or other transferral device.

Test by logging in using the keys

If you are using the default case throughout, then you should be able to log in from the client user without the login password. But be aware that if you set a password on the private key, then that will be needed instead. Make sure you read the prompts before typing in a password of some sort.

If you cannot do so, then check the contents of the sshd_config file on the server, the names and permissions of all the key files, and if you look in the authorisation file, then you may spot the problem in the names of users or devices. Note that the permission level is important – if you have too little protection, ssh will ignore the file or directory, ensure that only the owner can read the relevant file and its owning directory; ssh does not like you being too lax in your security.

If the client's home directory is encrypted

That was all very well for the standard case. However, portable devices often have encrypted home directories, so is this a problem?

Sort answer: No. For clients, access to the home directory is no problem. If you do not leave your home directory available to anyone stealing your device, then you can leave the password for the private key empty.

If the server's home directory is encrypted

However, on the server, the situation is very different. When a request comes in from some client, the user's home directory may not be mounted, that is it has not been decrypted for anyone's use. For any machine that may be logged onto from afar, it is probably a good idea to use encryption, as that will mean that if one user is compromised, the attacker will still not able to access the other user's data – unless the other users are logged in.

The system's root user has special permissions to be able to read the mounted encrypted directories of users, but cannot mount them without the passwords. So in that case, it cannot in advance read the user's .ssh directory to access the authorisation file. Fortunately, there is a way out of this apparent impasse.

The server's configuration can be set up so it can search several different locations for the appropriate file. Hence, you can define the search list to include areas that are not encrypted for those users that have encrypted home directories. This is not a serious security issue since the only keys in the authorisation file are public ones.

The first step is to create a file somewhere else, not in the home directory, that contains the keys for authentication. For example, I created a set of directories under /home/.ssh/ one for each user, like this:

```
sudo mkdir /home/.ssh
sudo mkdir /home/.ssh/user
sudo chmod 700 /home/.ssh/user
sudo chown user:user /home/.ssh/user
```

That set of commands, creates the relevant directories, changes the permissions so that only the owner can manipulate them, and then change the ownership to the user to whom they are to belong.

And then logging in to each *user*, I placed a copy of the authorization file like this:

```
cp ~/.ssh/authorized_keys /home/.ssh/user/authorized_keys
```

where *user* is the name of the server user to be logged in to.

Looking again at the sshd configuration file, /etc/ssh/sshd_config, find the line for the parameter:

```
AuthorizedKeysFile
```

This contains a list of possible names of files that might contain the files of authentication keys. The rest of line will be a list of file names that the server will search for looking for the appropriate key for the login access. If it is commented out, then first uncomment it, and then add to the list, the names of the alternative locations for the key files. For example, it may then look like this:

```
AuthorizedKeysFile %h/.ssh/authorized_keys /home/.ssh/%u/authorized_keys
```

The % signs will be substituted by various names appropriate to the invocation. So, %h means the home directory of the server user. %u means the name of the server user.

Now, for uses that do not have their home directories encrypted, then the authorisation file can remain in their home directory. For those with encrypted home directories, then copies can be placed in the alternative location which is accessible by root. All names listed on the line will be searched until a suitable key is found.

Always remember, that after you have changed the sshd_config file, you must restart the ssh service:

```
sudo service ssh restart
```

Getting right server environment after logging in

Logging in is all very well, but not much good if you cannot read your own encrypted home directory. It will not be accessible until you mount it. Getting round this problem is a bit more tricky. After logging in over ssh you should then, if under a Ubuntu or Debian system, issue the commands:

```
ecryptfs-mount-private
cd
. .profile
```

These will in turn decrypt the home directory for you, but will ask for your login password in order to do so. The second command will then re-establish the working directory to the home directory. Finally, it will execute the .profile file, which all users should have, to do further set up of their environment.

I put these commands into a file and placed them in /home/.ssh/*user*/.profile to make them readily accessible when logging in.

There is in fact something further you can do in order to make invocation of these commands automatic when ssh-ing to the server, which is described in the next section.

Automatically getting the home directory decrypted

This is a little tricky and depends on the exact way the home directory is configured. For Debian and Ubuntu systems this is how it will work. If you log in normally, there is a normal directory hidden behind the mounted encrypted home directory. That hidden directory is the one that you will see when you log in over ssh, as though it is your real home directory. But you cannot see it when you log in to the server since it will automatically mount your home directory and overlay this hidden one. You need a different method to get to it.

Since we have now got access via ssh, then we have an alternative way available to us. First, make sure you have logged out from the server. In fact, to be on the safe side, you might as well re-boot the server to be absolutely certain the directory is not mounted.

Then you can log in to the server from your client, and in doing so you will expose the hidden directory as your home directory, since the home directory has not yet been decrypted and mounted. If you have created your alternative .profile as in the previous section, then all you now need to do is set up a symbolic link in your hidden home directory to this alternative .profile:

```
ln -s /home/.ssh/user/.profile .profile
```

Now when you log in, the commands to mount the proper home directory will be automatically executed, prompting you for the login password to allow the decryption if necessary.

Note when I did this for a Linux Mint system, the home directory had NOT been given write permissions, not even for the owner. So I had to first change those permissions to allow me to write into it before I could create the symbolic link, and then change the permissions back again afterwards. I understand the reasons why the permissions are set that way – it is far safer to prevent accidents on the original hidden directory.

Graphical interfaces on applications

So far, we have looked only at getting access to a common line interface to the server. This may not be quite what you want, especially if you are used to a graphical interface. Let's see how we can mange that next.

The ssh command you use to log in with, has an option which will allow so-called forwarding of the X protocol which is what drives the windowing system. The command looks like this:

```
ssh -X user@server
```

Note that it will not show you the desktop, only individual windows within your local client desktop.

You can now start graphical applications from the server, via the command line, and display them locally on the client, like this:

```
xeyes &
```

and you will see a new window pop up to demonstrate it works.

A very useful tip, is always to put that final '&' sign after the command to start it up, because that will enable the terminal window to continue to operate as a terminal for later commands. If you do not, then it will hang and be unresponsive, until the new application has completed.

If you start a file browser, for Linux Mint it is called *caja*, then you can open applications from there and get them to operate as normally.

When terminating a conenction by exiting from the secure shell (exit command), sometimes, especially after have used an X connection, I noticed that the *ssh* common may hang. In which case a ctrl-C is required to kill the shell. Another way of ending a connection is to provide it with an end-of-file indicator, and on Unix systems that is achieved with the ctrl-D key.

A word of warning. If you are already logged onto the server directly, then the system can get confused with which display to use. For example, if you already have a Libreoffice window open on the server's desktop, then trying to start another one on a client will likely fail, as it usually will merely open another window in the Libreoffice application, and not start a completely new instance of it. This behaviour will depend on the actual application. The moral is not be logged into the server when using *ssh* to it.

Graphics and encrypted home directory

Typically, this is a problem, and I do not yet have a satisfactorily complete answer. When the graphical system (known as X) starts up, it uses a local file to keep a list of where the authorised displays are, whether local, remote, how many, etc. This information is located in a file called .Xauthority and is stored in the user's home directory.

Now, when logging in to the server, the order of events is not optimal. In particular, the X system cannot access its authority file unless the home directory is decrypted. Unfortunately, it seems to try to do it before the log in has taken place, and so cannot see its file. In this case, I tried to do a similar job a worked with the authorisation file, but X is too fastidious about what type of file it wants, and promptly removed my attempted link to an alternative location. It will accept only a .Xauthority file in the home directory.

The partial good news is that you can get the effect you need, by first logging in via ssh without X in one terminal window, then open another and ssh across with the X parameter, and since the first one mounted the home directory properly, the second can find what it needs. Troublesome, but possible.

Other security changes to make

If you particularly wish to improve security to prevent unwanted logins, then reference [7] has some suggestions. Among them are some sensible changes to the default configuration on the server.

One important possibility is to prevent root log in via ssh by changing the sshd_config file to include the line:

```
PermitRootLogin no
```

The default situation is that you can login to root "without password"; what that actually means that passwords cannot be used, only keys – providing you can login as root in any case. Under Debian etc. it is not possible to login as root anyway as it has no password at all (note: no password does not equal an empty password, it is diametrically opposite to it). Instead you have to log in as an administrator user, and then sudo to get to the commands you need.

References

[1] To set up the server for using ssh, with keys where you require it: https://help.ubuntu.com/community/SSH/OpenSSH/Configuring

[2] Generating keys for login use https://help.ubuntu.com/community/SSH/OpenSSH/Keys

[3] More general information: https://www.digitalocean.com/community/tutorials/how-to-use-ssh-to-connect-to-a-remote-server-in-ubuntu

[4] Another tutorial: http://support.suso.com/supki/SSH_Tutorial_for_Linux

[5] Encrypted home directory set up: http://unix.stackexchange.com/questions/47122/cant-do-ssh-public-key-login-under-encrypted-home

[6] Windows to Linux via ssh: http://www.codeproject.com/Articles/497728/HowplustoplusUseplusSSHplustoplusAccessplusaplusLi

[7] Improve security on server: https://wiki.centos.org/HowTos/Network/SecuringSSH