The Shell Game

By icarus

Table of Contents

Introduction

If you've been using the Internet for any length of time, you'll be aware of the immense power this network of connected computers gives you. It allows you, for example, to access your email from anywhere in the world, to share data between users in different countries, and − in one memorable example − to participate in a live birth from a million miles away!

By allowing you to connect to, or "telnet" into, remote computers, the Internet also puts tremendous computing power at your disposal. Telnet allows you to remotely administer computers, to gain access to their information and computing power they possess, and to use that information and horsepower to solve your own problems in a faster, more efficient manner.

However, telnet is a relatively insecure way of working over the Internet. A telnet connection is typically unencrypted, and offers experienced hackers − or bored twelve−year−olds − a number of opportunities to tap into your connection and siphon off information from the data stream flowing back and forth. What is needed is a more secure communication protocol, one which is immune to IP−based attacks, and which uses hard−to−crack cryptographic techniques to protect the data it carries.

SSH.

Like telnet, SSH is a program designed to let you log in to other computers on a network. However, unlike telnet, all the data flowing back and forth in an SSH session is encrypted, and thus secured from hackers attempting to eavesdrop on the connection. Passwords, for example, are sent over a telnet connection in clear−text, and are vulnerable to interception − however, SSH always encrypts data transmissions and thus secures sensitive information from falling into the hands of others.

SSH also offers numerous improvements to the other remote login programs − rlogin, rsh and rcp. Where rlogin and rsh depend on a flat file to establish whether or not to allow remote hosts and users access, SSH relies on public/private key authentication to avoid the use of IP−spoofing or DNS−based attacks.

Finally, SSH allows X11 forwarding, allowing the encryption of all X11 data, and TCP port forwarding, which makes it possible to communicate with other ports on the remote system [and systems that may be further connected to it] via the secure SSH channel, as well.

Sounds like a system administrator's dream come true? You're not far wrong...

Kicking The Tires

If you're planning to use SSH, the first thing to do is make sure that it's available on both the client [usually your local system] and the server [the remote system].

The easiest way to check this is to telnet to port 22 of both hosts, which is the port the SSH daemon traditionally runs on. If SSH services are available, you'll be rewarded with an identification string containing the version number, like this:

```
$ telnet 22 Connected to popeye. Escape character is '^]'.
SSH−1.99−2.1.0 SSH Secure Shell (non−commercial)
```

Nothing? Well, you could politely ask your ISP/Web hosting service/friendly neighbourhood geek to install it for you − or, if you have "root" access to the system, and are comfortable with installing new software on your system, you could download and install it yourself.

The primary distribution site for SSH is now http://www.ssh.org or ftp://ftp.ssh.org/pub/ssh. You'll find that there are two versions − SSH1, currently in version 1.2.27, and SSH2, currently in version 2.1.0. Pick whichever one you like − I'll be explaining how to use them both.

When making a decision on which one to use, I also suggest that you take a close look at their licensing terms, and the conditions under which you are allowed to use them − of the two, SSH1's license is a little more flexible. You should also be aware that the SSH1 and SSH2 protocols are not compatible with each other.

Another option here is to use the OpenSSH package, available at http://www.openssh.com − this package does away with many of the licensing restrictions of SSH1 and SSH2. While this article does not specifically detail the installation and usage procedures for OpenSSH, it's not very hard to adapt these procedures to that software, given that many of the fundamentals of OpenSSH are similar to SSH1.

Once you've downloaded the package, you need to build it − under UNIX, this is the traditional

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

cycle. It's usually a good idea to run these commands as "root".

During the install part of the cycle, you'll notice a set of host keys being generated − this is the private/public key pair for your system, and the two keys are usually stored in the files /etc/ssh_host_key and /etc/ssh_host_key.pub respectively.

Once the binaries have been installed in the right places, you need to start up the SSH daemon. Make sure that you're still "root", change to the directory containing the sshd binary [under Linux, the default path is /usr/local/sbin/sshd] and run it:

```
$ /usr/local/sbin/sshd
```

Now verify that the daemon is running via a telnet to port 22:

```
$ telnet localhost 22 Trying 127.0.0.1... Connected to
localhost. Escape character is '^]'. SSH-1.99-2.1.0 SSH Secure
Shell (non-commercial)
```

Remember that you need SSH at both ends of the connection; in the absence of this, SSH will revert back to using insecure rsh mechanisms to perform a remote login.

Next, I'll be showing you how to generate your own public/private key pair, which you'll be using to authenticate your remote logins.

Gassing It Up

If you're using SSH1, the procedure for using private/public key authentication [also known as RSA authentication] to log into a remote server is very simple. I'll explain it with an example, which assumes that the remote server is named "brutus" and the local system, or client, is named "popeye".

The first thing you need to do is generate a key pair for yourself. Log in to popeye, and run this command from your shell:

```
$ ssh-keygen
```

The key generator will go to work generating a key pair for you.

```
me@popeye:~/ $ ssh-keygen Initializing random number
generator... Generating p: .....................++ (distance
366) Generating q: ..............++ (distance 216) Computing
the keys... Testing the keys... Key generation complete. Enter
file in which to save the key (/home/me/.ssh/identity): Enter
passphrase: Enter the same passphrase again: Your
identification has been saved in /home/me/.ssh/identity. Your
public key is: 1024 35
234565536183551314996877522865573256060327600076719722913911282824162335360
427841873459538384562535811911069044049207983128292709699342973257355481300
683108768866195489712409399828775614944588372935178313183113696793807410180
973411619392897121912532080212214044974785496080800517135372627363843532960
4240340657901 me@popeye.localdomain.com Your public key has
been saved in /home/me/.ssh/identity.pub
```

Once the key generation process is complete, you'll be asked for a password for your private key. This is optional − you can enter a null passphrase − but recommended. Your passphrase may be any combination of letters and numbers, and can also be a complete sentence. Should you decide to change it later, simply use

```
$ ssh-keygen -p
```

Your public key will be saved to ~/.ssh/identity.pub while your private key will be located in ~/.ssh/identity

The public key may be distributed to all and sundry, and should be world−readable. The private key should not be readable by anyone but the owner. Remember that in public−key cryptography, it is not possible to deduce the private key from the public key − which is why this authentication method is so secure.

Next, you need to add this public key to the remote server. Telnet to brutus [the remote host], log in and create a directory in your home area named ~/.ssh. Within that directory, create a file named ~/.ssh/authorized_keys and insert the contents of your ~/.ssh/identity.pub on popeye into that file.

This "authorized_keys" file contains the public keys which are authorized to log in to your account on brutus.

Each key in the file should be on a separate line.

If you don't have telnet access to the remote host, you could also upload your "identity.pub" file via FTP and rename it to "authorized_keys". Alternatively, if you're trying to set this up on a restricted server, you might need to email the system administrator with your public key so that he can add it to the appropriate file.

Done? Log out of brutus.

Test Drive!

Now comes the real test. Use SSH to log in to the remote system with this command:

```
me@popeye:~/.ssh $ ssh −v brutus
```

The "−v" option tells SSH to spew out debugging information. If all goes well, you'll see something like this:

```
me@popeye:~/.ssh $ ssh −v brutus SSH Version 1.2.27
[i586−unknown−linux], protocol version 1.5. Standard version.
Does not use RSAREF. popeye.localdomain.com: Reading
configuration data /etc/ssh_config popeye.localdomain.com:
Connecting to brutus port 22. popeye.localdomain.com:
Connection established. popeye.localdomain.com: Remote
protocol version 1.5, remote software version 1.2.27
popeye.localdomain.com: Waiting for server public key.
popeye.localdomain.com: Received server public key (768 bits)
and host key (1024 bits). popeye.localdomain.com: Host
'brutus' is known and matches the host key.
popeye.localdomain.com: Initializing random; seed file
/home/me/.ssh/random_seed popeye.localdomain.com: Encryption
type: idea popeye.localdomain.com: Sent encrypted session key.
popeye.localdomain.com: Installing crc compensation attack
detector. popeye.localdomain.com: Received encrypted
confirmation. popeye.localdomain.com: Trying rhosts or
/etc/hosts.equiv with RSA host authentication.
popeye.localdomain.com: Server refused our rhosts
authentication or host key. popeye.localdomain.com: No agent.
popeye.localdomain.com: Trying RSA authentication with key
'me@popeye.localdomain.com' popeye.localdomain.com: Received
RSA challenge from server. popeye.localdomain.com: Sending
response to host key RSA challenge. popeye.localdomain.com:
Remote: RSA authentication accepted. popeye.localdomain.com:
RSA authentication accepted by server. popeye.localdomain.com:
Requesting pty. popeye.localdomain.com: Requesting shell.
popeye.localdomain.com: Entering interactive session. Last
login: Wed May 24 14:50:50 2000 from popeye.localdomain.com
You have mail. me@brutus:~/ $
```

Now, wasn't that simple?

What's happening behind the scenes here is very interesting. When popeye first connects to brutus, it sends the user's public key to brutus. brutus then sends popeye a challenge, usually a random number encrypted with the user's public key. popeye receives the challenge, decrypts it with the private key, and sends it back to brutus.

Since one of the principles of RSA authentication is that data encrypted with a public key can only be

decrypted by the corresponding private key, popeye thus proves its identity without actually disclosing the private key, and is granted access to brutus.

A few points of interest:

* In case you're asked whether or not to accept the host key, you should usually accept it − this host key is added to SSH's database of "known hosts".

* In case your username on the remote system is different from that on the local system, you need to specify your username on the remote host on the command line. So, in the example above, if me@popeye wanted to log in as john@brutus, the command would look like this:

```
me@popeye:~/.ssh $ ssh −v −l john brutus Last login: Wed May
24 14:50:50 2000 from popeye.localdomain.com You have mail.
john@brutus:~/ $
```

* You can also use ssh to run a command on the remote system, rather than logging you in. For example,

```
me@popeye:~/.ssh $ ssh brutus ls bin mail misc_docs
public_html tmp me@popeye:~/.ssh $
```

Lost Your Keys? No Sweat!

SSH1 also offers you the option of a second authentication method, which might seem simpler if you're already used to rhosts−based authentication; it's a combination of that technique and RSA−based server authentication.

In this case, all you need to do is add the local system's public server key [usually in /etc/ssh_host_key.pub] to your ~/.ssh/known_hosts file on the remote system. That file would then look something like this:

```
me@brutus:~/.ssh $ cat known_hosts popeye.localdomain.com 1024
37
15919278210870935985759745878566263189422581949559583739713865003396317973
40283134655659371514861065336042326248391624445265772426517645510591464201
04922432594528243809831589978848291465587084629175415375869773793235631123
02903009784247280791473268872555661008346945504094443451694065398482898 3
me@brutus:~/.ssh $
```

As you can see, the "known_hosts" file on brutus now contains popeye's public server key.

Next, take a look at your ~/.rhosts file on brutus, and make sure that it contains an entry for popeye, which includes both the host name and the name of the user allowed to log in.

```
me@brutus:~/ $ cat .rhosts popeye.localdomain.com me
me@brutus:~/.ssh $
```

Now, log in as usual via SSH:

```
me@popeye:~/ $ ssh brutus Last login: Wed May 24 14:50:50 2000
from popeye.localdomain.com You have mail. me@brutus:~/ $
```

As a fallback, if both these authentication methods fail, SSH will ask for your password on the remote system, and authenticate your login on that basis. This is the most primitive method of SSH authentication − however, even this is more secure than plain ol' telnet, as your password is first encrypted and then transmitted to the remote system.

Test Drive Part Deux

If you're using SSH2 instead of SSH1, the basic principles of RSA−based authentication remain the same − only the filenames differ. To generate your key pair, you should run

```
$ ssh−keygen
```

or

```
$ ssh−keygen2
```

If this is the first key pair that you are creating, your public key will be ~/.ssh2/id_dsa_1024_a.pub, while your private key will be named ~/.ssh2/id_dsa_1024_a Next, you need to create an "identification" file on the local system − this file contains a list of the private keys to be used when attempting SSH logins to remote servers. At it's most basic, the file contains of a series of lines, each containing the "IdKey" parameter, followed by the name of a private key file.

```
me@popeye:~/.ssh2 $ cat identification IdKey id_dsa_1024_a
me@popeye:~/.ssh2 $
```

Next, telnet to the remote server and create a file named "authorization" in the ~/.ssh2 directory there. This file contains a list of public keys to be used when authenticating client logins.

```
me@brutus:~/.ssh2 $ cat authorization Key id_dsa_1024_a.pub
me@brutus:~/.ssh2 $
```

Obviously, you need to ensure that the keys listed in this file are actually present on the remote server, in the ~/.ssh2 directory. As described above, you can either transfer them to the remote host via FTP or email. Multiple keys are permitted − they're tried one after the other. Now, all you need to do is log in via SSH2

```
me@popeye:~/ $ ssh −v brutus warning: Development−time
debugging not compiled in. warning: To enable, configure with
−−enable−debug and recompile. warning: Development−time
debugging not compiled in. warning: To enable, configure with
−−enable−debug and recompile. debug: connecting to brutus...
debug: entering event loop debug: ssh_client_wrap: creating
transport protocol debug:
Ssh2Client/sshclient.c:1015/ssh_client_wrap: creating userauth
protocol debug: remote version: SSH−2.0−2.0.13
(non−commercial) debug:
Ssh2Client/sshclient.c:349/keycheck_key_match: Host key found
from database. debug:
Ssh2AuthPubKeyClient/authc−pubkey.c:295/ssh_client_auth_pubkey_send_signat
e: Constructing and sending signature... debug:
```

```
Ssh2AuthPubKeyClient/authc−pubkey.c:383/ssh_client_auth_pubkey_send_signat
e: ssh_client_auth_pubkey_send_signature: reading
/home/me/.ssh2/id_dsa_1024_a debug:
Ssh2/ssh2.c:472/client_authentication_notify: Authentication
successful. debug: DISPLAY not set; X11 forwarding disabled.
Last login: Wed May 24 2000 16:09:18 +0530 You have mail.
me@brutus:~$
```

and you're done! Since SSH1 and SSH2 are two completely different protocols, an SSH2 client cannot authenticate against an SSH1 server, or vice−versa. However, it is possible to configure the SSH2 daemon to intercept SSH1 connections and hand them off to the SSH1 daemon − detailed instructions for doing so, together with a list of the configuration parameters to be altered, are available in your SSH documentation.

The Tool Box

In addition to allowing you to execute secure remote logins, SSH also comes with some additional tools designed to help you transfer files between systems.

The first, and most useful, of these is scp, a file−copy utility which allows you to securely copy files from one system to another. Usage is extremely simple:

```
$ scp
```

So, if I wanted to copy my bookmarks from the remote server to my local system, I could use this command

```
me@popeye:~/ $ scp brutus:lynx_bookmarks.html .
```

The reverse is also true.

```
me@popeye:~/ $ scp myfile brutus:myfile.txt
```

You can also add the "−r" option to recursively copy directories, or the "−p" option to preserve date and time stamps, file permissions and the like

Another useful utility, which ships only with SSH2, is sftp, a secure FTP client. sftp can be used to open secure FTP connections to hosts running an SSH2 server, and use this connection to transfer files using regular FTP commands. Here's an example of how to use it:

```
me@popeye:~/ $ sftp brutus sftp> ls .. ..lynx_cookies
..bash_logout ..bash_profile tmp ..xauth ..bash_history mail
..vimrc ..ncftp public_html ..muttrc bin ..ssh2 ..plan sftp>
get .muttrc ..muttrc | 2 kB | 2.0 kB/s | TOC: 00:00:01 | 100%
sftp> quit me@popeye:~/ $
```

And the third useful utility that comes with SSH is the SSH agent, a program that allows you to store authentication keys in memory, and use them for every SSH connection you attempt without needing to tap in your passphrase each time. Think of it like a cache for all your passphrases.

The usual way to run ssh−agent is to use it to spawn a new shell, like this:

```
$ ssh−agent bash
```

At this stage, you can begin adding your keys to the agent with the ssh−add command.

```
$ ssh−add me@popeye:~/ $ ssh−add ~/.ssh/identity Need
passphrase for /home/me/.ssh/identity
```

```
(me@popeye.localdomain.com). Enter passphrase: Identity added:
/home/me/.ssh/identity (me@popeye.localdomain.com)
```

You can list all keys currently in memory with

```
me@popeye:~/ $ ssh-add -l 1024 35
120655361835592114384513149968775228655732560603276000767197229139112828 24
233536121427841873459562535811991106904404920798312829270969934297325735 54
307268310876886619548971240939982877561494458837293517831318311369679380 74
187297341161939289712191253208021221404490051713537262736384353296174240 34
57901 me@popeye.localdomain.com
```

Now that the agent is running with the key in memory, you can open as many SSH connections as you like, and you will not be prompted for a passphrase each time.

You can add as many keys to the agent as you like. Deleting a key can be accomplished with the "-d" parameter, while the "-D" parameter will delete all keys in memory.

Any Port In A Storm

Finally, SSH also allows you to forward TCP/IP connections from one port to another on different hosts − this allows you to use the secure channel for communication that usually travels unencrypted, such as mail passwords. SSH allows you to forward local ports to remote hosts, and vice versa.

In the example below, an SSH connection has been established to forward all connections to port 10 on popeye to port 110 on brutus.

```
root@popeye:~/ $ ssh −L 10:brutus:110 −l me brutus Last login:
Fri May 26 16:54:57 2000 from popeye.localdomain.com No mail.
me@brutus:~/ $
```

Now, if I opened a telnet connection to port 10 on popeye in another window

```
me@popeye:~/ $ telnet popeye 10 Connected to brutus. Escape
character is '^]'. +OK brutus POP3 server ready
```

my connection would be automatically forwarded to port 110 − the POP3 mail server − on brutus, for so long as the secure channel remained open. Other services could be secured in a similar manner.

Do note that you need to have appropriate privileges to forward ports in this manner.

Of Clients And Servers...

If you're on a *NIX system, your SSH distribution contains everything you need to build the SSH daemon [sshd] and the client [ssh]. However, if you're running a Windows−based PC, and need to connect to an SSH server, things get a little murkier.

If you're looking for something that's free − but not necessarily easy or even good−looking − take a look at FiSSH, available from http://doghouse.mit.edu/FiSSH/, or puTTY at http://www.chiark.greenend.org.uk/~sgtatham/putty/.

If, however, you don't mind putting some money down for your software, I'd recommend that you take a look at Van Dyke Technologies' wonderful SecureCRT package − it's a powerful terminal emulator that includes support for both SSH1 and SSH2 protocols, and it comes with a manual for the novice user [unlike the two packages listed above]. It also includes support for all the authentication methods that SSH supports − host−based authentication, RSA authentication, and password authentication − whereas my initial experiments with puTTY suggest that that package only includes support for simple password authentication.

You can download a trial version of SecureCRT from http://www.vandyke.com

Another good alternative is the SSH client marketed by DataFellows, makers of the popular F−Secure anti−virus package − you'll find it at http://www.datafellows.com/products/ssh. DataFellows also offers an SSH server, and a choice of two clients, depending on whether you're connecting to an SSH1 or SSH2 server.

Well, that's about it − hopefully, you now have a better idea of what SSH is all about, and why it's a good thing to add to your network. Should you need more information, I'd recommend the official Web site at http://www.ssh.org/, or the SSH FAQ at http://www.tigerlair.com/ssh/faq/ssh−faq.html

Happy SSHing!