SSH Techniques

Welcome back!

Last month we went over the motivation behind using ssh to encrypt communications between two computers, went through the process of compiling and installing the ssh client and server programs, made a first connection using those programs, and made a lovely garden salad. If you missed that article (or lost the recipe for the dressing ;−), please read that one first.

FIXME − relative link for first article.

We've already seen one of the primary uses of ssh: it allows you to open up a terminal session to a remote system. By using "ssh" instead of telnet or rsh, you get the same ability to type commands on remote systems, but your sessions are encrypted to protect them from prying eyes.

What else does ssh offer? Let's start with the security features in the program and move on to some of the practical uses of the tool.

Host Authentication

Let's take a look at that first connection I made last month:

```
[wstearns@sparrow wstearns]$ ssh goober
Host key not found from the list of known hosts.

Are you sure you want to continue connecting (yes/no)?
yes
Host 'goober' added to the list of known hosts.
wstearns's password: password_entered
You have mail.
[wstearns@goober wstearns]$ ls
```

Each machine running the ssh server has an identifier called a *host key*. The server, Goober, sent this back to the client machine. Since the client hadn't seen this host key before, it let me know that and asked if I wanted to continue. Since I said yes, it stored this host key in ~/.ssh/known_hosts . When I connect again in the future, I'll see:

```
[wstearns@sparrow wstearns]$ ssh goober
wstearns's password: password_entered
You have new mail.
bash#
```

As part of the login process, the ssh program compared the key it again received from goober to the key it got the first time. Since the keys are identical, it continues on with the login process.

Let's say someone removed the real goober and replaced it with their own computer. Since the host keys are randomly generated, the new computer would have a different host key. When I attempt to connect to goober but reach the replacement, ssh will tell me loudly:

```
[wstearns@sparrow wstearns]$ ssh goober
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@       WARNING: 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 the host key has just been changed.
Please contact your system administrator.
Add correct host key in /home/wstearns/.ssh/known_hosts to get rid of this message.
Agent forwarding is disabled to avoid attacks by corrupted servers.
X11 forwarding is disabled to avoid attacks by corrupted servers.

Are you sure you want to continue connecting (yes/no)?
```

As loudly as it can without a sound card, at least. *smile*

The only time you should ignore this message is if you're absolutely certain the key has been replaced by the good guys.

If you have any doubt about the identity of a key, simply compare the line for that host in ~/.ssh/known_hosts with the one on that host in /etc/ssh/ssh_host_key.pub .

Use of ssh keys

When I connected to the host called goober last month, it asked me for my password, much like a telnet program would have. You always have the option of continuing to use the same passwords you've always used with telnet.

In much the same way that ssh uses a unique host key to identify hosts, ssh allows you to have your own key that you can use to identify *yourself*. This key can be used to identify you to multiple hosts; we'll see how in a minute.

First, log in as yourself on your main work machine − the one you'll be ssh'ing *from*. If you don't already have a user key, run the **ssh−keygen** utility as follows:

```
[wstearns@sparrow wstearns]$ ssh−keygen −b 1024
Initializing random number generator...
Generating p:  .......++ (distance 150)
Generating q:  ..........++ (distance 126)
Computing the keys...
Testing the keys...
Key generation complete.
Enter file in which to save the key (/home/wstearns/.ssh/identity):
Enter passphrase:
Enter the same passphrase again:
Your identification has been saved in /home/wstearns/.ssh/identity.
Your public key is:
1024 41 135...a lot of digits deleted...8412590733 wstearns@sparrow
Your public key has been saved in /home/wstearns/.ssh/identity.pub
[wstearns@sparrow wstearns]$
```

A side note: The passphrase is essentially a long password. See **man ssh−keygen** for more info on how to pick one. It should definitely be longer than a password because this protects access to multiple machines. **Do not** simply press enter when asked for your passphrase − this is a tremendous security risk.

When the program is done, it will leave two files in the .ssh directory off of your home directory: identity and identity.pub . The identity file is the one you must keep private; nobody else should see this file. identity.pub, on the other hand, is what goes to the hosts to which you'll connect.

Identity.pub has a single readable line. Append it to ~/.ssh/authorized_keys . Since ssh is so concerned about someone getting access they shouldn't, I usually do the following on the remote hosts as well:

```
chmod 700 ~
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

If you used the root account to get your keys on the system, make sure that your home directory, the .ssh directory, and ~/.ssh/authorized_keys are all owned by you. Ssh may refuse access if these permissions and ownerships are set incorrectly.

Back on the client computer, try connecting to the server:

```
[wstearns@sparrow wstearns]$ ssh goober
Enter passphrase for RSA key 'wstearns@sparrow':
Last login: Wed Apr 26 23:48:43 2000 from sparrow
No mail.
[wstearns@goober wstearns]$
```

Note: I was no longer asked for my usual login password over on goober − I was asked for, and typed in, my *passphrase*: the same one I entered when I created the user key.

By copying identity.pub over to ~/.ssh/authorized_keys on all the machines with which you need to connect, you can get access to them all with a single passphrase, rather than having to assign different passwords to them or risk having a shared password broken on one machine, giving an attacker a way into the others. Because ssh uses a public and private key approach, even if an attacker got access to your public key, stored in authorized_keys, on one machine, they couldn't use it to get access to another machine that also had your public key. They also couldn't use it to get access back to the box where you have your private key.

Instead of the easily sniffed old password scheme, you now have a pair of objects that do a good job of authenticating you to remote servers: your private key and your passphrase. Neither will get you access by itself; if someone is going to use ssh to impersonate you, they would need both. It's still not perfect, but it's certainly better than passwords.

ssh−agent

Typing that long passphrase can get annoying, especially when I need to log into a bunch of machines quickly. Couldn't I type my passphrase once and have the machine remember it? Of course; if it weren't possible, I wouldn't have asked the question. :−)

The ssh package includes a tool called **ssh−agent**. I run this tool once before I start X windows. Here's how I start up X on one of my text consoles. There are probably more elegant ways to do this: if you have a suggestion, send it in for possible inclusion as a tip of the week.

```
[wstearns@sparrow wstearns]$ touch ~/agent
[wstearns@sparrow wstearns]$ chmod 600 ~/agent
[wstearns@sparrow wstearns]$ ssh−agent >~/agent
[wstearns@sparrow wstearns]$ . agent
[wstearns@sparrow wstearns]$ ssh−add
Need passphrase for /home/wstearns/.ssh/identity (wstearns@sparrow).
Enter passphrase:
Identity added: /home/wstearns/.ssh/identity (wstearns@sparrow)
[wstearns@sparrow wstearns]$ startx −− −bpp 16 &sleep 20 ; exit
```

When X starts up, all of the terminals I start under it know how to find the ssh−agent running in the background. Now I can ssh to remote systems without even having to type my passphrase again:

```
[wstearns@sparrow wstearns]$ ssh goober
Last login: Wed Apr 26 23:49:16 2000 from sparrow
No mail.
[wstearns@goober wstearns]$
```

I only need to enter my passphrase once, when I boot the machine. My private key still stays secure, though; if it were ever taken from this system, it would be worthless without the passphrase.

There's one more really neat trick that becomes available once you have ssh−agent working. Lets say I've ssh'ed over to goober. I want to ssh from goober over to boomer, which also has my public key in ~/.ssh/authorized_hosts . Because I'm running ssh−agent, I can simply type:

```
[wstearns@goober wstearns]$ ssh boomer
Last login: Wed Apr 26 23:49:16 2000 from sparrow
No mail.
[wstearns@boomer wstearns]$
```

Even though my private key is still back on sparrow, ssh−agent handles making the new connection from goober to boomer for me.

Copying files with scp

We've only used ssh for terminal sessions so far. Let's use the encryption capabilities of ssh to send a file between systems. It's pretty straightforward:

```
[wstearns@goober wstearns]$scp −p /home/wstearns/agenda goober:/tmp
[wstearns@goober wstearns]$
```

This copies agenda to goober's tmp directory. The "−p" preserves the times and modes of the copied file. Because ssh actually carries the contents of the file over to the remote system, the file is encrypted just like the terminal sessions we've seen.

Notice that scp never asked for either a password or passphrase − ssh−agent provides the passphrase for me. This is another of the neat techniques ssh offers.

Protecting pop3, imap or smtp session with tunneling

Opening encrypted terminal sessions and copying files between systems is useful, but Tatu Ylonen and the ssh team deserve a round of applause for making the encryption available to other programs.

Let's say goober is also my mail server. While I have a terminal session open to it, my mail software goes out to pull down my mail from goober's pop server. Here's how the connections look; remember that the characters on the lower connection are encrypted.

```
Sparrow               Goober

mail −−−−−−−−−−−−−−−−> port 110, ipop3d

ssh −−−−−−−−−−−−−−−−−> port 22, sshd
```

As we've seen, all the characters and screen displays that go over the ssh connection are encrypted − nobody can read them in transit. The pop connection, however, carries your username and password as clearly visible text. It's hardly worth going to all of the trouble of encrypting your terminal sessions if someone can simply watch your pop mail connection to get your username and password.

What if we could somehow convince the ssh client and server to carry your pop traffic as well? Could we somehow stuff the mail requests and messages into the encrypted connection?

```
Sparrow                Goober

mail −+              +−> port 110, ipop3d
      |              |
ssh −−+==============+−> port 22, sshd
```

Ssh handles this by *forwarding* connections. When I start up the ssh connection, I tell the ssh client to listen on some port on sparrow, say 1234. I tell ssh to encrypt and carry any traffic over to goober and deposit it into goober's port 110, bringing the responses back to sparrow's port 1234. It means that the mail program sees local port 1234 as it's pop mail server. Ssh handles getting the packets back and forth to the real mail server on goober.

To actually make this happen, we only have to make two minor changes. First, we start up the ssh connection and tell it to listen on port 1234:

```
[wstearns@sparrow wstearns]$ ssh −L 1234:goober:110 goober
Last login: Thu Apr 26 22:49:16 2000 from sparrow
No mail.
[wstearns@goober wstearns]$
```

We leave this connection open while we do the following steps. Before we go on to change the mail setup, let's check that port 1234 is actually getting carried over to goober:

```
[wstearns@sparrow wstearns]$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
+OK POP3 goober v12.5 server ready
QUIT
+OK Sayonara
Connection closed by foreign host.
[wstearns@sparrow wstearns]$
```

Note that even though I'm connecting to sparrow's port 1234, I'm seeing what I'd normally see if I connected to goober's port 110 − success!

The only step left is to tell your mail program to get it's mail from local port 1234. Although each program is different, you're looking for a way to specify the name and port number used for the pop mail server. Tell your mail client to use **localhost** for the mail server, and use port **1234** as the port to which to connect. That's it!

Once you've pulled down your mail, you can close down the terminal session to goober. Remember to open up the session again before pulling down your mail next time.

Running X apps

OK, I've got one more trick to show you this month, and this one's really cool. Let's say that I want to run an X windows app over on goober, but I really don't want to go over to goober's console. Especially since goober is at my main office, 6 hours away. I'd like to have the application run on goober's processor, using goober's memory, but have the output show up on my screen and use my own keyboard and mouse to control it. It's really easy.

```
[wstearns@sparrow wstearns]$ ssh goober
[wstearns@goober wstearns]$ xload &[wstearns@goober wstearns]$
```

Within a second or two, the xload application starts up − on my screen! I'm actually seeing an application that's running on goober, but it's display is being carried to my machine and my keystrokes and mouse movements are being carried back to goober. The display, keystrokes, and mouse movements are all encrypted, so nobody can see what I see or type.

Where do we go from here?

The man pages for ssh, ssh−agent, and scp can tell you more about the techniques we've seen here. There's an ssh mailing list for help; send a message to majordomo@clinet.fi with

```
subscribe ssh
```

in the body. Finally, take a look at these links for more info:

- The home site for SSH Communications Security, the company started by SSH's author to promote ssh.
- The ssh faq.
- The OpenSSH project, an implementation of the last freely available version of ssh with updates.
- PuTTY: A free Win32 Telnet/SSH Client.
- A Java SSH client.
- The zedz ftp site which carries much of the encryption software for linux.

William is an Open−Source developer, enthusiast, writer, and advocate from New Hampshire, USA. His day job at SANS pays him to work on network security and Linux projects.

Last updated 12/18/2003.