SSH: The Secure Shell Server

Having the OpenSSH client tools installed on your machine is useful for connecting to other hosts, but without the OpenSSH server installed, you can't use SSH to connect to your own machine. In this lesson we'll cover the installation of the OpenSSH server and some related concepts.

The Secure Shell Server

The OpenSSH server runs as a process in the background on a machine and it listens for incoming connection requests on a specified *TCP port*. You may recall from the networking concepts lesson that TCP, the **T**ransmission **C**ontrol**P**rotocol, provides a method for data packets to be directed to a numbered port. This functionality allows a *service*, such as the OpenSSH server, to listen for incoming connections without disrupting other services that are listening for connections. The standard port that ssh servers listen on is port 22. If you try to initiate an ssh connection from a client and do not specify a target port, the client will assume that port 22 is to be used. An extensive list of well known ports is available in the ***/etc/services*** file. Let's go ahead and use **yum** to install the **openssh-server** package:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum install openssh-server
[sudo] password for dbassett1:
Loaded plugins: fastestmirror
Determining fastest mirrors
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package openssh-server.x86_64 0:5.3p1-70.el6_2.2 will be
installed
--> Processing Dependency: libwrap.so.0()(64bit) for package:
openssh-server-5.3p1-70.el6_2.2.x86_64
--> Running transaction check
---> Package tcp_wrappers-libs.x86_64 0:7.6-57.el6 will be
installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================
================================================================
==================
 Package                                    Arch
Version                                    Repository
Size
```

```
================================================================================
================================================================================
==================
Installing:
 openssh-server                             x86_64
5.3p1-70.el6_2.2                           updates
297 k
Installing for dependencies:
 tcp_wrappers-libs                          x86_64
7.6-57.el6                                 base
62 k

Transaction Summary
================================================================================
================================================================================
==================
Install       2 Package(s)

Total download size: 359 k
Installed size: 780 k
Is this ok [y/N]: y
Downloading Packages:
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------
Total
3.6 MB/s | 359 kB     00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : tcp_wrappers-libs-7.6-57.el6.x86_64
1/2
  Installing : openssh-server-5.3p1-70.el6_2.2.x86_64
2/2

Installed:
  openssh-server.x86_64 0:5.3p1-70.el6_2.2

Dependency Installed:
  tcp_wrappers-libs.x86_64 0:7.6-57.el6

Complete!
```

The ssh server is now installed. We can test to make sure it's working using ssh to connect to *localhost*. Localhost is a special hostname reserved for a special IP address, 127.0.0.1. This address is assigned to a *loopback* interface. On your machine this

interface is called "lo." A loopback interface takes any packet that it receives and spits it right back out. This means that any packets sent to the address 127.0.0.1 from your machine will also be received by your machine. So using your ssh client to connect to port 22 at the address 127.0.0.1 will connect to the local machine on port 22. Let's go ahead and connect to the ssh server that you just installed:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ssh localhost
ssh: connect to host localhost port 22: Connection refused
```

That didn't work out so well. To see why, use ps to look for the ssh server's process, which should be called "sshd." It's not even running—that explains why we couldn't connect to it! If a service you expect to be able to connect to is not responding, check to find out whether it's actually running on the host. Unfortunately, when an ssh server is not running, it probably means you'll have to go to work directly on the machine's console. Fortunately, you are already logged into your machine's console, so no ssh server is necessary to connect to it. We can start the sshd process by using the **service** command. This handy command is used to control the services that will run on your machine. The general format of the command is **service *service_name action***, where *action* can be **start**, **restart**, **stop**, or **reload**. Let's use the **start** action to get our ssh server running:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo service sshd start

Starting sshd: [  OK  ]
```

As long as you see an "OK" in the brackets, your ssh server should be running. Now let's try to connect to localhost again:

Connecting to localhost with ssh:

```
[username@username-m0 ~]$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be
established.
RSA key fingerprint is
44:51:fb:df:57:db:a3:76:df:24:bc:e0:e6:60:81:ab.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of
known hosts.
username@localhost's password:
Last login: Wed Mar 14 11:01:22 2012
[username@username-m0 ~]$
```

Now your ssh server is working! But what's the deal with that RSA key fingerprint and known hosts stuff? This is another security feature that's built into ssh. Each ssh server will generate a public/private key pair for itself. The first time you connect to the server, you don't have its public key, so your ssh client asks you to confirm that you are

connecting to the server that you think you are. When you confirm that you are, you download the ssh server's public key and store it in the **known_hosts** file in the **~/.ssh** directory. When you connect to this server in the future, your ssh client uses this stored public key to verify the identity of the ssh server using the same method that we discussed earlier for using public/private key pairs to authenticate users. This security feature prevents something called a *man in the middle attack*. In this kind of attack, a hacker could redirect traffic intended for a legitimate ssh server to his own ssh server, where he can then trick users into providing their passwords. He can then collect these passwords and use them to get into the legitimate server. When you manage an ssh server, you have to make sure that the ssh server's private key is safe.

While you're logged into your own machine via ssh, issue the command **w**. The w command gives you information about who is logged in, how they logged in, and statistics about what they have done since they logged in:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ w
 11:10:46 up 5 days, 23:54,  2 users,  load average: 0.00, 0.00,
0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU
WHAT
username tty0     -                11:01    0.00s  3.43s  0.29s
ssh localhost
username pts/0    localhost        11:01    0.00s  0.38s  0.00s
w
```

If you are logged into only one console on your machine, your output will look like that. In our example, we see that someone is logged in on the first console (tty0) and that person is currently running the command **ssh localhost**. On the next line, we see that same user, this time logged into pts/0 and running the **w** command. A single pts is called a *pty* or *pseudo-terminal*. A pseudo-terminal is given out to users for things like remote logins, rather than a tty, which is given out for logins on the local console. On the second user line, we see that w tells us where the user is logged in from, in this case, localhost. Finally, w gives us some statistics about when the user logged in (LOGIN@), how long it's been since the user did anything (IDLE), and the amount of CPU that the user is using. JCPU describes the total amount of CPU being consumed by jobs in that particular tty or pty, and PCPU tells us how much CPU is being used by the job specified in the WHAT column.

Now, add a new user called "testuser" to your system. If you need a reminder on how to do that, you can find instructions here. When you have added testuser, open another console on your Linux Learning Environment machine and log in as yourself. Now we'l login as another user using ssh. Connect to your local machine as testuser using **ssh testuser@localhost**. Switch back to the other console where you're logged in and issue the **w** command again. You'll see something like this:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ w
 14:26:53 up 6 days,  3:10,  3 users,  load average: 0.00, 0.00,
0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU
WHAT
username tty0     -                11:01    0.00s  4.21s  0.00s
w
username tty1     -                11:13    8.00s  0.00s  0.00s
ssh testuser@localhost
testuser pts/0    localhost        14:26    4.00s  0.32s  0.32s
-bash
```

Source: http://courses.oreillyschool.com/sysadmin2/ssh_server.html