By icarus

Table of Contents

Hooking Up

Most relatively-experienced Internet users are already familiar with the benefits of SSH, secure shell technology that makes it possible to securely connect to other hosts over TCP/IP. Unlike regular telnet, which is unencrypted and offers hackers numerous opportunities to break into your connection and siphon off sensitive information, SSH is a 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.

Now, most of the time, SSH is used as a replacement for regular telnet, allowing users to securely log in to other hosts on a network. However, in addition to this, SSH also comes with one very interesting - yet not very well-known - feature: the ability to create secure "tunnels" between two hosts for encrypted data communication between other ports as well. This means that, for example, you could use SSH to create a secure tunnel between your local host and your mail server so that your mail password is never transmitted in cleartext across the network (as is usually the case) every time you check your mail. Or you could use SSH to build an encrypted bridge between two or more firewall-protected hosts, so that network sniffers never get to intercept the data flowing back and forth between the two.

By allowing such encrypted connections between two (or more) hosts, SSH provides harassed network administrators with a powerful weapon in their daily balancing act of making their network more secure while simultaneously giving users as much flexibility as possible. SSH tunneling and port forwarding allow any user on a TCP-based network to communicate and transact with other hosts on the network in a reliable and secure fashion, with minimal risk of data interception or corruption.

Sounds interesting? Keep reading.

Kicking The Tyres

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:

```
  [me@olympus] $ telnet localhost 22Connected to olympus.Escape character is '^]'.SSH-1.99-OpenSSH_3
```

Nothing? Well, you could politely ask your ISP/Web hosting service/friendly neighbourhood geek to install it for you - or, if you have super-user access to the system, and are comfortable with installing new software on your system, you could download and install it yourself.
This article uses OpenSSH, an open-source alternative to commercial SSH that does away with many of the licensing restrictions of SSH1 and SSH2. Drop by the official Web site at http://www.openssh.org/and get yourself the latest stable release of the software (this tutorial uses OpenSSH 3.5). Note that you will also need a copy of the zlib library, available from http://www.gzip.org/zlib/ (this tutorial uses zlib 1.1.4) and the OpenSSL library, available from http://www.openssl.org/(this tutorial uses OpenSSL 0.9.7).
Once you’ve downloaded the source code archive to your Linux box (mine is named "olympus"), log in as "root".

```
  [me@olympus] $ su -Password: ****
```

You’ll first need to compile and install zlib. Extract the source to a temporary directory.

```
  [root@olympus] $ cd /tmp[root@olympus] $ tar -xzvf zlib-1.1.4.tar.gz
```

Next, configure the package using the provided "configure" script,

```
  [root@olympus] $ cd /tmp/zlib-1.1.4[root@olympus] $ ./configure
```

and compile and install it.

```
  [root@olympus] $ make[root@olympus] $ make install
```

Unless you specified a different path to the "configure" script, zlib will have been installed to the directory "/usr/local/lib".
Next up, OpenSSL. Extract the source to a temporary directory,

```
  [root@olympus] $ cd /tmp[root@olympus] $ tar -xzvf openssl-0.9.7a.tar.gz
```

configure it,

```
  [root@olympus] $ cd /tmp/openssl-0.9.7a[root@olympus] $./config
```

and compile and install it.

```
  [root@olympus] $ make[root@olympus] $ make test[root@olympus] $ make install
```

The compilation process here is fairly involved and may take a few minutes - get yourself a cup of coffee while you’re waiting for it to happen. By the time you get back, OpenSSL should be installed to the directory "/usr/local/ssl".
Finally, it’s time to install the OpenSSH package itself. Again, extract the source to a temporary directory,

```
   [root@olympus] $ cd /tmp[root@olympus] $ tar -xzvf openssh-3.5p1.tar.gz
```

and configure the software via the provided "configure" script. Remember to tell "configure" where it can find the libraries you just installed as well.

```
   [root@olympus] $ cd /tmp/openssh-3.5p1[root@olympus] $ ./configure --with-ssl-dir=/usr/local/ssl/
```

Once the software has been configured, you can compile and install it.

```
   [root@olympus] $ make[root@olympus] $ make install
```

In this case, since I specified an installation path to the "configure" script, OpenSSH would have been installed to the "/usr/local/ssh" directory.
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 "/usr/local/ssh/etc/ssh_host_key" and "/usr/local/ssh/etc/ssh_host_key.pub" respectively.
Once the software has been installed, you need to start up the "sshd" daemon.

```
   [root@olympus] $ /usr/local/ssh/sbin/sshdPrivilege separation user "sshd" does not exist
```

Oops! Something obviously went wrong somewhere...
Actually, the reason for the error above is fairly simple. As the OpenSSH manual puts it, "privilege separation is a method in OpenSSH by which operations that require root privilege are performed by a separate privileged monitor process [...] the purpose is to prevent privilege escalation by containing corruption to an unprivileged process."
You can correct this error by creating a user and group for the "sshd" daemon to run as, by executing the following commands:

```
   [root@olympus] $ mkdir /var/empty[root@olympus] $ chown root:sys /var/empty[root@olympus] $ chmod
```

Now, try restarting the daemon,

```
   [root@olympus] $ /usr/local/ssh/bin/sshd
```

and all should be well.
You can verify that the daemon is, in fact, alive via a telnet to port 22:

```
  [me@olympus] $ telnet localhost 22Trying 127.0.0.1...Connected to localhost.Escape character is '^
```

Remember that you need an OpenSSH daemon 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.

Test Drive

The procedure for using SSH-based private/public key 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 "olympus".
The first thing you need to do is generate a key pair for yourself. Log in to "olympus", and run this command from your shell:

```
  [me@olympus] $ /usr/local/bin/ssh-keygen -t rsa
```

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

```
  Generating public/private rsa key pair.Enter file in which to save the key (/home/me/.ssh/id_rsa):
```

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

```
  [me@olympus] $ /usr/local/bin/ssh-keygen -p
```

Your public key will be saved to "~/.ssh/id_rsa.pub" while your private key will be located in "~/.ssh/id_rsa".
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 "authorized_keys" and insert the contents of your "~/.ssh/id_rsa.pub" on "olympus" 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. Ensure that the file has 0600 permissions, while the "~/.ssh" directory has 0700 permissions.
If you don’t have telnet access to the remote host, you could also upload your "id_rsa.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".

Et Tu, Brute?

Now that you have your two hosts talking to each other over an encrypted connection, the next step is to set up a secure channel between them for non-telnet type activities - for example, setting up a secure tunnel for mail transfer.
In order to do this, you need to use the port forwarding support built into OpenSSH. Port forwarding essentially means that connections made to a port on one host are automatically and transparently forwarded to another port on another host. Since SSH is taking care of the forwarding for you, the connection also gets encrypted - a nice (and very useful) bonus.
The best way to demonstrate how this works is with an example. Let’s suppose that I would like to read my mail on "brutus", the network’s POP3 mail server, from my personal Linux box, "olympus". Normally, I would configure my mail client to connect to port 110 on "brutus" to retrieve my mail - this would involve sending my username and mail password to "brutus" in cleartext across the network, a technique that we have determined to be unsuitable for purposes of this tutorial.
With port forwarding, I have an alternative. I can have SSH forward a port (say, 9000) on my local host, "olympus", to port 110 on "brutus", and protect all traffic passing between the two (including my mail password) by creating a secure tunnel between the two hosts and ports. Once the tunnel is established, my mail client would no longer connect to port 110 on "brutus" to get my mail; rather, I would configure it to use port 9000 on my local host, "olympus", instead, and SSH would take care of automatically passing the data to post 110 on "brutus" via the tunnel.
Here’s how to go about doing this:

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus
```

Translated into English, the above command merely says "listen for connections to port 9000 on this local host, and use the remote host named brutus to forward all those connections to port 110 on the host named localhost".
SSH will now connect and log in to "brutus", and simultaneously begin forwarding port 9000.

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus Lastlogin: Fri Mar 28 10:18:59
```

You can now verify that the port is indeed being forwarded by switching to another terminal on "olympus" and opening up a telnet connection to port 9000.

```
[me@olympus] $ telnet localhost 9000Trying 127.0.0.1...Connected to localhostEscape character is '
```

As you can see, connections to port 9000 on your local host are being transmitted to port 110 (the POP3 server port) on the host named "brutus". All data transmitted in this session will be encrypted and decrypted by the SSH daemons running at the two ends of the connection.
Note that this connection remains available for the duration of your SSH session - the moment you log out of "brutus", the port forwarding will also stop.
If this is not what you want, you can add the "-N" command-line argument to tell SSH *not* to execute any command on the remote side once the connection has been established. Background the task, and you’ll have port forwarding without a login on the remote host!

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus -N &
```

Note that if you do this, you will need to manually kill the process when you want to stop forwarding the specified ports.
You can also forward more than one port at a time by specifying them all on the same command line:

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 -L9001:localhost:25 brutus
```

No Forwarding Address

Now that you have your two hosts talking to each other over an encrypted connection, the next step is to set up a secure channel between them for non-telnet type activities - for example, setting up a secure tunnel for mail transfer.
In order to do this, you need to use the port forwarding support built into OpenSSH. Port forwarding essentially means that connections made to a port on one host are automatically and transparently forwarded to another port on another host. Since SSH is taking care of the forwarding for you, the connection also gets encrypted - a nice (and very useful) bonus.
The best way to demonstrate how this works is with an example. Let’s suppose that I would like to read my mail on "brutus", the network’s POP3 mail server, from my personal Linux box, "olympus". Normally, I would configure my mail client to connect to port 110 on "brutus" to retrieve my mail - this would involve sending my username and mail password to "brutus" in cleartext across the network, a technique that we have determined to be unsuitable for purposes of this tutorial.
With port forwarding, I have an alternative. I can have SSH forward a port (say, 9000) on my local host, "olympus", to port 110 on "brutus", and protect all traffic passing between the two (including my mail password) by creating a secure tunnel between the two hosts and ports. Once the tunnel is established, my mail client would no longer connect to port 110 on "brutus" to get my mail; rather, I would configure it to use port 9000 on my local host, "olympus", instead, and SSH would take care of automatically passing the data to post 110 on "brutus" via the tunnel.
Here’s how to go about doing this:

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus
```

Translated into English, the above command merely says "listen for connections to port 9000 on this local host, and use the remote host named brutus to forward all those connections to port 110 on the host named localhost".
SSH will now connect and log in to "brutus", and simultaneously begin forwarding port 9000.

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus Lastlogin: Fri Mar 28 10:18:59
```

You can now verify that the port is indeed being forwarded by switching to another terminal on "olympus" and opening up a telnet connection to port 9000.

```
[me@olympus] $ telnet localhost 9000Trying 127.0.0.1...Connected to localhostEscape character is '
```

As you can see, connections to port 9000 on your local host are being transmitted to port 110 (the POP3 server port) on the host named "brutus". All data transmitted in this session will be encrypted and decrypted by the SSH daemons running at the two ends of the connection.
Note that this connection remains available for the duration of your SSH session - the moment you log out of "brutus", the port forwarding will also stop.
If this is not what you want, you can add the "-N" command-line argument to tell SSH *not* to execute any command on the remote side once the connection has been established. Background the task, and you’ll have port forwarding without a login on the remote host!

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus -N &
```

Note that if you do this, you will need to manually kill the process when you want to stop forwarding the specified ports.
You can also forward more than one port at a time by specifying them all on the same command line:

```
[me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 -L9001:localhost:25 brutus
```

Any Port In A Storm

One of the things that frequently goes unmentioned when discussing SSH port forwarding - perhaps because it's not so obvious at first glance - it that you can use the remote host to forward connections to *any* other named host (not just to itself) on the network.
If you look at the example on the previous page again,

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:localhost:110 brutus
```

you will notice that I am using the remote host "brutus" to open connections to port 9000 on a host named "localhost". Since "brutus" automatically resolves the host name "localhost" to itself, I could also write the command above as

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -L 9000:brutus:110 brutus
```

and obtain an equivalent result.
This opens up an interesting possibility - using an SSH connection between two hosts to create a connection to a third host. Can it be done? Yes indeedy - take a look:

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -L 9001:medusa:25 brutus
```

In this case, all connections made to port 9001 on my local machine "olympus" will automatically get forwarded to port 25 (the SMTP port) on the new host "medusa" via the host "brutus". Try it and see for yourself:

```
  [me@olympus] $ telnet localhost 9001Trying 127.0.0.1...Connected to localhostEscape character is '
```

Neat, huh?

Remote Control

SSH also allows you to do the reverse - forward connections made to a port on the remote host to the local host, or some other host.
In order to better understand this, let’s look at another example. Suppose I wanted all connections made to port 9000 on the remote host "brutus" to be forwarded to port 80 (the Web server port) on my local machine "olympus". Here’s how:

```
  [me@olympus] $ /usr/local/ssh/bin/ssh -R 9000:olympus:80 brutus
```

Once SSH connects and logs me in, it will automatically set up a listener on port 9000 on the host named "brutus". All connections to this port will then get forwarded to port 80 on the host named "olympus" (my local host).
You can verify this by logging in to "brutus" and attempting a telnet connection to port 9000:

```
  [me@brutus] $ telnet localhost 9000Trying 127.0.0.1...Connected to localhostEscape character is '^
```

In And Out

Port forwarding comes in handy when dealing with firewalls as well. By creating a secure channel between two hosts, each on different sides of a firewall, SSH makes it possible for any computer outside the firewall to connect to any computer inside it.

The most common scenario here is when roaming users need to connect to their intranet Web server, which is protected behind a corporate firewall, from an external location. A direct connection in such cases is not possible - the firewall will reject all requests coming from outside the local network. SSH can play an important role here.

Let’s consider an example, to better understand how this could work. Let’s assume that the corporate Web server is running on a machine named "www", inside the firewall, and that there exists a machine named "medusa" on the same physical network, but outside the firewall. We can also assume that both "medusa" and "www" are running OpenSSH.

Next, let’s assume that there exists a roaming host "remote", which is connected to the Internet on a dial-up connection and is outside the firewall...but desperately needs to get inside so that its owner can access the corporate intranet.

With port forwarding, accomplishing this is fairly simple. First, an SSH connection should be established between "medusa" and "www", which are on opposite sides of the firewall, and then an arbitrary port (say, 8080) on "medusa" should be forwarded to port 80 (the Web server port) on "www".

```
 [me@www] $ /usr/local/ssh/bin/ssh -R 8080:www:80 medusa
```

Since "medusa" is outside the firewall, it can accept connections from the external host "remote" on port 8080. These connections are then forwarded through the secure tunnel to port 80 on "www", and the resulting Web pages transmitted back to "remote" via the reverse route. The end result: the roaming user can gain access to a server inside the corporate firewall without compromising the security of the system.

Log Out

And that’s about it for the moment. In this article, I introduced you to OpenSSH, a free open-source implementation of the SSH protocol. After a quick crash course in installing and configuring SSH, I took you through the process of creating a key pair, and using it to securely connect to other hosts. With the basics out of the way, I then moved to the main focus of this article - using SSH to create secure tunnels between ports on different hosts, in an effort to add greater security to the data packets traveling across a network. I walked you through a number of possible scenarios for this capability, including securing your connection to your incoming mail server via local port forwarding, and creating secure channels for hosts outside your firewall to communicate with hosts inside it via remote port forwarding.
In case you’d like to read more about the capabilities and technologies discussed in this article, you should take a look at the following Web sites:
The official OpenSSH Web site, at http://www.openssh.org/
The OpenSSH FAQ, at http://www.openssh.org/faq.html
The OpenSSH mailing list, at http://www.openssh.org/list.html
The Shell Game, at http://www.devshed.com/Server_Side/Administration/SSH
Getting Started With SSH, at http://kimmo.suominen.com/ssh/
SSH tutorials for Linux, at http://www.suso.org/linux/tutorials/ssh.phtml and http://www.linux.ie/articles/tutorials/ssh.php
I hope you enjoyed this article, and that you found the information in it useful in securing your network. Till next time...stay healthy!
Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!