

# SSH Port Forwarding

Contributed by Linux  
Tuesday, 08 August 2006

Introduction SSH is typically used for logging into remote servers so you have shell access to do maintenance, read your email, restart services, or whatever administration you require. SSH also offers some other native services, such as file copy (using scp and sftp) and remote command execution (using ssh with a command on the command line after the hostname).

Introduction SSH is typically used for logging into remote servers so you have shell access to do maintenance, read your email, restart services, or whatever administration you require. SSH also offers some other native services, such as file copy (using scp and sftp) and remote command execution (using ssh with a command on the command line after the hostname). Whenever we SSH from one machine to another, we establish a secure encrypted session. This first article in this SSH series[1] looked at properly verifying a server's host key, so that we can be sure that no attacker is able to perform a man-in-the-middle attack and gain access to read or manipulate what we do in that session. Other articles in this series looked at removing the need for static passwords using SSH user identities[2], and then using ssh-agent[3] to automate the task of typing passphrases. SSH also has a wonderful feature called SSH Port Forwarding, sometimes called SSH Tunneling, which allows you to establish a secure SSH session and then tunnel arbitrary TCP connections through it. Tunnels can be created at any time, with almost no effort and no programming, which makes them very appealing. In this article we look at SSH Port Forwarding in detail, as it is a very useful but often misunderstood technology. SSH Port Forwarding can be used for secure communications in a myriad of different ways. Let's start with an example. LocalForward Example Say you have a mail client on your desktop, and currently use it to get your email from your mail server via POP, the Post Office Protocol, on port 110.[4] You may want to protect your POP connection for several reasons, such as keeping your password from going across the line in the clear[5], or just to make sure no one's sniffing the email you're downloading. Normally, your mail client will establish a TCP connection to the mail server on port 110, supply your username and password, and download your email. You can try this yourself using telnet or nc on the command line: xahria@desktop\$ nc mailserver 110

```
+OK SuperDuper POP3 mail server (mailserver.my_ism.net) ready.
```

```
USER xahria
```

```
+OK
```

```
PASS twinnies
```

```
+OK User successfully logged on.
```

```
LIST
```

```
+OK 48 1420253
```

```
1 1689
```

```
2 1359
```

```
3 59905
```

```
...
```

```
47 3476
```

```
48 3925
```

```
.
```

```
QUIT
```

```
+OK SuperDuper POP3 mail server signing off.
```

```
xahria@desktop$
```

We can wrap this TCP connection inside an SSH session using SSH Port Forwarding. If you have SSH access to the machine that offers your service (POP, port 110 in this case) then SSH to it. If you don't, you can SSH to a server on the same network if the network is trusted. (See the security implications of port forwarding later in this article.) In this case, let's assume we don't have SSH access to the mail server, but we can log into a shell server on the same network, and create a tunnel for our cleartext POP connection: # first, show that nothing's listening on our local machine on port 9999:

```
xahria@desktop$ nc localhost 9999
```

```
Connection refused.
```

```
xahria@desktop$ ssh -L 9999:mailserver:110 shellserver
```

```
xahria@shellserver's password: *****
```

```
xahria@shellserver$ hostname
```

```
shellserver
```

From a different window on your desktop machine, connect to your local machine (localhost) on port 9999:

```
xahria@desktop$ nc localhost 9999
```

```
+OK SuperDuper POP3 mail server (mailserver.my_ism.net) ready.
```

```
USER xahria
```

```
+OK
```

```
PASS twinnies
```

```
...
```

Before we connected to the shellserver with SSH, nothing was listening on port 9999 on our desktop - once we'd logged in to the mail server with our tunnel, this port was bound by our SSH process, and the TCP connection to local port 9999 was magically tunneled through SSH to the other side. Let's describe how this works in detail, using the example above.

- You launch the `/usr/bin/ssh` SSH client on the command line.
  - The SSH client logs into the remote machine using whatever authentication method (password, Pubkey, etc) is available.
  - The SSH client binds the local port you specified, port 9999, on the loopback interface, 127.0.0.1.
  - You can do anything in your remote machine that you want -- tar up some files, write to some users, delete `/etc/shadow...` This interactive login is completely usable, or you can just let it hang around doing nothing.
  - When a process connects to 127.0.0.1 on port 9999 on the client machine, the `/usr/bin/ssh` client program accepts the connection.
  - The SSH client informs the server, over the encrypted channel, to create a connection to the destination, in this case mailserver port 110.
  - The client takes any bits sent to this port (9999), sends them to the server inside the encrypted SSH session, who decrypts them and then sends them in the clear to the destination, port 110 of the mailserver.
  - The server takes any bits received from the destination, mailserver's port 110, and sends them inside the SSH session back to the client, who decrypts and sends them in the clear to the process that connected to the client's bound port, port 9999.
  - When the connection is closed by either endpoint, it is torn down inside the SSH session as well.
- SSH Port Forward Debugging Let's see it in action by using the verbose option to ssh: `xahria@desktop$ ssh -v -L 9999:mailserver:110 shellserver`

```
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Rhosts Authentication disabled, originating port will not be trusted.
debug1: Connecting to shellserver [296.62.257.251] port 22.
debug1: Connection established.
debug1: identity file /home/bri/.ssh/identity type 0
debug1: identity file /home/bri/.ssh/id_rsa type 1
debug1: identity file /home/bri/.ssh/id_dsa type 2
...
debug1: Next authentication method: password
xahria@shellserver's password: *****
debug1: Authentication succeeded (password).
debug1: Connections to local port 9999 forwarded to remote address localhost:110
debug1: Local forwarding listening on 127.0.0.1 port 9999.
debug1: channel 0: new [client-session]
debug1: Entering interactive session.
debug1: channel 0: request pty-req
debug1: channel 0: request shell
xahria@shellserver$
```

As you can see, there's a brief mention of port 9999 being bound and available for tunneling. We haven't made a connection to this port yet, so no tunnel is active yet. You can use the `~#` escape sequence to see the connections in use. This sequence only works after a carriage return, so hit enter a few times before trying it: `xahria@shellserver$ (enter)`

```
xahria@shellserver$ (enter)
xahria@shellserver$ (enter)
xahria@shellserver$ ~#
The following connections are open:
 #1 client-session (t4 r0 i0/0 o0/0 fd 5/6)
xahria@shellserver$
```

You can see that there's only one connection, our actual SSH session from which we're typing those unix commands. Now, in a different window if we do a telnet localhost 9999, we'll open up a new connection through the tunnel, and we can see it from our SSH session using `~#` `xahria@shellserver$ (enter)`

```
xahria@shellserver$ ~#
The following connections are open:
 #1 client-session (t4 r0 i0/0 o0/0 fd 5/6)
 #2 direct-tcpip: listening port 9999 for mailserver port 110,
   connect from 127.0.0.1 port 42789 (t4 r1 i0/0
   o0/0 fd 8/8)
```

You can see that now we have both the SSH session we're using, plus a tunnel, the second entry. It tells you all you need to know about the connection -- it came from our local machine (127.0.0.1) source port 42789, which we could look up with `netstat` or `lsof` output if we were curious about it. RemoteForward Example SSH Forwards actually come in two flavours. The one I've shown above is a local forward, where the ssh client machine is listening for new connections to be tunneled. A Remote Forward is just the opposite - a tunnel initiated on the server side that goes back through the client machine. The classic example of using a Remote Forward goes something like this. You're at work, and the VPN access is going to be down for maintenance for the weekend. However you really have some important work to do, but you'd

rather work from the comfort of your desk at home, rather than being stuck at work all weekend. There's no way for you to SSH to your work desktop because it's behind the firewall. Before you leave for the evening, you SSH from your work desktop back to your home network. Your ~/.ssh/config file has the following snippet: `laine@work$ tail ~/.ssh/config`

```
Host home-with-tunnel
  Hostname 204.225.288.29
  RemoteForward 2222:localhost:22
  User laineboo
```

```
laine@work$ ssh home-with-tunnel
laineboo@204.225.228.29's password: *****
```

```
laineboo@home$ ping -i 5 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=0.1 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=255 time=0.2 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=255 time=0.2 ms
```

...  
We've set up a tunnel using the RemoteForward option in the SSH configuration file. (We could have set it up on the command line using the -R option if we'd preferred.) Just to make sure our firewall doesn't kill the connection for inactivity, we run a ping for grins. Then we head on home. Later that evening, we can sit down on our home machine and see that we're logged in: `laineboo@home$ last -1`

```
laineboo pts/18 firewall.my_work.com Tue Nov 23 22:28 still logged in
```

```
laineboo@home$ ps -t pts/18
PID TTY TIME CMD
3794 pts/18 00:00:00 ksh
4027 pts/18 00:00:00 ping -i 5 127.0.0.1
```

Now comes the payoff - our tunnel is listening on our home machine on port 2222, and will be tunneled back through the corporate firewall to our work machine's port 22. So to SSH to work from home, since we have our tunnel ready, we simply point `/usr/bin/ssh` to port 2222: `laineboo@home$ ssh -p 2222 laine@localhost`

```
laine@localhost's password: *****
```

```
laine@work$
Success!
```