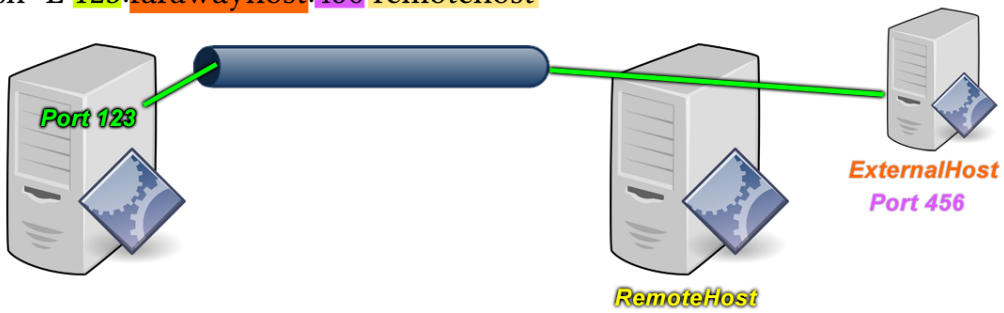
SSH LOCAL TUNNEL/ PORT FORWARD

ssh -L 123:localhost:456 remotehost

local: -L ---- given port on local (client) host to be forwarded to host, port on other side.
ssh -L sourcePort:forwardToHost:onPort connectToHost
Connect ssh to connectToHost, and forward all connection attempts to the local sourcePort to port onPort on forwardToHost, which can be reached from the connectToHost machine.

ssh -L 123:farawayhost:456 remotehost

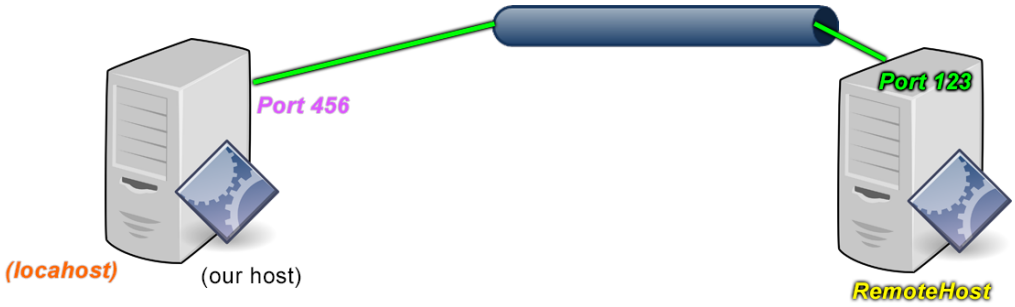


ssh -L 80:localhost:80 SUPERSERVER
You specify that a connection made to the local port 80 is to be forwarded to port 80 on SUPERSERVER. That means if someone connects to your computer with a webbrowser, he gets the response of the webserver running on SUPERSERVER. You, on your local machine, have no webserver running.

ssh -L 8080:127.0.0.1:80 user@webserver
Then in your browser on local use URL http://localhost:8080/
it will connect to local machines port 8080, which ssh will forward on to remote ssh, and it will then make a request to 127.0.0.1:80. Note 127.0.0.1 is actually the remote server's localhost, but it could have been a host/IP available at the remote machine's network.

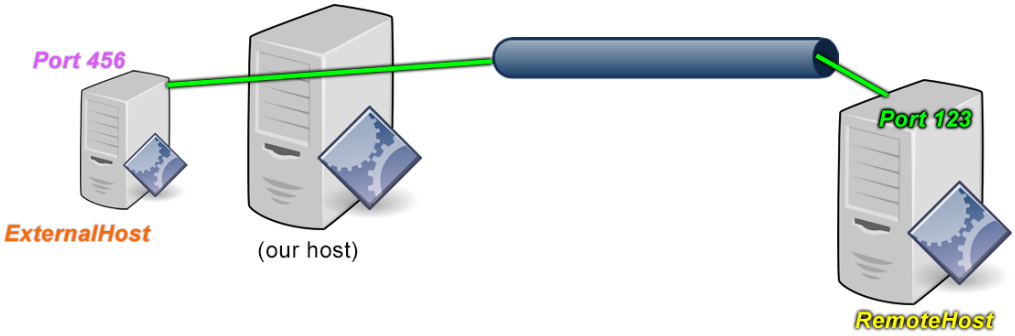
SSH REMOTE/REVERSE TUNNEL/ PORT FORWARD

ssh -R 123:localhost:456 remotehost



remote: -R - Port on remote host is to be forwarded to the given host and port on the local side.
ssh -R sourcePort:forwardToHost:onPort connectToHost
Connect with ssh to connectToHost, forward all connection attempts to remote sourcePort to port onPort on the machine forwardToHost, which can be reached from your local machine.

ssh -R 123:nearhost:456 remotehost



ssh -R 80:localhost:80 tinyserver
You specify that a connection made to the port 80 of tinyserver is to be forwarded to port 80 on your local machine. That means if someone connects to the small and slow server with a webbrowser, he gets the response of the webserver running on your local machine. The tinyserver, which has not enough diskspace for the big website, has no webserver running. But people connecting to tinyserver think so.

Other things could be: The powerful machine has five webservers running on five different ports. If a user connects to one of the five tinyservers at port 80 with his webbrowser, the request is redirected to the corresponding webserver running on the powerful machine. That would be
ssh -R 80:localhost:30180 tinyserver1
ssh -R 80:localhost:30280 tinyserver2
etc.

Or maybe your machine is only the connection between the powerful and the small servers. Then it would be (for one of the tinyservers that play to have their own webservers):

ssh -R 80:SUPERSERVER:30180 tinyserver1
ssh -R 80:SUPERSERVER:30280 tinyserver2
etc

ssh -R 10123:127.0.0.1:123 user@webserver
Asks ssh to create a listening port on the remote machine which it will forward back (reverse) to the local ssh to forward on. So, after ssh connects to webserver, the remote ssh creates and listens on a port 10123. A process on webserver connecting to 10123, ssh will pick it up and send it back to the local machine's ssh, which sends it on to 127.0.01:123 port.

SSH Security and You - /bin/false is *not* security - Jordan Sissel, Wed, 28 Dec 2005

While at RIT, I discovered that a few important machines at the datacenter allowed to authenticate against them via ssh. Everyone's shells appear to be set to /bin/false (or some derivative) on said machines, so the only thing you'll see after you authenticate is the login banner and your connection will close. I thought to myself, "Fine, no shell for me. I wonder if port forwarding works?" The sysadmin was tasked with securing these machines forgot something very important about ssh2: channels. I use them often for doing agent, x11, or port forwarding. So what happens if we try to port forward without requesting a shell (*ssh -N*)? It allows you to do the requested port forward and keeps the connection alive. SSH stays connected because it never executes the shell, so it never gets told to die. Whoops!

The fix is common sshd: AllowGroups to restrict ssh-authenticatable users by group- put all the users who need to ssh to your machines into a single group and prevent unauthorized users from authenticating (getting a shell, port forwarding, etc).

What is /bin/false?

Many times you will have a system where you need a user to exist in the account database (say, /etc/passwd) but don't want to give them shell access to your machine(s). A common solution to this is to set a user's shell to /bin/false. This has the effect of rejecting shell login attempts over ssh, telnet, or other shell-requesting protocols. It may have other side effects too, but those are beyond the scope of this article. It does not keep them from using said account to authenticate over ssh and using non-shell tools such as port forwarding. A default sshd config will often allow tunneling and other non-shell activity.

Hole #1: Potential Firewall Bypass

So, to make things more interesting, there are two obvious holes I can exploit here. The first, is firewall-bypass. ITS employs lots of ACLs limiting access to machines by IP ranges. This is a normal practice in the world. However, what if the machine I am port forwarding through is one of these trusted machines? You just gave me access to your supposedly locked-down network. Don't do that.

Hole #2: Anonymous traffic

I can make my traffic far more anonymous by using ssh's port-forward or SOCKS proxy feature. OpenSSH does not appear to log port-forward-only sessions, so chances are you can get away with using this half-secured server as a proxy. I haven't done all the research, but ssh port-forward-only sessions only seem to show up in process listings and not standard audit logs. This stuff needs to be logged if you're going to allow it.

Hole #3: Resource Starvation

The third one is less obvious, but quite easy. You setup a remote port forward (*ssh -R*) pointed at "itself" (the machine you're logging into) and then a local port forward (*ssh -L*) to the machine so you can just touch it with telnet and walk away. This creates a large problem on the end machine becuase you will eventually take up all the available file descriptors, and since unix lives on file descriptors, you just DoS'd the machine. So if some naughty person manages to guess a password of one of your 30000 users, he/she can happily perform resource starvation attacks 'till the end of the day despite your wishes that I stay off your machine. Like I said, **/bin/false is not security**.

Example DoS

I used 3 xterms for this (wow, high-tech!). I could've used one shell, but I like seeing debug output.

(terminal 1) whack% ssh -vN -L4141:localhost:4141 kenya
(terminal 2) whack% ssh -vN -R4141:localhost:4141 kenya
(terminal 3) whack% telnet localhost 4141

When the telnet command executes, the other two terminals are flooded with debug information detailing new port forwarded connections happening, etc. Since you've just created a loop, you can now kill the telnet session and the loop maintains it's stability. Simply wait a few minutes and you'll fill up the system's open file table. During this test, I checked the growth of the number of open file descriptors on kenya (the "target" of our DoS):

```
kenya(~) [1003] % while :; do sysctl kern.openfiles; sleep 1; done
kern.openfiles: 242
kern.openfiles: 242
kern.openfiles: 278
kern.openfiles: 652
kern.openfiles: 896
kern.openfiles: 1082
kern.openfiles: 1246
```

The number was stable at 242 before the attack began, and rose rapidly to something like 100-300 file descriptors per second. That is quite significant, and will very quickly hose a host. Notably, there is a rapid decelleration in the number of files opened per second, but it steadies (for me) around 20-30 per second after about 7000 open sockets.

Fixing It

Modern systems will generally have a pam module that will help- possibly allowing you to reject authentication requests over ssh simply because the user's shell is set to /bin/false, or /usr/sbin/nologin (or wherever that is on your system).

Other solutions include fixing your sshd config to either: 1) restrict which users are allowed via AllowUsers/ AllowGroups, and/or 2) deny tunneling/forwarding: AllowTcpForwarding, X11Forwarding, PermitTunnel

So, ssh sessions not requesting shells (*ssh -N*) do not show up in utmp, therefore are not listed in w(1) output, and not in last(1), etc. Unless there's a more in-depth audit log, you just made your traffic at least somewhat anonymous (assuming you're actually port-forwarding). So go ahead and abuse that.