SSH TRICKS - FORWARDING

I love the SSH protocol. Not only does it secure telnet and ftp -two highly useful and highly insecure protocols- but it has all these other little tricks buried in it. It is this wealth of features which makes SSH so useful, but also conversely what can also make it so dangerous when improperly implemented. In this article I am going to be specifically talking about port forwarding using SSH. You need to think of port forwarding as tunneling, meaning that a person can tunnel almost anything through a SSH channel. Lets take a look..

SSH and Trust Relationships
First thing to understand is that SSH has a default port of 22, but this port can be changed. When a client connects to a SSH server, anything traveling via that established channel will be encrypted. You also need to understand the issues of trust relationships as we go through this article. A trust relationship is where one machine grants another machine certain privileges based upon where that machine is situated. I know these two ideas do not seem related right now, but just bear them in mind and you will soon see how they relate.

Local Port Forwarding
What happens here is that the client connects to the ssh server , then asks the ssh server to forward a port it can see, to a specified port on the client. For example, lets say the client is a machine called *clntpc,* the SSH server's name is *bigbox,* and the target server is *bullseye*. Now *clntpc* wants to check the POP3 service on *bullseye*, but does not have the access to do so. But *bigbox* can access *bullseye*, and *clntpc* can ssh to *bigbox*. So what *clntpc* does is ssh to *bigbox*, and tell *bigbox* to tunnel out the POP3 port on *bullseye* to a specified local port on *clntpc*. Once this has happened, *clntpc* can connect to itself at the specified local port and access the POP3 service on *bullseye*. Lets take a look at this process in detail:

- *clntpc* connects to *bigbox* and creates the tunnel : *ssh -L 10101:bullseye:110 bob@bigbox*
- thus *bigbox* tunnels the POP3 port on *bullseye* to port 10101 on *clntpc*
- to access POP3, *clntpc* can do something like : *telnet localhost 10101*

What happened in the above example is that *clntpc* could not access the POP3 resource on *bullseye*, but *bigbox* could. So *clntpc* exploited that trust relationship to get *bigbox* to tunnel the POP3 resource out to *clntpc*. If *clntpc* wanted to make the locally forwarded port accessible to people accessing *clntpc*, then in setting up the tunnel a *-g* flag would have been used. Also important to remember is that the communication between *clntpc* and *bigbox* is all encrypted and happening over the ssh channel, this includes the POP3 traffic, however this does not apply to the traffic between *bigbox* and *bullseye*. Remember that the format for the ssh local forwarding command is:
ssh -g -L [local_port]:[target_server]:[server_port] [user]@[ssh_server]

Remote Port Forwarding
This is the other aspect of ssh tunneling, and in this instance the client does not ask for target port to be sent back to itself, rather it sends a target port to another system's port. For example, let stay with our three machines and the POP3 traffic. Lets same *bigbox* wants people to access the POP3 email on *bullseye*, but does not want to make the port publicly available. What can happen is that *bigbox* can forward the *bullseye* POP3 port to *clntpc*. This would allow people in the network with *clntpc* to connect to a specified port on *clntpc* to access the POP3 service on *bullseye*. Also since this traffic is traveling in a ssh tunnel, the traffic is encrypted and *bigbox* does not need to make the POP3 port publicly available. Think of it as a cheap and fast VPN solution for one port. Lets look at the details:

- *bigbox* connects to *clntpc* and creates the tunnel: *ssh -R 11011:bullseye:110 mack@clntpc*
- thus *bigbox* tunnels the POP3 port on *bullseye* to port 11011 on *clntpc*
- to access POP3, people can do something like: *telnet clntpc 11011*

What has happened in the above example is that *bigbox* did not want to make the POP3 port publicly available and wanted to encrypt the POP3 traffic. Now since *bigbox* could see both *clntpc* and *bullseye*, but they could not see one another, it could act as the creator of the tunnel as it was trusted by both sides (remember trust relationships). Once the tunnel was made, all traffic between *bigbox* and *clntpc* went through the ssh tunnel and was encrypted, but remember that the traffic from *bigbox* to *bullseye* and the traffic between *clntpc* and any clients accessing the forwarded port would not be encrypted by ssh. If, as in this example, you want other people to have access to the forwarded port, then you must remember to set the *GatewayPorts yes* option in the sshd configuration file (generally sshd_config) on the client side (in this example it would need to be set on *clntpc*). However, setting this makes all forwarded ports publicly available which may be a hassle. The format for the command is:

```
ssh -R [port_to _be forwarded_to]:[target_server]:[forwarded_port] [user]@[ssh_server]
```

Other tweaks
You will have seen in using these examples that they open up an ssh session, and when that session closes so does the tunnel. Now this behavior could put a dampener on your testing. But what you can try are some of the following;

- To send the ssh session to the background use the *-f* flag, in using this flag you must execute a command though, so use something like *sleep 99999*. This means that the full command would be:
  ```
  ssh -g -f -L [local_port]:[target_server]:[server_port] [user]@[ssh_server] "sleep 99999"
  ```
- If you only want the command to stay open for the length of a single tunneled session, you could use *sleep 10* as the command, then when the 10 seconds is up and the session wants to close, it will see that there is an active tunnel. It will then wait for the tunnel activity to stop before closing the session.
- If you do not want to type your password in when you connect to setup the tunnel, you can setup passwordless logins using certificates for authentication in ssh.

Final Thoughts
Just remember something when you play with this functionality. Yes, it is seriously cool. But it also can very easily be used to circumvent established firewall rules and security policies. Remember that those rules and policies were put in place for a reason and you sidestep them at your own risk. Also all you system administrators out there, bear this functionality in mind when you're setting up your networks. And lastly, all playing with this is at your own risk. You screw up and let an attacker onto your network, that is your problem.... otherwise have fun.