SSH Bouncing - How to get through firewalls easily.

By Brian Hatch.

Summary: *Often you'll have firewalls or other network equipment that doesn't allow direct SSH access to machines behind it. Using a bit of trickery, you can get through without seemingly jumping through any hoops.*

Have you ever been in the situation that you wanted to SSH directly to a machine, but there has been some device in between that prevents it? Say you have a Linux firewall that protects your DMZ, and you have a boatload of machines behind it that you want to manage. There are all sorts of methods that are used to do so, and all have some level of annoyance.

SSH to the intermediate host

The first and most simple solution is to SSH to the machine in the way, say the firewall. The firewall administrator can just set up one or more non-privileged accounts for users who need access to the machines behind it. This is a pain, of course - if you want to upload a file, you need to upload it to the firewall via `sftp`/`scp`, and then upload it to the target server. What a pain. And security-wise, you now have all these random firewall accounts running amok, probably not your favourite situation.

Of course, it's still nicer than Windows networking, but we can do better.

Non-standard SSH ports

You can set up a bunch of ports that tunnel into the target machines. You might have firewall port 5000 go to port 22 (the SSH port) on machine1, firewall:5001 go to machine2, firewall:5002 go to machine3, etc. For example,

```
#!/bin/sh
# Set up forwards for inbound SSH


EXT_IP=205.382.29.20    # External IP address
EXT_IFACE=eth0          # External Interface
INT_IFACE=eth1          # Internal Interface

# handy dandy tcp forward function
tcp_forward () {
  local ext_port int_ip
  echo "$1" | {
    read int_ip  ext_port
     # create prerouting and appropriate forward from the tuple
     iptables -A PREROUTING -t nat -p tcp -d $EXT_IP \
        --dport $ext_port -j DNAT \
        --to-destination $int_ip:22
     iptables -A FORWARD -i $EXT_IFACE -o $INT_IFACE \
        -p tcp -d $int_ip --dport 22 -m state \
        --state NEW -j ACCEPT

 }

 tcp_forward " 192.168.1.1      5000"
 tcp_forward " 192.168.1.2      5001"
 tcp_forward " 192.168.1.3      5002"
 tcp_forward " 192.168.1.4      5003"
```

```
   ...
   tcp_forward " 192.168.1.58     5057"
   tcp_forward " 192.168.1.59     5058"
```

What problems do we have with this setup? Well, you need to manage the forwards, which is rather a pain. Also, you now have these ports open to the outside world, which means you need to create ACLs for them on the firewall or the target or both, lest anyone be able to try to guess passwords.

The other problem with this is that you'll get ssh host key conflicts unless you're careful -- you appear to connect to the machine 'firewall' but you get different keys when you hit the actual machine behind it. To get around this, you can use `$HOME/.ssh/config` sections like this:

```
Host machine1
Hostname firewall.my_network.com
Port 5000
HostKeyAlias machine1

Host machine2
Hostname firewall.my_network.com
Port 5001
HostKeyAlias machine2
```

Then you can just `ssh machine1` and not need to remember the port, and due to the `HostKeyAlias` option each machine will have it's own key recognised correctly, rather than sharing the one for the firewall.

Netcat SSH bounce

This is my preferred method, and it can be used to create a seamless connection. What you do is SSH to the intermediate machine (the firewall in this example) and from that machine you run Netcat (`nc`). Netcat can be used in all sorts of situations, such as a replacement for telnet:

```
$ nc www.some_host.com 80
GET / HTTP/1.0

...
```

When used as a telnet-like replacement, all it does is open up a connection to the remote port and transfer the data, unaltered, to and from it and your keyboard/screen. So how do we use this to help out with our SSH connection?

OpenSSH supports the ability to use a proxy command. A proxy command is a program (shell script, binary, etc) that `/usr/bin/ssh` will run, rather than making an actual TCP connection to the target. The job of the proxy command is to establish a connection to the target. `/usr/bin/ssh` talks to this command, and doesn't care how it does its work.

So, what will our proxy command do?

- The proxy command will SSH to the firewall
- On the firewall, it will run Netcat as follows:

  ```
  nc -w 1 target_host 22
  ```

The `nc command says 'connect to port 22 on the target host, and wait one second after the connection is dead before closing it.' Now Netcat's stdin/stdout are going to be connected to the SSH server on the target, and the /usr/bin/ssh client on your desktop. To the client program, it looks just like it's hit the target directly, the proxy does the work of getting them together.`

So, how do we create this proxy? How 'bout a shell script:

```
$ cat netcat-proxy-command
#!/bin/sh
bouncehost=$1
target=$2

ssh bouncehost   nc -w 1 $target 22
```

```
ssh bouncehost   nc -w 1 $target 22
```

Then point to this proxy command via your `$HOME/.ssh/config` file:

```
$ head $HOME/.ssh/config
Host machine1
Hostname machine1
HostKeyAlias machine1
ProxyCommand netcat-proxy-command firewall.my_network.com 192.168.1.1

Host machine2
Hostname machine2
HostKeyAlias machine2
ProxyCommand netcat-proxy-command firewall.my_network.com 192.168.1.2

...
```

Or, to make it even easier to copy/paste, use the fact that `%h` in a `$HOME/.ssh/config` file is replaced with the hostname, and you can use the following:

```
$ head $HOME/.ssh/config
Host machine1
Hostname 192.168.1.1
HostKeyAlias machine1
ProxyCommand netcat-proxy-command firewall.my_network.com %h

Host machine2
Hostname 192.168.1.2
HostKeyAlias machine2
ProxyCommand netcat-proxy-command firewall.my_network.com %h
...
```

All the logic of how to actually get to the host is in the `config` file, all the magic in getting there is in the proxy script, and you can connect 'directly' to the target machine at the command line like this:

```
$ ssh machine1
$ scp machine1:/path/to/some/file .
```

Now doing this requires that you can connect to the firewall without a password[1] If you can't, then you'll want to to enable SSH key based security. If you don't know how to do that yet, see one of the Previous Articles that covers it.

There are many other options that I didn't cover here, such as VPN technologies, Portknocking and fun tunnels like chownat (http://chownat.lucidx.com/). While these can all be exciting, I'm trying to stick to pretty portable tools that are likely pre-installed on your machines anyway.

Next time, we'll see how to tighten security a bit by making changes to the firewall user's configuration.

NOTES:

[1] If you don't have passwordless authentication to the firewall, you'll need to type the firewall password each time too. This is annoying, but not a show stopper.

Brian Hatch is Chief Hacker at Onsight, Inc and author of ***Hacking Linux Exposed*** and Building Linux VPNs. He can't understand how a few months have gone by since he had time to write. Oh wait, maybe it's the number of kids in his home, and the massive distance between him and any free babysitting -- i.e. relatives... Brian can be reached at brian@hackinglinuxexposed.com.
