CHAPTER 1

Invisibility Cloak

Imagine that you could hide a server from the Internet but still have access to your ISP's superior bandwidth. Without making any changes, you would be able to securely use it as a file repository, among many other things.

You'd also have full access to the command line so that you could start and stop or even install any services that you wanted to use. The choice would be yours, whether you ran those services briefly and then closed them down, or left them running and visible to the outside world for a period of time.

This is possible to achieve using a technique called port knocking. You can disguise your server by closing all network ports to the outside world, genuinely making it invisible. The single port that you might choose to open up at will, by using a prearranged "door-knock," could be for your SSH server or for some other service. In this chapter, you'll see how you can create an invisible server along with some options that you might want to consider.

Background

By disguising the very existence of a server on the Internet, at best you can run a machine in private, and at worst, even if its existence is known, you will reduce the attack surface that an attacker can target by limiting the time ports are open and even partially visible.

Probing Ports

Before beginning, let's take a closer look at network ports on a server, so you'll have a frame of reference. If you've ever used security tools such as Nmap, then you may be familiar with the initially confusing premise that some ports appear to be closed when in fact they are not. Nmap makes the distinction between whether a nonopen port has a service (a daemon) listening behind it or not.

Nmap refers to closed ports as those that don't have a daemon listening behind them but do appear to be open or at least potentially available. If Nmap refers to filtered ports, it means that a firewall of some kind is preventing access to the IP address that is scanning the system in question. This is partly to do with TCP RST packets, and there are also three other states that Nmap reports back on: unfiltered, open|filtered, and closed|filtered. If you want more information on how these states are different, go to `https://nmap.org/book/man-port-scanning-basics.html`.

Confusing a Port Scanner

Now that you know how ports may present themselves to port scanners, let's look at how to obfuscate the response you give back in order to confuse sophisticated port scanning techniques. The most obvious tool of choice, thanks to its powerful feature set, would be the kernel-based firewall Netfilter, more commonly known as iptables.

Here's how it works. For TCP packets, you want to manipulate how you respond to port probes by using iptables to generate a REJECT request. For other protocols you want to simply DROP the packets. This way, you get closed, not filtered, responses from Nmap. Based on what I've gathered from most online opinions (and it seems that this argument is both contentious and changeable), a closed port is the best response that you can hope for. This is because you're not openly admitting to blocking any ports with a firewall, nor is the port simply open because a daemon is running behind it.

To explain a little further, under normal circumstances, an unreachable port would usually generate an ICMP Port Unreachable response. You don't want these errors to be generated, however, because that would mean a server was listening on that port in the first place and you would give your server's presence away. The tweaked REJECT response that you want to generate instead is applied as follows:—`reject-with tcp-reset`. This helps you to respond as if the port were unused and closed, and also not filtered.

You simply append this snippet to the end of each of your iptables rules:

```
-j REJECT—reject-with tcp-reset
```

By using this technique, you're simply making sure you're not giving away unnecessary information about your system.

Note that in the port knocking example that you're about to look at, you won't be using that iptables option. This is because you won't be running additional services to your SSH server. However, this background information will help you understand how an attacker might approach a machine's ports and how you can apply a—`reject-with tcp-reset` option to other services.

There is some debate about using iptables DROP versus REJECT responses in your rules. If you're interested, you'll find some insightful information on the subject at

`www.chiark.greenend.org.uk/~peterb/network/drop-vs-reject`.

Installing knockd

Now that you are armed with some useful background information, I'll walk you through how to install a port knocker on your server. As we continue you might consider which

services you may wish to run hidden from the Internet at large. There might be an occasion to run a web server or a mail server on an unusual port for a few hours, for example.

Packages

Let's look at installing the package that will give your system port knocking functionality. Called knockd, this package is installed in different ways depending on your system.

On Debian derivatives you install the package as follows:

```
# apt-get install knockd
```

On Red Hat derivatives you install it as follows:

```
# yum install knockd
```

A main config file controls most of the config required for knockd. On a Debian Jessie server, this file resides at `/etc/knockd.conf`. Take a look at Listing 1.1, which shows my main config file, to see how knockd works.

LISTING 1.1 The main config file. The port sequences and (importantly) `-I INPUT` have been altered from the defaults.

```
[options]
        UseSyslog
[openSSH]
        sequence     = 6,1450,8156,22045,23501,24691
        seq_timeout = 5
        command    = /sbin/iptables -I INPUT -s %IP% -p tcp—dport 22 -j
   ACCEPT
        tcpflags       = syn

[closeSSH]
        sequence     = 3011,6145,7298
        seq_timeout = 5
        command    = /sbin/iptables -D INPUT -s %IP% -p tcp—dport 22 -j
   ACCEPT
        tcpflags       = syn
```

Changing Default Settings

In Listing 1.1, you can see a section for setting up your options at the top. The other two sections are the actions that you want to perform when knockd opens up SSH access or when you shut down your port access. Both sections also include the default port knocking

sequence to trigger those actions under the `sequence` option. After installing knockd, I immediately changed those ports from the defaults to avoid reducing the effectiveness of my server security. The defaults are ports 7000, 8000, and 9000 to open up SSH access and ports 9000, 8000, 7000 to close access. As you can see, I've added more ports to open up the access so someone will be less likely to stumble across their combination with an arbitrary port scan.

After changing any settings, you can simply restart knockd as follows on systemd-based operating systems:

```
# systemctl restart knockd.service
```

After installing knockd, if you want more background information on the package, then Debian Jessie has a brief README file that you can find at `/usr/share/doc/knockd/README`.

This helpful README file discusses how knockd works, among other things. It uses a library called `libpcap`, which is also used by several other packages such as `tcpdump`, `ngrep`, and `iftop` (which capture packets for inspection). Thanks to its clever design, knockd doesn't even need to bind to the ports, which it's covertly listening on, in order to monitor raw traffic.

Altering Filesystem Locations

Events such as connections, disconnections, or errors are logged directly to your system's syslog, and may be written to the `/var/log/messages` or `/var/log/syslog` file. If you don't want this information to be buried among other system log activities, or go to the bother of parsing an unwieldy log file, then you can create your own custom log file. I prefer to do this so that debugging is much clearer, and I might use an automated tool or a custom shell script to e-mail logs to myself daily so that I can monitor suspicious events. Because I'm keeping all knockd logs in one place, the information is easier to parse for scripts and other tools.

```
[options]
        LogFile = /var/log/portknocking.log
```

Changing the logfile's location is a common solution, but you can also alter where the Process ID file is written when the knockd service is launched. You can change the location under the `[options]` section of the config file, as follows:

```
[options]
       PidFile = /var/tmp/run/file
```

Some Config Options

Now that you've got a better understanding of how the main config file is set up, you can examine how to configure it for your needs. Among a number of other tasks, you will consider how the timeouts of certain options play a part in setting up your server.

Starting the Service

Don't be alarmed if you see an error message saying that knockd is disabled. This is a precaution so that until you have finished setting it up, knockd won't introduce unwelcome changes to iptables.

On Debian Jessie the error message asks you to change the following parameter to 1 in the file `/etc/default/knockd`:

```
START_KNOCKD=1
```

Clearly, you should only do this after double-checking your configuration or making sure that your out-of-band access is working as expected.

Changing the Default Network Interface

Once you have configured your preferred port sequence, you might want to tweak other parameters. In the config file (`/etc/default/knockd`), you have the opportunity to alter the `KNOCKD_OPTS` settings. The example within that file is commented out and means that you can alter the network interface that knockd is listening on, as follows:

```
KNOCKD_OPTS="-i eth1"
```

These options will be appended to the knockd service, and you will need to restart your service to make the changes live, as follows (on systemd machines):

```
# systemctl restart knockd
```

Packet Types and Timing

In the `/etc/knockd.conf` file, you can alter a few settings to finely tune how clients can connect to you. Referring back to Listing 1.1, under the `[openSSH]` section, you will add more options as follows:

```
[openSSH]
        tcpflags = syn
```

```
seq_timeout = 10
cmd_timeout = 15
```

The `tcpflags` option means that you can expect a specific type of TCP packet to be sent for knockd to accept it. That's a TCP "SYN" in this case. The TCP flags that you can use are `fin`, `syn`, `rst`, `psh`, `ack`, and `urg`. If the specified type of TCP packet isn't received, then knockd will simply ignore those packets. Be aware that this isn't how knockd usually works. Normally an incorrect packet would stop the entire knocking sequence from working, which would mean that the client would have to start again in order to connect. You can separate multiple TCP packet types by using commas, and it appears that newer versions (from version 0.5 according to the changelog) of knockd can use exclamation marks to negate the packet type, such as `!ack`.

Back to the other options in the example. You may have noticed that `seq_timeout` is already present in Listing 1.1 by default. However, because you have increased the number of ports within your sequence setting, you have upped the `seq_timeout` value to 10 rather than 5. This is needed because on a slow connection, such as via your smartphone, timeouts may occur.

The final option in the example is `cmd_timeout`. This option applies to what happens after knockd has received a successful knock. The sequence of events is as follows. First, once the port knocking has been confirmed as valid, knockd will run the `start_command` (refer to Listing 1.1 if you need a reminder). If this setting is present, then after knockd has executed the `start_command` option, it will only wait for the time specified in `cmd_timeout` before running the `stop_command` action.

This is the preferred way to open up your SSH server for access, and then promptly close it down once your connection is established. You shouldn't have any problems continuing with your session, but new connections will need to run through the port-knocking sequence again in order to connect. Think of this action as closing the door behind you once you've entered. Your server will become invisible again, and only your associated traffic will be visible.

Testing Your Install

Because you are dealing with the security of a server, you should run a few tests to see that knockd is working as expected. Ideally you will have access to another client machine to run some tests from. To be completely sure that knockd is opening and closing ports correctly, I like to test by connecting from a completely different IP address. If you don't have access to a connection with a different IP address, then you might be able to drop your Internet connection periodically so that your ISP will allocate you a new dynamic IP address to test from. Some broadband providers will do this after a reboot or your mobile provider might too in addition.

Port Knocking Clients

You can use different clients to create a knocking sequence in order to initialize a connection and open up your SSH port. You can even manually use tools such as Nmap, netcat, or Telnet to manually probe the required ports in sequence. The documentation also mentions that you can use the hping, sendip, and packit packages if they are available. Let's look at an example of the `knock` command that comes with the knockd package.

If you used the `openSSH` section shown in Listing 1.1, then you would set up your simple `knock` command with the following syntax:

```
# knock [options] <host> <port[:proto]> <port[:proto]> <port[:proto]>
```

I have configured TCP ports in Listing 1.1, so you can run the `knock` command as follows:

```
# knock 11.11.11.11 6:tcp 1450:tcp 8156:tcp 22045:tcp 23501:tcp
24691:tcp
```

The target host has the IP address 11.11.11.11 in this example. If you want, you can also put a combination of UDP and TCP ports in Listing 1.1; your client-side knocking sequence might look like this:

```
# knock 11.11.11.11 6:tcp 1450:udp 8156:udp 22045:tcp 23501:udp
24691:tcp
```

One nice shortcut is that if you only want to use UDP ports, then you can simply add `-u` to the start of the command rather than specifying them explicitly. You can run a command for UDP ports like this:

```
# knock -u 11 22 33 44 55
```

Let's return to your server's config file to see how TCP and UDP can be interchanged within your valid knocking sequence. In order to mix protocols, you would simply alter the `sequence` line under the `openSSH` section as follows:

```
[openSSH]
        sequence = 6:tcp 1450:udp 8156:udp 22045:tcp 23501:udp
24691:tcp
```

Making Your Server Invisible

Once you are confident that your installation is working as you'd like, you can lock your server down to hide it from attackers. An attacker may be aware of the IP address bound to your server or may somehow be able to view traffic sent and received from that IP address (for example, they might work for the ISP that the server was hosted with).

Otherwise, it should be invisible to Internet users. I would experiment with your firewall if this is not the case. To achieve Nmap's closed port status, however, the following approach works for me.

Testing Your iptables

As mentioned earlier, I will use the trusted iptables. Ideally you should have physical access to the server before locking it down, in case you make a mistake. Failing that, you should have out-of-band access of some type, such as access via a virtual machine's console, a secondary network interface that you can log in through, or a dial-up modem attached to the machine. Be warned that unless you've tested your configuration on a development box first, there's a very high chance of making a mistake and causing problems. Even though I've used port knocking before, I still get caught out and lock myself out of a server occasionally.

With that warning in mind, let's begin by looking at your iptables commands. Be careful when integrating these rules with any rules you already have. It might be easier to overwrite your existing rules after backing them up. First, you need to make sure that your server can speak to itself over the localhost interface, as follows:

```
# iptables -A INPUT -s 127.0.0.0/8 -j ACCEPT
```

You must now ensure that any existing connections are acknowledged and responded to, as follows:

```
# iptables -A INPUT -m conntrack—ctstate ESTABLISHED,RELATED -j
ACCEPT
```

You're using `conntrack` to keep track of associated connections. Your connections can continue to operate once they have been initiated successfully. Now, assuming that you're only going to open up TCP port 22 for your SSH server and no other services, you can continue. As a reminder of how to do this, referring to Listing 1.1, add the following command to open up TCP port 22:

```
command = /sbin/iptables -I INPUT -s %IP% -p tcp—dport 22 -j ACCEPT
```

Pay attention to this line. If you left an "append" by using `-A INPUT` in the command, you would be locked out by iptables. It must be an `-I` for "insert" so that the rule is entered as the first rule and takes precedence over the others.

You might be wondering what the `%IP%` variable is. Port knocking is clever enough to substitute the connecting IP address in the `-s` field, in place of the `%IP%` value.

Now here's where you have to be careful. There's no going back if this doesn't work the way you'd expect, so make sure that you have tested the rules on a virtual machine or that you

have out-of-band access to the server just in case. You block all inbound traffic to your server as follows:

```
# iptables -A INPUT -j DROP
```

If you run the following command to check your iptables rules, then you won't see any mention of TCP port 22 and your SSH port:

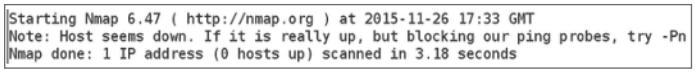
```
# iptables -nvL
```

You will, however, see such a rule (very briefly if you've set up a low value for the `cmd_timeout` setting) in iptables once you have successfully logged in.

If you are having problems at this stage, then keep reading for some ways to troubleshoot your configuration and increase your levels of logging. Otherwise, you should now have a server whose ports all report as nonexistent, thus making the server invisible, as shown in Figure 1.1.

FIGURE 1.1

Nmap seems to think there's no machine on that IP address.



Saving iptables Rules

To ensure that your iptables rules survive a reboot, you should install a package called `iptables-persistent` on Debian derivatives, as follows:

```
# apt-get install iptables-persistent
```

You can then save your rules with a command like this one:

```
# /etc/init.d/iptables-persistent save
```

Or, you can revert to the saved config by running this command:

```
# /etc/init.d/iptables-persistent reload
```

On Red Hat derivatives (on presystemd machines), you can use this command:

```
# /sbin/service iptables save
```

And to restore rules, you run this command:

```
# /sbin/service iptables reload
```

For the above to work on systemd Red Hat derivatives, you could try installing this package first:

```
# yum install iptables-services
```

Further Considerations

There are a few other aspects to port knocking that might be helpful to you. Let's have a look at them now.

Smartphone Client

On my Android smartphone, my preferred SSH app is called JuiceSSH (`https://juicessh.com`). It has a third-party plug-in that allows you to configure a knocking sequence as part of your SSH handshake. This means that there's no excuse for you not to employ port knocking, even when you're on the road and without a laptop.

Troubleshooting

If you run the command `tail -f logfile.log` on your port knocking log file, you will see various stages being written to the log. This will include whether a valid port is knocked, and importantly, if it was knocked in the correct sequence or not.

A debugging option also gives you the opportunity to increase the levels of logging produced by knockd. If you carefully open the file `/etc/init.d/knockd` and look for the `OPTIONS` line, then you can add an uppercase D character (Shift+d) to any existing values on this line as follows:

```
OPTIONS="-d -D"
```

The additional logging should be switched off once you have diagnosed and solved your issue to prevent disk space from filling up unnecessarily. The `-d` simply means run knockd as a daemon in case you're wondering. This should remain as it is for normal operation.

Back to the client for a moment. You can also add verbosity to the output, which the "knock" client generates by adding a `-v` option. Tied in with the debugging option, this should give you helpful feedback from both the client and server sides of your connections.

Security Considerations

When it comes to the public information associated with your server, a reminder that your ISP should not be publishing forward or reverse DNS information about the IP address that you are using for your server. Your IP address should appear to be unused and unallocated in order for it to be invisible.

Even on your public services such as HTTP, you need to remember to obfuscate the versions of the daemons that are in use. The common way to do this with the world's most popular web server, Apache, is to change the "ServerTokens" to "Prod" and set "ServerSignature" to "Off". These are hardly cutting-edge configuration changes, but might mean an automated attack ignores your server when a new zero-day exploit is discovered because your Apache version number wasn't in its attack database.

Another aspect to consider is discussed in the knockd documentation. It mentions that if you use `-l` or `--lookup` service launch options to resolve hostnames for your log entries, then it might be a security risk. There's a chance of some information being leaked to an attacker if you do. The attacker may be able to determine the first port of a sequence if it's possible to capture DNS traffic from your server.

Ephemeral Sequences

What about using a different approach to knocking sequences? It's also possible to use port knocking with a predefined list of port sequences that expire after they are used just once. Referring back to Listing 1.1 and your main config file, you can add this option to your `open` and `close` sections to enable one-time sequences if you want:

```
[openSSH]
        One_Time_Sequences = /usr/local/etc/portknocking_codes
```

If you remove the `sequence` line in Listing 1.1 and replace it with this code, then knockd will take its sequences from the file specified in the path instead.

The way that knockd deals with one-time sequences is unusual. It reads in the next available sequence from that file and then comments out that line following a successful connection with a valid knock. It simply adds a hash or pound character to the start of the line that has the sequence present. The next connection triggers the same sequence.

The documentation states that you should leave space at the start of each line. Otherwise, when the # character is added to the start of the line, you might find it has been overwritten unintentionally, meaning that you're locked out.

Inside your sequences file, you can add one sequence per line. That file follows the same format as the `sequences` option within the main config file.

The documentation also points out that you can put comments in by preceding them with a # character but bad things will happen (such as being locked out of your server) if you edit the sequences file when knockd is already running.

Once you understand the basic features of knockd, it is an excellent addition to experiment with. During testing, you could enter telephone numbers that you are able to memorize or some other sequence of numbers so that you're not continually looking up an insecure list.

For example, you might consider rotating through five telephone numbers, split up into valid port numbers.

Summary

In addition to making your server invisible, I've covered how your server appears to the Internet before launching an attack. In order to fully obfuscate your server using port knocking, you should think carefully about public information such as reverse DNS entry, which might give away an IP address as being in use. You might also consider using NAT to hide a server and dynamically change its IP address periodically, only letting administrators know which IP address is in use at a given time via a secret hostname, published covertly in DNS on an unusual Domain Name.

There are many other facets to protecting a server; still, I've hopefully covered enough ground to make you consider what information is leaked to the public and may potentially be used in future exploits, as well as hiding a server if you need to do so.