COMP3052.SECSEC LAB: FIREWALLS AND PORT SCANNING

(**Thank you to Mike Pound**)

LAB DESCRIPTION

In this lab you'll be familiarizing yourself with general firewall administration in Linux. This will involve setting up services, then using firewall rules to allow and disallow specific types of network traffic. You will also gain experience with one of the most popular network scanning utilities — Nmap — which can be used to hunt for vulnerable machines and ports on a network. Finally, you will learn how to use SSH tunneling to circumvent firewall restrictions.

INTRODUCTION

Firewalls perform an extremely useful function on any network. They are used to redirect, allow or prevent the passage of certain packets to other machines. In doing so, they protect vulnerable software that may be communicating using ports on those machines. Firewalls can also be used to block access to external services or websites. However, it's actually quite easy to circumvent these restrictions using SSH and proxies.

Whether you're white-hat or black-hat, Nmap is a powerful tool in your inventory. It will perform a variety of port scanning techniques on a network or target machine, and will report back which services are running. It can often even tell you which OS and version they are running. Nmap is often used as the first step in a penetration test or hack of a target machine; once you know which ports are open, you can find out which ports are vulnerable.

CLONING VIRTUAL MACHINES

Today we want to test the effectiveness of port scanning software and firewalls. It makes sense, therefore, that we will use two machines: one which operates as a server, and another that provides the testing environment. Rather than the time-consuming process of duplicating virtual disks, let's create a linked clone — a copy machine where only the difference image is stored. This is much more efficient!

Please refer to the separate "Cloning VMs" and "Networking VMs" lab manuals, where you will find instructions on setting up a network of machines. Return here once you have two Kali installations running, and communicating.

LOGIN INFORMATION

As with previous labs, a description of the Kali operating system and the general setup of the virtual machines is given in the first lab document. Both machines have the same login credentials, which are listed here for reference. As always, avoid logging in as root: we will notice!

Normal User	**Root**
Username: sec	Username: root
Password: security	Password: toor

SETTING UP THE SERVER

Select the running VM that will act as your server. We want to install a few services on this machine to get used to opening ports, before we set up a firewall. Install and/or run the following (they may already be installed, in which case, don't worry):

SSH: This is already installed, simply use the command: `sudo service ssh start`

HTTP: Already installed: `sudo service apache2 start`

FTP: Install: `sudo apt-get install vsftpd; sudo service vsftpd start`

Telnet: Already installed and accessed via xinetd: `sudo service xinetd start`

SQL: Already installed: `sudo service mysql start`

Once these are installed and running, you can minimize this machine and start work on the testing VM.

> **Side-note: In normal circumstances, there is no reason to use ftp and telnet, they are incredibly outdated and insecure, and leak passwords everywhere. There is only one decent reason to use telnet:**
>
> **`sec@kali:~$ telnet towel.blinkenlights.nl`**
>
> **`You can use Ctrl+] then type quit to leave telnet.`**

To test the server, use the other machine to connect to ssh and the webserver. You shouldn't have any issues using SSH, and the default Apache2 website will show when you visit the machine in a browser.

PORT SCANNING

Port scanning is a process whereby software scans a network for active machines, then scans the ports at those IP addresses for active services. Knowing what services are running is useful for security auditing and penetration testing. If you're a hacker, it's useful to determine the weak points of a system before you attempt to gain entry.

Port scanning can be achieved by combining a number of network protocols, such as ping, and essentially flooding a machine with messages on different ports. Its responses to messages indicate what is, or is not, running. Over time, a detailed picture can be built up of the target machine, even as far as what software versions are installed. Knowing versions can lead to targeted exploits, like known buffer overflows in some versions of some packages.

Most of what we need for network analysis is combined into the Nmap tool. Let's start by assuming we know nothing about the network we're looking into, and would like to know if there are any machines on there we can analyse, so-called host discovery. Bring up your testing VM and initiate an Nmap scan of the network: `sudo nmap -sn 10.0.2.1/27`

```
sec@kali:~$ sudo nmap -sn 10.0.2.1/27
Starting Nmap 7.80 ( https://nmap.org ) at 2021-03-16 13:00 EDT
Nmap scan report for 10.0.2.1
Host is up (0.00042s latency).
MAC Address: 52:54:00:12:35:00 (QEMU virtual NIC)
Nmap scan report for 10.0.2.2
Host is up (0.00034s latency).
MAC Address: 52:54:00:12:35:00 (QEMU virtual NIC)
Nmap scan report for 10.0.2.3
Host is up (0.00030s latency).
MAC Address: 08:00:27:76:0F:28 (Oracle VirtualBox virtual NIC)
Nmap scan report for 10.0.2.4
Host is up (0.00072s latency).
MAC Address: 08:00:27:C4:98:AA (Oracle VirtualBox virtual NIC)
Nmap scan report for 10.0.2.15
Host is up.
Nmap done: 32 IP addresses (5 hosts up) scanned in 0.59 seconds
sec@kali:~$
```

The –sn flag instructs Nmap not to perform detailed scans of ports on these machines, just to return their IP addresses. Notice the types of devices Nmap has found, in this lab we are interested in the two Oracle VirtualBox virtual NIC devices, which are the two virtual machines we're using. Note the IP addresses if you don't already know them. If you can't see the device type, be sure to run using sudo — Nmap sometimes uses raw sockets and other techniques that require root privileges for its more detailed scans.

Now that we know which machine we want to target (we'll refer to 10.0.2.4 in this example, but you should be targeting the "server" kali VM) we can initiate a more detailed port scan: `sudo nmap -sV 10.0.2.4`

```
sec@kali:~$ sudo nmap -sV 10.0.2.4
Starting Nmap 7.80 ( https://nmap.org ) at 2021-03-16 13:01 EDT
Nmap scan report for 10.0.2.4
Host is up (0.00066s latency).
Not shown: 995 closed ports
PORT     STATE SERVICE    VERSION
21/tcp   open  tcpwrapped
22/tcp   open  ssh        OpenSSH 8.3p1 Debian 1 (protocol 2.0)
23/tcp   open  telnet     Linux telnetd
80/tcp   open  http       Apache httpd 2.4.43 ((Debian))
3306/tcp open  mysql      MySQL 5.5.5-10.3.23-MariaDB-1
MAC Address: 08:00:27:C4:98:AA (Oracle VirtualBox virtual NIC)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https:/
/nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 8.95 seconds
sec@kali:~$
```

This will take a minute. The -sV flag indicates that we want to try to obtain version information for the software running behind each port as well. (Your details may be different.) We've found out a lot about this machine! You'll see in a later lab that we can use this information to target attacks at specific ports. For now, we'll simply use Nmap as a test mechanism for our firewall.

IPTABLES

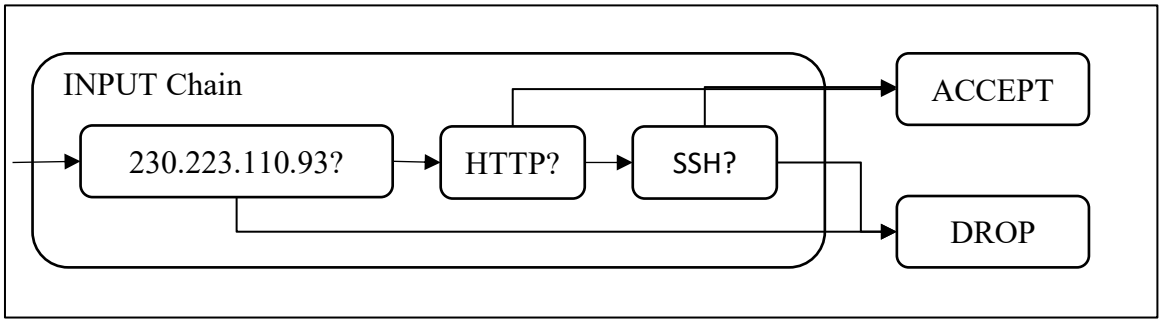
`iptables` is part of the core Linux packet filtering framework. `netfilter` acts as a set of hooks to allow modules (like with PAM from a different lab) to add functionality to the filtering. `iptables` acts as a set of rules that govern what happens to packets on the way in, through, and the way out. We could, for example, block all access in and out of a certain port — something that IT Services does for the Bittorrent protocol ports, for example. On the other hand, we could allow SSH connections in, but only from certain ports, or from the whole internet.

In this lab we'll add a set of `iptables` rules to our server machine to manage the services we just started. We'll operate a white list policy. That is, we'll block everything by default, then white list certain ports to allow access. The opposite is more risky, if we let everything through by default, we could leave ports open we aren't intending, we can't assume it'll cross our minds to block everything in the black list. A black list policy is more appropriate for something like blocking specific host IPs, for example of known botnet machines. This is something that denyhosts (http://denyhosts.sourceforge.net/) does.

TABLES, CHAINS AND RULES

`iptables` uses different tables for different aspects of a firewall. For example, there is a filtering table that performs general packet filtering, and a nat table that handles network address translation. In this lab, we will focus only on the filtering table. This table is the default, so we can just assume anything we do to the rules is happening on this table.

Within each table, there are different chains of rules that are used in different situations. In the filtering table, these are the INPUT, FORWARD and OUTPUT chains. Each chain is a set of rules that are applied one after another. These will ultimately result in a jump to a specific result for that packet, if the packet matches the criteria of the rule. Here is an example (simplified) diagram for a possible INPUT chain:



When a packet reaches this machine, it is passed to the iptables FILTERING table's INPUT chain for processing. In this example, the first rule checks whether the sender's IP address matches 230.223.110.93, if it does, the rule matches and the packet is immediately dropped. If the rule isn't matched, the next rule is tested, which determines whether or not the packet is an HTTP request (port 80), if it is, it is accepted. If not, the next rule is tested. The policy of this chain is to drop all packets that are not accepted by a rule, so if the SSH rule does not accept the packet, it is dropped at the end of the chain.

Notice that the ordering of these rules can affect the output. If the HTTP request was first, it would accept a port 80 request from 230.223.110.93 before it was rejected by the next rule. There is also a choice to be made with how we implement the drop policy. You can either set the chain itself to drop if no rules are matched, or you can simply add a "match all" drop rule at the end. There are subtle differences that we won't go into here, but we'll use the latter approach this time.

BUILDING A FIREWALL

Let's get started building a firewall for the Kali Server VM. All access to iptables can be achieved through the command line, but rather than fire off single commands, we'll write a script file that contains all of the commands. Load the lab5/buildfirewall.sh file in gedit (or any other editor), on the server machine. This file contains some comments that we'll use to structure the file. New rules are usually appended to the end of chains, so inserting a new rule at the start is more inconvenient. Since we're running in a script, we can simply clear

the entire rule table, then re-insert each rule one after another. We'll look at the input rules first, add the required lines into the file:

```
# Clear all current rules
iptables -F

# Input chain
# Rules to accept packets if they meet criteria

# Drop all remaining packets
iptables -A INPUT -j DROP
```

The -F flag means flush, add this at the beginning of your script. Next we want to add a rule to drop packets by default, so that we can begin to emulate the diagram above. Save and run this script now (don't forget sudo), then try and connect to the server using the other machine.

You can also check the current rules and policies using `iptables -L`. More interestingly, try running Nmap as before. Notice that it can't scan the other host at all (if it can, your firewall script isn't working!). This is because the firewall is dropping all packets, particularly Ping requests. This isn't the same as responding with a connection refused, which would at least let Nmap know there was a machine there.

Now that we're dropping everything by default, we should add some rules to let certain things through. We'll begin with SSH, where we'll want to accept packets from elsewhere. We'll add a rule for this:

```
# Rules to accept packets if they meet criteria
iptables -A INPUT -i eth0 -p tcp --dport ssh -j ACCEPT
```

`-A INPUT` : Specifies that we're using the INPUT chain (remember that the filtering table is used by default).
-i eth0 : This rule applies to packets coming in on eth0 (Ethernet adapter)
`-p tcp` : This rule applies to TCP packets.
`--dport 22` : This packet must have a destination port of 22.
`-j ACCEPT` : If this rule matches, accept the packet.

It's also good practice to allow incoming connections from established communication. This is in case things change port at some stage. Add this, too:

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

-m conntrack : Uses the connection tracking module, this is a stateful firewall module
--ctstate : A parameter for conntrack, specifying we allow all packets from established connections

Add these two rules, re-run the script, and you should now have SSH access to the server.

We're almost there for SSH. We should probably restrict output packets too — you never know when a rogue package will start communicating out of your machine! This is the same structure as with the input, we add a line that will drop all OUTPUT packets, then a line to permit established SSH packets (remember to re-run the script):

```
# Output chain
# Rules to accept packets if they meet criteria
iptables -A OUTPUT -o eth0 -p tcp --sport ssh -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT

# Drop all remaining packets
iptables -A OUTPUT -j DROP
```

SSH should still work, but other output ports are now blocked. Notice, for example, that we can't now initiate SSH connections from the server to elsewhere, because it requires its packets to already be established. If you Nmap the server now, use the `-Pn` flag to prevent it relying on ping. It should find SSH open, but not a lot else.

Unsurprisingly, we can't view the website on this server at the moment. Add the necessary HTTP rules now, they are identical to the ssh rules, except they use port 80. Also add loopback rules to the input and output chains, which allows the server to connect to itself on localhost:

```
iptables -I INPUT -i lo -j ACCEPT
iptables -I OUTPUT -o lo -j ACCEPT
```

We won't open up telnet, ftp or mysql because that's a security risk. The important thing, though, is that the firewall is now dropping on those ports, and an intruder has no way of accessing those services. If you needed to open them one day, you could just add the necessary lines.

For reference, your buildfirewall.sh script should look roughly like the one below. We've also included ICMP rules to allow us to ping the server, can you spot why this only allows pinging to the server, not from it?

```
# Clear all current rules
iptables -F

# Input chain
# Rules to accept packets if they meet criteria
iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -i eth0 -p tcp --dport ssh -j ACCEPT
iptables -A INPUT -i eth0 -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -I INPUT -i lo -j ACCEPT

# Drop all remaining packets
iptables -A INPUT -j DROP

# Output chain
# Rules to accept packets if they meet criteria
iptables -A OUTPUT -o eth0 -p tcp --sport ssh -j ACCEPT
iptables -A OUTPUT -o eth0 -p tcp --sport 80 -j ACCEPT
iptables -A OUTPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
iptables -I OUTPUT -o lo -j ACCEPT

# Drop all remaining packets
iptables -A OUTPUT -j DROP
```

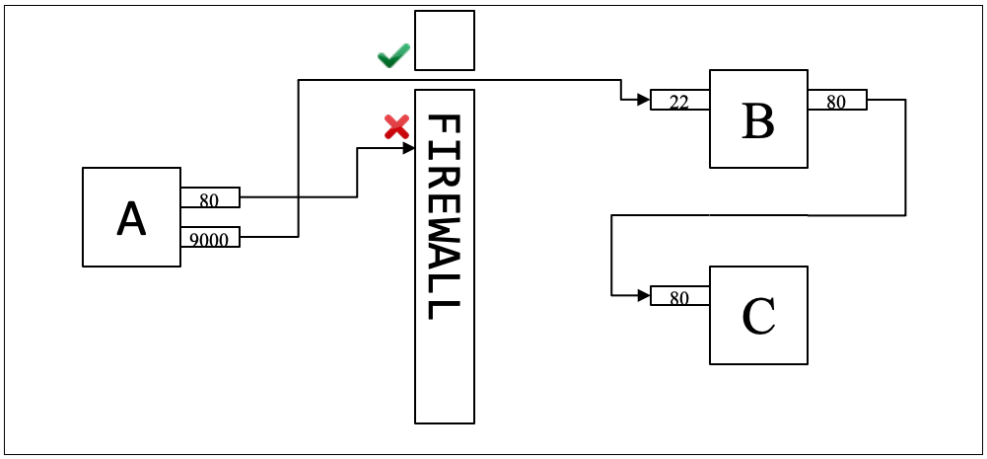
SSH TUNNELING

Sometimes you may find that a firewall upstream of your machine is blocking a port you need. The university network blocks most ports apart from vital services such as SSH and RDP (which now runs through a gateway). While often this results in trivial issues (such as no access to steam games), there can be more serious problems. Some software packages authenticate on unusual ports, and the University firewall means they assume they are unlicensed, and won't work. Here we'll briefly see how an SSH tunnel can be used to route packets past a firewall, avoiding any access restrictions.

On the **VM server machine**, run `sudo iptables -F` to disable the firewall (we need DNS and other features). Next on the **VM testing machine** run `sudo ~lab5/blockweb.sh` this will setup simple firewall rules that block packets out on port 80 and 443. This prevents web access from a browser, no packets out means no web requests! Try using the Firefox browser to access the web now (e.g. www.nottingham.edu.cn). You won't be able to access it on the test machine, but you will on the server machine.

We're looking at HTTP in this example, but this same tunneling principle applies to any port you wish to use. The principle is to redirect web requests through an encrypted ssh connection on port 22, which is usually open. Once the machine you are connected to receives the request it will forward on the packets, then return the response from the web

server. In essence, you are browsing the web as if directly on another machine, the proxy. This diagram illustrates the principle:



We're on machine A, and we want to communicate via port 80 with computer C, the webserver. The firewall blocks port 80, so we embed an HTTP communication channel within an encrypted SSH tunnel, which passes through the firewall on port 22. We assign port 9000 to run this SSH service. Machine B forwards any communications between B and C onto A. All we need to do is have access to a machine that isn't behind this firewall, and the ability to connect to it via SSH.

In this case, machine A is our testing machine, machine B is our server machine, and C is a web server e.g. google.co.uk. Connect via SSH to the server using the following command (your IP might be different):

```
sec@kali:~$ ssh -D 9000 -C 10.0.2.4
```

Once you've authenticated, leave this connection running, it's now providing a dynamic (-D) and compressed (-C) forwarding service on port 9000. All we need to do now is set up the browser to use a proxy. Open up Firefox, select Menu (≡) -> Preferences … come to the Network Settings and click "Settings…". Then select "Manual proxy configuration", SOCKS Host. Enter 127.0.0.1 and port 9000, and accept the settings. Try browsing the web now. If it's setup correctly, you should have full internet access, despite the fact that your machine can't send packets on port 80 or 443. As a last test, what happens if you close the SSH tunnel and attempt to browse the web?

CONCLUSION

In this lab session you've learned how to manage the Linux firewall through the `iptables` utility. We've also looked at Nmap, a port scanner, and the effect that firewall rules will have on the output of these programs. Finally we used an SSH tunnel to demonstrate that while a port is open, we can encrypt our packets through it, and gain access around firewall restrictions.