Assignment 2

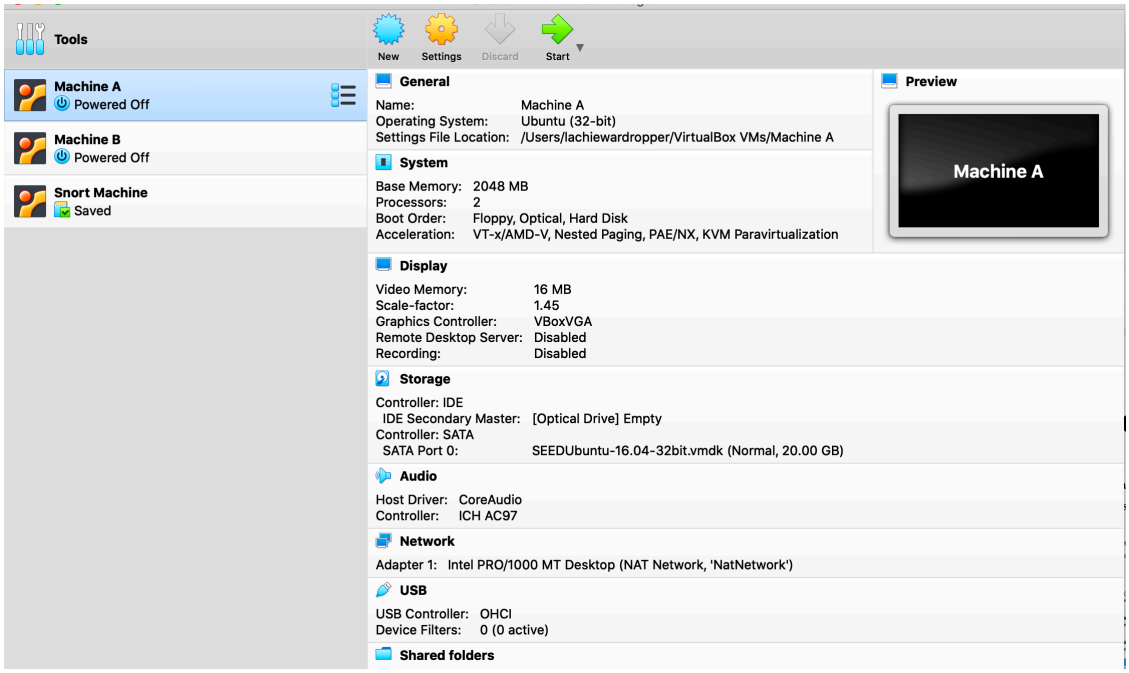
COMS3000 SEMESTER 2 2020

LACHLAN WARDROPPER – 44397580

Table of Contents

Introduction

This assignment was completed using a Virtual Machine running SEED Ubuntu16.04 VM(32-bit). The local machine used was a MacBook Pro, and a total of three virtual machines were created to complete the tasks in the lab.



Virtual machines used for lab tasks

Goals & Objectives

Iptables:

The goal of these tasks is to understand how Iptables can be used to control inbound and outbound connections between virtual machines and/or external sites. Additionally, the lab aims to show how different types of connections can be controlled through the use of the Iptables tools, including both TCP and ICMP.

Nmap:

The goal of these tasks is to understand how different Nmap scans are performed, as well as what they identify about a machine. Additionally, the tasks aim to show certain limitations of different Nmap scans based on how they evade firewall protection measures and filtering devices. By completing these lab tasks, it should become clear how normal, UDP and Xmas scans work to distinguish between open and closed ports on a machine.

Snort:

The goal of these tasks is to understand how Snort is used to control and alert a machine about a potential network risk. Specifically, these tasks should gain insight into how Snort can be used to counter the measures taken by Nmap scans to evade detection from machines. Additionally, a learning outcome of these tasks will be how a machine can be alerted when a specific web domain is accessed, as well as how a machine can be alerted when another machine fails to login to the SSH server.

Timeline

The tasks for this lab were completed on the following dates:

Iptables:

- Task 1: 21/10/20
- Task 2: 21/10/20
- Task 3: 21/10/20
- Task 4: 21/10/20

Nmap:

- Task 1: 21/10/20
- Task 2: 21/10/20
- Task 3: 21/10/20

Snort:

- Task 1: 22/10/20
- Task 2: 22/10/20
- Task 3: 22/10/20
- Task 4: 22/10/20
- Task 5: 22/10/20

Task 1 – Iptables

Task 1.1 – Prevent Machine A Doing telnet to Machine B

The first task that utilised Iptables involved preventing Machine A from establishing a telnet connection with Machine B. Firstly, two separate virtual machines were created and run, and the corresponding IP addresses were found using the '*ifconfig'* Linux command. From this, the respective addresses were 10.0.2.15 (Machine A) and 10.0.2.4 (Machine B). Next, it was clear that Machine A would require root privileges in order to alter restriction on outgoing telnet connections, and thus the '*su'* command was used (*Figure 1*).

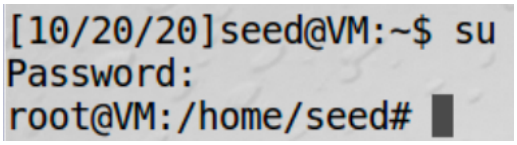


Figure 1: Root access in Machine A

Once root mode was established, the next step was to see what existing Iptables existed on the machine by using the *'-L'* tag. As shown in *Figure 2*, there were no manually created Iptables on the machine prior to commencing the task. Furthermore, *Figure 3* shows a telnet connection being established between Machine A and Machine B before Iptables were used to block outgoing connections. Thus, the results of the created Iptables can be verified as being the cause for any effects on establishing a telnet connection.

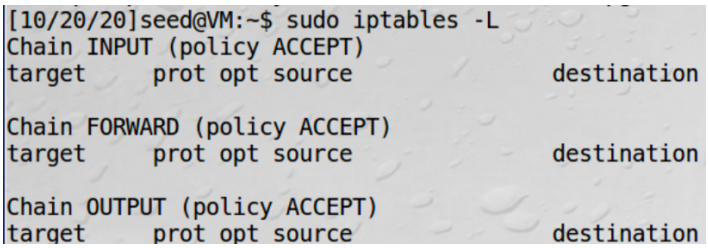


Figure 2: Existing Iptables prior to task

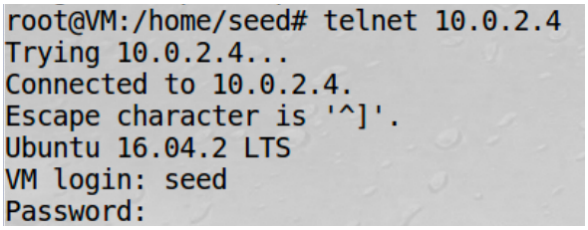


Figure 3: Establishing telnet connection before beginning task

Next, the Iptables manual was used to access all possible commands that could block outgoing telnet connections. It was found that the most relevant tags were the *'-A', '-p', '-s', '—dport'* and *'-j'* tags. The '*-A'* command was used in order to append the rule being created to the end of the selected chain [1]. The *'-p'* parameter defined the protocol of the rule or packet to check. For this task, the protocol being used was TCP, since telnet uses TCP to connect to a remote system. The *'-s'* parameter was used to define the source address, or in this case the IP address of Machine B (10.0.2.4). The *'—dport'* tag specifies the destination port, and thus was set to *telnet*, in order to allow the firewall to block requests going to this service. Finally, the *'-j'* parameter was used to define what to do if the packet matches the description of the Iptables. In this case, it was set to *REJECT*, which ensures that Machine A will send back an error packet in response to the matched packet. The full Iptables rule can be seen below in *Figure 4*.

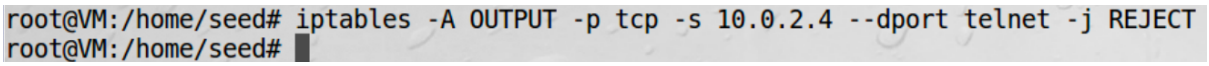


Figure 4: Iptables rule used to stop Machine A establishing telnet connection with Machine B

Once the Iptables rule was created, the next step was to attempt to establish a connection with Machine B by using the IP address found earlier (10.0.2.4). As expected, due to the Iptables rule, the connection was refused (*Figure 5*).

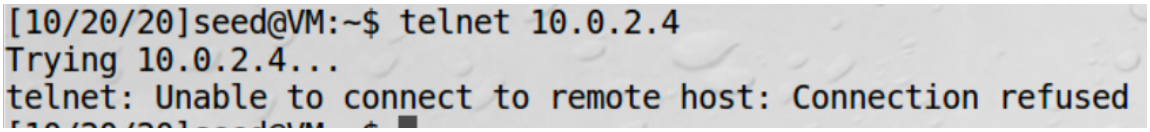


Figure 5: Rejected telnet request

The 'Connection refused' output also explicates the success of the command, as it shows the REJECT clause of the command working. In order to confirm the creation of the Iptables command, the *'-L'* tag was used to list all existing Iptables. As shown, the created command is listed, and shows that any telnet connections with 10.0.2.4 should be rejected (*Figure 6*).

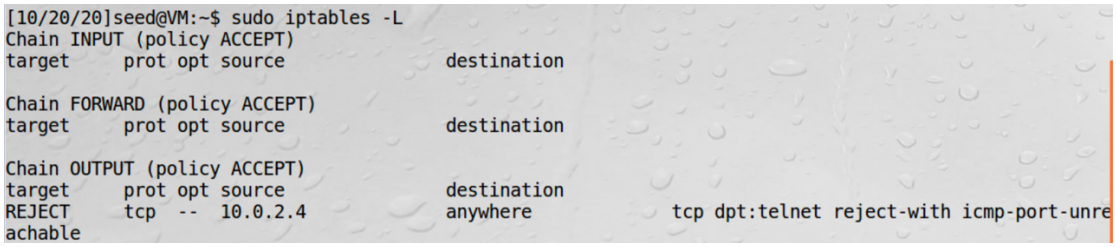


Figure 6: All Iptables commands

Thus, it is clear that Machine A was successfully blocked from establishing a telnet connection with Machine B. Additionally, the validity of the outcome was confirmed by establishing a telnet connection prior to creating the rule that blocked an outgoing connection from being made to the IP address of Machine B.

Task 1.2 – Prevent Machine A Being telnet from Machine B

This task again required the use of the Iptables firewall protection, however instead preventing Machine B from establishing a connection with Machine A. Much like the last task, the first step was to use the *ifconfig* command to find the IP addresses of each machine. Machine A (10.0.2.15) and Machine B (1.0.2.4) were found to be unchanged from the previous task. The same tags and parameters were used in this task as in *Task 1.1*, however the values were altered in order to prevent incoming telnet connections from being established. Firstly, the *'-A'* tag was altered from a value of OUTPUT to INPUT, which specified that this rule must consider incoming packets rather than outgoing ones. The *'-p', '-s'* and *'—dport'* tags remained the same as in *Task 1.1,* since this Iptable rule again had to be used over a telnet connection with Machine B (10.0.2.4). The *'-j'* values was however altered from REJECT to DROP, which instead meant that any incoming packets should be blocked from Machine A, rather than rejected. The full Iptables command can be seen in *Figure 7* below.

```
[10/20/20]seed@VM:~$ sudo iptables -A INPUT -p tcp -s 10.0.2.4 --dport telnet -j DROP
```

Figure 7: Iptables rule used to block Machine B establishing telnet connection with Machine A

In order to validate the Iptables rule was created, the *'L'* tag was used to list all existing Iptable commands. As shown, the newly created rule had been successfully stored, and specified to drop and incoming tcp packets received from the IP address 10.0.2.4 (*Figure 8*).

```
[10/20/20]seed@VM:~$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       tcp  --  10.0.2.4             anywhere             tcp dpt:telnet

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Figure 8: Existing Iptables rules on Machine A

Next, a telnet connection with Machine A (10.0.2.15) was created from Machine B, and as shown by the output in *Figure 9*, eventually times out and fails to be established. This explicates that Machine A successfully dropped the incoming TCP packets from Machine B, resulting in an eventual timeout and unsuccessful attempt to establish a telnet connection.

```
[10/20/20]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
telnet: Unable to connect to remote host: Connection timed out
```

Figure 9: Unsuccessful telnet connection to Machine A from Machine B

Thus, this task successfully identified how Iptables firewall protection can be used to prevent an incoming telnet connection from a specific machine using by its IP address.

Task 1.3 – Prevent Machine A Accessing External Website

This task required the use of Iptables to block Machine A (10.0.2.15) from accessing an external website. It was decided that *www.youtube.com* would be used for the task. In order to validate that the machine had access to YouTube, it was accessed via the web and was shown to be working as intended (*Figure 10*).

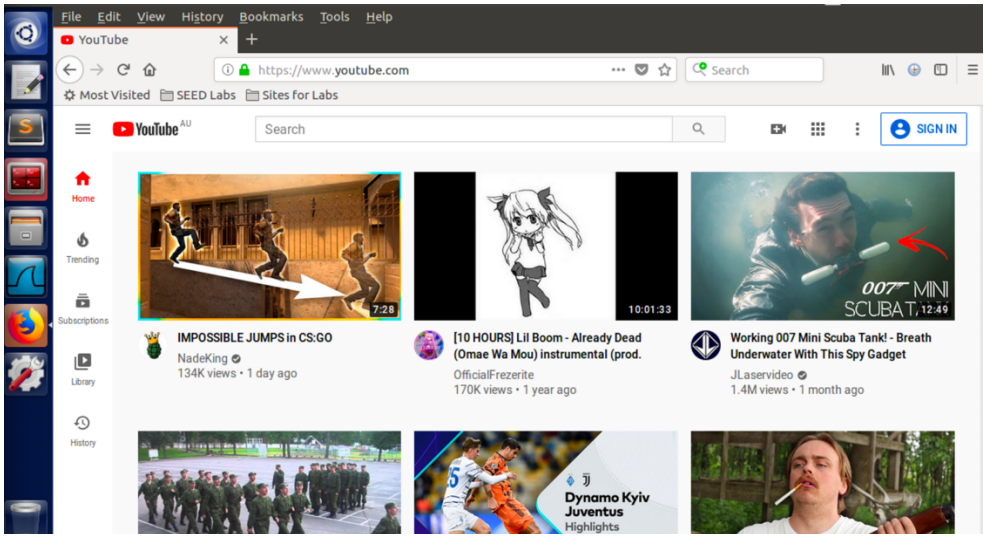


Figure 10: YouTube

In order to find the IP address range used by YouTube, the first step was to use the Linux *ping* command to access the site from the shell and find the IP address returned by the server. It shows the outgoing and incoming packets between the machine and the server as observed in *Figure 11.*

```
root@VM:/home/seed# ping www.youtube.com
PING youtube-ui.l.google.com (142.250.66.206) 56(84) bytes of data.
64 bytes from syd09s23-in-f14.1e100.net (142.250.66.206): icmp_seq=1 ttl=111 time=55.5 ms
64 bytes from syd09s23-in-f14.1e100.net (142.250.66.206): icmp_seq=2 ttl=111 time=54.3 ms
64 bytes from syd09s23-in-f14.1e100.net (142.250.66.206): icmp_seq=3 ttl=111 time=67.1 ms
64 bytes from syd09s23-in-f14.1e100.net (142.250.66.206): icmp_seq=4 ttl=111 time=52.5 ms
64 bytes from syd09s23-in-f14.1e100.net (142.250.66.206): icmp_seq=5 ttl=111 time=49.9 ms
^C
--- youtube-ui.l.google.com ping statistics ---
```

Figure 11: 'Pinging' of YouTube server

Next, whilst in root mode, the Iptables firewall protection was used to specify Machine A to reject accessing the specific IP address of YouTube. This IP address was found in the ping command and returned a value of 142.250.66.205. Thus, this value was used as the *'-d'* tag, which specifies the destination address to block a connection with. Additionally, the *'A'* parameter was defined as OUTPUT, since the task required Machine A to prevent request packets being sent to the YouTube server. Additionally, the *'-j'* tag was set to REJECT, which meant that the message displayed when trying to establish a connection with the YouTube servers would be '*Destination Port Unreachable'*. The full Iptables rule can be seen below in *Figure 12,* including the tags and associated values mentioned above.

```
root@VM:/home/seed# sudo iptables -A OUTPUT -d 142.250.66.206 -j REJECT
```

Figure 12: Iptables rule to block connection with YouTube server

In order to validate the Iptables rule was successfully creates, the *'-L'* parameter was used to list all existing Iptables rules. As is evident in *Figure 13,* the rule was successfully created and shows that any outgoing requests to the specified IP address should be rejected with the *'Port Unreachable'* message.

```
root@VM:/home/seed# sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
REJECT     all  --  anywhere             syd09s23-in-f14.1e100.net  reject-with icmp-port-unreachable
```
Figure 13: Existing Iptables on Machine A

In order to confirm the Iptables host-based firewall would successfully block Machine A from accessing the address, the *ping* command was again used (*Figure 14*).

```
root@VM:/home/seed# ping 142.250.66.206
```
Figure 14: Pinging YouTube server

As expected, the result of the *ping* command was a stream of '*Destination Port Unreachable'* messages every time the machine attempted to send a packet to the IP address. Consequently, it was evident that the newly created Iptables rule was working as intended. Additionally, the result was verified by reloading the YouTube homepage, which displayed an "*Unable to connect"* message. Consequently, it is clear that access to the external website was successfully blocked.

```
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destination Port Unreachable
From 10.0.2.15 icmp_seq=1 Destinat^C

--- 142.250.66.206 ping statistics ---
0 packets transmitted, 0 received, +18468 errors
```
Figure 15: Unsuccessful sending of packets to YouTube server

Thus, the key lesson from this task is how Iptables can be used to prevent a local Machine from accessing an external website by using the IP address specified by the *ping* command.

Task 1.4 – Prevent Machine A Response to ICMP Requests

This task required the use of two virtual machines, Machine A (10.0.2.15) and Machine B (10.0.2.4). Specifically, it again used Iptables to prevent Machine A from responding to ping requests from Machine B. Initially, Machine B was used to send a ping request to confirm the validity of the statements used in this task. As clear in *Figure 16*, the ping request was successful, and Machine A is reciprocating the packets sent by Machine B.

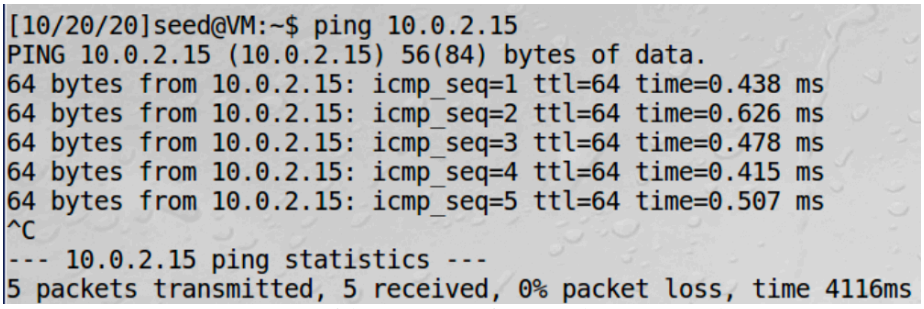


Figure 16: Successful ping request from Machine B to Machine A

Once it was clear that Machine A was responding to ICMP requests from Machine B, the next step was to again use Iptables to define a rule that would prevent this from occurring. Firstly, the *'-A'* parameter was used to append the rule being created to the end of the selected chain. The next value was set to INPUT for this example. This was due to the fact the Machine A was to be receiving ICMP requests, and consequently the Iptables rule had to act on incoming requests rather than outgoing ones. Next, the *'-p'* tag used to define the protocol of the rule or packet to check was set to a value of ICMP, since this was the type of request Machine A was to reject. Next, the *'—icmp-type'* was set to a value of *echo-request*. This value corresponds to the type of request used in a ping request, and consequently had to be set in order for the Iptables rule to respond to the request of Machine B. Finally, the *'-j'* tag was set to DROP, which specified for Machine A to ignore the incoming requests, rather than reject them. The full Iptables command can be seen in *Figure 17* below.

```
root@VM:/home/seed# sudo iptables -A INPUT -p icmp --icmp-type echo-request -j DROP
```

Figure 17: Iptables command to block ICMP request from Machine B

Next, to confirm the Iptables rule was successfully created, the *'-L'* tag was used to list all existing rules on Machine A. As shown by *Figure 18*, the rule was successfully initialised, and shows that any ICMP echo requests should be dropped from any destination machine.

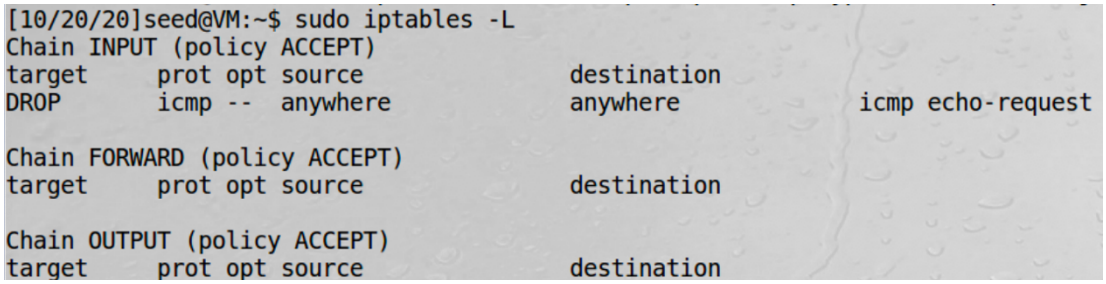


Figure 18: Successfully created Iptables rule

In order to test the created Iptables rule, Machine B was used to ping Machine A (10.0.2.15). As shown by the console output in *Figure 19*, 49 total packets were received by Machine A, however 0% were acknowledged. This is shown by the 100% packet loss value, which verifies that the DROP command worked as intended.

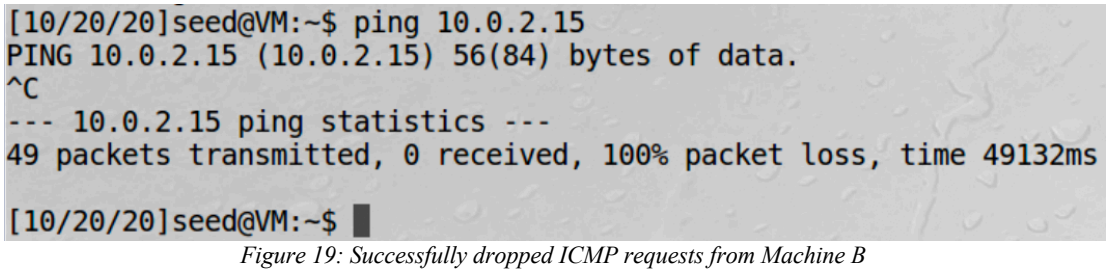


Figure 19: Successfully dropped ICMP requests from Machine B

In summary, this task showed how Iptables can be utilised to drop incoming ICMP requests from other machines. It also showed how the various parameters and tags specified in the Iptables manual allows one to highly customise the conditions for the host-based firewall to act.

Task 2 – Nmap

Task 2.1 – Nmap Normal Scan

The first Nmap task required the use of the Network Mapper tool to conduct a normal scan on the virtual machine. The Nmap tool is an open source tool used for network scanning and vulnerability discovery [SOURCE]. Since it does not come pre-installed on the machine, the first step was to install the library using the command shown in *Figure 20.* The command was executed in root mode to ensure that Nmap had complete access to files in the virtual machine.

```
root@VM:/home/seed# sudo apt-get install nmap
```

Figure 20: Installation of Nmap

Next, the normal scan was executed by using the virtual machine's IP address (10.0.2.15) as a specification for the target port to scan. Once the command was executed, the results were outputted to the console screen, specifying information regarding the machine's OS network. Firstly, the output displayed information showing there was a total of 994 closed ports, as well as information about ports in an 'open' state. There were six total open ports, being 21, 22, 23, 53, 80 and 3128. Additionally, the normal scan showed information pertaining to the service each port was using, which helps gain an understanding of how each port is being used by the virtual machine. The command used to scan the machine as well as the output can be seen in *Figure 21* below.

```
root@VM:/home/seed# nmap 10.0.2.15

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-20 22:42 EDT
Nmap scan report for 10.0.2.15
Host is up (0.0000040s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
23/tcp   open  telnet
53/tcp   open  domain
80/tcp   open  http
3128/tcp open  squid-http

Nmap done: 1 IP address (1 host up) scanned in 6.03 seconds
root@VM:/home/seed#
```

Figure 21: Nmap normal scan command and output

From this task, it became clear how the network mapper tool Nmap can be used to identify how different ports are used to bind services, which provided key insights into how a machine's vulnerabilities can be exploited through specific ports.

Task 2.2 – Nmap UDP Scan

This task again used the network scanner tool Nmap, however this time to perform a UDP scan rather than a normal scan. This particular scan is used to identify exploitable UDP services, which can often be a target by hackers since UDP scanning is overlooked by many people due to the longer amount of time required to complete [2]. The scan works by sending UDP packets to every targeted port. For the majority of ports, the packet will be empty, however for some

will contain a 'protocol-specific' payload [2]. Based on the information received back, the port is assigned a state of open, open filtered, closed or filtered. The UDP scan is initialised with the *'sU'* tag. The Nmap scan as well as the output can be seen in *Figure 22.*

```
root@VM:/home/seed# nmap -sU 10.0.2.15

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-20 22:43 EDT
Nmap scan report for 10.0.2.15
Host is up (0.0000060s latency).
Not shown: 996 closed ports
PORT     STATE         SERVICE
53/udp   open          domain
68/udp   open|filtered dhcpc
631/udp  open|filtered ipp
5353/udp open|filtered zeroconf

Nmap done: 1 IP address (1 host up) scanned in 2.85 seconds
root@VM:/home/seed#
```

Figure 22: Nmap UDP scan and output

As shown in the Nmap scan report, the output shows 996 closed ports, and four ports not in a closed state (53, 68, 631, 5353). Port 53 shows to be in an open state, which means that it responded to the UDP packet sent by the scan. More interestingly are the other three ports, which are shown to be in an open filtered state. A key challenge with UDP scanning is that open ports are more likely to respond to protocol-specific payloads, rather than empty probes which are used for the majority of ports. Thus, if no response is established, Nmap is unable to determine whether a port is opened or filtered, hence the state 'Open/Filtered'. The main reason for this occurring is because firewalls and filtering devices may drop packets without responding to the UDP packet, which means that the Nmap scan has no way of knowing if the port is indeed open.

In order to gain a better understanding of how the UDP scan works, *Wireshark* was run alongside the Nmap scan, where it then became clear to see that certain destination ports were unreachable under an ICMP protocol. This means that the empty or protocol-specific payloads being sent by the UDP scan were not being received by the port, implying that it is closed. This can be seen in *Figure 23.*

	Source	Destination	Protocol	Length	Info
5:27.1113142…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:27.1113175…	10.0.2.15	10.0.2.15	UDP	44	39031 → 1234 Len=0
5:27.1113196…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:27.1113229…	10.0.2.15	10.0.2.15	UDP	44	39031 → 20742 Len=0
5:27.1113249…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:27.1113281…	10.0.2.15	10.0.2.15	UDP	44	39031 → 1026 Len=0
5:27.1113302…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:27.1113531…	::1	::1	UDP	64	45736 → 47179 Len=0
5:28.2150257…	10.0.2.15	10.0.2.15	UDP	44	39031 → 1038 Len=0
5:28.2150486…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:28.2150736…	10.0.2.15	10.0.2.15	UDP	44	39031 → 5555 Len=0
5:28.2150798…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:28.2150895…	10.0.2.15	10.0.2.15	UDP	44	39031 → 47808 Len=0
5:28.2150950…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:28.2151032…	10.0.2.15	10.0.2.15	UDP	44	39031 → 664 Len=0
5:28.2151084…	10.0.2.15	10.0.2.15	ICMP	72	Destination unreachable (Port unreachable)
5:28.2151167…	10.0.2.15	10.0.2.15	UDP	44	39031 → 23531 Len=0

Figure 23: Wireshark output showing closed ports

Furthermore, if a specific port is investigated further inside *Wireshark*, then it becomes even clearer that this is the case. By looking at information about the UDP request, it can be seen that specific ports are unverified by the scan, meaning that the packet was not able to be received. Consequently, these are confirmed to be closed by Nmap, which is then outputted (as seen in *Figure 22*).

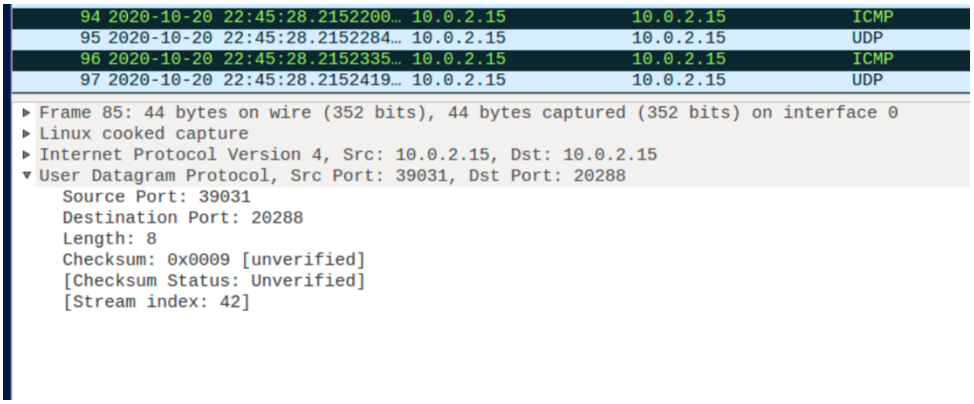


Figure 24: Wireshark output of specific port UDP request

In summary, this tasked showed how a UDP scan can be used to determine whether specific ports are open or closed. This is done through Nmap sending either empty packets or protocol-specific payloads depending on the port and waiting for a response. Additionally, it was shown that a limitation to the Nmap UDP scan is that the firewalls or filtering devices of some destinations ignore the packets, meaning that it is not decisive whether a port is open or closed.

Task 2.3 – Nmap Xmas Scan

The final Nmap task utilised the network mapper to perform an Xmas scan of the virtual machine. This scan is similar to a TCP FIN and NULL scan however with some slight differences. All three of these scans exploit a subtle loophole in the TCP RFC to differentiate between open and closed ports [3]. Essentially, as Nmap scans a system, and packet that does not contain SYN, RST or ACK bits should return an RST if the port is closed and no response if it is open. Thus, the Xmas scan is able to determine whether a port is open or closed. The slight difference with an Xmas scan compared to the other two is that the scan will set the FIN, PSH and URG flags, whereas a Null scan sets no bits and a FIN scan only sets the TCP FIN bit.

The scan works by slipping through specific non-stateful firewalls and filtering routers. Generally, these firewalls protect TCP connections from being received. This occurs by the firewalls identifying packets with the SYN bit set and ACK cleared and blocking them. Thus, the Xmas scan is effective as it does not use the SYN bit, meaning firewalls do not detect it. A downside to the Xmas scan however is that it cannot distinguish open ports from filtered ports (much like the UDP scan in *Task 2.2*), and thus displays non-closed ports as *open filtered.* What this means is that any non-responsive ports are marked with this tag, as they could have either received the packets (open) or dropped them without any response (filtered). The scan was executed by using the *'sX'* tag as well as the IP address of the machine to scan (10.0.2.15). The full command as well as the console output can be seen in *Figure 25*. It must be noted that this scan was performed in root mode, to ensure that Nmap had complete access privileges for the machine.

```
root@VM:/home/seed# nmap -sX 10.0.2.15

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-20 22:48 EDT
Nmap scan report for 10.0.2.15
Host is up (0.000042s latency).
Not shown: 994 closed ports
PORT     STATE         SERVICE
21/tcp   open|filtered ftp
22/tcp   open|filtered ssh
23/tcp   open|filtered telnet
53/tcp   open|filtered domain
80/tcp   open|filtered http
3128/tcp open|filtered squid-http

Nmap done: 1 IP address (1 host up) scanned in 100.42 seconds
root@VM:/home/seed#
```

Figure 25: Nmap Xmas scan on virtual machine

As shown in the results, 994 ports were found closed, and 6 were returned with the '*open/filtered'* tag, meaning that they either received the packet or dropped it without acknowledging so. In order to gain a better understanding of how the Xmas scan worked, *Wireshark* was again used to observe the behaviour of the network mapper. The output in *Figure 26* shows the FIN, PSH and URG flags being sent to specific ports, which is what occurs during the Xmas scan. Hence, it is clear that the packets were successfully being sent to the machine.

```
10.0.2.15          TCP        56 51862 → 23 [FIN, PSH, URG] Seq=2824875789 Win=1024 Urg=0 Len=0
10.0.2.15          TCP        56 51862 → 22 [FIN, PSH, URG] Seq=2824875789 Win=1024 Urg=0 Len=0
10.0.2.15          TCP        56 51861 → 554 [FIN, PSH, URG] Seq=2824941324 Win=1024 Urg=0 Len=0
10.0.2.15          TCP        56 554 → 51861 [RST, ACK] Seq=0 Ack=2824941325 Win=0 Len=0
```

Figure 26: Wireshark output of Nmap Xmas scan

Furthermore, in order to understand the Xmas scan, a specific packet was observed in the *Wireshark* output. As expected, the Fin, Push and Urgent flags were set to 1, whilst all other flags were set to 0. Additionally, as is evident in *Figure 27,* the SYN flag is set to 0, which is what allows the Xmas scan to go unnoticed by firewall protection measures.

```
▸ Frame 9: 56 bytes on wire (448 bits), 56 bytes captured (448 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.15
▾ Transmission Control Protocol, Src Port: 51861, Dst Port: 22, Seq: 2824941324, Len: 0
    Source Port: 51861
    Destination Port: 22
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 2824941324
    Acknowledgment number: 0
    Header Length: 20 bytes
  ▾ Flags: 0x029 (FIN, PSH, URG)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..1. .... = Urgent: Set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
    ▸ .... .... ...1 = Fin: Set
      [TCP Flags: ·····U·P··F]
    Window size value: 1024
    [Calculated window size: 1024]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xf184 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
```

Figure 27: Packet information from Nmap Xmas scan

In summary, this task showed how a Nmap Xmas scan manipulates the setting of specific flags to avoid detection from firewalls, in order to check whether specific ports are closed or open. Additionally, the scan is unable to distinguish between an open or filtered port, since both outcomes lead to no response from the specific port.

Task 3 – Snort

Task 3.1 – Snort Nmap Normal Scan Detection

In order to complete the Snort tasks, the package first had to be installed using the command found in *Figure 28*. Snort is a network intrusion detection system and intrusion prevention system. It allows a user to define rules that detect and alert them when a certain type of packet is found. *Task 3.1* aims to use the Snort tool to detect and alert Machine A (10.0.2.9) of a Nmap normal scan being performed by Machine B (10.0.2.10).

```
[10/22/20]seed@VM:~$ sudo apt-get install snort
```

Figure 28: Installation of Snort package

Once Snort had been installed, and the configuration and network had been set to '*enp0s3'* and *'10.0.2.9'* respectively, the next step was to alter the *snort.conf* file. Specifically, the *HOME_NET* address was changed to the local IP address, and the *EXTERNAL_NET* variable was set to all network addresses that were not the *HOME_NET* (as seen in *Figure 29*).

```
ipvar HOME_NET 10.0.2.9/32

# Set up the external network addresses. Leave as "any" in most situations
ipvar EXTERNAL_NET !$HOME_NET
```

Figure 29: Altering configuration file for Snort

Once the configuration file had been altered to match the values found form the '*ifconfig'* command from Machine A, the next step was to define a rule to alert the machine of a Nmap normal scan occurring. The first step of this process was to access the rules file in a text editor, in order to create and save rules for Snort to use when the associated command was run from the console. *Figure 30* shows the command used to access the Snort local rules file.

```
[10/22/20]seed@VM:~$ sudo gedit /etc/snort/rules/local.rules
```

Figure 30: Command to access Snort local rules file

Inside the Snort local rules file, the next step was to define a rule that would alert Machine A when Machine B performed a Nmap normal scan. In order to do this, the Snort documentation was used to find the appropriate tags and parameters to define within the rule. Firstly, the 'alert' command was used to generate an alert using a selected alert method, before then logging the packet. Next, the TCP protocol was defined, since a Nmap normal scan sends RST and ACK packets via the TCP protocol to ports to test whether they are open. Next, the source IP address and port numbers were set to 'any', since the alert had to be initialised if any machine was to perform a Nmap normal scan. After, this, the destination IP address was set to the local network of Machine A (10.0.2.9), and 'any' was selected for the value of ports. This means that the alert should be activated if any port in Machine A receives a TCP packet from any port and any other local network. Finally, the 'msg' parameter was set to a relevant description explicating Snort had found an occurrence of a Nmap normal scan, and the rule was given a unique 'sid' to ensure that the rule could be identified given that many rules can exist in the *local.rules* file. The final rule created can be seen below in *Figure 31.*

```
# $Id: local.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
# ----------------
# LOCAL RULES
# ----------------
# This file intentionally does not come with signatures.  Put your local
# additions here.

alert tcp any any -> 10.0.2.9 any (msg: "This rule found Nmap Normal Scan"; sid 10000001;)
```

Figure 31: Snort rule to alert Machine A of Nmap normal scan

After the rule was created, a Nmap normal scan on the IP address of Machine A was performed from Machine B. The command used as well as the output of the Nmap normal scan from Machine B can be seen in *Figure 32.*

```
[10/22/20]seed@VM:~$ nmap 10.0.2.9

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-22 07:22 EDT
Nmap scan report for 10.0.2.9
Host is up (0.00075s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
23/tcp   open  telnet
53/tcp   open  domain
80/tcp   open  http
3128/tcp open  squid-http

Nmap done: 1 IP address (1 host up) scanned in 0.21 seconds
```

Figure 32: Nmap normal scan performed by Machine B

Before the Nmap normal scan was performed by Machine B, a Snort command was used in the console of Machine A to ensure that the scan would trigger the rule created above. This command again made use of relevant documentation from the Snort website in order to configure correctly. Firstly, the '-A' tag was used with a value of 'console' to specify that Snort should output alert data to the console screen instead of logging to files [4]. Next, the '-q' tag was used to specify Snort to run quietly, meaning that no initialisation information is displayed. Next, the '-u' and '-g' tags were used to change the default user and group IDs to Snort. Finally, the '-c' tag was added to specify which configuration file to use, and the '-i' tag was used to define the interface with the value '*enp0s3'*, since that is the interface local Machine A listens on. The full command with the described tags can be seen in *Figure 33*.

```
[10/23/20]seed@VM:~$ sudo snort -A console -q -u snort -g snort -c /etc/snort/snort.conf -i enp0s3
```

Figure 33: Command-line argument and tags to initialise Snort

Once Snort was running, the next step was to re-run the Nmap normal scan from Machine B and observe the output from the Machine A console (as specified in the tags of *Figure 33*). *Figure 34* shows that the created rule in the rules file successfully detected a Nmap normal scan from Machine B.

```
10/22-07:21:50.858689  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:50.858705  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:50.858927  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:50.872069  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:50.873865  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:51.183438  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
10/22-07:21:46.538949  [**] [1:10000001:0] This rule found Nmap Normal Scan [**]
 [Priority: 0] {TCP} 91.189.91.39:80 -> 10.0.2.9:37064
```

Figure 34: Snort output during Nmap normal scan

In order to confirm the success of the Snort rule, *Wireshark* was used to observe the packets being sent form Machine B. Upon inspecting a single packet, it was clear that the TCP protocol was being used to send packets to specific destination ports. Thus, *Wireshark* confirmed the successful outcome of the task, as shown in detail in *Figure 35*.

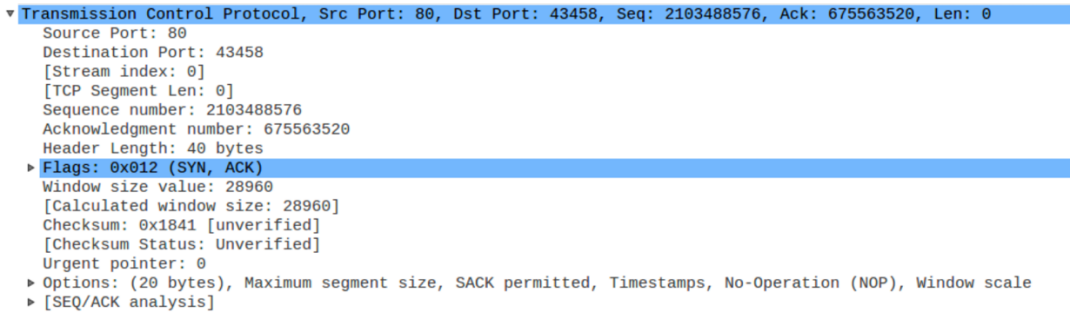


Figure 35: Inspection of single TCP packet

In summary, this task aimed to use the tools of the Snort package to define a rule that would detect and alert when a Nmap normal scan occurred. This was through defining the alert to occur when any TCP packets were found by Machine A on any ports and outputting an appropriate message.

Task 3.2 – Snort Nmap UDP Scan Detection

Much like *Task 3.1*, this tasked aimed to use Snort to define a rule that would detect a certain type of packet found when performing a Nmap scan. However, instead of a Nmap normal scan, this task required the use of Snort to alert Machine A when a Nmap UDP scan was occurring. The key differences between the two scans, as seen in *Section 2,* is that a UDP scan uses the UDP protocol instead of TCP. Thus, the rule defined for this task had to make use of this protocol. As in *Task 3.1*, the source IP address and ports were set to ‘any’ and ‘any’, meaning that the alert would occur if any other machine performed a UDP scan. Additionally, the destination IP address and ports were set to the local network ’10.0.2.9’ and ‘any’ respectively. This ensures that an alert will occur if UDP packets are found in any port of Machine A. Finally, an appropriate alert message and unique ‘sid’ were assigned to the rule, to ensure it was distinguishable from previous alerts. The full rule can be seen in *Figure 36*.

```
alert udp any any -> 10.0.2.9 any (msg: "This rule found Nmap UDP Scan"; sid: 10000003;)
```

Figure 36: Snort rule to alert of UDP scan

Once the rule had been created and saved, the same command-line arguments as were used in *Task 3.1* initialised Snort to begin detecting for a Nmap UDP scan. This command ensured that initialisation messages were suppressed, all output was to the console, and that the correct

configuration file was used to reference any rules. Next, from Machine B, a Nmap UDP scan was initialised using the 'sU' tag, and the destination IP address was set to the address of Machine A (10.0.2.9). After the scan was initialised, the output from the Machine A console clearly showed the rule had triggered multiple alerts, which can be seen in *Figure 37*.

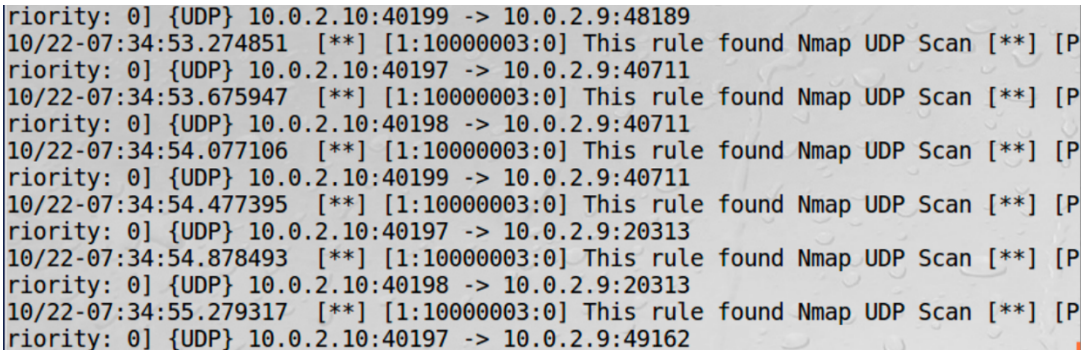


Figure 37: Alert showing Snort detected UDP scan

In order to confirm the success of the created rule, *Wireshark* was again used, where it was observed that ports were receiving UDP packets. In addition, since it is known from previous tasks that the machine sends ICMP packets back if the port is closed, this was also found in *Wireshark*. In order to extend this task, it could have been possible to create an alert to notify the console of these returning ICMP packets being sent. The *Wireshark* output at the time of the Nmap UDP scan can be seen below in *Figure 38*.

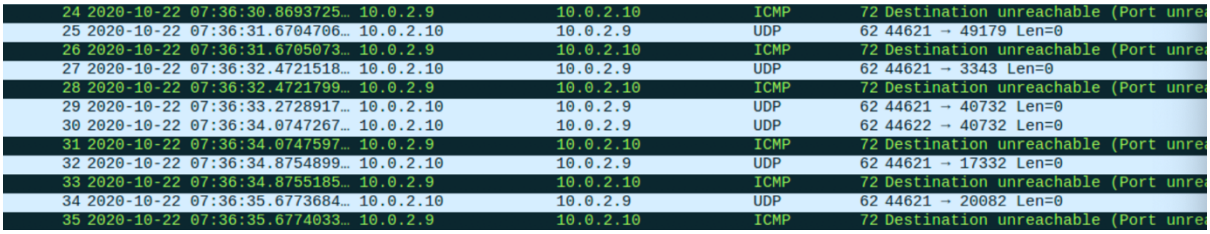


Figure 38: Wireshark during Nmap UDP scan

In summary, this task required the use of Snort to detect a Nmap UDP scan from a separate machine. The results clearly showed the rule successfully achieved this, by searching for incoming UDP packets from any other destination IP address or port. This task shows how Snort can be used to protect a machine from possible attacks by alerting a user of any scans to identify potential threats.

Task 3.3 – Snort Nmap Xmas Scan Detection

The final Nmap related task required the use of Snort to detect and alert Machine A when an Xmas scan was being performed by another machine. Much like the previous tasks, the first step was to access the *local.rules* file and define a new rule that would perform this action. In order to ensure the rule created would correctly identify an Xmas scan, the information found in *Task 2.3* was used. That is, an Xmas scan avoids firewalls and detection devices by setting the FIN, PSH, and URG flags. This is because the majority of firewalls search for SYN flags being sent in a TCP packet. Thus, when defining the rule seen in *Figure 39*, this had to be considered.

Like the previous two tasks, the created rule specified the IP address and ports of the external machine to be 'any' and 'any'. Additionally, the receiving machine values were assigned the local network IP address (10.0.2.9), and port numbers 'any', since an Xmas scan sends packets

to all ports. Furthermore, the TCP protocol was defined in this rule, since that is the protocol an Xmas scan uses to send packets to ports. As stated above, the flags FIN, PSH, and URG are set within a TCP packet in an Xmas scan. Thus, these flags should be searched within all packets being sent to the destination machine. Snort allows for this rule to be set using the 'flags' parameter, where the values 'FPU' was assigned. Finally, an appropriate message and unique 'sid' were assigned to the rule, to ensure it was clear whether an alert was successful. The full command is shown in *Figure 39*.

```
#
# LOCAL RULES
# ---------------
# This file intentionally does not come with signatures.  Put your local
# additions here.

alert tcp any any -> 10.0.2.9 any (msg: "This rule found Nmap Xmas Scan"; flags: FPU; sid 10000006;)
```

Figure 39: Snort rule to detect Xmas scan

Next, snort was initialised in the console using the same command as found in *Figure 33* (see *Task 3.1*), and a Nmap Xmas scan began on Machine B. This used the '-sX' tag to specify the type of scan to be an Xmas scan, and the destination IP address was set as the local network of Machine A. This scan as well as the output can be seen in *Figure 40*.

```
root@VM:/home/seed# nmap -sX 10.0.2.9

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-22 07:42 EDT
Nmap scan report for 10.0.2.9
Host is up (0.00064s latency).
Not shown: 994 closed ports
PORT     STATE         SERVICE
21/tcp   open|filtered ftp
22/tcp   open|filtered ssh
23/tcp   open|filtered telnet
53/tcp   open|filtered domain
80/tcp   open|filtered http
3128/tcp open|filtered squid-http
MAC Address: 08:00:27:5A:F8:8D (Oracle VirtualBox virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 98.49 seconds
```

Figure 40: Nmap Xmas scan

Whilst the Xmas scan was occurring from Machine B, the console output from Machine A was observed. As explicated in *Figure 41*, the output suggests that Snort successfully detected the scan using the rule defined above.

```
10/22-07:43:11.786648  [**] [1:1228:7] SCAN nmap XMAS [**] [Classification: Atte
mpted Information Leak] [Priority: 2] {TCP} 10.0.2.10:64963 -> 10.0.2.9:5060
10/22-07:43:11.867023  [**] [1:10000006:0] This rule found Nmap Xmas Scan [**] [
Priority: 0] {TCP} 10.0.2.10:64964 -> 10.0.2.9:6543
10/22-07:43:11.867023  [**] [1:1228:7] SCAN nmap XMAS [**] [Classification: Atte
mpted Information Leak] [Priority: 2] {TCP} 10.0.2.10:64964 -> 10.0.2.9:6543
10/22-07:43:11.947782  [**] [1:10000006:0] This rule found Nmap Xmas Scan [**] [
Priority: 0] {TCP} 10.0.2.10:64963 -> 10.0.2.9:55056
10/22-07:43:11.947782  [**] [1:1228:7] SCAN nmap XMAS [**] [Classification: Atte
mpted Information Leak] [Priority: 2] {TCP} 10.0.2.10:64963 -> 10.0.2.9:55056
10/22-07:43:12.027801  [**] [1:10000006:0] This rule found Nmap Xmas Scan [**] [
Priority: 0] {TCP} 10.0.2.10:64963 -> 10.0.2.9:1183
```

Figure 41: Snort output during Nmap Xmas scan

Furthermore, the *Wireshark* output shown in *Figure 42* shows that the Xmas scan sent TCP packets with the FIN, PSH and URG flags set. Thus, this means that the Snort rule and the

'flags' parameter defined successfully distinguished these packets from other TCP packets. Hence, it is clear that this task was successful in detecting and alerting Machine A of an incoming Nmap Xmas scan from another machine.

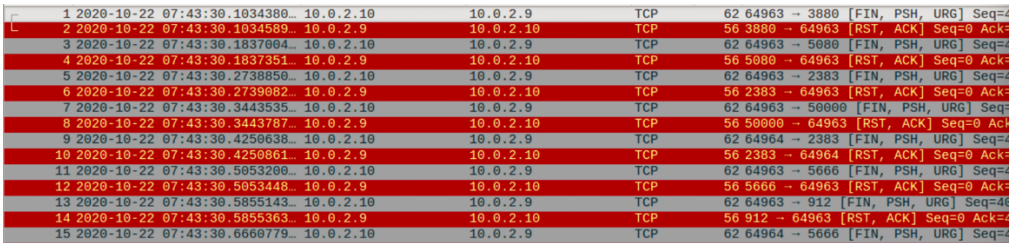


Figure 42: Wireshark output during Nmap Xmas scan

In summary, this task showed how Snort can be utilised to detect and alert a machine when an Xmas scan is occurring. This is incredibly valuable, given that the Xmas is designed to be able to evade most firewalls and detection methods. Thus, it acts as a much more reliable way of checking whether a computer is vulnerable to an attack through this method.

Task 3.4 – Snort Accessing Website Alert

This task aimed to use Snort as a detection tool to alert a machine when it is accessing a specific external website. Specifically, for this task the external website was defined as '*staff.uq.edu.au*'. Hence, the first step was to find the IP address range of this website's server, by using the Linux *ping* command. As shown by the output in *Figure 43*, the IP address of the website's server was found to be *130.102.184.123*.

```
[10/22/20]seed@VM:~$ ping staff.uq.edu.au
PING ng-prod.drupal.uq.edu.au (130.102.184.123) 56(84) bytes of data.
64 bytes from ng-cms.lb.dc.uq.edu.au (130.102.184.123): icmp_seq=1 ttl=234 time=
29.3 ms
64 bytes from ng-cms.lb.dc.uq.edu.au (130.102.184.123): icmp_seq=2 ttl=234 time=
62.8 ms
64 bytes from ng-cms.lb.dc.uq.edu.au (130.102.184.123): icmp_seq=3 ttl=234 time=
44.1 ms
64 bytes from ng-cms.lb.dc.uq.edu.au (130.102.184.123): icmp_seq=4 ttl=234 time=
42.6 ms
```

Figure 43: Pinging of staff.uq.edu.au server

The snort rule defined in *Figure 44* had to make use of the IP address found from pinging the *staff.uq.edu.au* server, since this had to be the destination address to trigger an alert. However, it was found that the *staff.uq.edu.au* website is in fact encrypted, which meant that specifying the IP address as the one found above would not work. This is because the TCP packets to the source cannot be filtered, meaning that the rule would not pick them up. Thus, the 'content' tag was used and assigned a value of *'130.102.184.123'*. This ensured that only access to the *staff.uq.edu.au* website would be detected by the machine, however, would not be affected by the encryption of packets. Furthermore, because this content was defined elsewhere, the source IP address was set to 'any'. Without defining the value of content, this would have triggered the rule for any website. Additionally, the source IP address was set to the local machine network address 10.0.2.9. Since it is known that a web connection used TCP packets, this was an obvious choice of parameters when defining the rules protocol. Finally, an appropriate message and 'sid' were added, to ensure the rule was identifiable. The full rule located in the *local.rules* file is shown below in *Figure 44*.

```
# LOCAL RULES
# ----------------
# This file intentionally does not come with signatures.  Put your local
# additions here.

alert tcp 10.0.2.9 any -> any any (content: "130.102.184.123"; msg: "Warning: This machine trying to access staff.uq.edu.au";
sid: 10000006;)
```

Figure 44: Alert to detect access of staff.uq.edu.au website

Before Snort was initialised, the *staff.uq.edu.au* website was opened in the virtual machine, where it is shown to be working as expected in *Figure 45*.

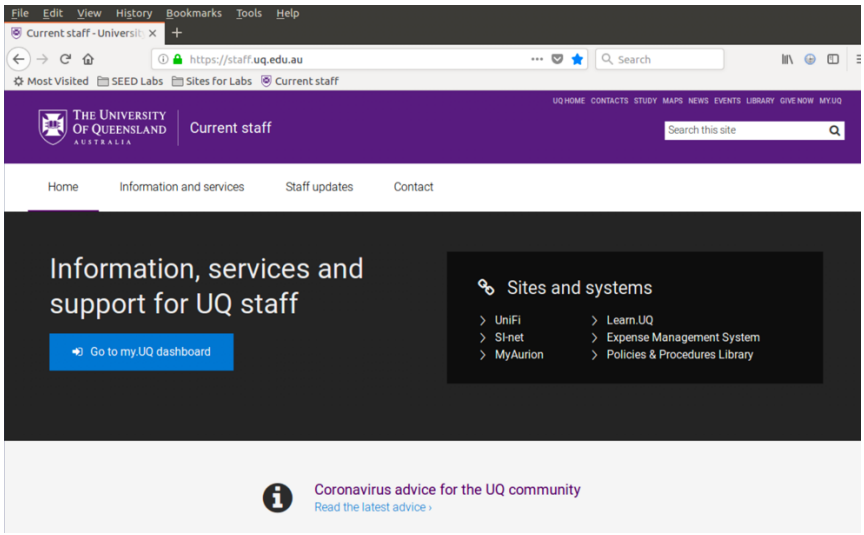


Figure 45: The staff.uq.edu.au website

Next, snort was initialised using the same command-line arguments as in all previous tasks, which ensured that any initialisation messages would be suppressed, the output of alerts would be sent to the console, and that the correct configuration file was selected. As is shown in *Figure 46*, the rule created worked as expected, with any outgoing TCP packets to the *staff.uq.edu.au* website being detected by Snort, since the IP address matched the *'contents'* tag in the rule definition.

```
root@VM:/home/seed# sudo snort -A console -q -u snort -g snort -c /etc/snort/sno
rt.conf -i enp0s3
10/23-8:57:45.345326 [**] [1:100000000:1] Warning: This machine is tring to acce
ss staff.uq.edu.au [**] [Priority: 0] {TCP} 10.0.2.9 -> 130.102.184.123
10/23-8:57:45.568674 [**] [1:100000000:1] Warning: This machine is tring to acce
ss staff.uq.edu.au [**] [Priority: 0] {TCP} 10.0.2.9 -> 130.102.184.123
10/23-8:57:45.823497 [**] [1:100000000:1] Warning: This machine is tring to acce
ss staff.uq.edu.au [**] [Priority: 0] {TCP} 10.0.2.9 -> 130.102.184.123
```

Figure 46: Snort output showing detection of Machine A accessing staff.uq.edu.au website

In summary, this task has shown how useful Snort can be in detecting whether a machine is accessing a specific website. This is because of the ability to interpret unencrypted TCP packets by using the *'contents'* tag. This tool has the ability to be used to track the movements of a person's browsing, which has potential benefits as well as dangers if used incorrectly.

Task 3.5 – Snort SSH Brute-Force Attack Alert

The final Snort task required the use of Snort to alert a machine of any potential threats of a brute force SSH attack occurring. Since SSH uses TCP packets to communicate, it would be possible to create a Snort rule that detected TCP packets matching a certain description. As provided in the report guide, the required maximum time period was 120 seconds and minimum number of attempts to connect within this period should be 5 before outputting an alert. Thus,

it was first important to create an SSH server on Machine A, as well as enabling root to be the username. The command used, as well as Machine A successfully logging in can be seen in *Figure 47*.

```
root@VM:/home/seed# ssh root@10.0.2.9
root@10.0.2.9's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)
```

Figure 47: Creating SSH server on Machine A

The next step was to create a Snort rule in the *locale.rules* file in Machine A, that would alert the console of a failed brute force attack. Firstly, since it was known that an SSH connection uses TCP packets, this was defined as the protocol for which the rule would check a packet matches other parameters. Next, the source IP address and ports were set to 'any' and 'any', since in a real scenario it would not be possible to know what machine is performing the attack. Additionally, the destination IP address was set to the network address of the local machine (10.0.2.9), since that is who the attack would take place on, and the destination ports were set to 'any'.

The '*flags'* tag was also used for this task, however in order to know what value to assign, additional research of how an SSH connection works was required. It was found that an SSH connection is started through a three-way handshake agreement, where a SYN flag is sent within a TCP packet to initialise a log-in request [5]. Thus, since this task was looking for failed login attempts, this would mean the Snort rule should detect the SYN flag being sent by a source machine. Thus, this tag was assigned the 'S+' value. Next, the '*threshold'* tag was used and assigned a value of '*type both'*, specifying that both end conditions must be true to trigger the alert, being the number of failed attempts and login timeframe. Using the '*track_by_src'* parameter also ensured that Snort was able to count the number of failed attempts for a specific IP address, which was essential for this task. Furthermore, the *'seconds'* and *'count'* tags were used to define the length this rule should last once first triggered, and the number of times it should be triggered before being sent to the console. Finally, an appropriate message and *'sid'* were defined to ensure that the alert was distinguishable when Snort was run. The full rule located in the *locale.rules* file can be seen in *Figure 48* below.

```
# LOCAL RULES
# ---------------
# This file intentionally does not come with signatures.  Put your local
# additions here.

alert tcp any any -> 10.0.2.9 any (msg: "SSH brute force occurring"; flags: S+; threshold: type
both, track by_src, count 5, seconds 120; sid: 100000009;)
```

Figure 48: Snort rule to detect possible SSH brute force attack

Next, Snort was initialised in the console using the command shown in *Figure 49.* This ensured that any output would be sent to the console, the correct configuration file was used, and that the correct interface was selected.

```
root@VM:~# sudo snort -A console -q -u snort -g snort -c /etc/snort/snort.conf -
i enp0s3
```

Figure 49: Initialisation of Snort

Machine B was then used to SSH into Machine A. This was done using the command shown in *Figure 50*. However, every time a password was prompted, an incorrect password was entered to investigate whether the rule would in fact work. An incorrect password was entered in Machine B ten times before then checking the output of Machine A.

```
[10/22/20]seed@VM:~$ ssh 10.0.2.10@10.0.2.9
The authenticity of host '10.0.2.9 (10.0.2.9)' can't be established.
ECDSA key fingerprint is SHA256:p1zAio6c1bI+8HDp5xa+eKRi561aFDaPE1/xq1eYzCI.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.9' (ECDSA) to the list of known hosts.
10.0.2.10@10.0.2.9's password:
Permission denied, please try again.
10.0.2.10@10.0.2.9's password:
Permission denied, please try again.
10.0.2.10@10.0.2.9's password:
Permission denied (publickey,password).
```

Figure 50: Failed SSH attempts

As expected, the console output showed two separate messages indicating five failed login attempts each. Additionally, the output shows that the source IP address matches that of Machine B, meaning that the Snort rule worked as intended. *Figure 51* shows the output of the Snort alert.

```
10/23-9:22:34.459284 [**] [1:10000009:1] SSH brute force occurring [**] [Priorit
y: 1] {TCP} 10.0.2.10 -> 10.0.2.9
10/23-9:22:37.826673 [**] [1:10000009:1] SSH brute force occurring [**] [Priorit
y: 1] {TCP} 10.0.2.10 -> 10.0.2.9
```

Figure 51: Snort alert showing risk of SSH brute force attack

In summary this task showed how Snort can also be used as a way to detect potential SSH brute force attacks on a machine. This is a powerful defence mechanism against a very common type of attack. This task also showed how key components of an SSH connection can be identified and used by Snort to define very specific rules. This again allows for a user to check for very particular things, meaning that alerts can tell someone incredibly valuable information depending on their needs.

Bibliography

[1] R. Russell, “iptables(8) - Linux man page,” 29 August 2020. [Online]. Available: https://linux.die.net/man/8/iptables. [Accessed 21 October 2020].

[2] nmap.org, “UDP Scan (-sU),” 14 May 2017. [Online]. Available: https://nmap.org/book/scan-methods-udp-scan.html. [Accessed 21 October 2020].

[3] nmap.org, “TCP FIN, NULL, and Xmas Scans (-sF, -sN, -sX),” 14 May 2017. [Online]. Available: https://nmap.org/book/scan-methods-null-fin-xmas-scan.html. [Accessed 21 October 2020].

[4] O'Reilly, “Snort Command Line Options,” 29 September 2020. [Online]. Available: https://www.oreilly.com/library/view/intrusion-detection-systems/0131407333/0131407333_ch02lev1sec4.html. [Accessed 22 October 2020].

[5] InetDaemon, “TCP 3-Way Handshake (SYN,SYN-ACK,ACK),” 18 May 2018. [Online]. Available: https://www.inetdaemon.com/tutorials/internet/tcp/3-way_handshake.shtml. [Accessed 22 October 2020].