MH4921
Supervised Independent Study II

Packet Sniffing & Spoofing

Brandon Goh Wen Heng

Academic Year 2018/19

Contents

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[1]. 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).

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

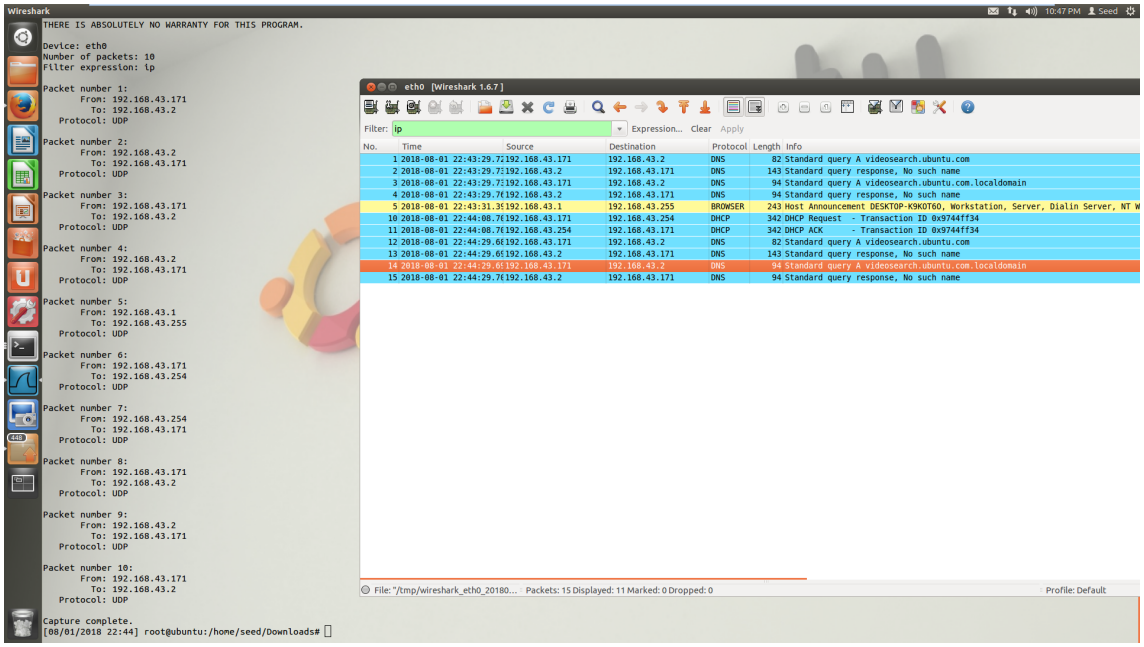


Figure 1: Sniffing Program Capture

From Figure 1, 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.

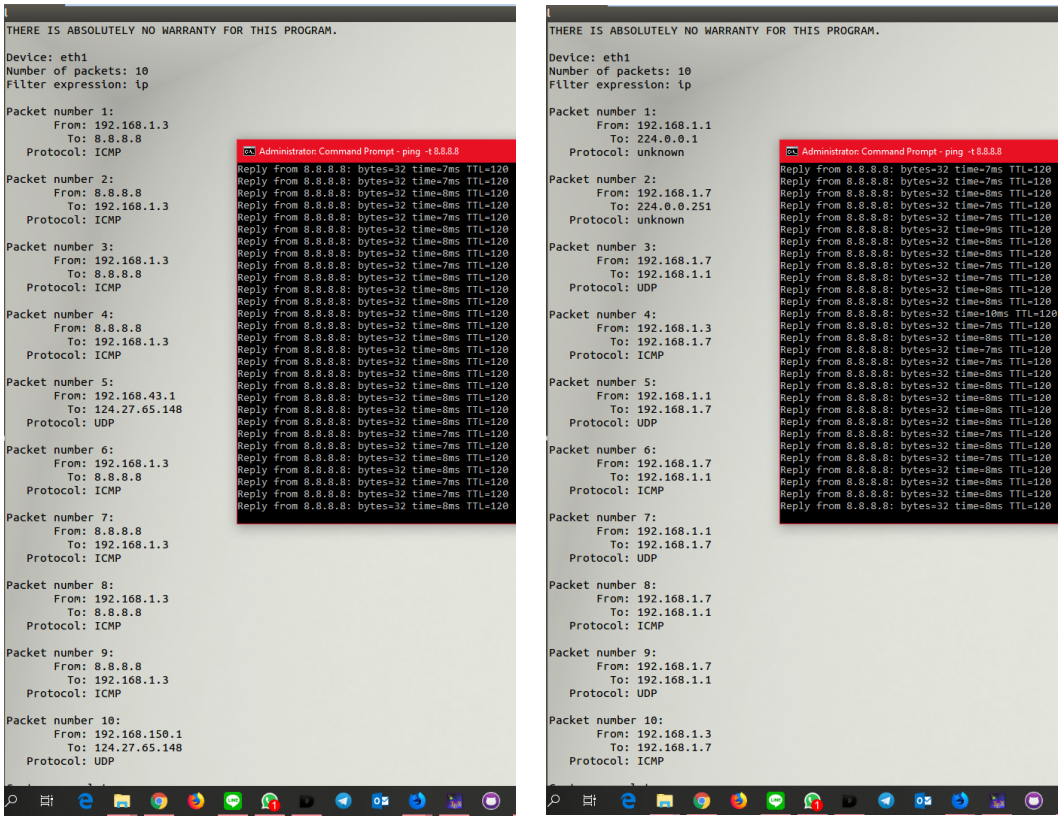
Windows (Command Prompt):

```
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 2a and 2b 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 2: 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 3: 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 4: 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 5 shows the data that was transmitted in the packet.

```
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                                              :
```

Figure 5: 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[2] 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.

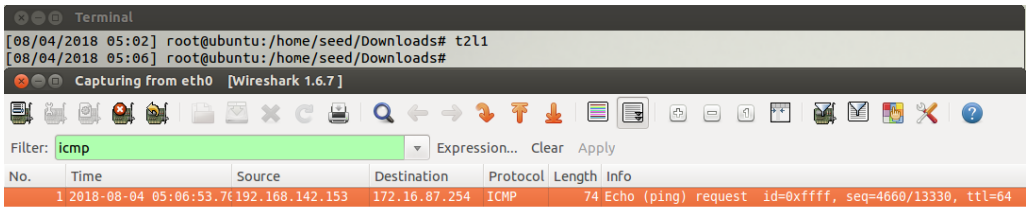


Figure 6: Spoofing Packet Sent

From Figure 6, 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.

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

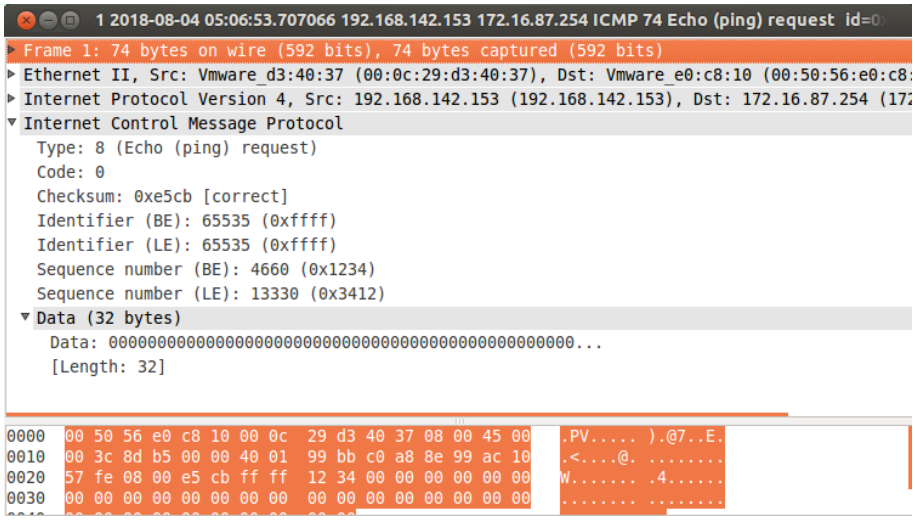


Figure 7: ICMP Packet Analysis

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.

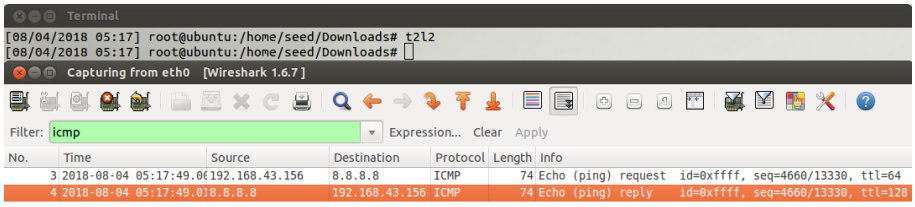


Figure 8: ICMP Reply

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.

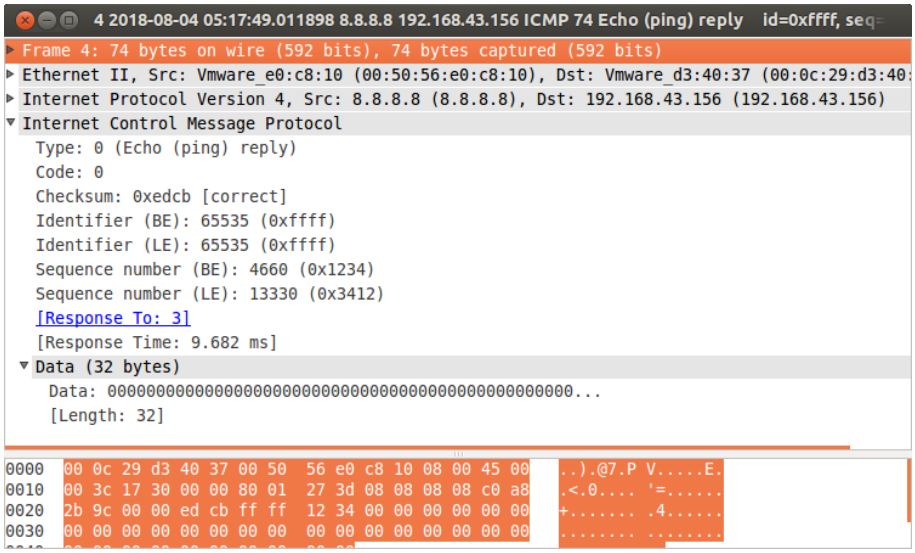


Figure 9: ICMP Reply Packet Detail

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).

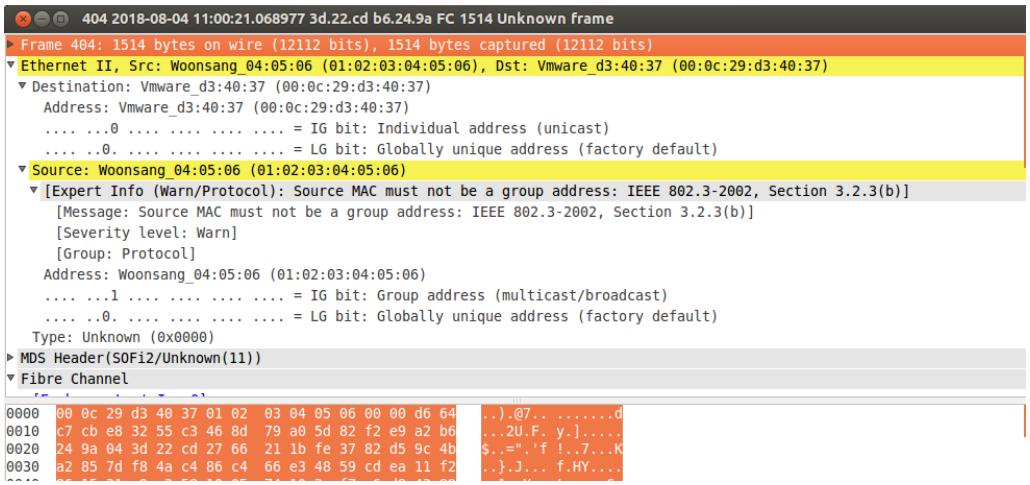


Figure 10: Ethernet Frame Spoofing

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) (*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.

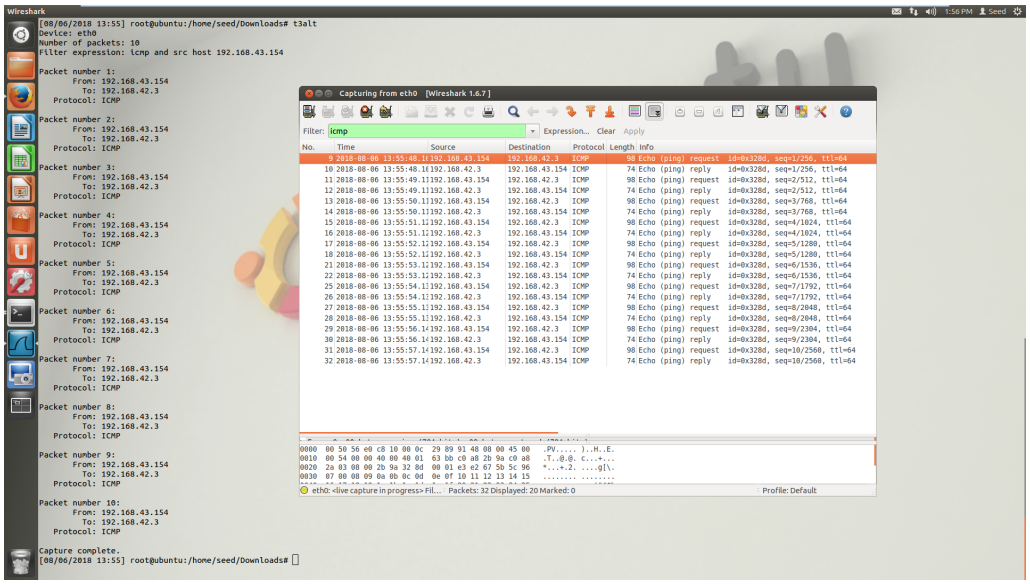


Figure 11: Successful ICMP Reply

From Figure 11, 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].

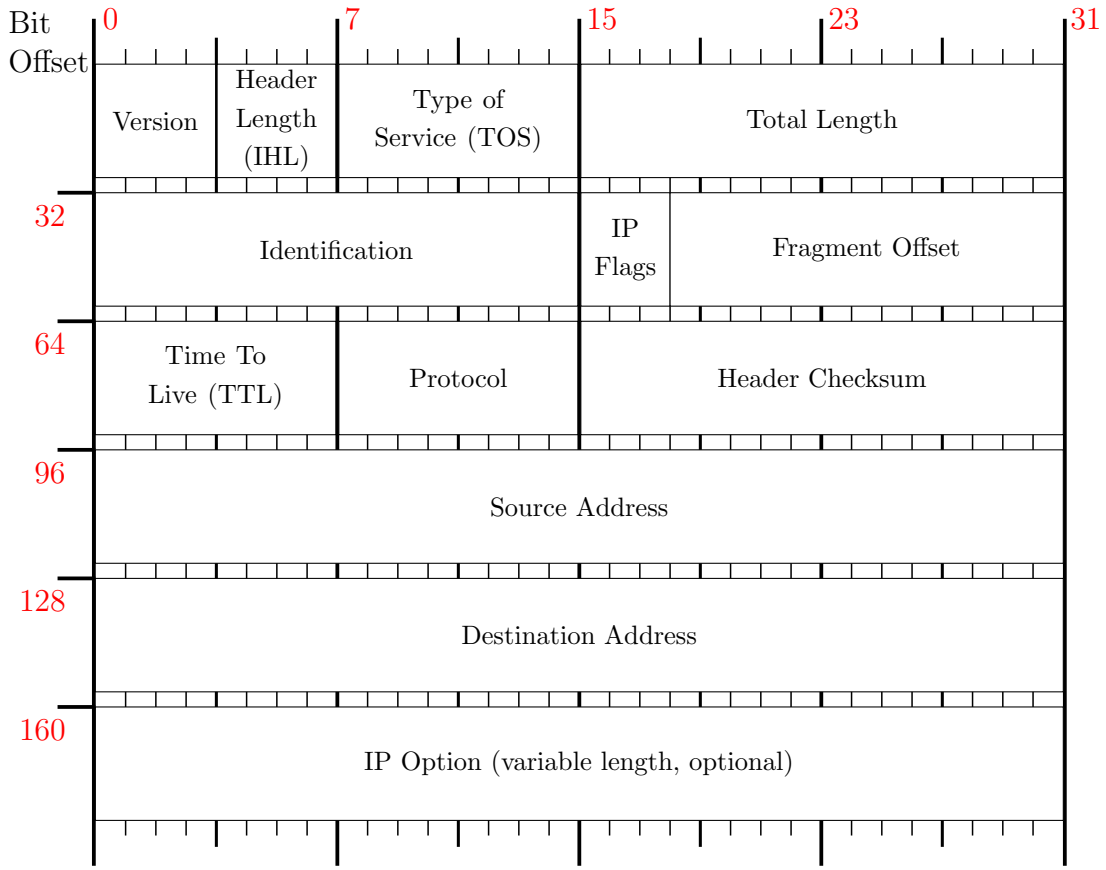
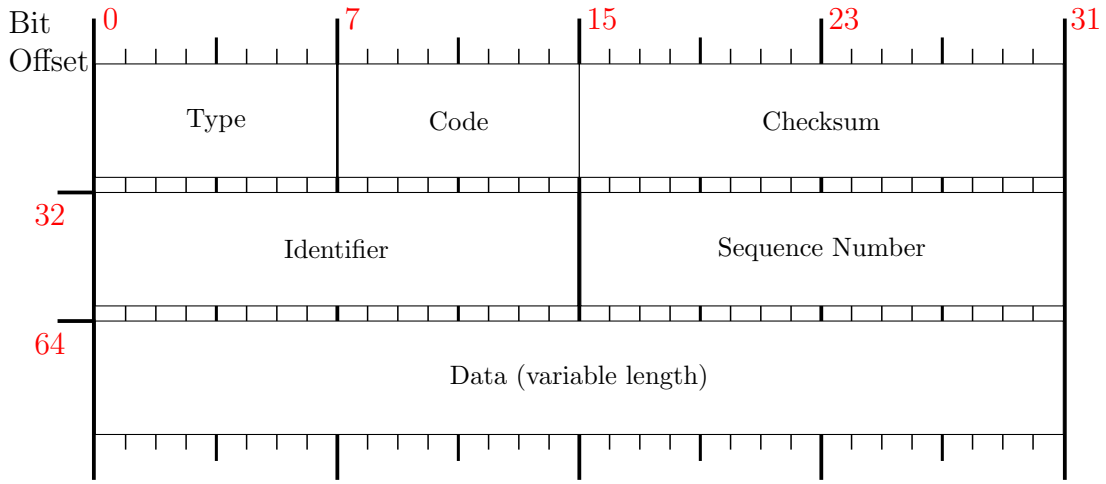


Figure 12: IPv4 Header

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 13: 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[3]. (*Note: VLAN Tag has been omitted as it is not relevant to our current task)

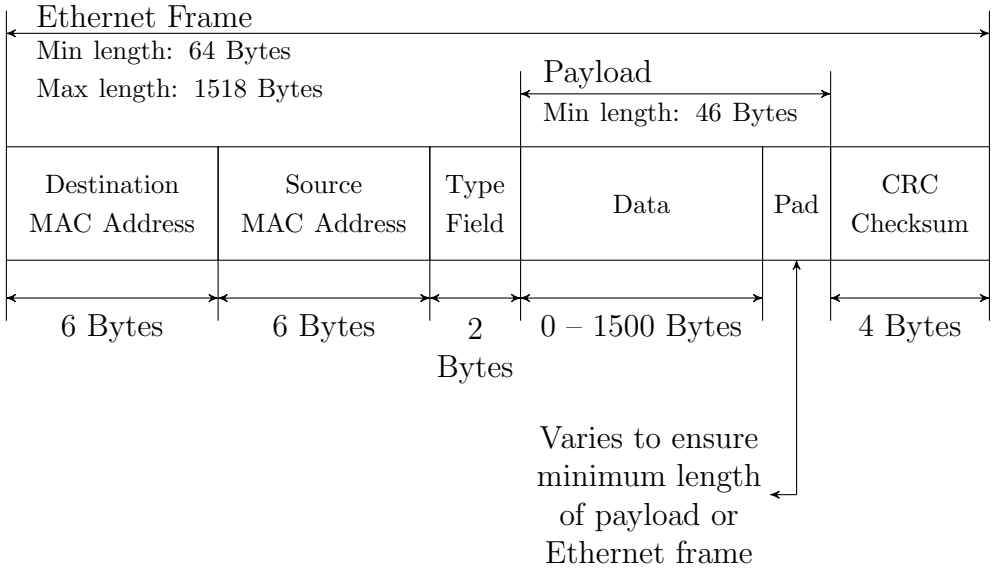


Figure 14: Ethernet II Frame

[3]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

```c
/* 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 */
```

```
49 struct sniff_ip {
50         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:
```

```
                        printf("   Protocol: unknown\n");
                        return;
        }

        /*
         *  OK, this packet is TCP.
         */

        /* define/compute tcp header offset */
        tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
        size_tcp = TH_OFF(tcp)*4;
        if (size_tcp < 20) {
                printf("   * Invalid TCP header length: %u bytes\n",
                 ↪  size_tcp);
                return;
        }

        printf("   Src port: %d\n", ntohs(tcp->th_sport));
        printf("   Dst port: %d\n", ntohs(tcp->th_dport));

        /* define/compute tcp payload (segment) offset */
        payload = (u_char *)(packet + SIZE_ETHERNET + size_ip +
         ↪  size_tcp);

        /* compute tcp payload (segment) size */
        size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

        /*
         * Print payload data; it might be binary, so don't just
         * treat it as a string.
         */
        if (size_payload > 0) {
                printf("   Payload (%d bytes):\n", size_payload);
                print_payload(payload, size_payload);
        }

return;
}


/* 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);
}
```

```
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 }
```