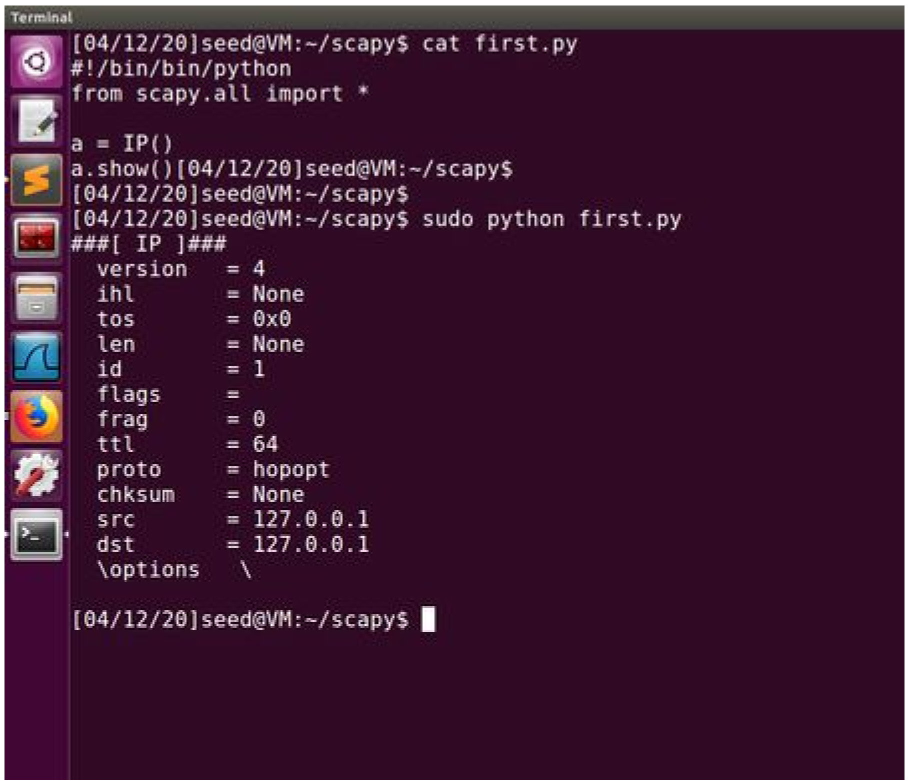
CSP 544: Packet Sniffing and Spoofing Lab **Darshan K (A20446137)**

In this and the rest of the tasks in this lab, we'll be making use of Scapy tool. This tool, unlike other tools, provides more functionalities than just the fixed one's offered by other tools. We can use this to integrate its functionalities into our own programs. In the screenshot below we can see how we're making use of a simple python program that prints the IP of the underlying machine. The screenshot has both the code and the output.



Lab Task 1.1A : Sniffing Packets (Running with sudo):

In this task we'll be making use of Wireshark. This tool cannot be used in our custom programs. Therefore we'll be using scapy for that purpose. We run a simple python packet that is used to sniff the packets and print us its contents. Below is th python program that we run:

```
!/usr/bin/python
from scapy.all import *
def  print_pkt(pkt):
```

```
                pkt.show()
        pkt = sniff(filter='icmp',prn=print_pkt)
```

Before running this program, I checked the IP of the machine to ping it. So that the ICMP packets sent by the ping command are captured by this program. The below screenshot shows how we ping to the same machine.

```
[04/12/20]seed@VM:~/scapy$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:e0:7c:7a
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::3b4b:808e:51b3:f3c1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:66569 errors:0 dropped:0 overruns:0 frame:0
          TX packets:36627 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:55190677 (55.1 MB)  TX bytes:10628982 (10.6 MB)

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:7873 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7873 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:721836 (721.8 KB)  TX bytes:721836 (721.8 KB)

[04/12/20]seed@VM:~/scapy$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.252 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.146 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.071 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.088 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.056 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.151 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.140 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.053 ms
64 bytes from 127.0.0.1: icmp_seq=9 ttl=64 time=0.074 ms
64 bytes from 127.0.0.1: icmp_seq=10 ttl=64 time=0.050 ms
^C
--- 127.0.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9296ms
rtt min/avg/max/mdev = 0.050/0.108/0.252/0.061 ms
[04/12/20]seed@VM:~/scapy$
```

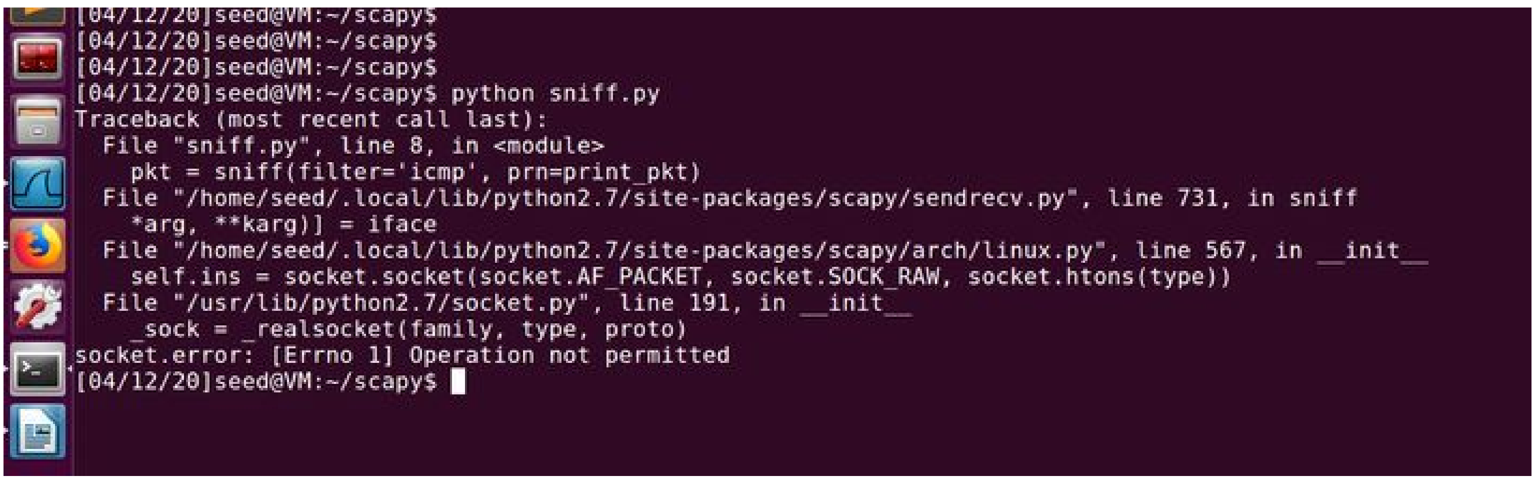
Here's the output of the same program when run with sudo.

```
^C[04/12/20]seed@VM:~/scapy$ sudo python sniff.py
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 24642
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xdc64
     src       = 127.0.0.1
     dst       = 127.0.0.1
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x7e00
        id        = 0x109e
        seq       = 0x1
###[ Raw ]###
           load      = '\x99\x97\x93^Cg\x0e\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       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
```

In the output we can see that it returns the ID and the checksum along with many other fields of each packet for both IP and ICMP packets and also returns the payload in the hex format.

Lab Task 1.1B : Sniffing Packets (Running without sudo):

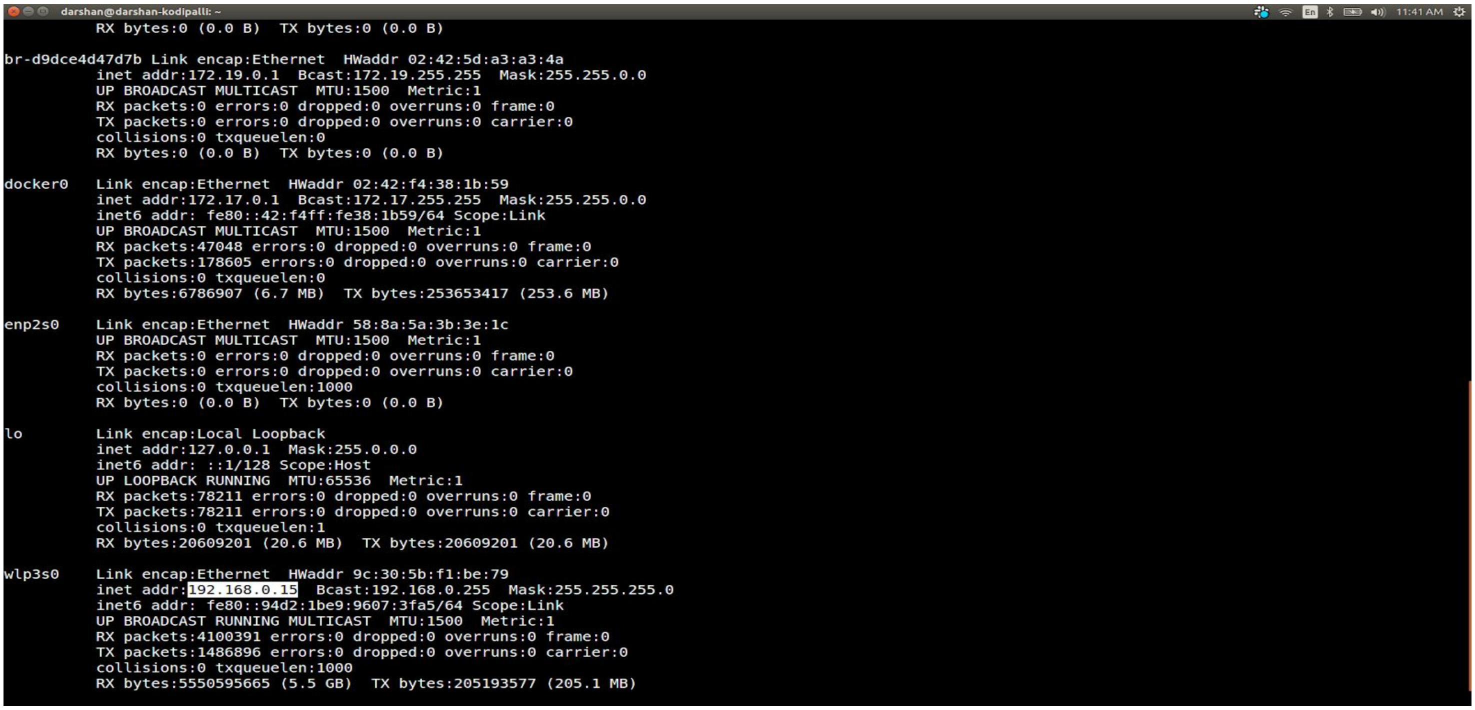
This is a repetition of the above task, instead we run the python program without the sudo. We can see there's an error as the root privilege is required for the python program to perform sniffing.



The imports we've used in this python program for scapy internally calls the socket which can only be accessed by the root. By not running this program as a root gives us the error that we've seen above.

Lab Task 1.2 : Spoofing ICMP Packets

In this task we spoof the IP packets with the arbitrary source IP address. We will be spoofing the ICMP echo request packet and sending it to the host machine which is on the same network as the VM is on. I first checked the IP of the host machine so that I can use the same IP as a Source IP address to perform out task.



We can see that the IP of the host machine is **192.168.0.15.** Now I'll make use of this IP to perform this task. I also ran the python code written in the previous task to sniff the packets on the host machine, to see if the attack was successful. Now the python code to send 1 packet to the host machine is as shown below..

```
#!/bin/bin/python
from scapy.all import *
a = IP()
a.dst = '192.168.0.15'
b = ICMP()
p = a/b
send(p)
```

After running the above code from the vm and monitoring it from the host machine, the outputs we see are shown below

```
[04/13/20]seed@VM:~/scapy$ cat
first.py        sniff.py        spoofICMP.py
[04/13/20]seed@VM:~/scapy$ cat spoofICMP.py

#!/bin/bin/python
from scapy.all import *
a = IP()
a.dst = '192.168.0.15'
b = ICMP()
p = a/b
send(p)

[04/13/20]seed@VM:~/scapy$ sudo python spoofICMP.py
.
Sent 1 packets.
[04/13/20]seed@VM:~/scapy$
```

Output from the host machine on sniffing the traffic:

```
darshan@darshan-kodipalli:~$ clear
darshan@darshan-kodipalli:~$ sudo python sniff.py
[sudo] password for darshan:
WARNING: No route found for IPv6 destination :: (no default route?)
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 28
     id        = 5789
     flags     = DF
     frag      = 0L
     ttl       = 63
     proto     = icmp
     chksum    = 0xa3d5
     src       = 192.168.0.15
     dst       = 192.168.0.15
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 28
     id        = 5789
     flags     = DF
     frag      = 0L
     ttl       = 63
     proto     = icmp
     chksum    = 0xa3d5
```

```
     dst       = 192.168.0.15
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 28
     id        = 5790
     flags     =
     frag      = 0L
     ttl       = 64
     proto     = icmp
     chksum    = 0xe2d4
     src       = 192.168.0.15
     dst       = 192.168.0.15
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xffff
        id        = 0x0
        seq       = 0x0
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 28
     id        = 5790
     flags     =
```

From the output from the above screenshots we can see that host machine which is in the same network as the VM is on, can sniff the packet and the second small screenshot shows the ICMP packet details.

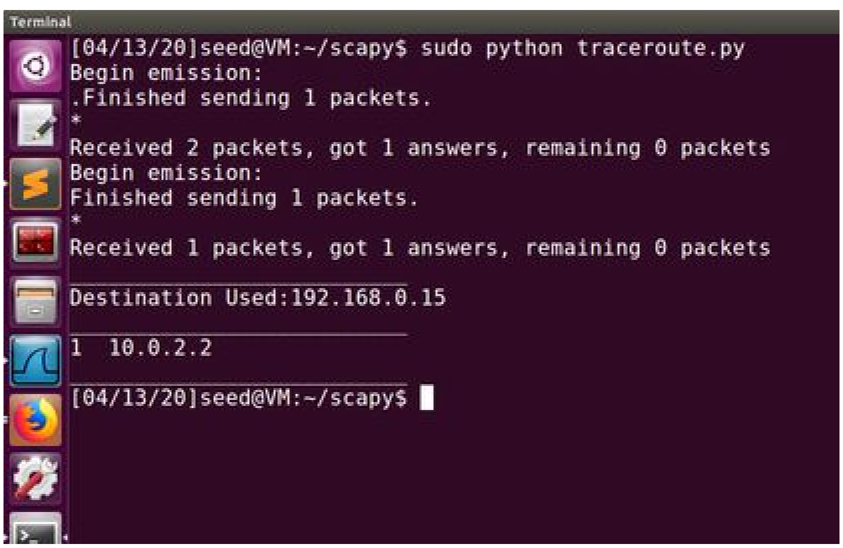
Lab Task 1.3 : Traceroute

In this task we're asked to find the distance in terms of the hops/routers the packet had crossed in order to reach the destination. Basically what a ***traceroute*** does.
In the python code, instead of running the program with different ***ttl*** values each time and recording the result, I ran the same program till it completed the traceroute with the status set to false and incremented the ***ttl*** in each run. And all the intermediate routers are stored in the ***intermediate_routers*** variable. Below is the complete python code that I've used for this task:

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

a = IP()
a.dst = '192.168.0.15'
b = ICMP()
status = True
time_to_live = 1
intermediate_routers = []

while status:
   a.ttl = time_to_live
   response, noresponse = sr(a/b)
   if response.res[0][1].type == 0:
        status = False
   Else:
        intermediate_routers.append(response.res[0][1].src)
        time_to_live+=1
```

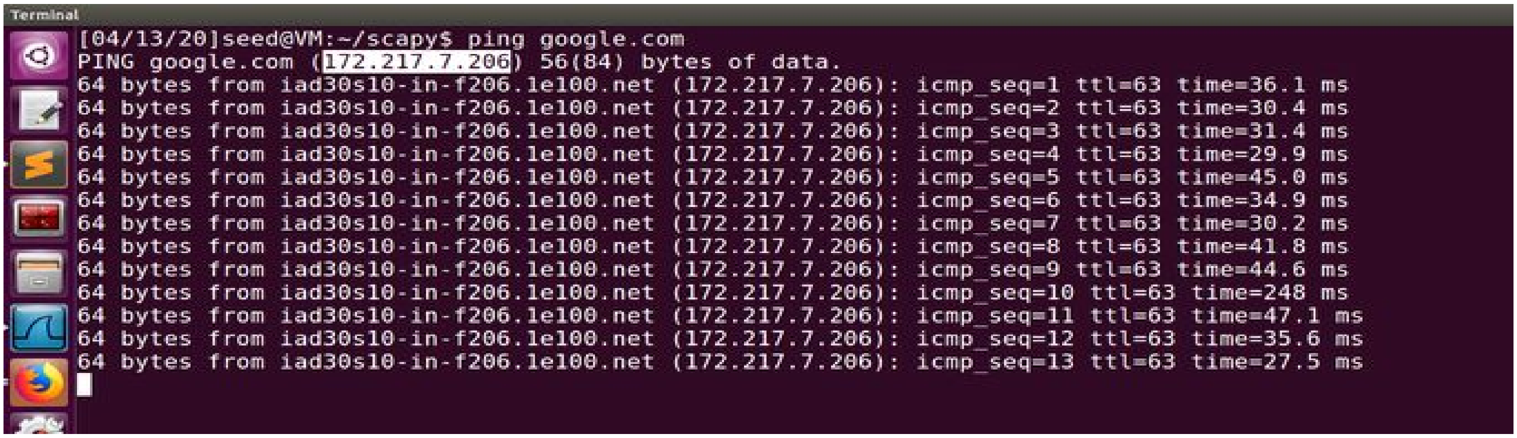
```
router_pointer = 1
print "______________________________"
print "Destination Used:" + a.dst
print "______________________________"

for hop in intermediate_routers:
        print router_pointer," " + hop
router_pointer+=1

print "______________________________"
```

In this task I’ll run the python program with 2 different destinations. First would be the address of the host machine. 192.168.0.15. Since the host machine is in the same network as the VM is on, we can see the packet only has to hit the router once. And the router hits the host machine since it is in the same network. Here is the output when run as with sudo

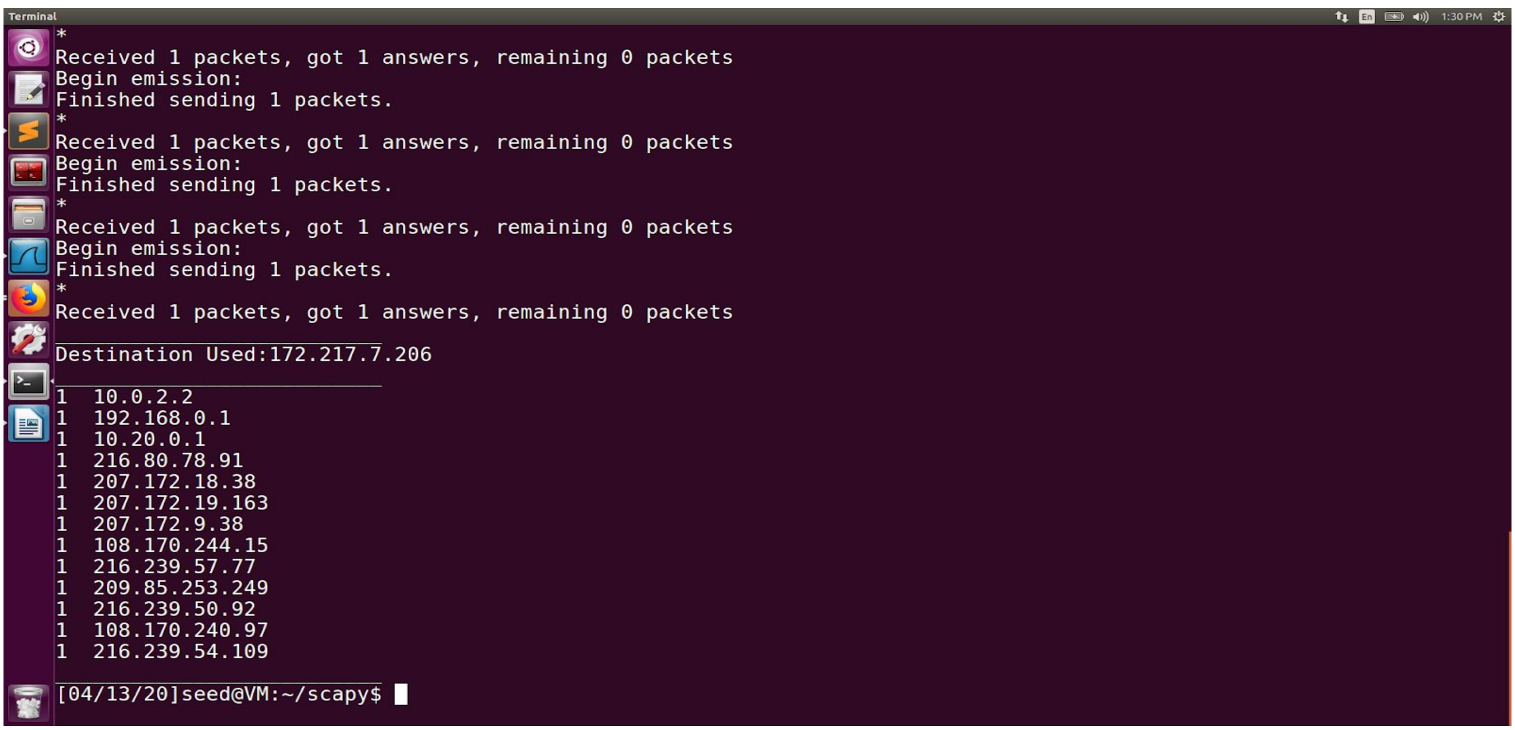


Now as said earlier, I’ll run the same program with different destination. I’ll first ping google.com and fetch it’s IP and then use it in our program to see what it returns.

From the result of ping, we can see the IP of google server in our case is **172.217.7.206**



I now use this IP and run the same program and here are the results.

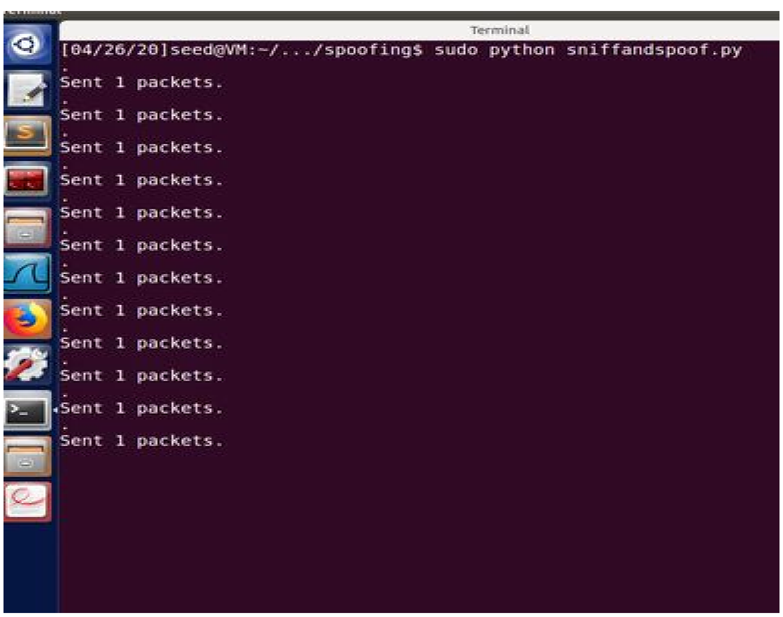


Lab Task 1.4 : Sniffing and-then Spoofing:

This task is a combination of all the tasks we've performed till now. For this task, I'm using the Seed VM from which I've been performing all the tasks and then the underlying host machine which is also a linux distros.. And is running on the same network as the VM is on. Below is the code I've used to successfully run this attack.

```
from scapy.all import *
def print_pkt(pkt):
   a = IP();
   a.src = pkt[IP].dst;
   a.dst = pkt[IP].src;
   b = ICMP()
   b.type = 'echo-reply'
   b.code = 0;
   b.id = pkt[ICMP].id
   b.seq = pkt[ICMP].seq
   p=a/b
   send(p)
pkt = sniff(filter='icmp[icmptype] == icmp-echo', prn=print_pkt)
```

When pinged to another machine on the same network as the same machine is in.. here is what we get when we run the above program.



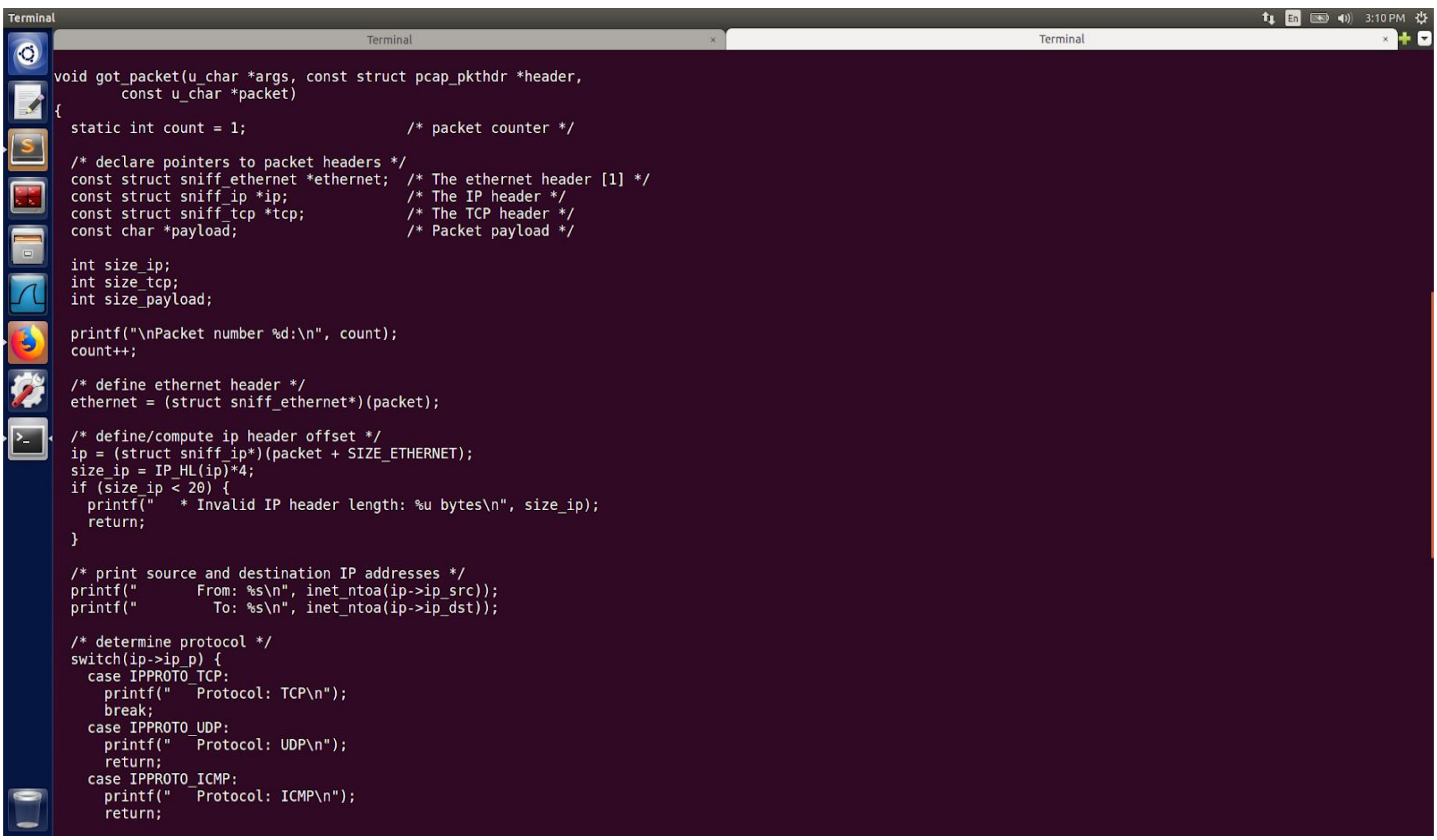
Spoofing is sniffing for the request and immediately sending the reply. The user pings a host 40.40.40.40, the attacker on VM A receives the ICMP packet which listens to traffic, spoofs an ICMP reply using raw socket by replacing the source ip as the destination ip and the destination ip as the source ip. The 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.

Lab Task 2.1 : Writing Packet Sniffing Program:

For this task we'll be making use of the *pcap* library. With this library, the task of the sniffers becomes invoking a simple sequence of procedures in the *pcap* library.

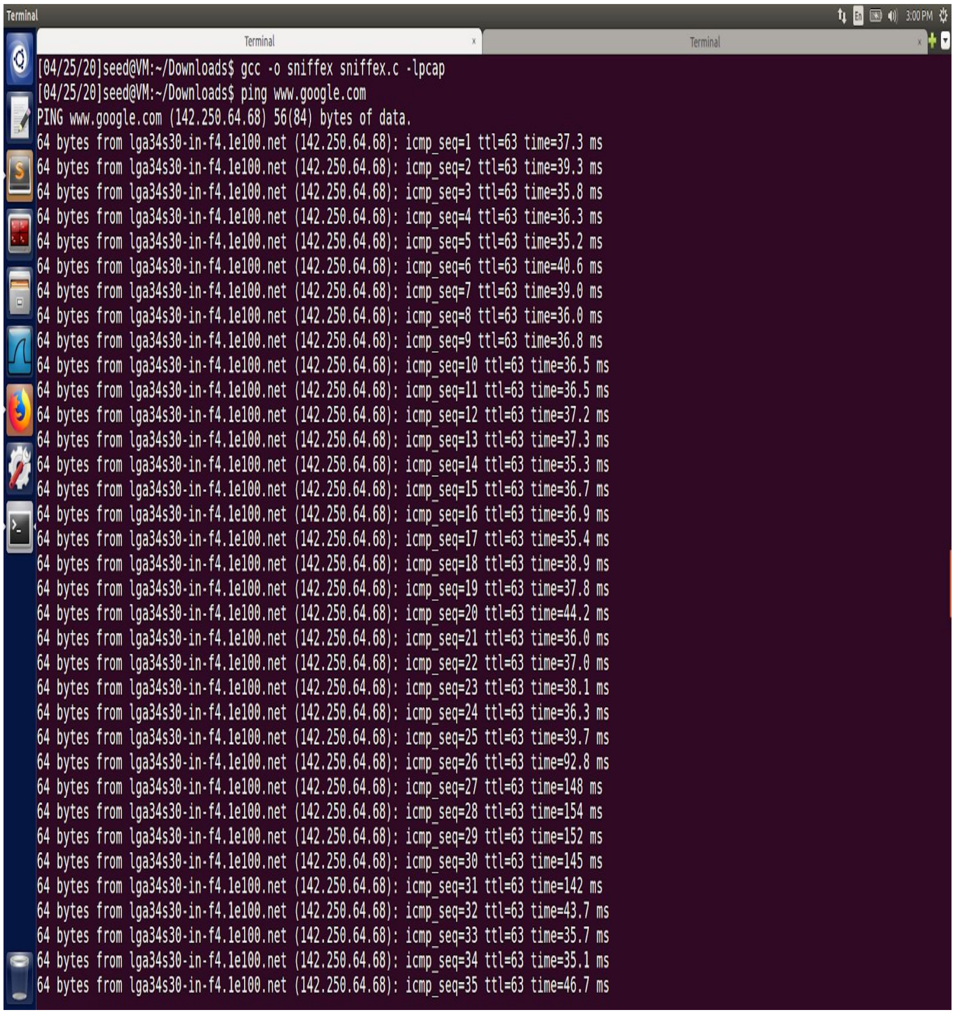
Task 2.1 A : Understanding how a Sniffer works:

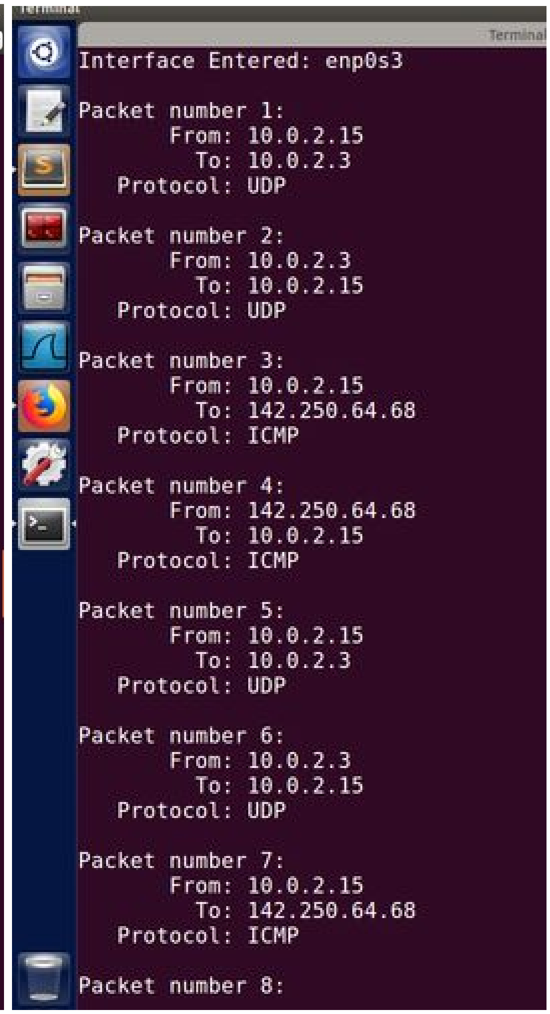
In this task, I executed the code provided in this link. And the screenshot below shows the code.



```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
  static int count = 1;                   /* packet counter */

  /* declare pointers to packet headers */
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
  const struct sniff_ip *ip;              /* The IP header */
  const struct sniff_tcp *tcp;            /* The TCP header */
  const char *payload;                    /* Packet payload */

  int size_ip;
  int size_tcp;
  int size_payload;

  printf("\nPacket number %d:\n", count);
  count++;

  /* define ethernet header */
  ethernet = (struct sniff_ethernet*)(packet);

  /* define/compute ip header offset */
  ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
  size_ip = IP_HL(ip)*4;
  if (size_ip < 20) {
    printf("   * Invalid IP header length: %u bytes\n", size_ip);
    return;
  }

  /* print source and destination IP addresses */
  printf("       From: %s\n", inet_ntoa(ip->ip_src));
  printf("         To: %s\n", inet_ntoa(ip->ip_dst));

  /* determine protocol */
  switch(ip->ip_p) {
    case IPPROTO_TCP:
      printf("   Protocol: TCP\n");
      break;
    case IPPROTO_UDP:
      printf("   Protocol: UDP\n");
      return;
    case IPPROTO_ICMP:
      printf("   Protocol: ICMP\n");
      return;
```

From the result of ifconfig, I can see for me it's **enp0s3,** So entering this in the code and executing gives us the below result. To send some packets from my machine, I executed a ping command to www.google.com. Here's the output





On the right hand side of the output, we can see the packet number, the protocol, IP source and destination.

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

In the code screenshot above we can see I’m first holding a pointer for the packet count. And the defining the standard eth header by calling this

ethernet = (struct sniff_ethernet*)(packet);

And then I’m printing the source and the destination address. And then setting up the protocol using the switch statements. Once we’re set till here, I’m defining the TCP header offset by calling **tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);** and computing the payload length offset and size. I’ve computed the payload size using ntohs function.

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.

Few of the libraries and headers we’ve used in this C program internally calls the socket which can only be accessed by the root. By not running this program as a root gives us the error that is shown below. It fails at accessing the **Socket**.

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.

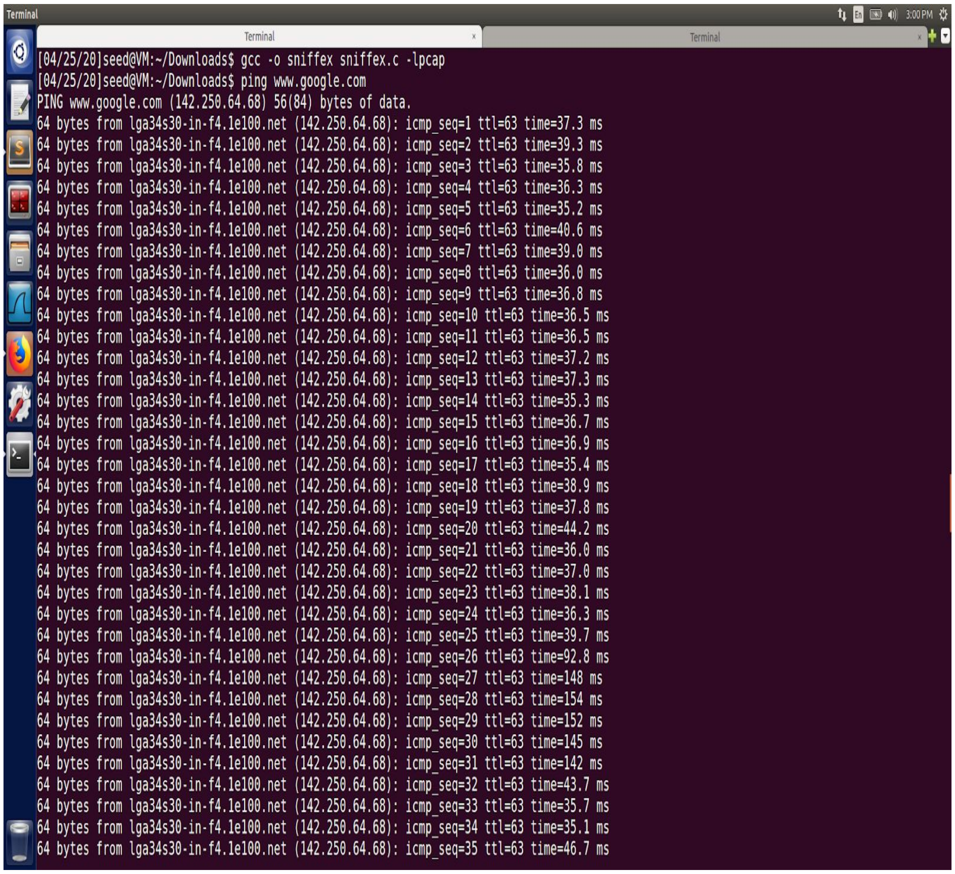
When promiscuous mode is on, sniffer program can capture all the packets in the same network regardless of the destination IP. When promiscuous mode is off, sniffer program cannot capture all the packets in the same network, it can only capture packets whose destination IP is the IP of the sniffer’s system.

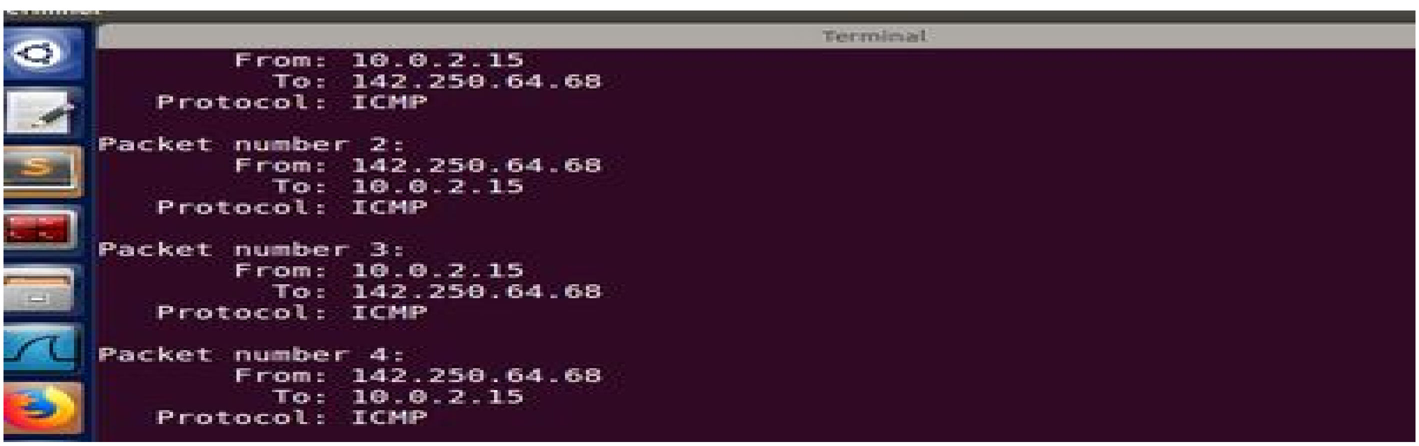
Task 2.1B: Writing Filters:

As done in the previous task, here's the snippet used for capturing the packets of a particular type.

```
ethernet = (struct sniff_ethernet*)(packet);
   ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);

   switch(ip->ip_p) {
    case IPPROTO_TCP:
           printf("   Protocol: TCP\n");
           break;
    case IPPROTO_ICMP:
           printf("   Protocol: ICMP\n");
           return;
    default:
           printf("   Protocol: unknown\n");
           return;
   }
```

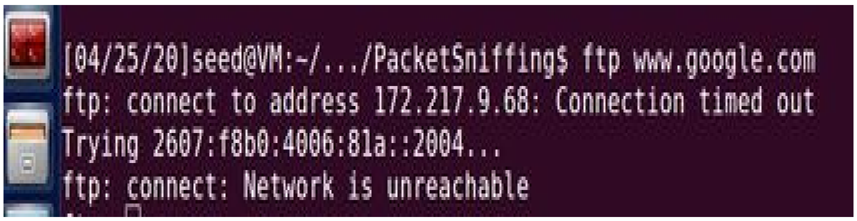
We first capture the packet and extract the protocol of that IP. And based on the protocol derived from the switch statement, we render the equivalent function. Below is the output when we send ICMP packet through the ping command to google.com





All the from addresses(IP's of the intermediate servers) we see in the above screenshots, are the responses we get back from the ping command to google.com's server.

The code we've written also works well for capturing the TCP packets. In the screenshot below we can see the TCP packets being captured on calling ftp (runs on 21) to google.com. Here although the connection is not established, it was initiated. The screenshot adjacent to it shows how the TCP packets are captured and they're all in the range of 10-100.

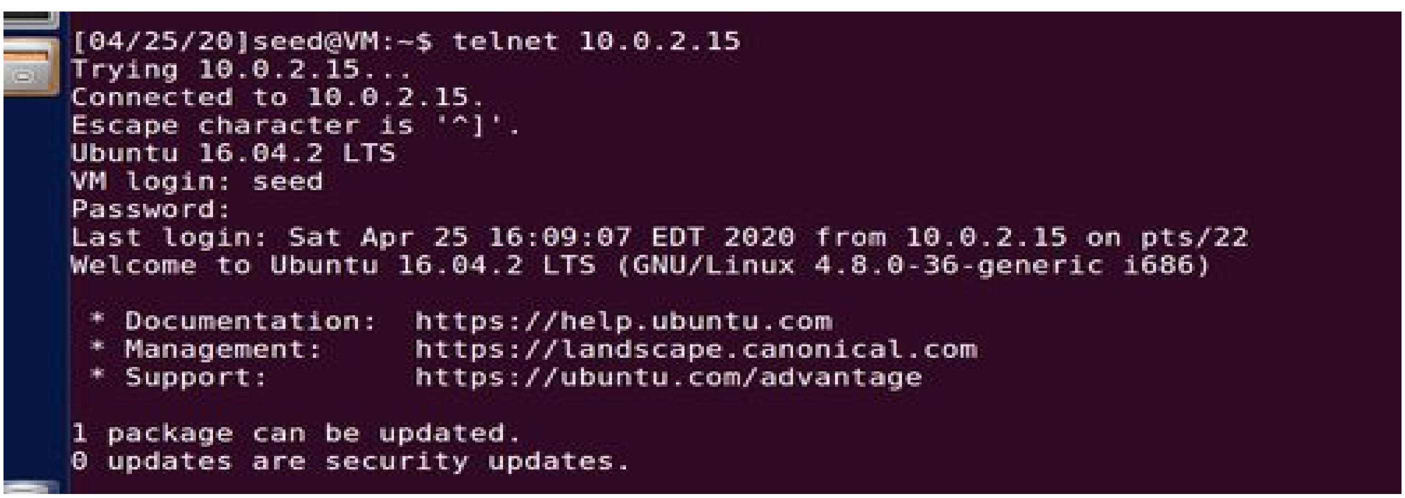


```
[04/25/20]seed@VM:~/.../PacketSniffing$ ftp www.google.com
ftp: connect to address 172.217.9.68: Connection timed out
Trying 2607:f8b0:4006:81a::2004...
ftp: connect: Network is unreachable
```

```
Packet number 1:
       From: 10.0.2.15
         To: 172.217.9.68
   Protocol: TCP
   Src port: 57030
   Dst port: 21

Packet number 2:
       From: 10.0.2.15
         To: 192.124.249.22
   Protocol: TCP
   Src port: 48286
   Dst port: 80

Packet number 3:
       From: 192.124.249.22
         To: 10.0.2.15
   Protocol: TCP
   Src port: 80
   Dst port: 48286
```

Task 2.1C: Sniffing Passwords:

In this task I'm trying to connect to a machine of IP 10.0.2.15 with the users details as seed/dees being the username and password to establish the connection. We can see in the first screenshot how we're calling the telnet and in the second screenshot we can see how it is being captured in the payload section of the screenshot.



```
[04/25/20]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
Connected to 10.0.2.15.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sat Apr 25 16:09:07 EDT 2020 from 10.0.2.15 on pts/22
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

1 package can be updated.
0 updates are security updates.
```

Task 2.2: Spoofing:

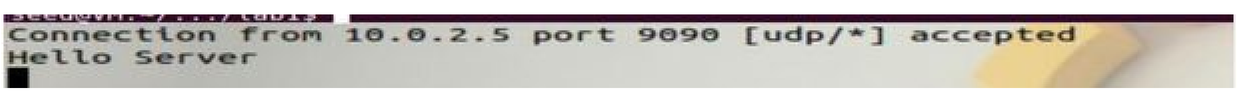
Below is the program for Spoofing the packets. In this task I'll make use of this program to successfully send out spoofed IP packets.

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
```

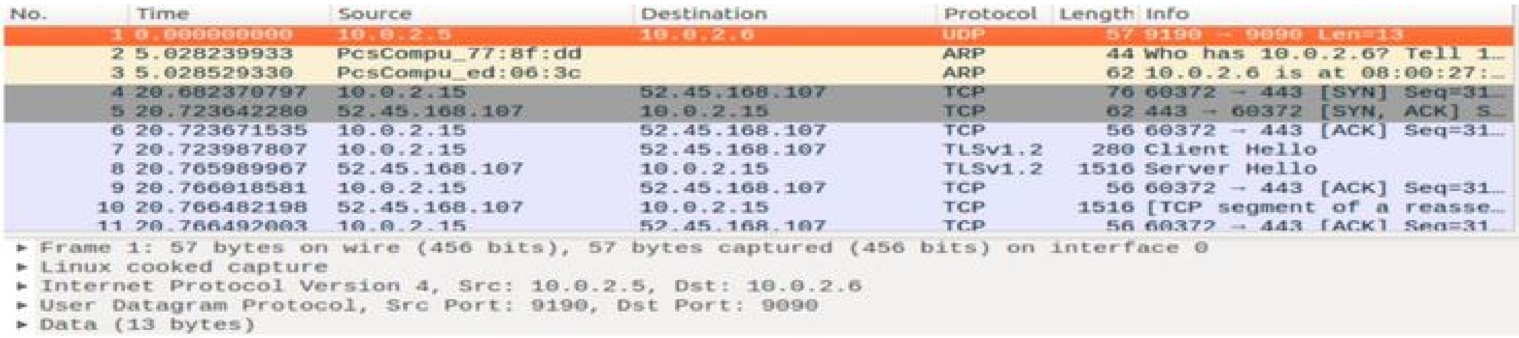
```
#include "myheader.h"

void send_raw_ip_packet(struct ipheader* ip)
{
      struct sockaddr_in dest_info;
      int enable = 1;
      int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
      setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                  &enable, sizeof(enable));
      dest_info.sin_family = AF_INET;
      dest_info.sin_addr = ip->iph_destip;
      sendto(sock, ip, ntohs(ip->iph_len), 0,
      (struct sockaddr *)&dest_info, sizeof(dest_info));
      close(sock);
}


int main() {
 char buffer[1500];
  memset(buffer, 0, 1500);
  struct ipheader *ip = (struct ipheader *) buffer;
  struct udpheader *udp = (struct udpheader *) (buffer +
                                    sizeof(struct ipheader));
  char *data = buffer + sizeof(struct ipheader) +
                  sizeof(struct udpheader);
  const char *msg = "Hello Server!\n";
  int data_len = strlen(msg);
  strncpy (data, msg, data_len);
  udp->udp_sport = htons(12345);
```

```
    udp->udp_dport = htons(9090);
    udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    udp->udp_sum =  0; /* Many OSes ignore this field, so we do not
                  calculate it. */
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.69");
    ip->iph_protocol = IPPROTO_UDP; // The value is 17.
    ip->iph_len = htons(sizeof(struct ipheader) +
                  sizeof(struct udpheader) + data_len);
    send_raw_ip_packet (ip);
  return 0;
  }
```

In this code the attacker is on the IP 10.0.2.15 as usual. He now sends the spoofed UDP packet with Hello server message so 10.0.2.6 from 10.0.2.5 as a destination and source IP addresses respectively. This program also expects all the headers defined in the myheader.h header file whose code is copied from the book’s code stored at [this](#) repository.



And this is confirmed by the wireshark that is configured to listen to all the packets encountered in the traffic. In the screenshot below we can see wireshark capturing that the source IP of the packet whose source and destination IP’s are different than that of the attacker’s.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.5	10.0.2.6	UDP	57	9190 → 9090 Len=13
2	5.028239933	PcsCompu_77:8f:dd		ARP	44	Who has 10.0.2.6? Tell 1…
3	5.028529330	PcsCompu_ed:06:3c		ARP	62	10.0.2.6 is at 08:00:27:…
4	20.682370797	10.0.2.15	52.45.168.107	TCP	76	60372 → 443 [SYN] Seq=31…
5	20.723642280	52.45.168.107	10.0.2.15	TCP	62	443 → 60372 [SYN, ACK] S…
6	20.723671535	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31…
7	20.723987807	10.0.2.15	52.45.168.107	TLSv1.2	280	Client Hello
8	20.765989967	52.45.168.107	10.0.2.15	TLSv1.2	1516	Server Hello
9	20.766018581	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31…
10	20.766482198	52.45.168.107	10.0.2.15	TCP	1516	[TCP segment of a reasse…
11	20.766492003	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31…

- ► Frame 1: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface 0
- ► Linux cooked capture
- ► Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
- ► User Datagram Protocol, Src Port: 9190, Dst Port: 9090
- ► Data (13 bytes)

Task2.2B: Spoof an ICMP Echo Request.

In this task the objective is to spoof an ICMP echo request packet on behalf of another machine. The code is very similar to the one in the previous one except for the main function which is modified as to this:

```
int main() {
  char buffer[1500];
  memset(buffer, 0, 1500);
  struct icmpheader *icmp = (struct icmpheader *)
                      (buffer + sizeof(struct ipheader));
  icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.
  icmp->icmp_chksum = 0;
  icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                           sizeof(struct icmpheader));
  struct ipheader *ip = (struct ipheader *) buffer;
  ip->iph_ver = 4;
  ip->iph_ihl = 5;
  ip->iph_ttl = 20;
  ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
  ip->iph_destip.s_addr = inet_addr("10.0.2.69");
  ip->iph_protocol = IPPROTO_ICMP;
  ip->iph_len = htons(sizeof(struct ipheader) +
                      sizeof(struct icmpheader));
  send_raw_ip_packet (ip);
```

```
    return 0;
}
```

```
seed@VM:~/.../lab1$ gcc spoof2.c -o spoof2
seed@VM:~/.../lab1$ sudo ./spoof2
[sudo] password for seed:
Sending the spoofed IP packet..
```

No.	Time	Source	Destination	Protocol	Length	Info
31	19.560653496	10.0.2.15	224.0.0.251	MDNS	183	Standard query 0x0000 PT…
Sublime Text	89521045	fe80::bdad:b2a1:55d…	ff02::fb	MDNS	203	Standard query 0x0000 PT…
33	20.000319345	PcsCompu_77:8f:dd	Broadcast	ARP	42	Who has 10.0.2.1? Tell 1…
34	20.594029658	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.1? Tell 1…
35	21.384030204	PcsCompu_77:8f:dd	Broadcast	ARP	42	Who has 10.0.2.1? Tell 1…
36	21.594100987	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.1? Tell 1…
37	22.333174967	10.0.2.5	10.0.2.6	ICMP	42	Echo (ping) request id=…
38	22.333441426	10.0.2.6	10.0.2.5	ICMP	60	Echo (ping) reply id=…
39	22.399732487	PcsCompu_77:8f:dd	Broadcast	ARP	42	Who has 10.0.2.1? Tell 1…
40	22.594810368	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.1? Tell 1…
41	23.423940344	PcsCompu_77:8f:dd	Broadcast	ARP	42	Who has 10.0.2.1? Tell 1…

```
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_77:8f:dd (08:00:27:77:8f:dd), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```

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

When the length is set to some arbitrary value, the IP packet will not be formed as expected. There are maximum chances that the packet can either be truncated or dropped as they can get bigger when they're sent.

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

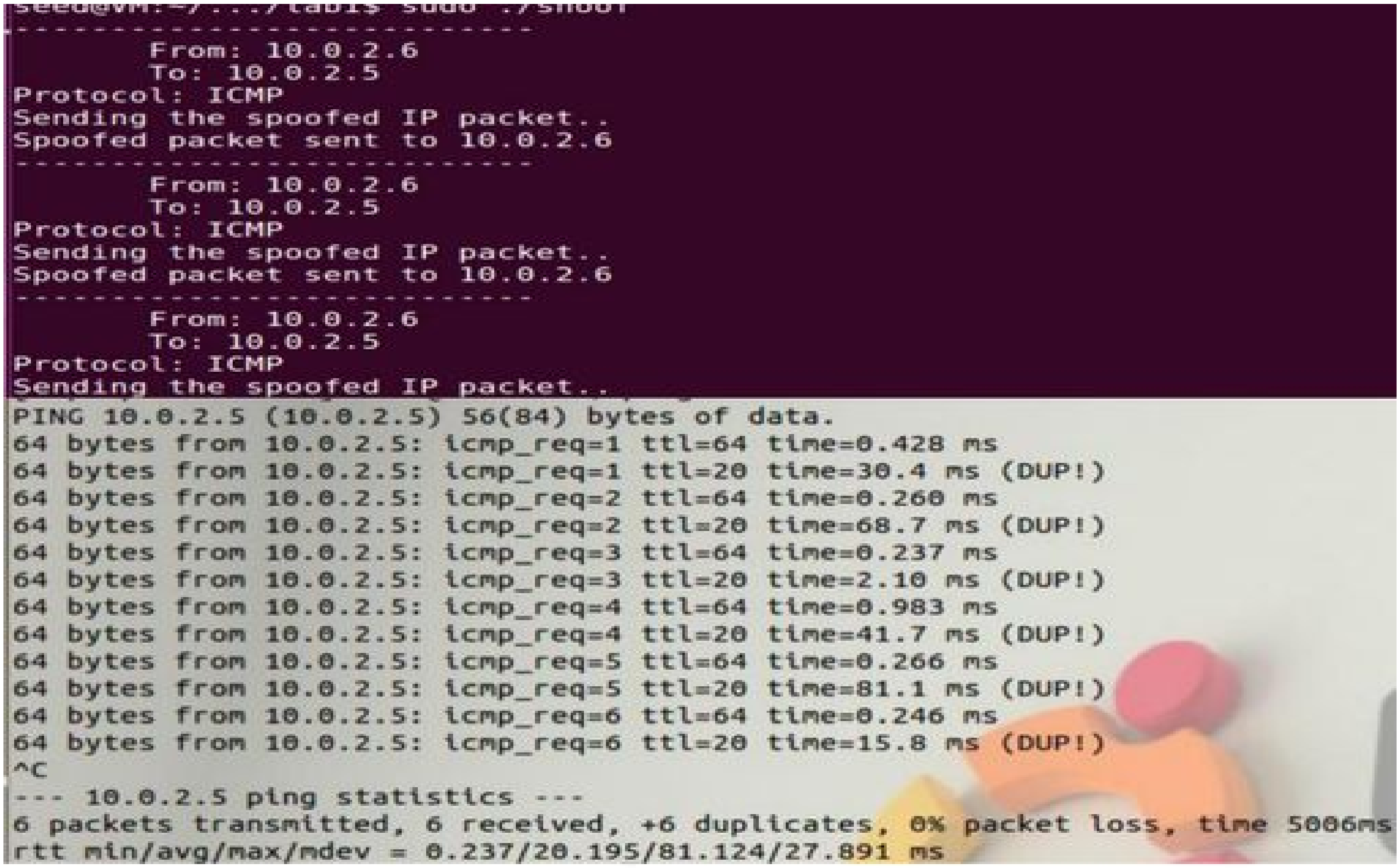
The checksum for the IP header is calculated by the Operating System before transmitting the packet over the network. So whatever the value specified, the operating system still calculates and transmits it.

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

Raw sockets gives the user the privilege to spoof a packet and set arbitrary values to any field in the packet headers. So when raw sockets are used, it is necessary to have root privileges to perform these tasks. When the spoof program is run without root privileges, it throws an error because to send a packet, the program needs to access the Network Interface Card.

Task2.3: Sniff and then Spoof

This task is a combination of the Sniff and Spoof techniques to implement the sniff and then spoof attack.



```
--------------------------
        From: 10.0.2.6
        To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
Spoofed packet sent to 10.0.2.6
--------------------------
        From: 10.0.2.6
        To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
Spoofed packet sent to 10.0.2.6
--------------------------
        From: 10.0.2.6
        To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_req=1 ttl=64 time=0.428 ms
64 bytes from 10.0.2.5: icmp_req=1 ttl=20 time=30.4 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=2 ttl=64 time=0.260 ms
64 bytes from 10.0.2.5: icmp_req=2 ttl=20 time=68.7 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=3 ttl=64 time=0.237 ms
64 bytes from 10.0.2.5: icmp_req=3 ttl=20 time=2.10 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=4 ttl=64 time=0.983 ms
64 bytes from 10.0.2.5: icmp_req=4 ttl=20 time=41.7 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=5 ttl=64 time=0.266 ms
64 bytes from 10.0.2.5: icmp_req=5 ttl=20 time=81.1 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=6 ttl=64 time=0.246 ms
64 bytes from 10.0.2.5: icmp_req=6 ttl=20 time=15.8 ms (DUP!)
^C
--- 10.0.2.5 ping statistics ---
6 packets transmitted, 6 received, +6 duplicates, 0% packet loss, time 5006ms
rtt min/avg/max/mdev = 0.237/20.195/81.124/27.891 ms
```

In this task the user pings a host whose IP is 10.0.2.5 on the network. And the attacker sniffs the ICMP request. And the immediately spoofs the ICMP reply to the source of the ICMP request.

Here, spoofing is sniffing for the request and immediately sending the reply. The user here pings a host 10.0.2.5 and the attacker on 10.0.2.6 receives the ICMP packet from ***pcap*** and spoofs and ICMP reply by replacing the source IP as the destination IP and the destination IP as the Source IP.