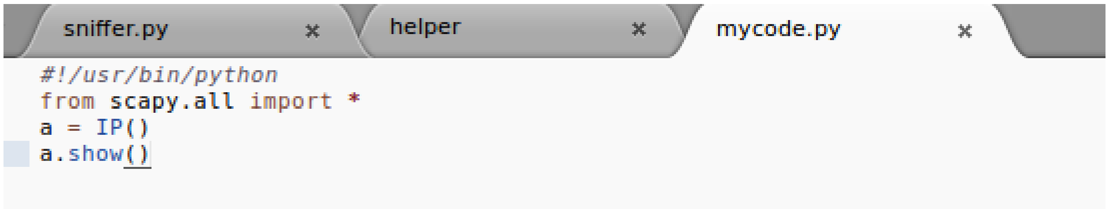
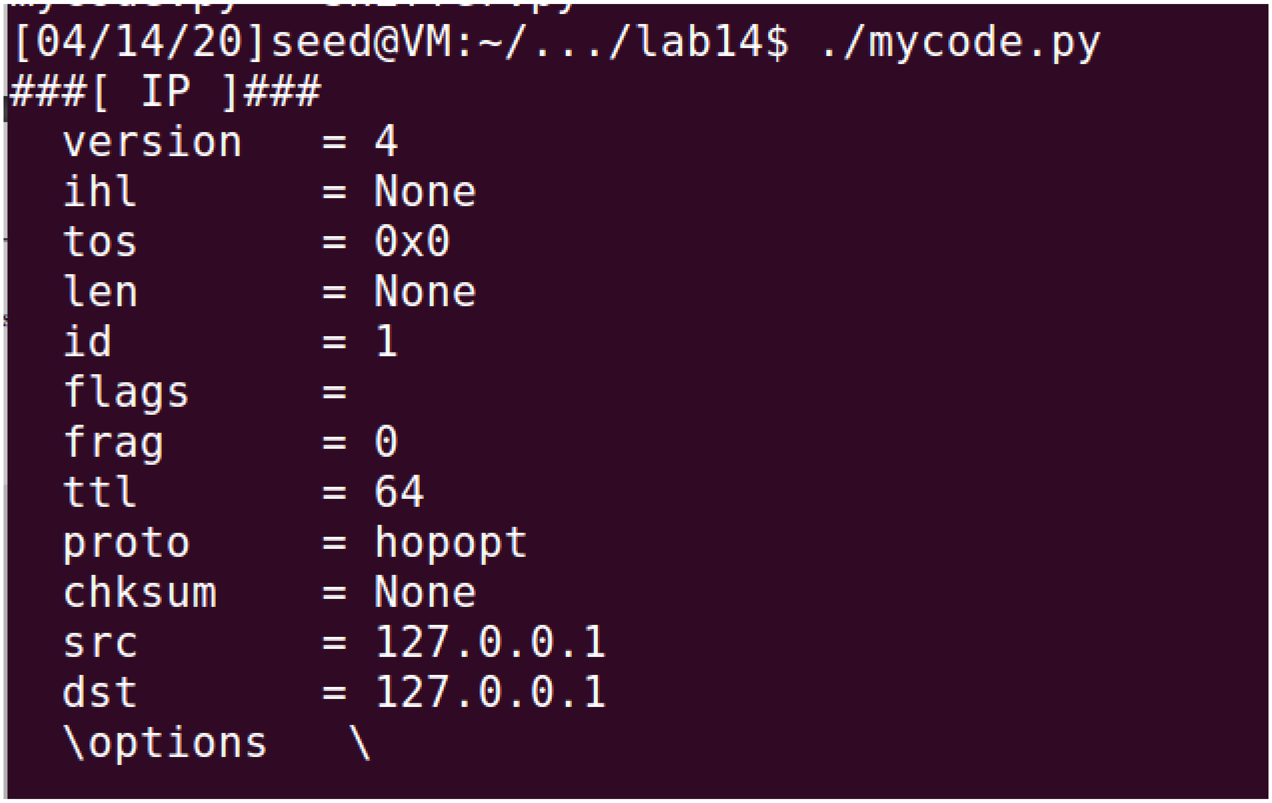
Lab 15
A20453991
Varun Gunda

Packet Sniffing and Spoofing Lab

Lab Task Set 1: Using Tools to Sniff and Spoof Packets

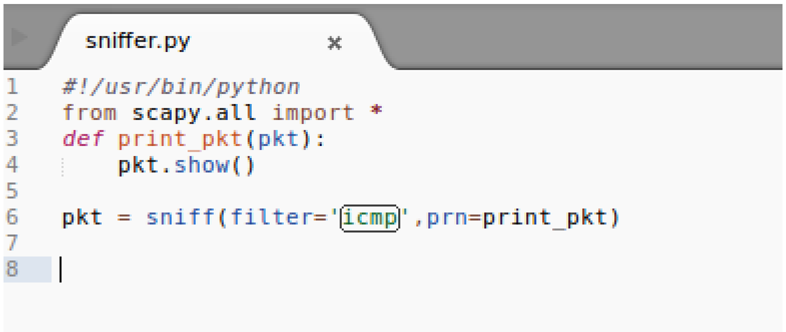


```
#!/usr/bin/python
from scapy.all import *
a = IP()
a.show()
```



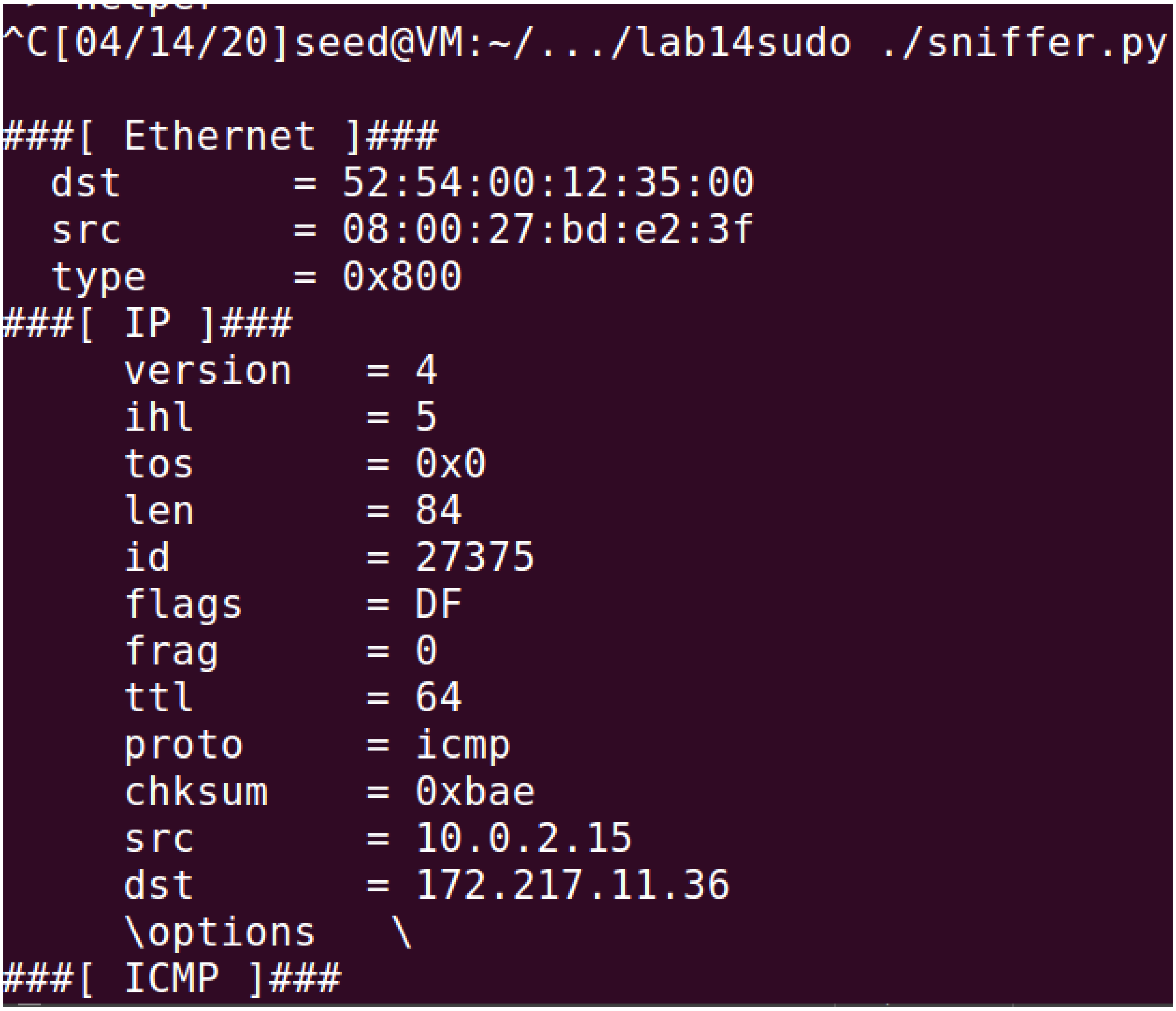
```
[04/14/20]seed@VM:~/.../lab14$ ./mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     = 
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
```

Task 1.1: Sniffing Packets:

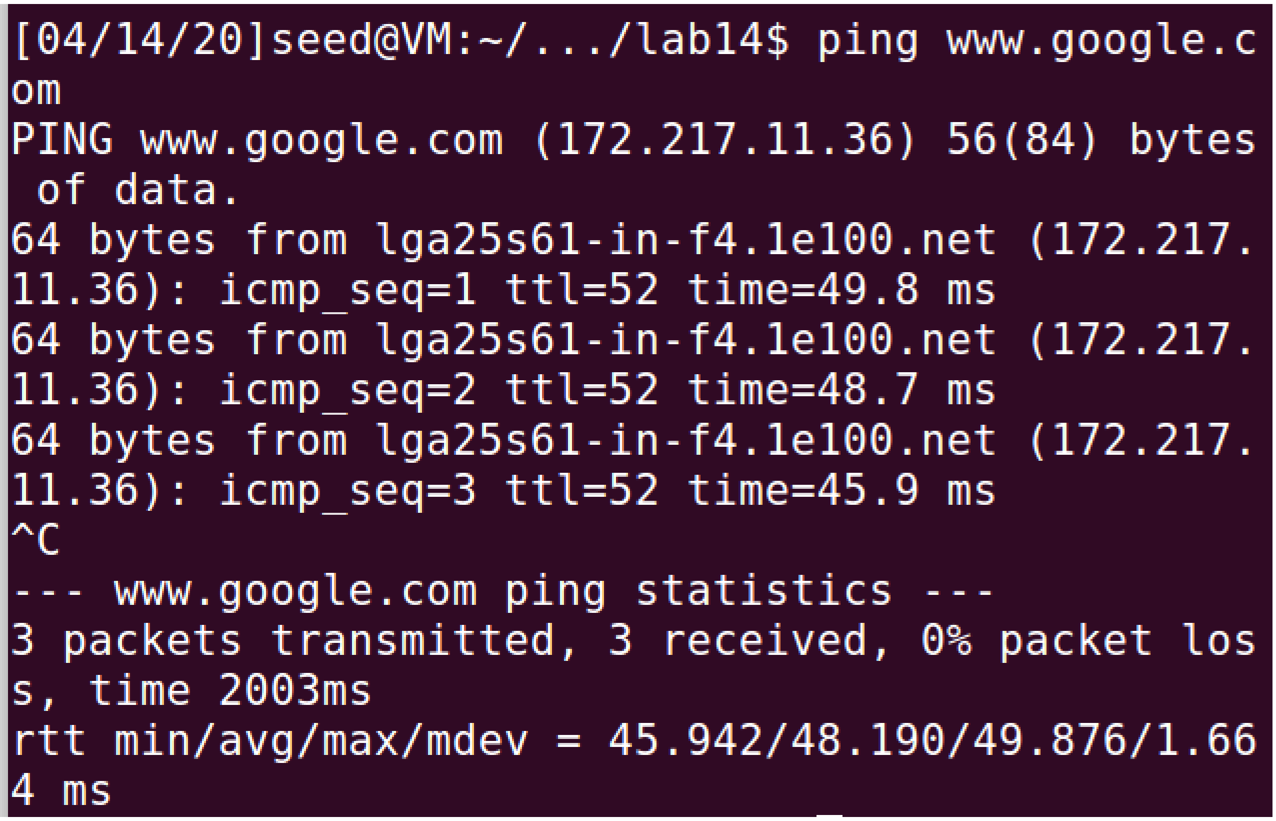


```
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)

```

With sudo:
On pinging google.com from another terminal:



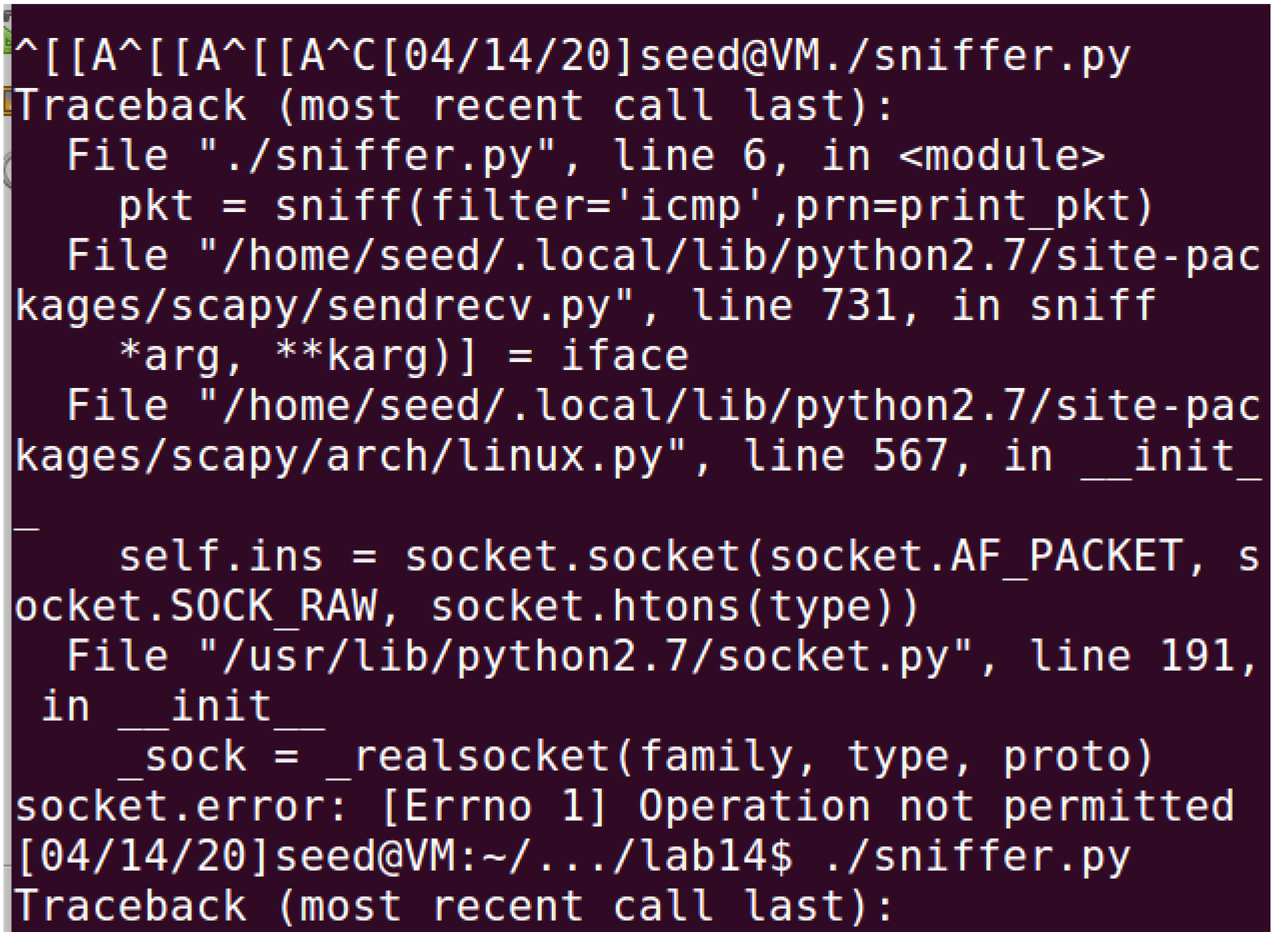
```
^C[04/14/20]seed@VM:~/.../lab14sudo ./sniffer.py

###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bd:e2:3f
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 27375
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xbae
     src       = 10.0.2.15
     dst       = 172.217.11.36
     \options   \
###[ ICMP ]###
```



```
[04/14/20]seed@VM:~/.../lab14$ ping www.google.c
om
PING www.google.com (172.217.11.36) 56(84) bytes
 of data.
64 bytes from lga25s61-in-f4.1e100.net (172.217.
11.36): icmp_seq=1 ttl=52 time=49.8 ms
64 bytes from lga25s61-in-f4.1e100.net (172.217.
11.36): icmp_seq=2 ttl=52 time=48.7 ms
64 bytes from lga25s61-in-f4.1e100.net (172.217.
11.36): icmp_seq=3 ttl=52 time=45.9 ms
^C
--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet los
s, time 2003ms
rtt min/avg/max/mdev = 45.942/48.190/49.876/1.66
4 ms
```

Without sudo:

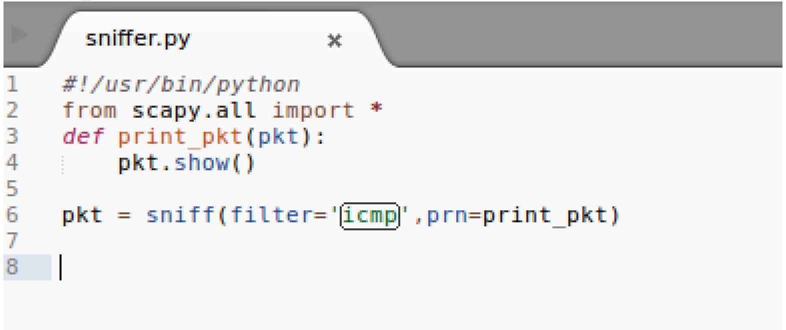
Without sudo, we don’t have permissions to sniff the packets:



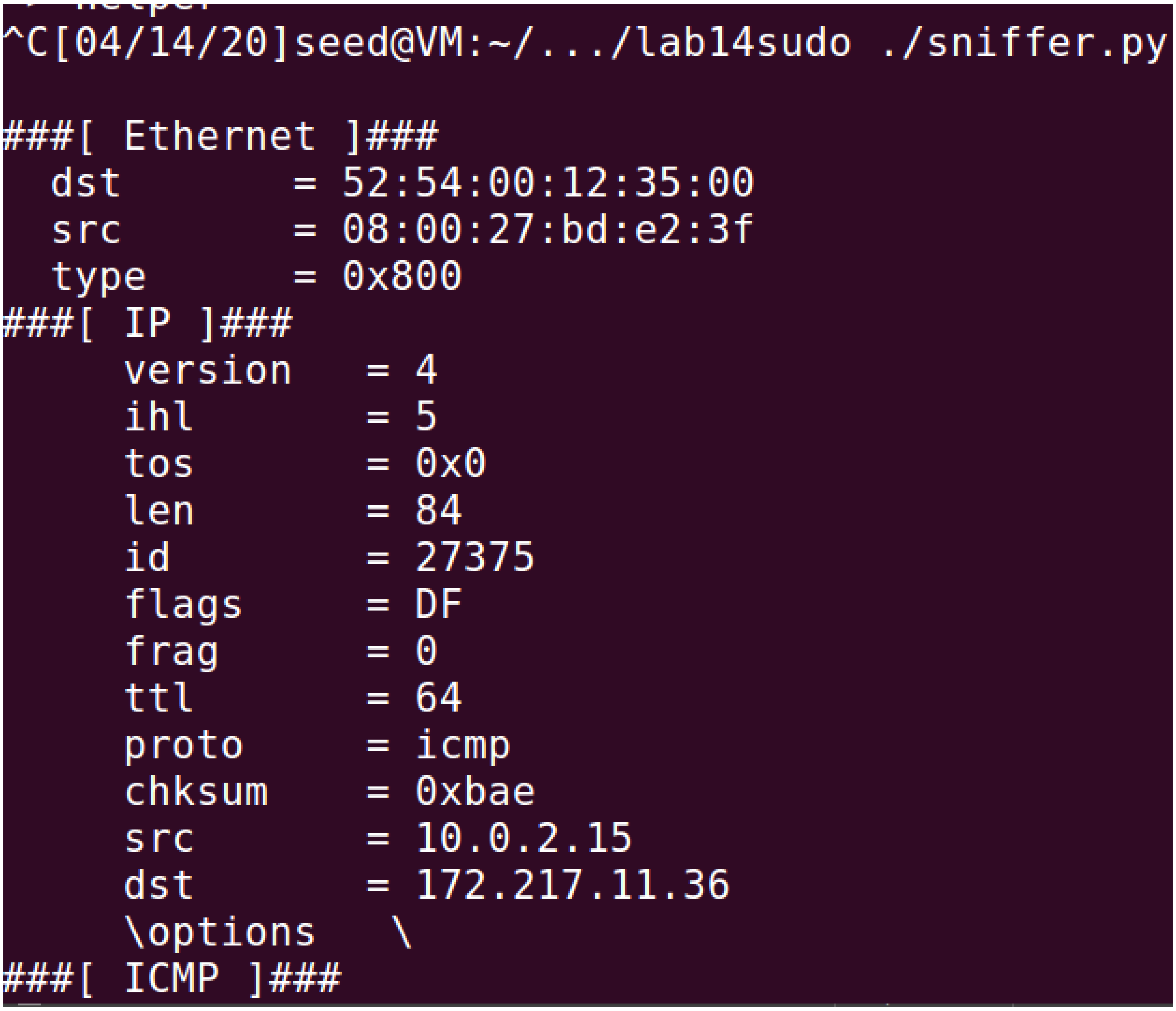
```
^[[A^[[A^[[A^C[04/14/20]seed@VM./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 6, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-pac
kages/scapy/sendrecv.py", line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-pac
kages/scapy/arch/linux.py", line 567, in __init_
_
    self.ins = socket.socket(socket.AF_PACKET, s
ocket.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
[04/14/20]seed@VM:~/.../lab14$ ./sniffer.py
Traceback (most recent call last):
```

Task 1.1B:

- **Capture only the ICMP packet**



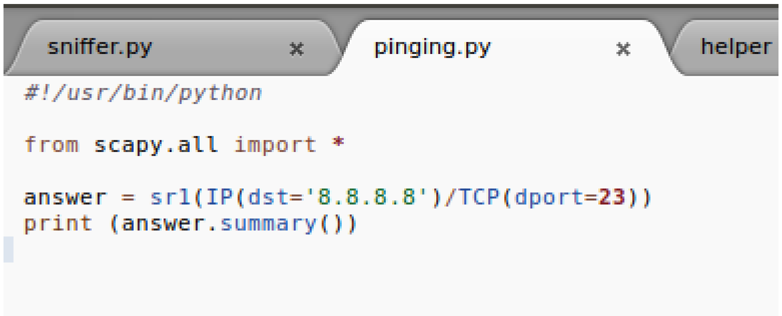
```
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='icmp',prn=print_pkt)

```



```
^C[04/14/20]seed@VM:~/.../lab14sudo ./sniffer.py

###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bd:e2:3f
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 27375
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xbae
     src       = 10.0.2.15
     dst       = 172.217.11.36
     \options   \
###[ ICMP ]###
```

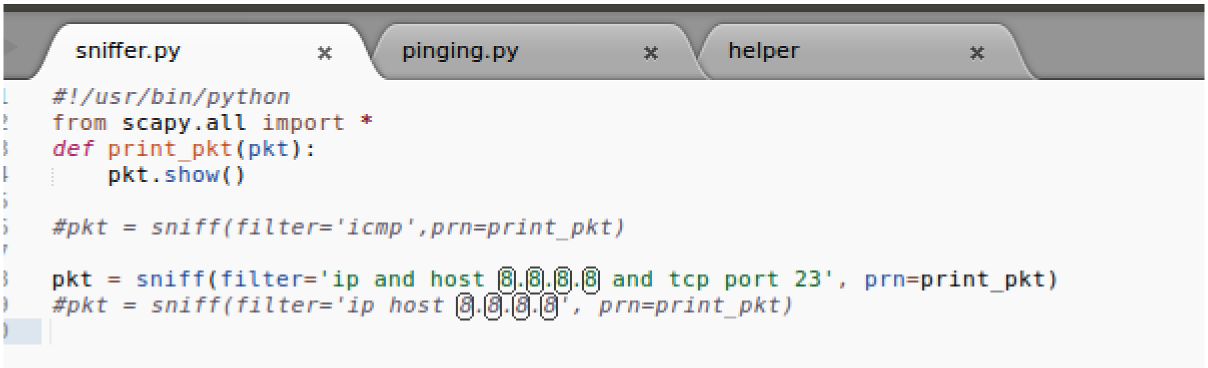
- **Capture any TCP packet that comes from a particular IP and with a destination port number 23**

The pinging script here is used to send the and receive packets



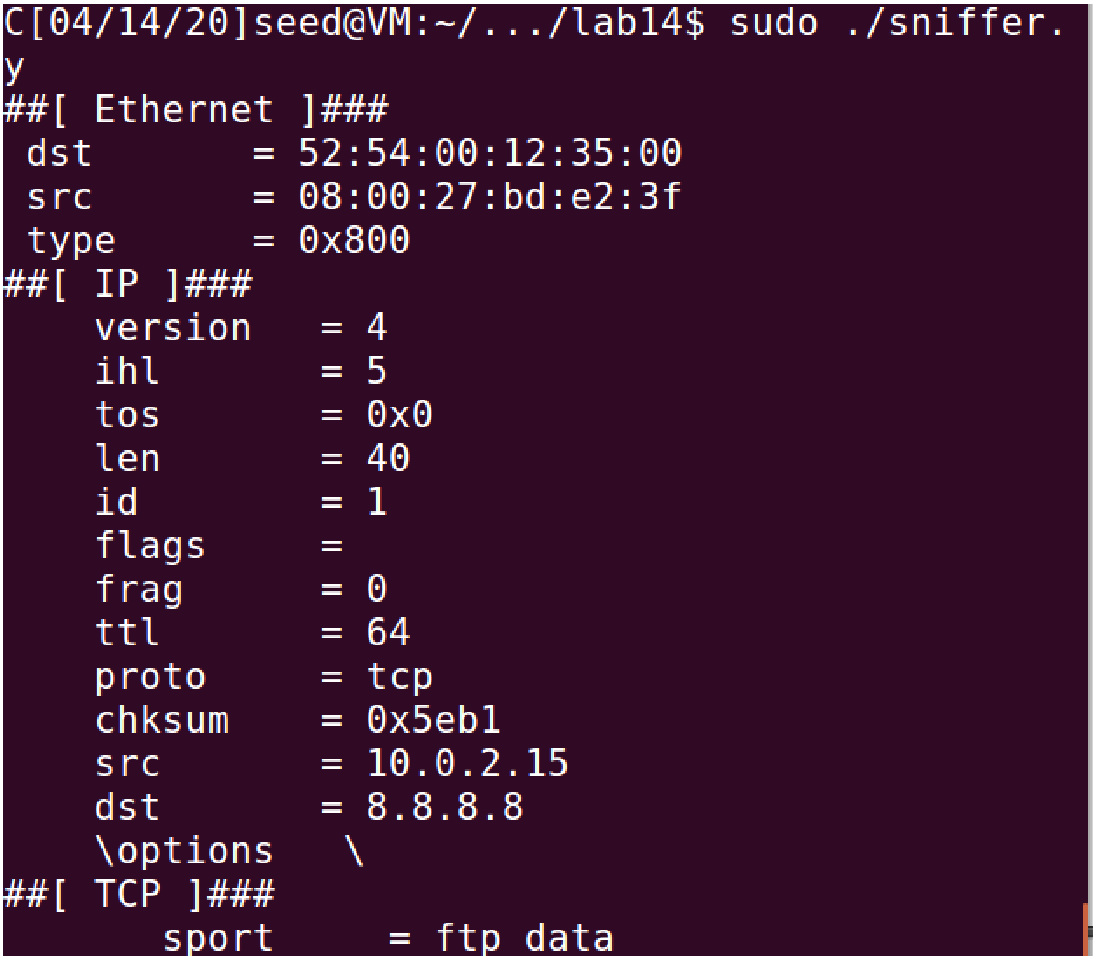
```
#!/usr/bin/python

from scapy.all import *

answer = sr1(IP(dst='8.8.8.8')/TCP(dport=23))
print (answer.summary())
```

The filter is shown here:



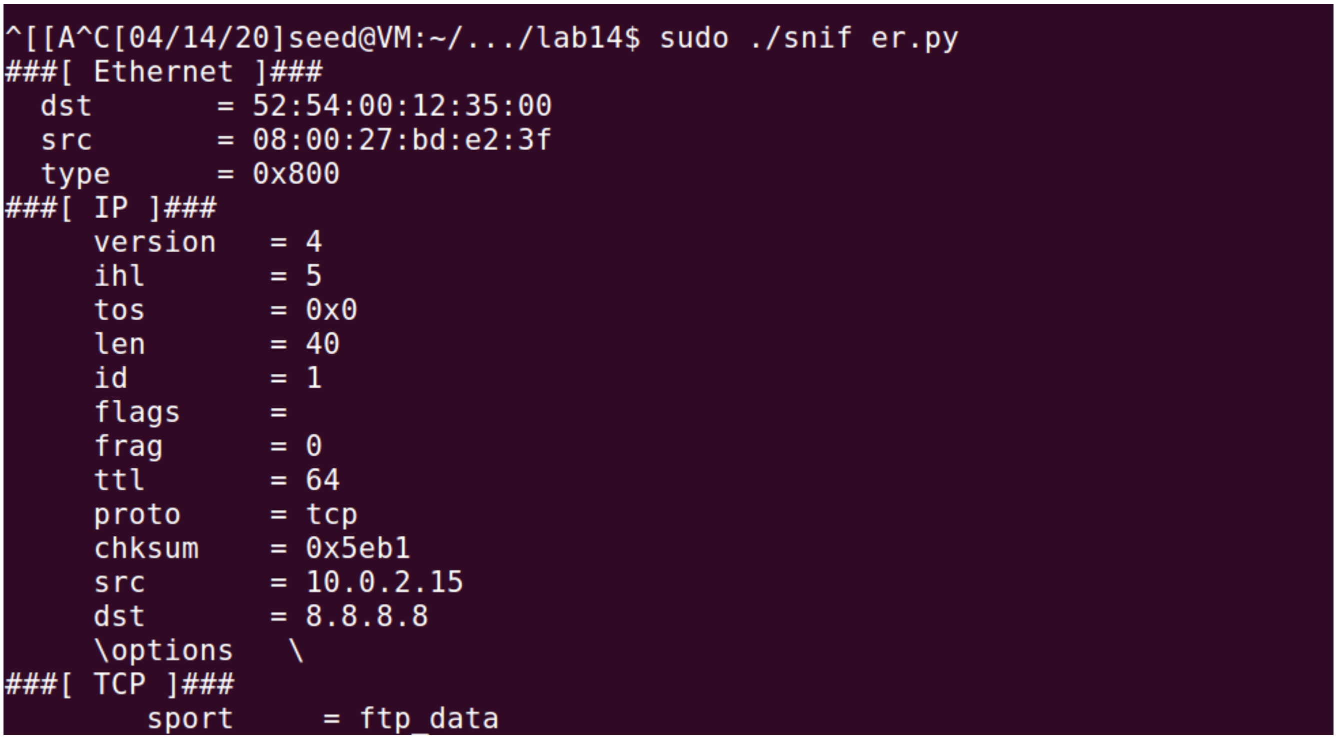
```
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

#pkt = sniff(filter='icmp',prn=print_pkt)

pkt = sniff(filter='ip and host 8.8.8.8 and tcp port 23', prn=print_pkt)
#pkt = sniff(filter='ip host 8.8.8.8', prn=print_pkt)
```

We can see that the packet is captured



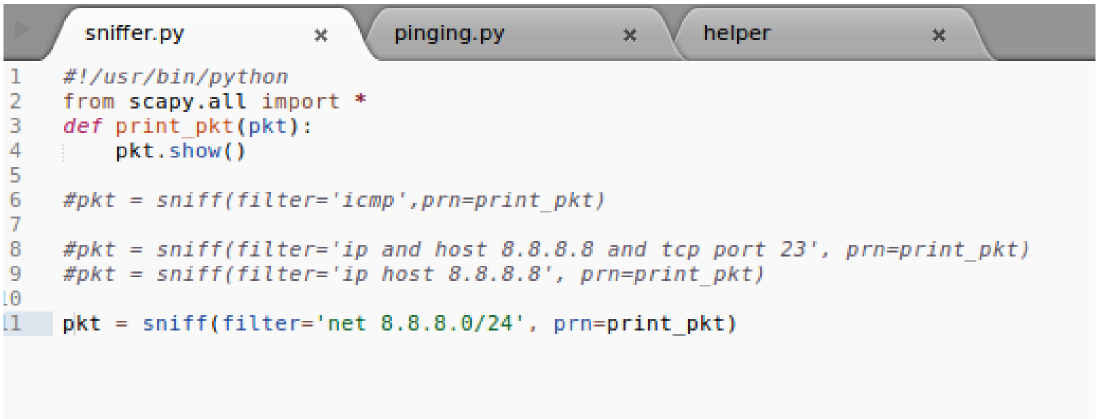
```
C[04/14/20]seed@VM:~/.../lab14$ sudo ./sniffer.
y
###[ Ethernet ]###
 dst       = 52:54:00:12:35:00
 src       = 08:00:27:bd:e2:3f
 type      = 0x800
###[ IP ]###
    version   = 4
    ihl       = 5
    tos       = 0x0
    len       = 40
    id        = 1
    flags     =
    frag      = 0
    ttl       = 64
    proto     = tcp
    chksum    = 0x5eb1
    src       = 10.0.2.15
    dst       = 8.8.8.8
    \options   \
###[ TCP ]###
       sport     = ftp data
```

- **Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.**



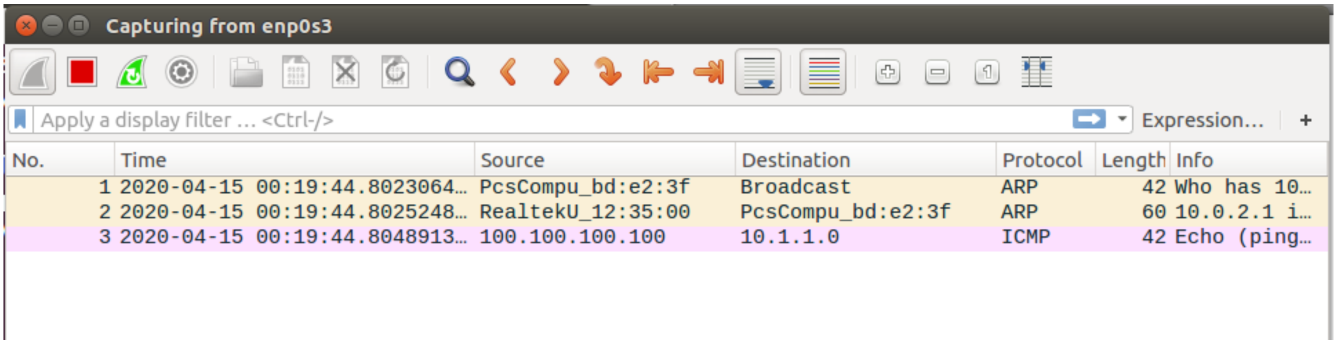
```
^[[A^C[04/14/20]seed@VM:~/.../lab14$ sudo ./snif er.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bd:e2:3f
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x5eb1
     src       = 10.0.2.15
     dst       = 8.8.8.8
     \options   \
###[ TCP ]###
        sport     = ftp_data
```

The filter is changed as seen here. Now when we send the packet, it is caught by the sniffer.



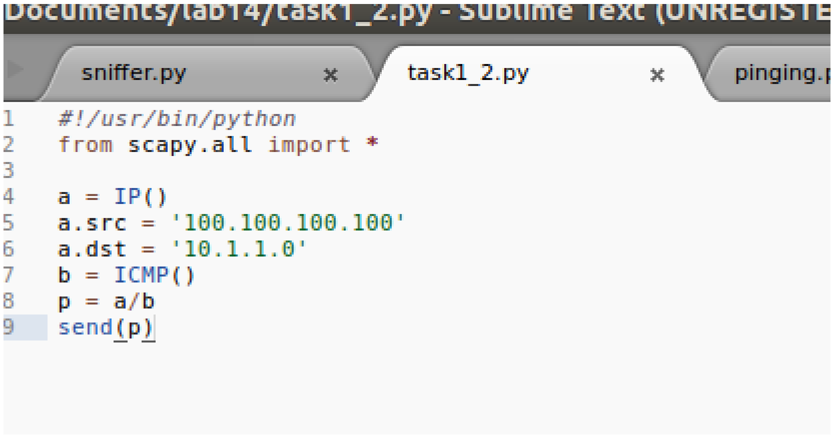
```python
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

#pkt = sniff(filter='icmp',prn=print_pkt)

#pkt = sniff(filter='ip and host 8.8.8.8 and tcp port 23', prn=print_pkt)
#pkt = sniff(filter='ip host 8.8.8.8', prn=print_pkt)

pkt = sniff(filter='net 8.8.8.0/24', prn=print_pkt)
```

Task 1.2: Spoofing ICMP Packets



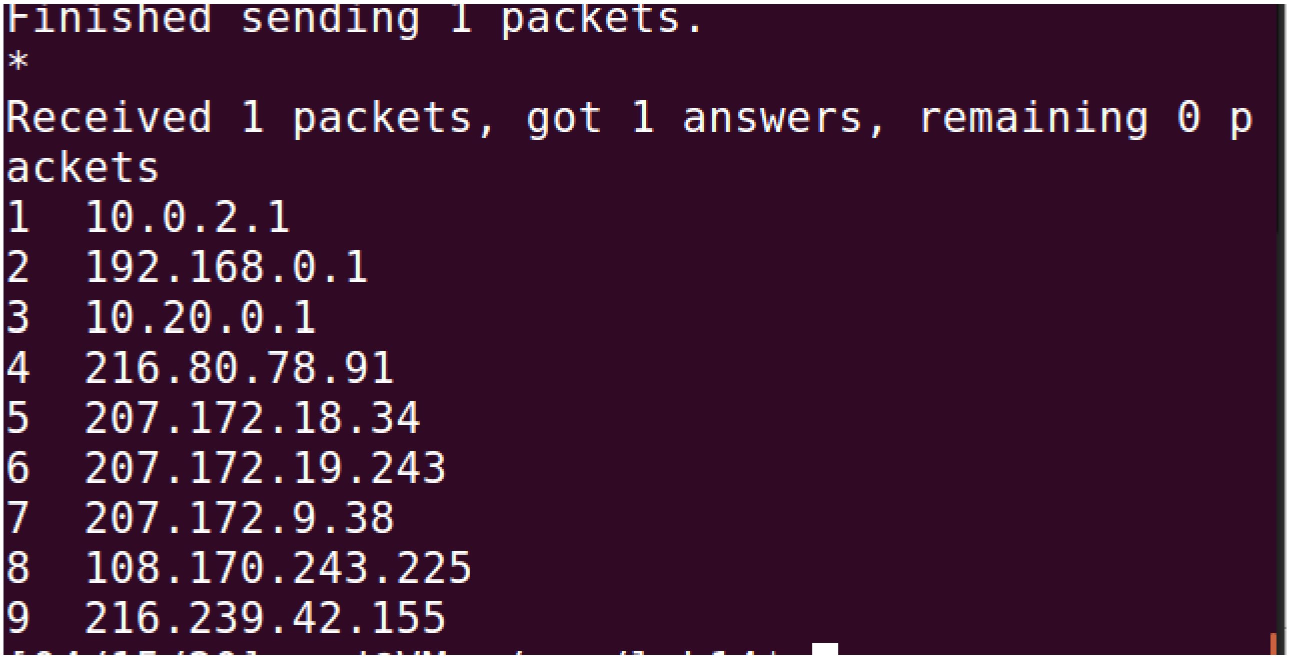
No.	Time	Source	Destination	Protocol	Length	Info
1	2020-04-15 00:19:44.8023064…	PcsCompu_bd:e2:3f	Broadcast	ARP	42	Who has 10…
2	2020-04-15 00:19:44.8025248…	RealtekU_12:35:00	PcsCompu_bd:e2:3f	ARP	60	10.0.2.1 i…
3	2020-04-15 00:19:44.8048913…	100.100.100.100	10.1.1.0	ICMP	42	Echo (ping…

As seen from the above wireshark image, the ICMP echo request packer was spoofed with an arbitrary source IP address.



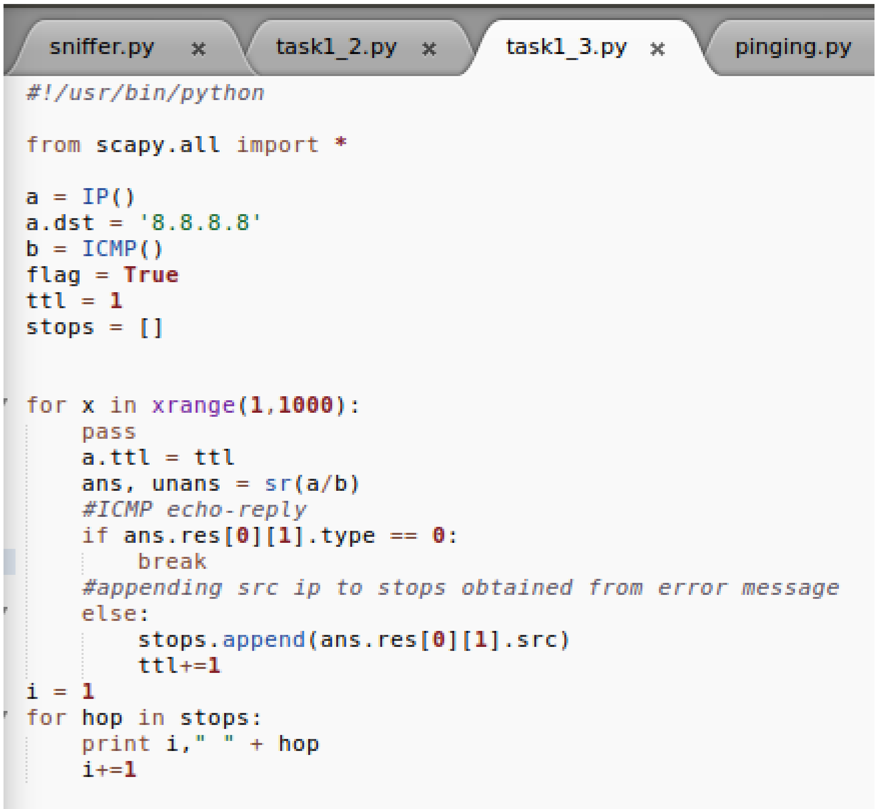
```
#!/usr/bin/python
from scapy.all import *

a = IP()
a.src = '100.100.100.100'
a.dst = '10.1.1.0'
b = ICMP()
p = a/b
send(p)
```

Task 1.3: Traceroute



```
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 p
ackets
1  10.0.2.1
2  192.168.0.1
3  10.20.0.1
4  216.80.78.91
5  207.172.18.34
6  207.172.19.243
7  207.172.9.38
8  108.170.243.225
9  216.239.42.155
```

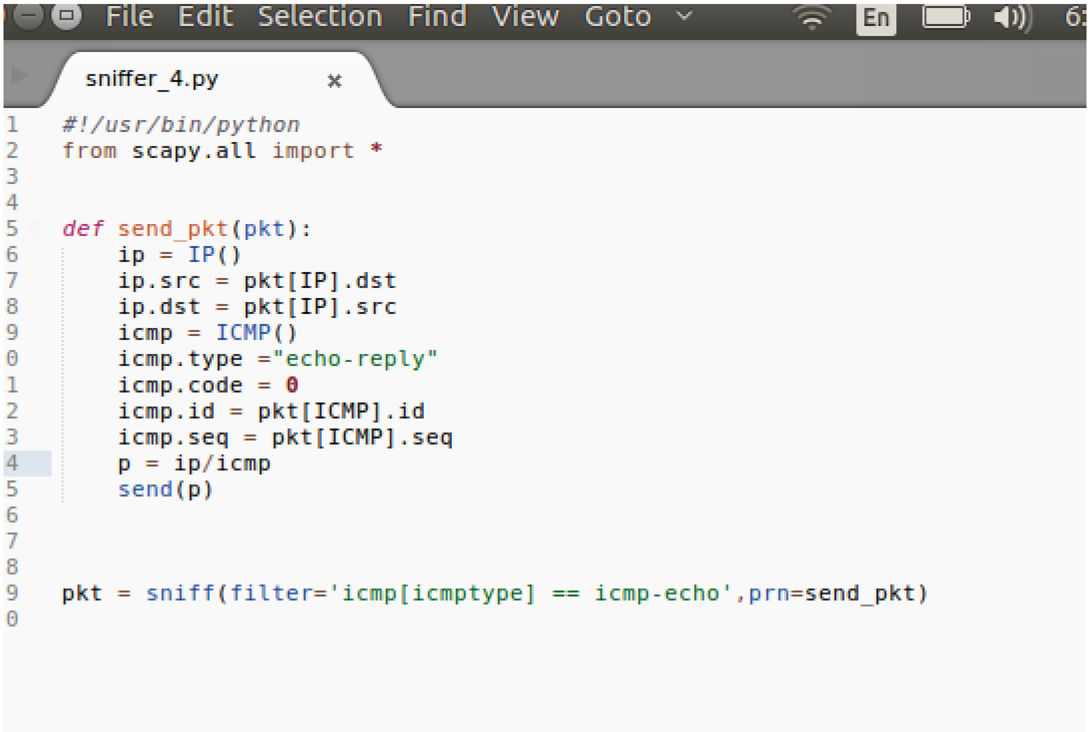
The code used is shown here and the hops are obtained as shown in the above image.



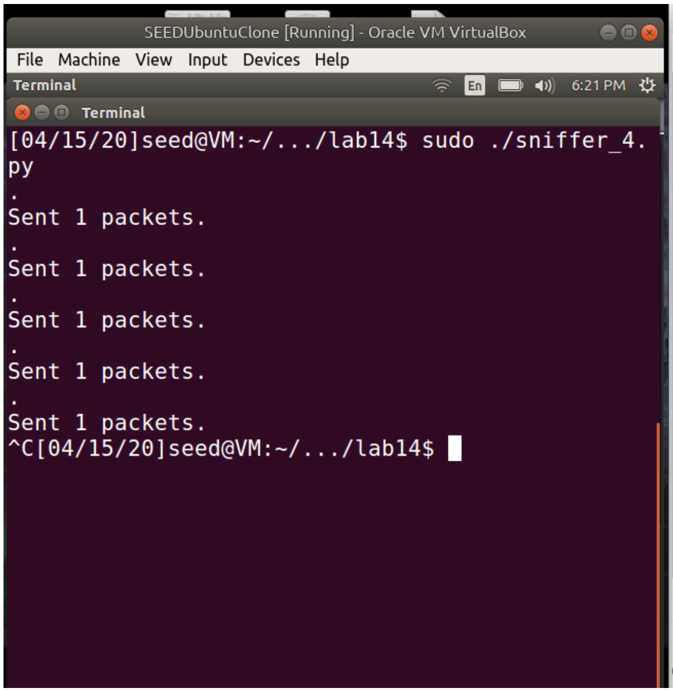
```
#!/usr/bin/python

from scapy.all import *

a = IP()
a.dst = '8.8.8.8'
b = ICMP()
flag = True
ttl = 1
stops = []


for x in xrange(1,1000):
    pass
    a.ttl = ttl
    ans, unans = sr(a/b)
    #ICMP echo-reply
    if ans.res[0][1].type == 0:
        break
    #appending src ip to stops obtained from error message
    else:
        stops.append(ans.res[0][1].src)
        ttl+=1
i = 1
for hop in stops:
    print i," " + hop
    i+=1
```

Task 1.4: Sniffing and-then Spoofing

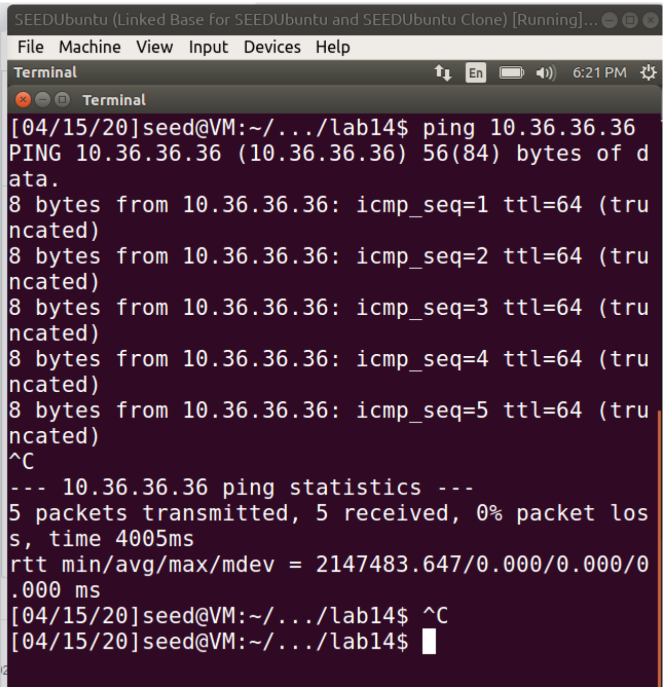
The code used in the sniffer program to sniff the packets in the network and send the response if packet is of echo type.



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


def send_pkt(pkt):
    ip = IP()
    ip.src = pkt[IP].dst
    ip.dst = pkt[IP].src
    icmp = ICMP()
    icmp.type ="echo-reply"
    icmp.code = 0
    icmp.id = pkt[ICMP].id
    icmp.seq = pkt[ICMP].seq
    p = ip/icmp
    send(p)



pkt = sniff(filter='icmp[icmptype] == icmp-echo',prn=send_pkt)
```


SEEDUbuntuClone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
En
6:21 PM
Terminal

```
[04/15/20]seed@VM:~/.../lab14$ sudo ./sniffer_4.
py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
^C[04/15/20]seed@VM:~/.../lab14$ 
```


SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone) [Running]...
File Machine View Input Devices Help
Terminal
En
6:21 PM
Terminal

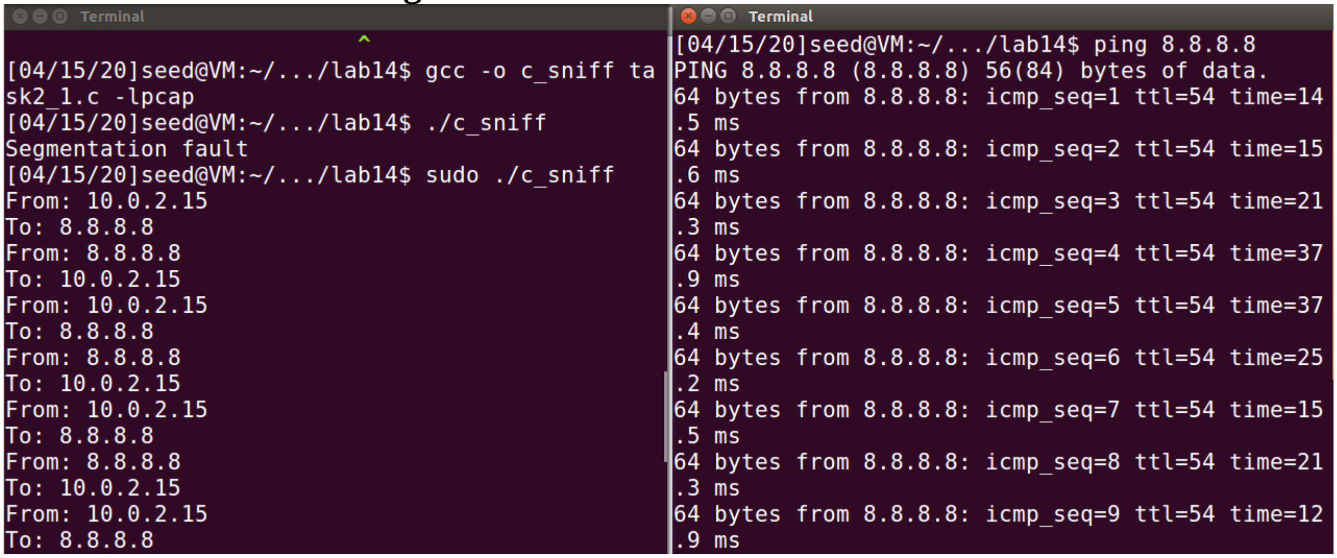
```
[04/15/20]seed@VM:~/.../lab14$ ping 10.36.36.36
PING 10.36.36.36 (10.36.36.36) 56(84) bytes of d
ata.
8 bytes from 10.36.36.36: icmp_seq=1 ttl=64 (tru
ncated)
8 bytes from 10.36.36.36: icmp_seq=2 ttl=64 (tru
ncated)
8 bytes from 10.36.36.36: icmp_seq=3 ttl=64 (tru
ncated)
8 bytes from 10.36.36.36: icmp_seq=4 ttl=64 (tru
ncated)
8 bytes from 10.36.36.36: icmp_seq=5 ttl=64 (tru
ncated)
^C
--- 10.36.36.36 ping statistics ---
5 packets transmitted, 5 received, 0% packet los
s, time 4005ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0
.000 ms
[04/15/20]seed@VM:~/.../lab14$ ^C
[04/15/20]seed@VM:~/.../lab14$ 
```

I ran two Vms as shown above, one for sending the packet and other to spoof the packet. There is no machine with ip address 10.36.36.36. However, the sniffer program sniffs this packet and sends the response on behalf of 10.36.36.36. Hence we see that our attack works.

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

3.1 Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

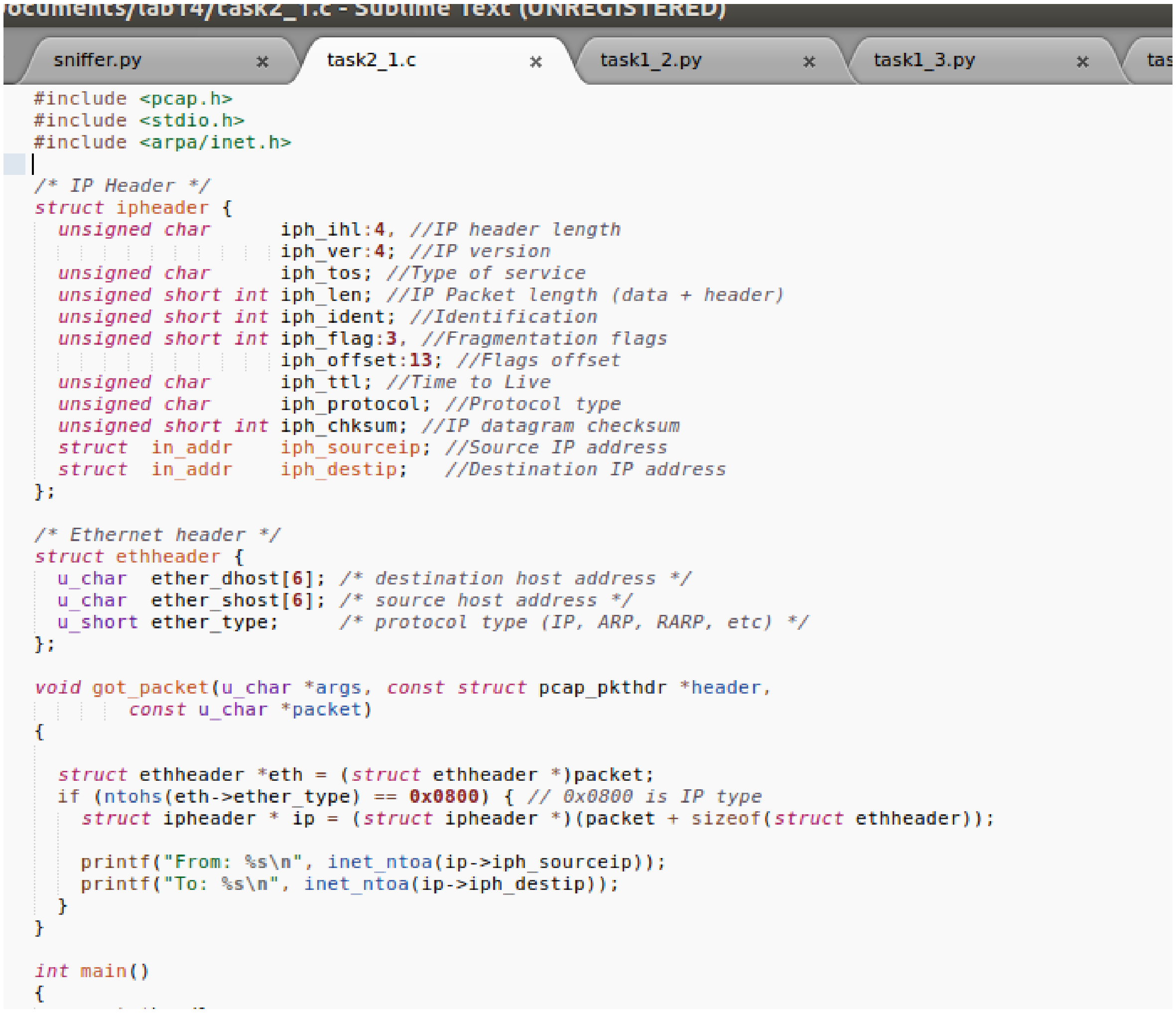


```
[04/15/20]seed@VM:~/.../lab14$ gcc -o c_sniff ta
sk2_1.c -lpcap
[04/15/20]seed@VM:~/.../lab14$ ./c_sniff
Segmentation fault
[04/15/20]seed@VM:~/.../lab14$ sudo ./c_sniff
From: 10.0.2.15
To: 8.8.8.8
From: 8.8.8.8
To: 10.0.2.15
From: 10.0.2.15
To: 8.8.8.8
From: 8.8.8.8
To: 10.0.2.15
From: 10.0.2.15
To: 8.8.8.8
From: 8.8.8.8
To: 10.0.2.15
From: 10.0.2.15
To: 8.8.8.8
```

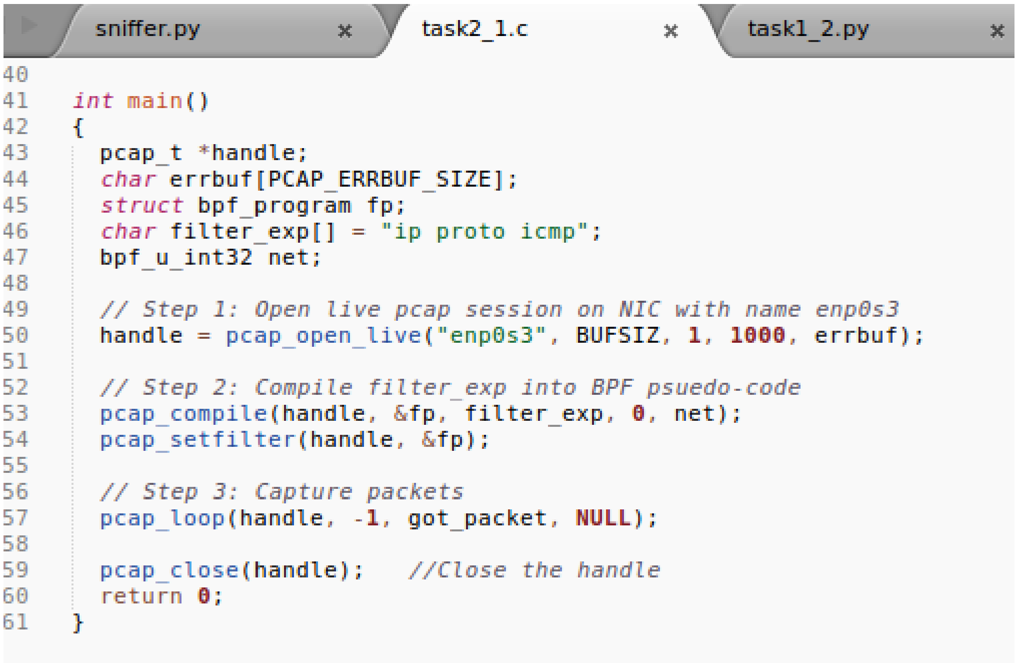
```
[04/15/20]seed@VM:~/.../lab14$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=54 time=14
.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=54 time=15
.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=54 time=21
.3 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=54 time=37
.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=54 time=37
.4 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=54 time=25
.2 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=54 time=15
.5 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=54 time=21
.3 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=54 time=12
.9 ms
```

As seen above, the c program successful running and displaying the source and destination address of each packet it captured.

The code that is used is shown in the next page.



```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>

/* IP Header */
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 + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation 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_destip;   //Destination IP address
};

/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{

  struct ethheader *eth = (struct ethheader *)packet;
  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

    printf("From: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("To: %s\n", inet_ntoa(ip->iph_destip));
  }
}

int main()
{
```



```

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

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 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;
}
```

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.

The sequence of library calls are: open device for capturing(pcap_open_live), set the BPF packet filter (pcap_compile and pcap_setfilter) and capture packets (pcap_loop) and finally close the handle (pcap_close)

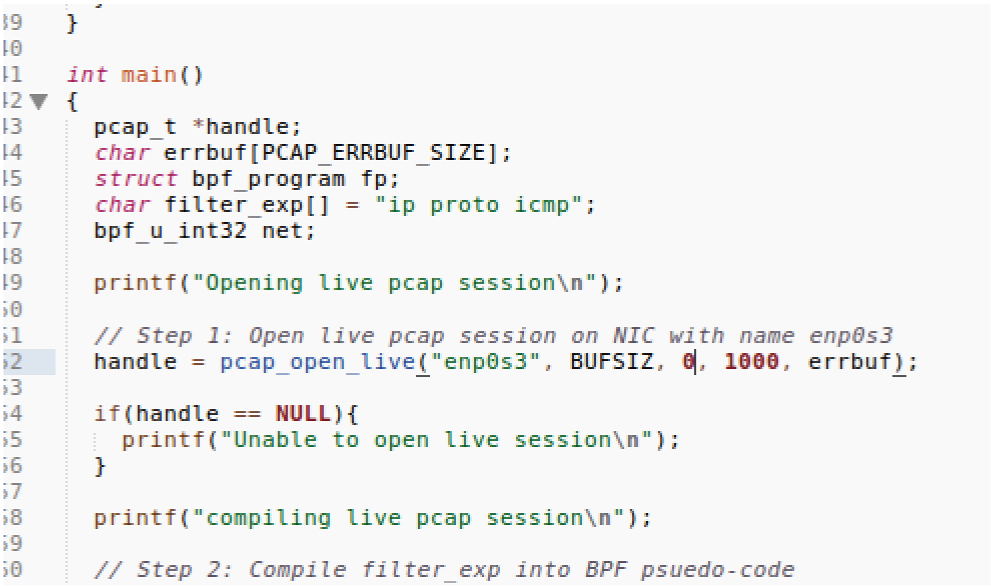
Opening live pcap session step initializes a raw socket and set network device into promiscuous mode and binds the socket to the card using setsocketopt(). In step2, pcap API compiles boolean predicate expressions to low-level BPF programs. In step 3, the library call pcap_loop() is used to enter the main execution loop of pcap session. Whenever a packet is captured by pcap, the callback function is invoked

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?

pcap_open_live library call requires root privilege. This is because only root processes and processes with the CAP_NET_RAW capabilities can create raw sockets and this creation is done at opening live pcap session stage. Hence, we need root privilege to run a sniffer program.

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.

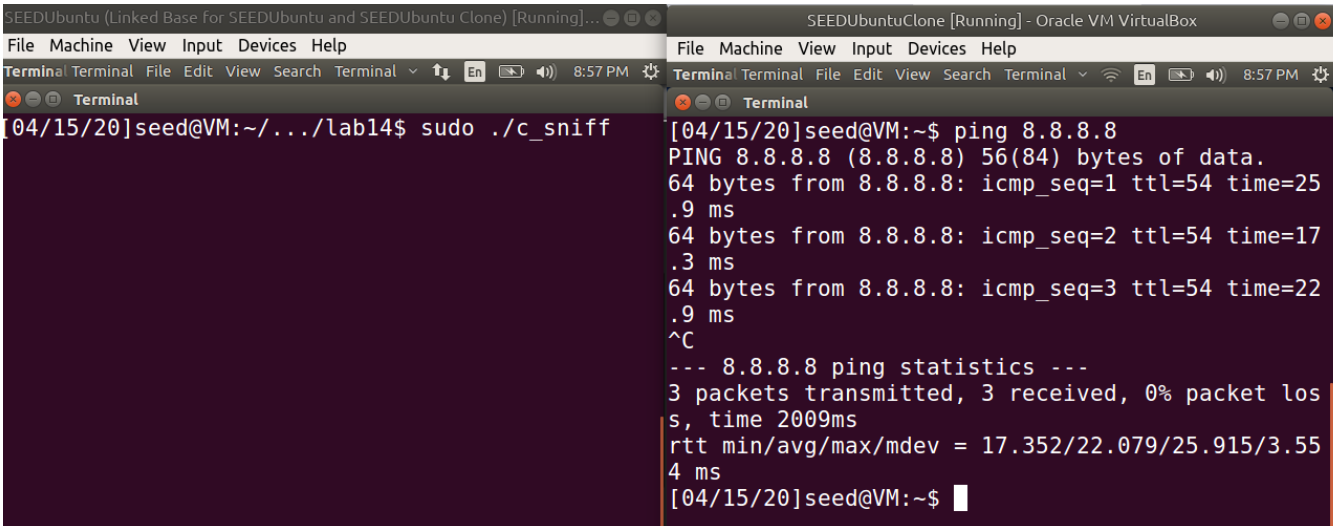
The promiscuous mode is turned off by sending 0 to the pcap_open_live session.



```
}

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

  printf("Opening live pcap session\n");

  // Step 1: Open live pcap session on NIC with name enp0s3
  handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

  if(handle == NULL){
    printf("Unable to open live session\n");
  }

  printf("compiling live pcap session\n");

  // Step 2: Compile filter_exp into BPF psuedo-code
```

As expected, without promiscuous mode set, the VM A on the same network as VM B, can't sniff the packets sent out by VM B as seen below.

Without Promiscuous mode:

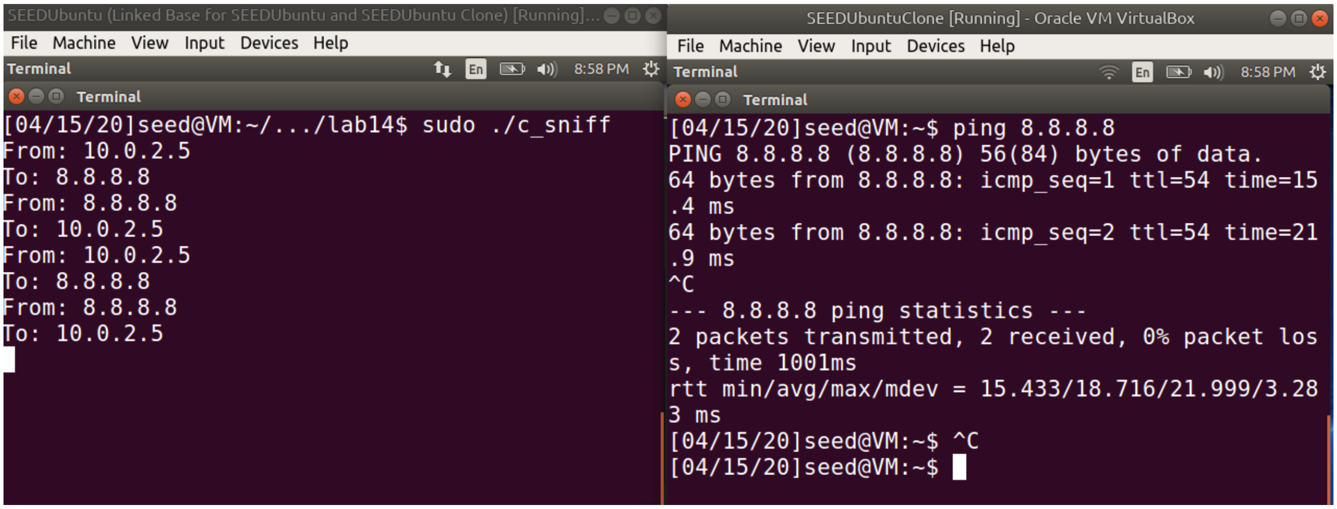


SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone) [Running]...
File Machine View Input Devices Help
Terminal Terminal File Edit View Search Terminal
8:57 PM
Terminal
SEEDUbuntuClone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Terminal File Edit View Search Terminal
8:57 PM
Terminal

```
[04/15/20]seed@VM:~/.../lab14$ sudo ./c_sniff
```

```
[04/15/20]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=54 time=25
.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=54 time=17
.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=54 time=22
.9 ms
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet los
s, time 2009ms
rtt min/avg/max/mdev = 17.352/22.079/25.915/3.55
4 ms
[04/15/20]seed@VM:~$
```

With Promiscuous mode:

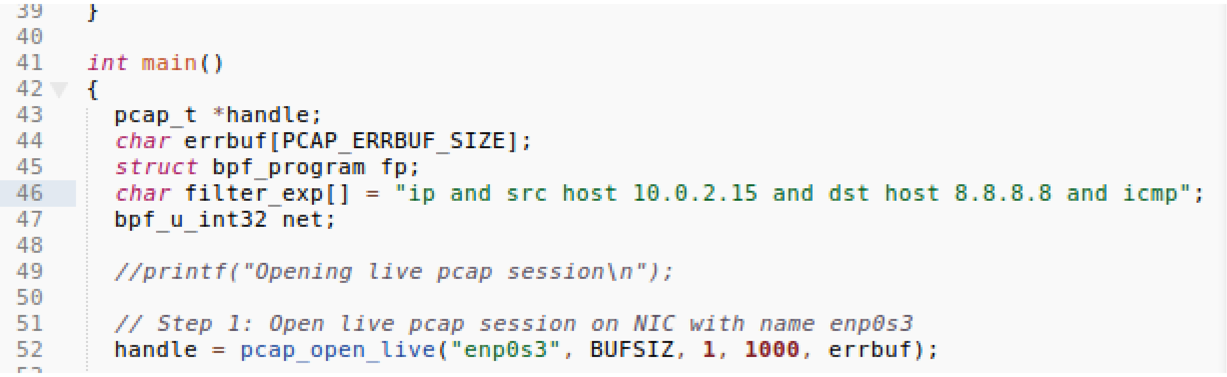


SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone) [Running]...
File Machine View Input Devices Help
Terminal
8:58 PM
Terminal
SEEDUbuntuClone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
8:58 PM
Terminal

```
[04/15/20]seed@VM:~/.../lab14$ sudo ./c_sniff
From: 10.0.2.5
To: 8.8.8.8
From: 8.8.8.8
To: 10.0.2.5
From: 10.0.2.5
To: 8.8.8.8
From: 8.8.8.8
To: 10.0.2.5
```

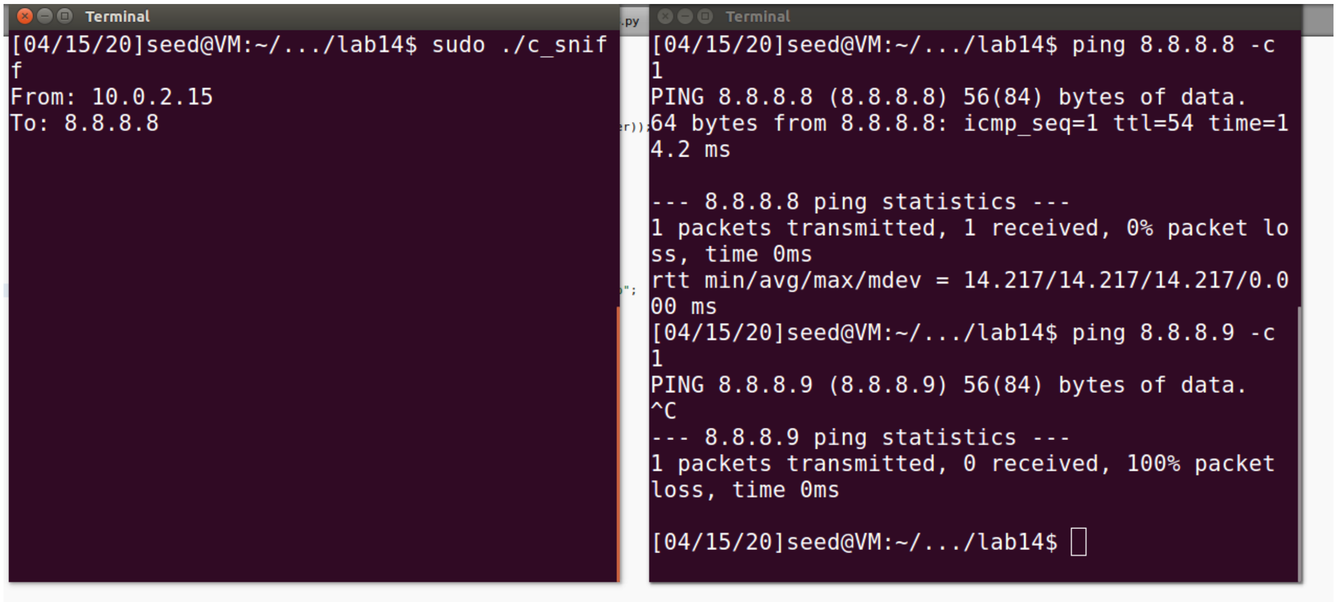
```
[04/15/20]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=54 time=15
.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=54 time=21
.9 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet los
s, time 1001ms
rtt min/avg/max/mdev = 15.433/18.716/21.999/3.28
3 ms
[04/15/20]seed@VM:~$ ^C
[04/15/20]seed@VM:~$
```

Task 2.1B: Writing Filters.
Capture the ICMP packets between two specific hosts.



```
39  }
40
41  int main()
42  {
43      pcap_t *handle;
44      char errbuf[PCAP_ERRBUF_SIZE];
45      struct bpf_program fp;
46      char filter_exp[] = "ip and src host 10.0.2.15 and dst host 8.8.8.8 and icmp";
47      bpf_u_int32 net;
48
49      //printf("Opening live pcap session\n");
50
51      // Step 1: Open live pcap session on NIC with name enp0s3
52      handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

The change is made in line 46 in task2_1.c as shown in the above picture to capture packets between two hosts. We can add another filter in line 46: **or (dst host 10.0.2.15 and src host 8.8.8.8)** so that packets in either direction between the two can be caught.



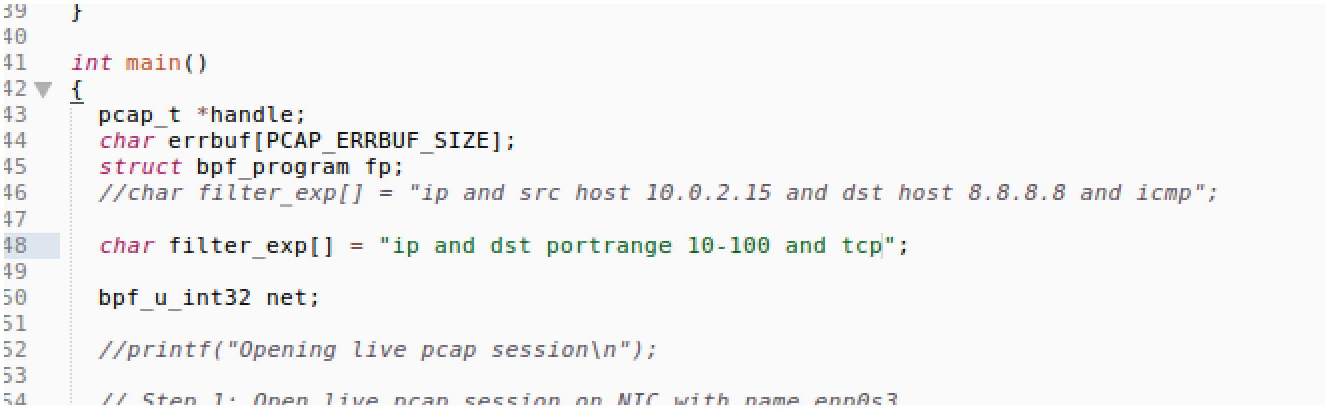
```
[04/15/20]seed@VM:~/.../lab14$ sudo ./c_snif
f
From: 10.0.2.15
To: 8.8.8.8
```

```
[04/15/20]seed@VM:~/.../lab14$ ping 8.8.8.8 -c
1
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=54 time=1
4.2 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet lo
ss, time 0ms
rtt min/avg/max/mdev = 14.217/14.217/14.217/0.0
00 ms
[04/15/20]seed@VM:~/.../lab14$ ping 8.8.8.9 -c
1
PING 8.8.8.9 (8.8.8.9) 56(84) bytes of data.
^C
--- 8.8.8.9 ping statistics ---
1 packets transmitted, 0 received, 100% packet
loss, time 0ms

[04/15/20]seed@VM:~/.../lab14$ 
```

As seen from above, the sniffer receives the packet and displays it since it has source as host 10.0.2.15 and destination has 8.8.8.8 but when the destination is 8.8.8.9, packet is rejected by sniffer program.

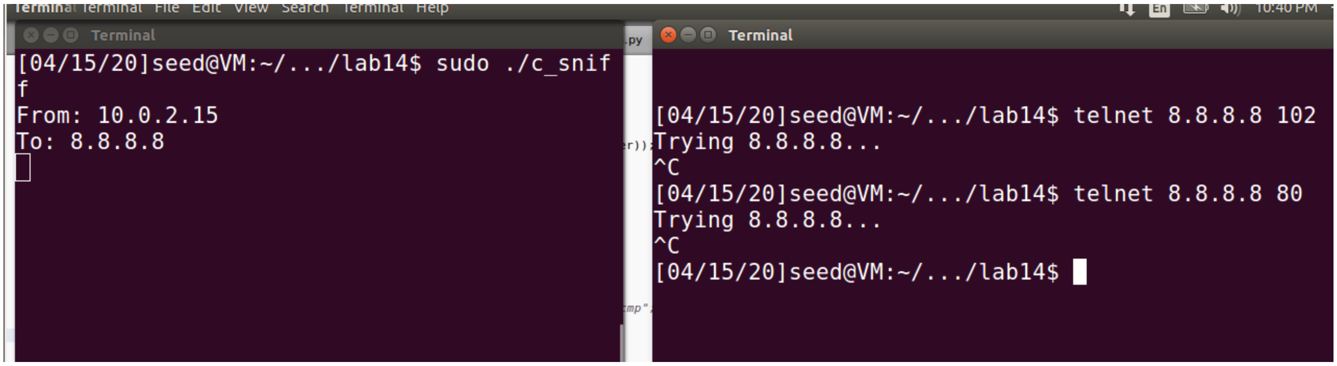
Capture the TCP packets with a destination port number in the range from 10 to 100

The line 48 below in task2_1.c contains the filter for this.



```
39    }
40
41    int main()
42    {
43      pcap_t *handle;
44      char errbuf[PCAP_ERRBUF_SIZE];
45      struct bpf_program fp;
46      //char filter_exp[] = "ip and src host 10.0.2.15 and dst host 8.8.8.8 and icmp";
47
48      char filter_exp[] = "ip and dst portrange 10-100 and tcp";
49
50      bpf_u_int32 net;
51
52      //printf("Opening live pcap session\n");
53
54      // Step 1: Open live pcap session on NIC with name enp0s3
```

As we can see below, when we try to connect to port 102 of 8.8.8.8, sniffer program

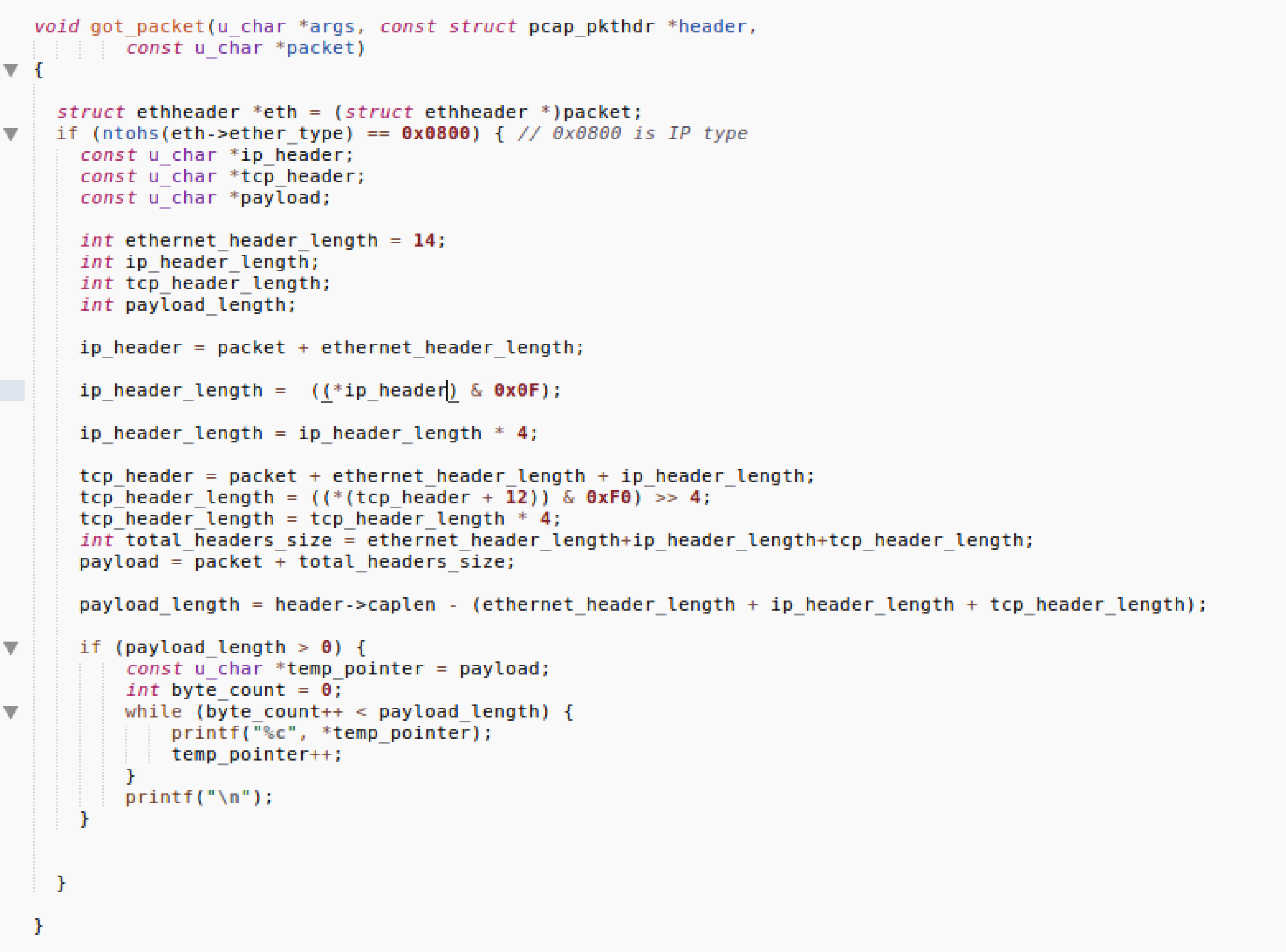


```
[04/15/20]seed@VM:~/.../lab14$ sudo ./c_snif
f
From: 10.0.2.15
To: 8.8.8.8
```

```
[04/15/20]seed@VM:~/.../lab14$ telnet 8.8.8.8 102
Trying 8.8.8.8...
^C
[04/15/20]seed@VM:~/.../lab14$ telnet 8.8.8.8 80
Trying 8.8.8.8...
^C
[04/15/20]seed@VM:~/.../lab14$
```

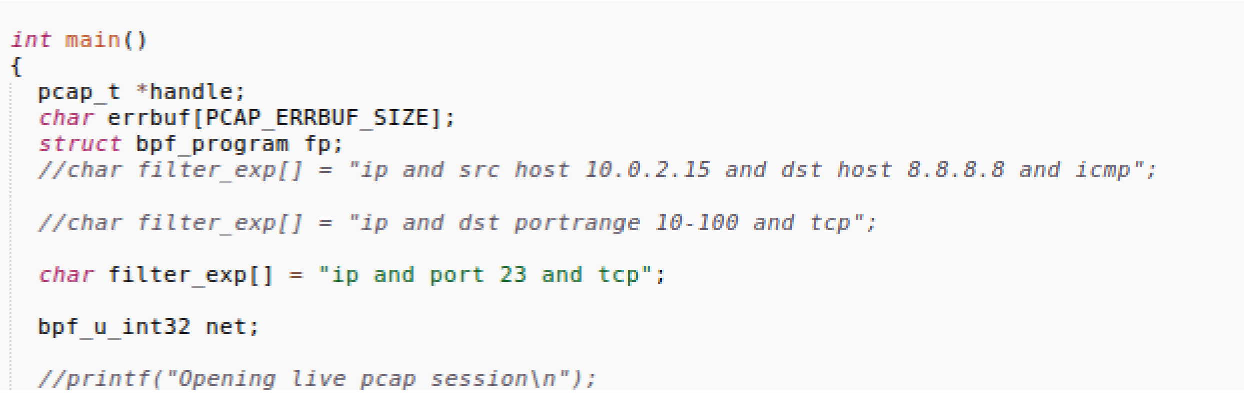
does not receive any packet but when we change the port number to 80, it displays the packet as shown.

Task 2.1C: Sniffing Passwords



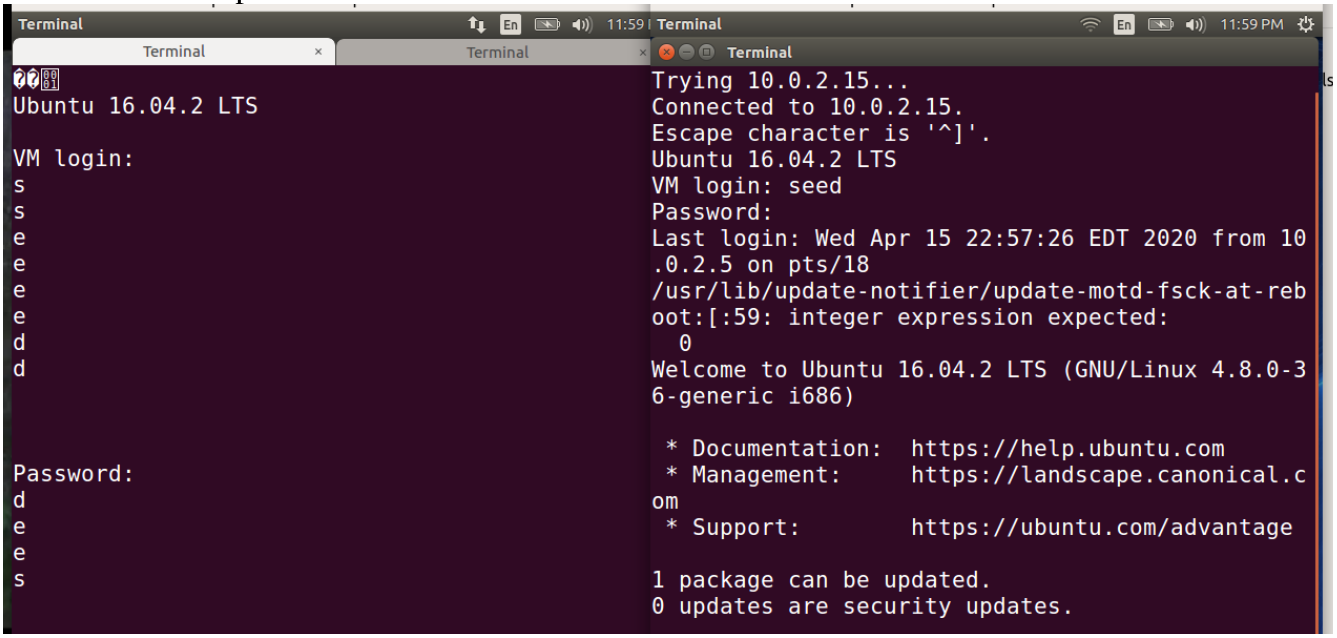
```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{

  struct ethheader *eth = (struct ethheader *)packet;
  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    const u_char *ip_header;
    const u_char *tcp_header;
    const u_char *payload;

    int ethernet_header_length = 14;
    int ip_header_length;
    int tcp_header_length;
    int payload_length;

    ip_header = packet + ethernet_header_length;

    ip_header_length =  ((*ip_header) & 0x0F);

    ip_header_length = ip_header_length * 4;

    tcp_header = packet + ethernet_header_length + ip_header_length;
    tcp_header_length = ((*(tcp_header + 12)) & 0xF0) >> 4;
    tcp_header_length = tcp_header_length * 4;
    int total_headers_size = ethernet_header_length+ip_header_length+tcp_header_length;
    payload = packet + total_headers_size;

    payload_length = header->caplen - (ethernet_header_length + ip_header_length + tcp_header_length);

    if (payload_length > 0) {
        const u_char *temp_pointer = payload;
        int byte_count = 0;
        while (byte_count++ < payload_length) {
            printf("%c", *temp_pointer);
            temp_pointer++;
        }
        printf("\n");
    }


  }

}
```

The main part of the code task2_1b.c is shown above. It listens to the packets that come to port 23 (since telnet uses port 23) as shown below and displays the content of the packet.



```
int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  //char filter_exp[] = "ip and src host 10.0.2.15 and dst host 8.8.8.8 and icmp";

  //char filter_exp[] = "ip and dst portrange 10-100 and tcp";

  char filter_exp[] = "ip and port 23 and tcp";

  bpf_u_int32 net;

  //printf("Opening live pcap session\n");
```

We can see the password when we tried to connect from VM B to VM A over telnet.



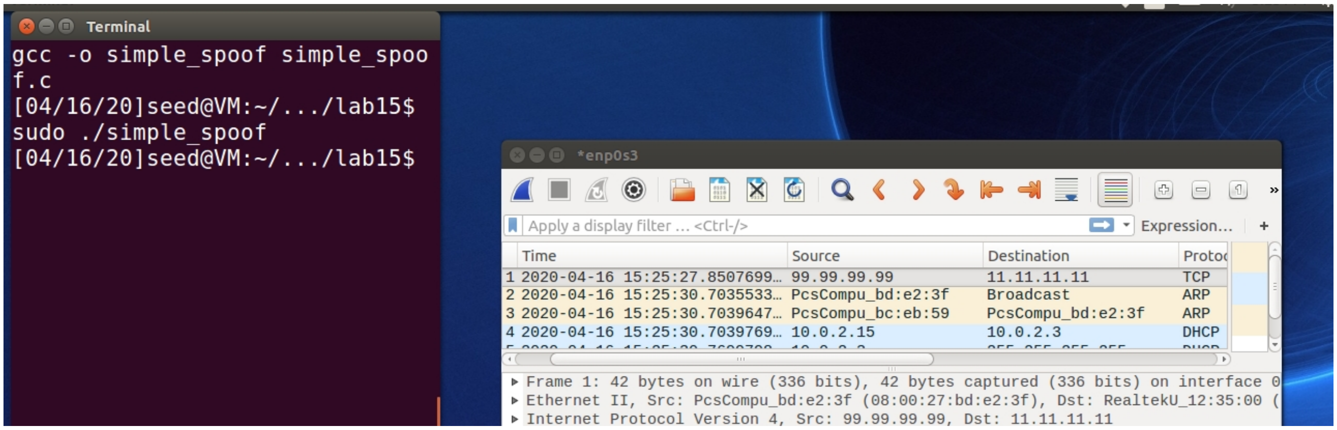
```
��
Ubuntu 16.04.2 LTS

VM login:
s
s
e
e
e
e
d
d



Password:
d
e
e
s
```

```
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: Wed Apr 15 22:57:26 EDT 2020 from 10
.0.2.5 on pts/18
/usr/lib/update-notifier/update-motd-fsck-at-reb
oot:[:59: integer expression expected:
  0
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-3
6-generic i686)

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

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

3.2 Task 2.2: Spoofing

Task 2.2A: Write a spoofing program.

As shown below, the c program is successfully running and spoofed an tcp request packet with source address 99.99.99.99 and destination address 11.11.11.11.

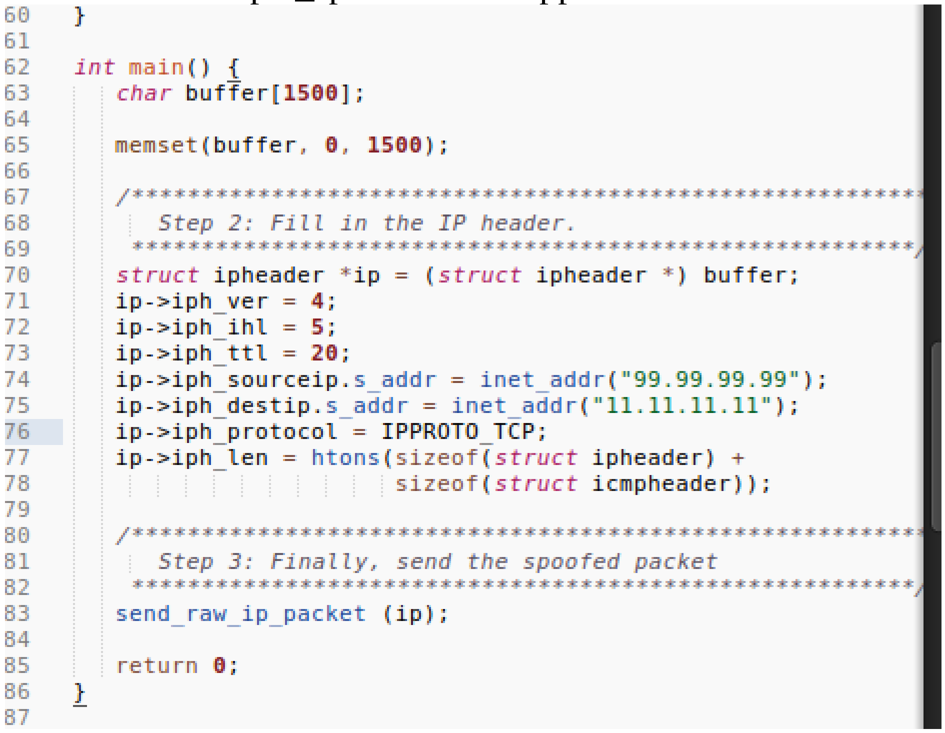


```
gcc -o simple_spoof simple_spoo
f.c
[04/16/20]seed@VM:~/.../lab15$
sudo ./simple_spoof
[04/16/20]seed@VM:~/.../lab15$
```

	Time	Source	Destination	Protoc
1	2020-04-16 15:25:27.8507699…	99.99.99.99	11.11.11.11	TCP
2	2020-04-16 15:25:30.7035533…	PcsCompu_bd:e2:3f	Broadcast	ARP
3	2020-04-16 15:25:30.7039647…	PcsCompu_bc:eb:59	PcsCompu_bd:e2:3f	ARP
4	2020-04-16 15:25:30.7039769…	10.0.2.15	10.0.2.3	DHCP

```
▶ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▶ Ethernet II, Src: PcsCompu_bd:e2:3f (08:00:27:bd:e2:3f), Dst: RealtekU_12:35:00 (
▶ Internet Protocol Version 4, Src: 99.99.99.99, Dst: 11.11.11.11
```

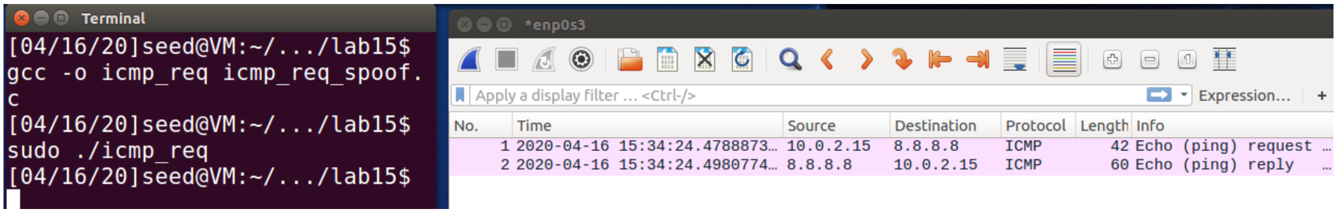
The code can be found in simple_spoof.c and snippet is here:



```
}

int main() {
    char buffer[1500];

    memset(buffer, 0, 1500);

    /**********************************************************
      Step 2: Fill in the IP header.
     **********************************************************/
    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("99.99.99.99");
    ip->iph_destip.s_addr = inet_addr("11.11.11.11");
    ip->iph_protocol = IPPROTO_TCP;
    ip->iph_len = htons(sizeof(struct ipheader) +
                        sizeof(struct icmpheader));

    /**********************************************************
      Step 3: Finally, send the spoofed packet
     **********************************************************/
    send_raw_ip_packet (ip);

    return 0;
}
```

Task 2.2B: Spoof an ICMP Echo Request.

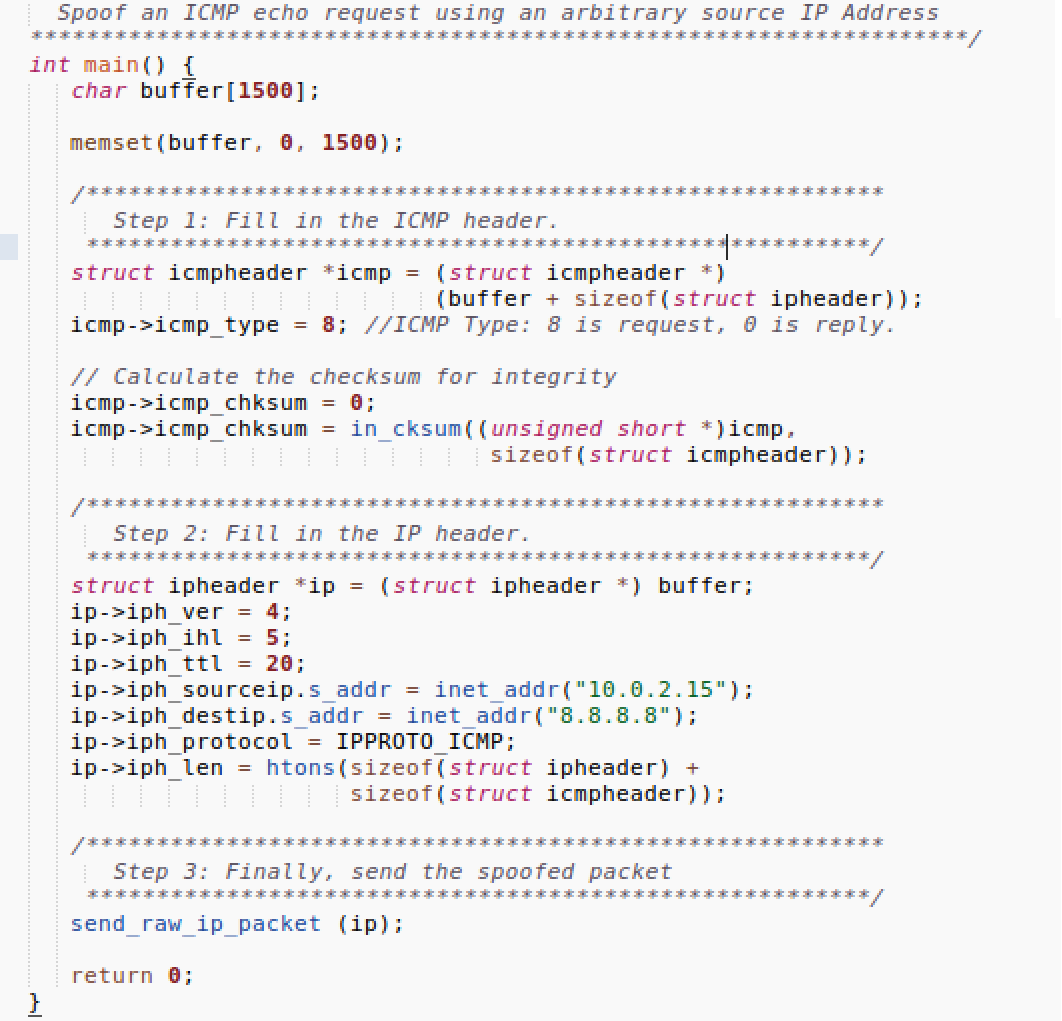
I have written the code icmp_req_spoof.c that spoofs a ICMP echo request. As seen in the image below, the request packet is sent and reply is received as seen in wireshark.



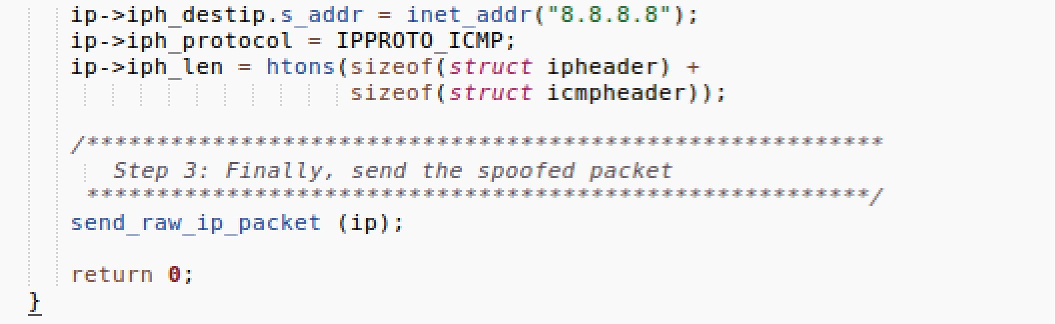
```
[04/16/20]seed@VM:~/.../lab15$
gcc -o icmp_req icmp_req_spoof.
c
[04/16/20]seed@VM:~/.../lab15$
sudo ./icmp_req
[04/16/20]seed@VM:~/.../lab15$
```

```
No.  Time                          Source     Destination  Protocol  Length  Info
  1  2020-04-16 15:34:24.4788873…  10.0.2.15  8.8.8.8      ICMP          42  Echo (ping) request  …
  2  2020-04-16 15:34:24.4980774…  8.8.8.8    10.0.2.15    ICMP          60  Echo (ping) reply    …
```

The code snippet used for this task is:



```
 Spoof an ICMP echo request using an arbitrary source IP Address
*********************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);

   /*********************************************************
      Step 1: Fill in the ICMP header.
    ********************************************************/
   struct icmpheader *icmp = (struct icmpheader *)
                              (buffer + sizeof(struct ipheader));
   icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                 sizeof(struct icmpheader));

   /*********************************************************
      Step 2: Fill in the IP header.
    ********************************************************/
   struct ipheader *ip = (struct ipheader *) buffer;
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("10.0.2.15");
   ip->iph_destip.s_addr = inet_addr("8.8.8.8");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));

   /*********************************************************
      Step 3: Finally, send the spoofed packet
    ********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```



```
   ip->iph_destip.s_addr = inet_addr("8.8.8.8");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));

   /*********************************************************
      Step 3: Finally, send the spoofed packet
    ********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```

Question 4. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
No. If the IP packet length filed is set to an arbitrary value and is not equal to the length of actual packet, an error will occur. Also, the maximum length of an IP Packet is about 65536 bytes.

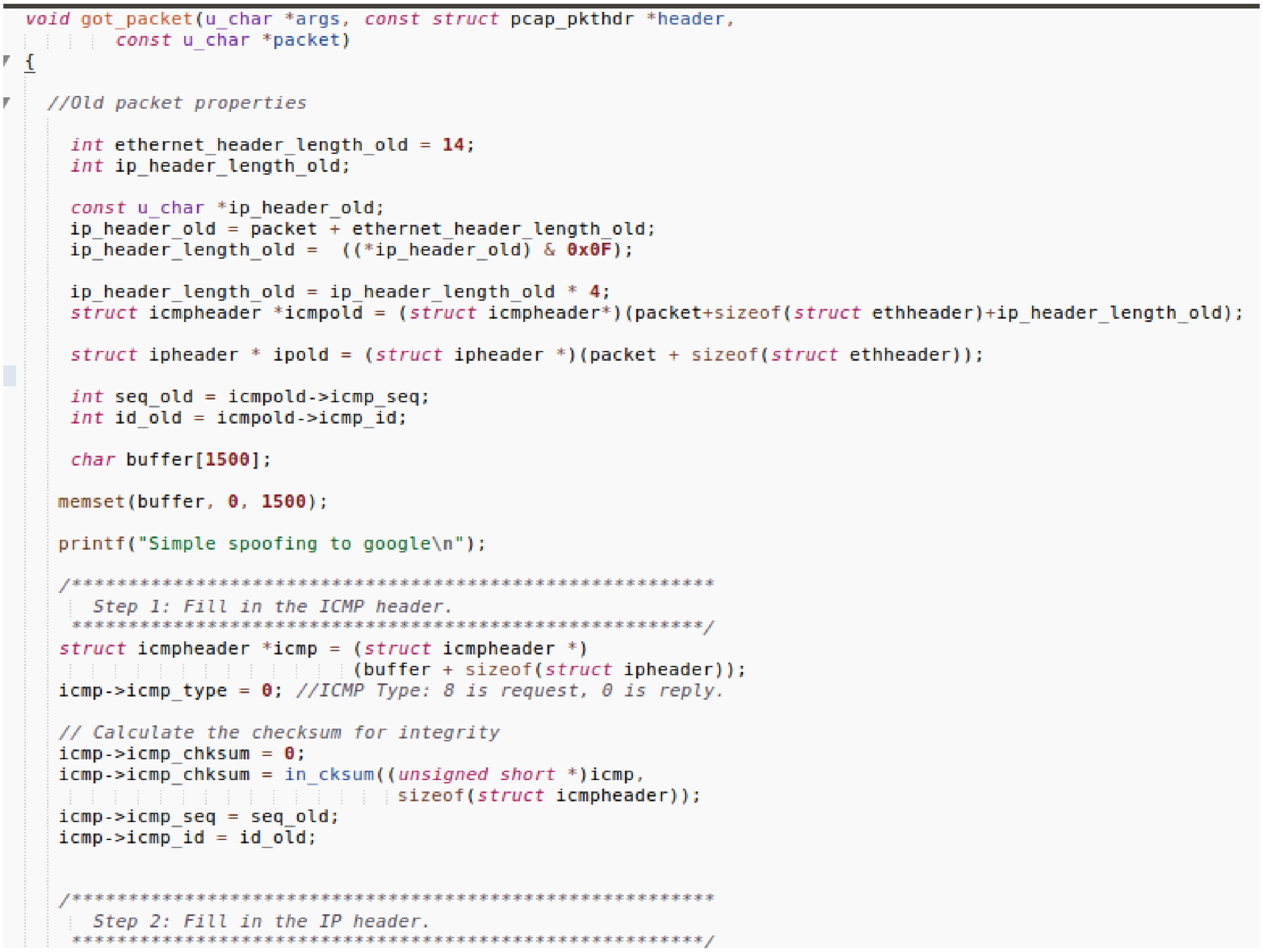
Question 5. Using the raw socket programming, do you have to calculate the checksum for the IP header?
No, since we are using raw packet, the os will send out the packet as is except for the checksum field which will be calculated by the system.

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?
For security reasons, only root processes and processes with the CAP_NET_RAW capabilities can create raw sockets. Hence, we need sudo and without this, the program will fail at socket creation line for example:

```
int sock = socket(AF_INET,SOCK_RAW,IPPROTO_RAW)
```

Task 2.3: Sniff and then Spoof

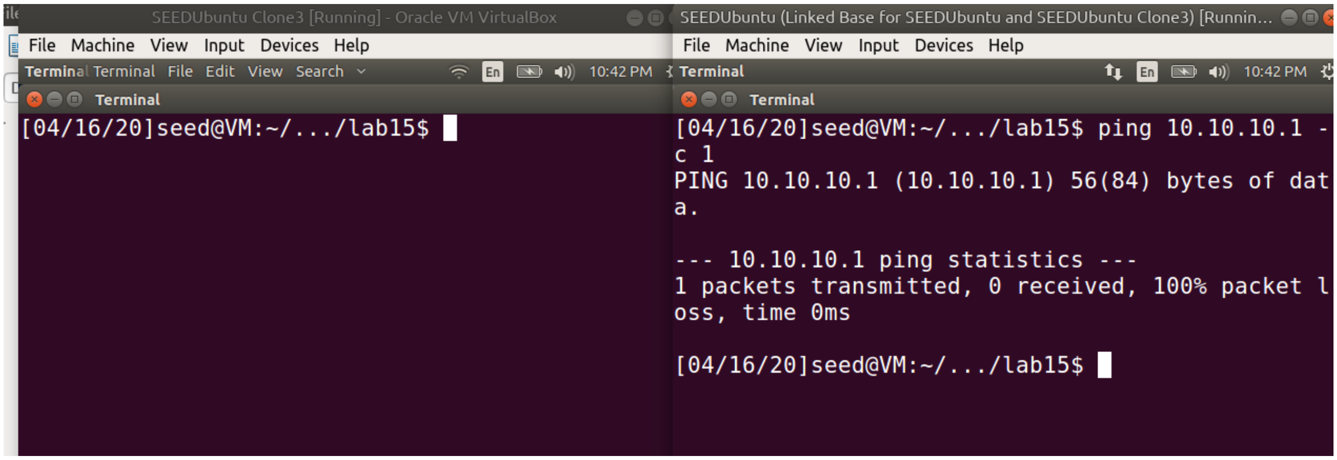
Combining both sniff and spoof, I got to sniff the packet from VM B from VM A and send the response immediately even though the pinged host doesn’t exist.



```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
        const u_char *packet)
{
  //Old packet properties

    int ethernet_header_length_old = 14;
    int ip_header_length_old;

    const u_char *ip_header_old;
    ip_header_old = packet + ethernet_header_length_old;
    ip_header_length_old =  ((*ip_header_old) & 0x0F);

    ip_header_length_old = ip_header_length_old * 4;
    struct icmpheader *icmpold = (struct icmpheader*)(packet+sizeof(struct ethheader)+ip_header_length_old);

    struct ipheader * ipold = (struct ipheader *)(packet + sizeof(struct ethheader));

    int seq_old = icmpold->icmp_seq;
    int id_old = icmpold->icmp_id;

    char buffer[1500];

   memset(buffer, 0, 1500);

   printf("Simple spoofing to google\n");

   /*********************************************************
      Step 1: Fill in the ICMP header.
    ********************************************************/
   struct icmpheader *icmp = (struct icmpheader *)
                             (buffer + sizeof(struct ipheader));
   icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                 sizeof(struct icmpheader));
   icmp->icmp_seq = seq_old;
   icmp->icmp_id = id_old;


   /*********************************************************
      Step 2: Fill in the IP header.
    ********************************************************/
```

Here is the code snippet where I created icmp header and modified sequence and id fields with the parameters of captured packet and send the data. Please see the file **sniffandspoof.c** for more code.

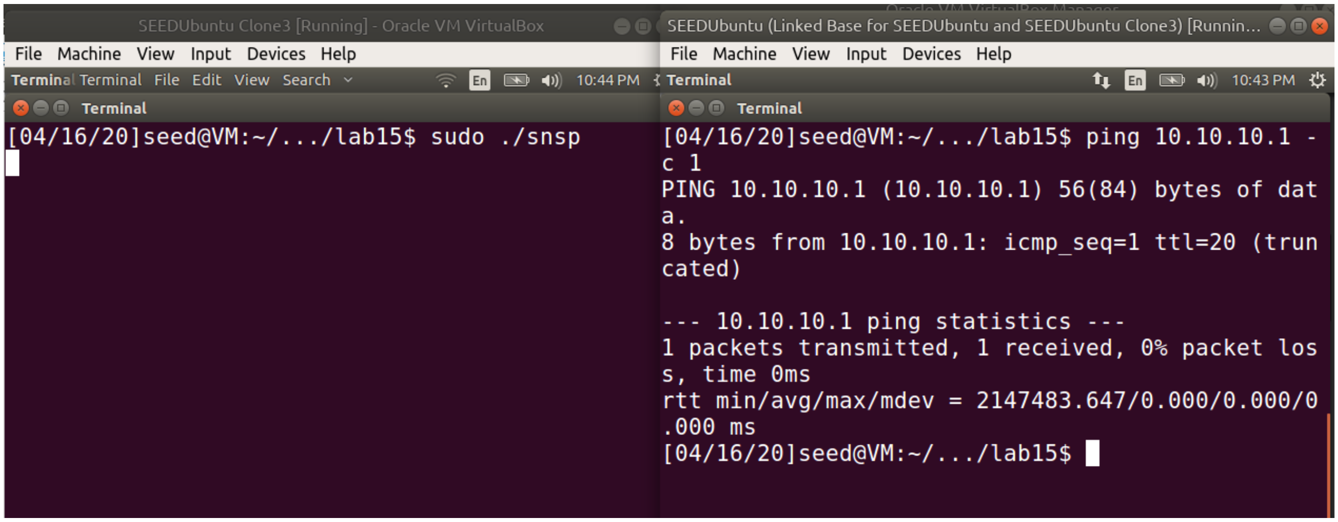
Before running **sniffandspoof.c**, the ping to 10.10.10.1 is failed as seen below.


SEEDUbuntu Clone3 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Terminal File Edit View Search
10:42 PM
Terminal
SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone3) [Runnin…
File Machine View Input Devices Help
Terminal
10:42 PM
Terminal

```
[04/16/20]seed@VM:~/.../lab15$
```

```
[04/16/20]seed@VM:~/.../lab15$ ping 10.10.10.1 -
c 1
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of dat
a.

--- 10.10.10.1 ping statistics ---
1 packets transmitted, 0 received, 100% packet l
oss, time 0ms

[04/16/20]seed@VM:~/.../lab15$
```

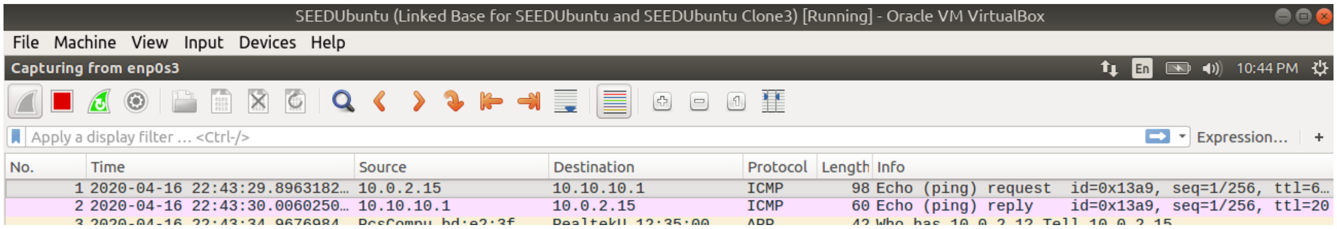
As seen below, sniffer is running in one vm and another vm is pinging non existant address 10.10.10.1. The sniffer catches this and sends the response.


SEEDUbuntu Clone3 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Terminal File Edit View Search
10:44 PM
Terminal
SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone3) [Runnin…
File Machine View Input Devices Help
Terminal
10:43 PM
Terminal

```
[04/16/20]seed@VM:~/.../lab15$ sudo ./snsp
```

```
[04/16/20]seed@VM:~/.../lab15$ ping 10.10.10.1 -
c 1
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of dat
a.
8 bytes from 10.10.10.1: icmp_seq=1 ttl=20 (trun
cated)

--- 10.10.10.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet los
s, time 0ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0
.000 ms
[04/16/20]seed@VM:~/.../lab15$
```

The wireshark also confirms the reception of the echo response packet.


SEEDUbuntu (Linked Base for SEEDUbuntu and SEEDUbuntu Clone3) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Capturing from enp0s3
10:44 PM
Apply a display filter … <Ctrl-/>
Expression…
+

No.	Time	Source	Destination	Protocol	Length	Info
1	2020-04-16 22:43:29.8963182…	10.0.2.15	10.10.10.1	ICMP	98	Echo (ping) request id=0x13a9, seq=1/256, ttl=6…
2	2020-04-16 22:43:30.0060250…	10.10.10.1	10.0.2.15	ICMP	60	Echo (ping) reply id=0x13a9, seq=1/256, ttl=20
3	2020-04-16 22:43:34.9676984…	PcsCompu_bd:e2:3f	RealtekU_12:35:00	ARP	42	Who has 10.0.2.12 Tell 10.0.2.15