INTERNET SECURITY LAB -1

Akarsh Shetty Umesh Mudelkadi
SUID- 317752264

1.a)

Code:

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

        pkt.show()

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

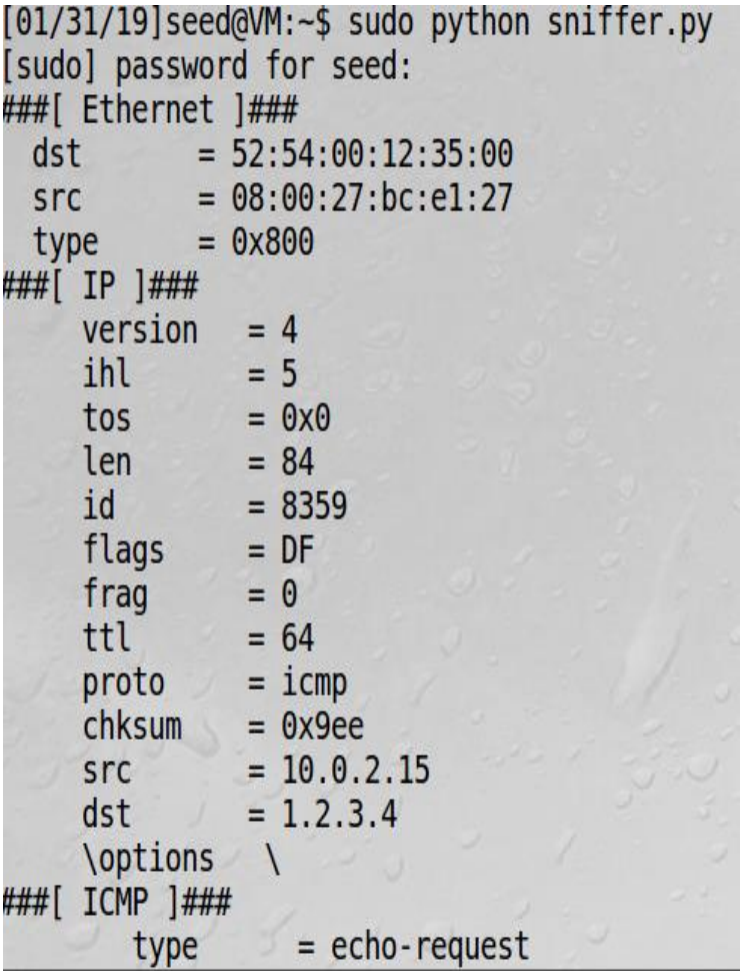
Running using root privilege:

Testing command: Ping 1.2.3.4 (from the same VM where I ran sniffing code).

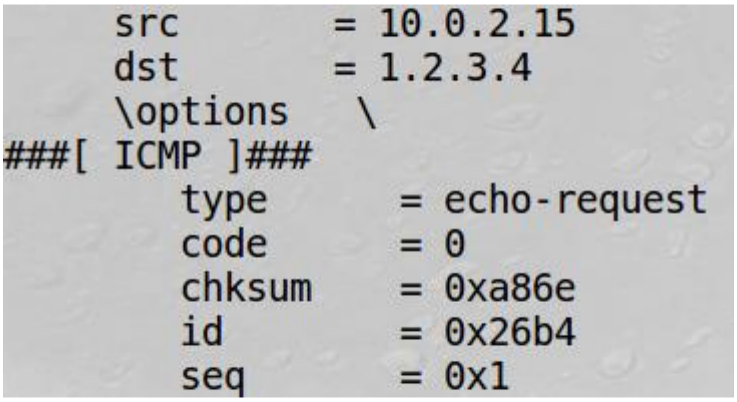
Observation: By running the above sniffing code using the root privilege we get the below results shown in the screenshots.

Explanation: The protocol of the packet sniffed is ICMP, it is of echo-request type. The echo request is made from IP 10.0.2.5(source) to target host IP 1.2.3.4(destination).

Screenshots:

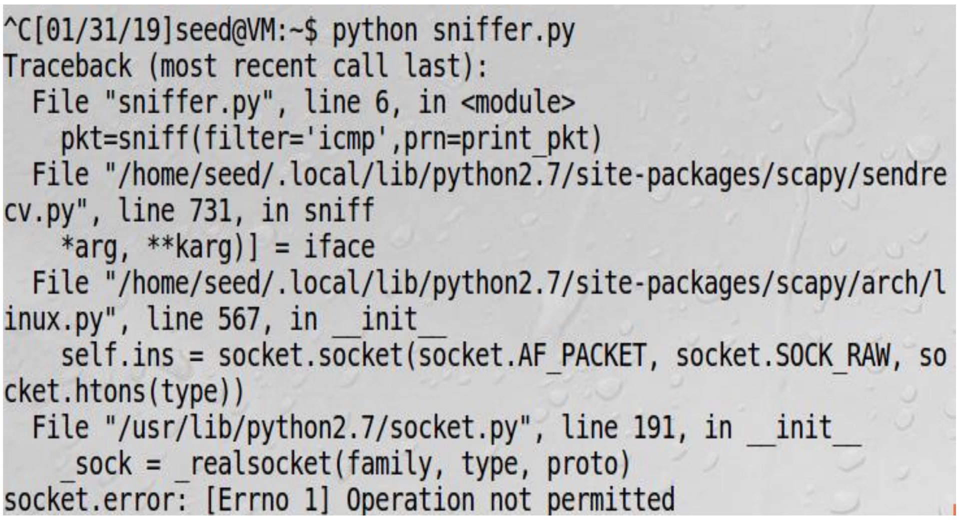


```
[01/31/19]seed@VM:~$ sudo python sniffer.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bc:e1:27
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 8359
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x9ee
     src       = 10.0.2.15
     dst       = 1.2.3.4
     \options   \
###[ ICMP ]###
        type      = echo-request
```



```
     src       = 10.0.2.15
     dst       = 1.2.3.4
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xa86e
        id        = 0x26b4
        seq       = 0x1
```

Running without using the root privilege:



```
^C[01/31/19]seed@VM:~$ python 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-packages/scapy/sendre
cv.py", line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/l
inux.py", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, so
cket.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
```

Observation: I was not able to sniff the packet by running the code without the root privilege. By looking at the error it looks like a socket issue.

Explanation: Scapy uses raw sockets to inject packets in Linux, thus to use those raw sockets we need root privilege.

1.b

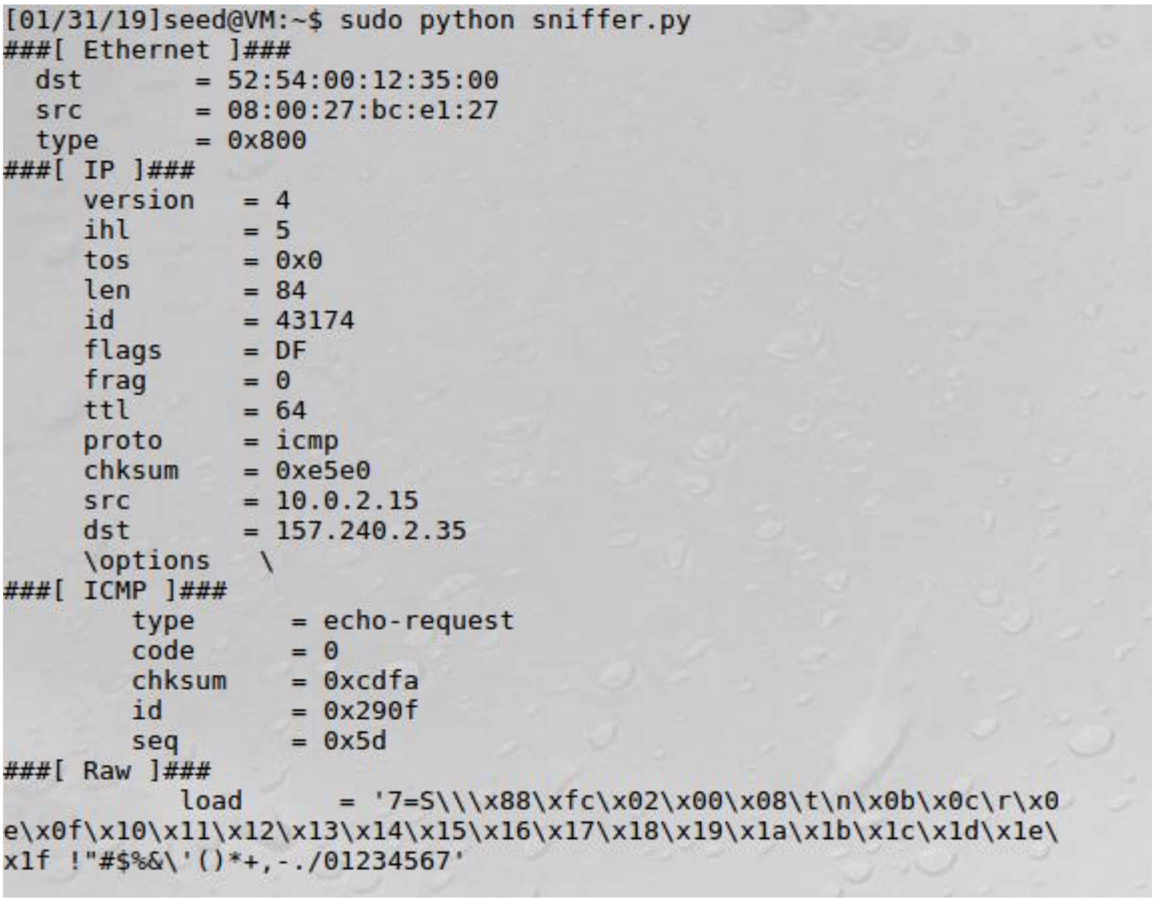
Capturing only ICMP packet:

Code:

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

        pkt.show()
```

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

Screenshot:



```
[01/31/19]seed@VM:~$ sudo python sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bc:e1:27
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 43174
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xe5e0
     src       = 10.0.2.15
     dst       = 157.240.2.35
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xcdfa
        id        = 0x290f
        seq       = 0x5d
###[ Raw ]###
           load      = '7=S\\\x88\xfc\x02\x00\x08\t\n\x0b\x0c\r\x0
e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\
x1f !"#$%&\'()*+,-./01234567'
```

***Command for testing*:** ping 157.240.2.35

Observation:

By using the above code for sniffing, I got the result shown above. Received a packet of protocol ICMP and type echo-request.

Explanation:

The sniffing code sniffs a packet going from source address 10.0.2.15 (my VM) to target host 157.240.2.35. Since it is pinging to target host, the packet type is echo-request.

Capturing TCP packet:

Code:

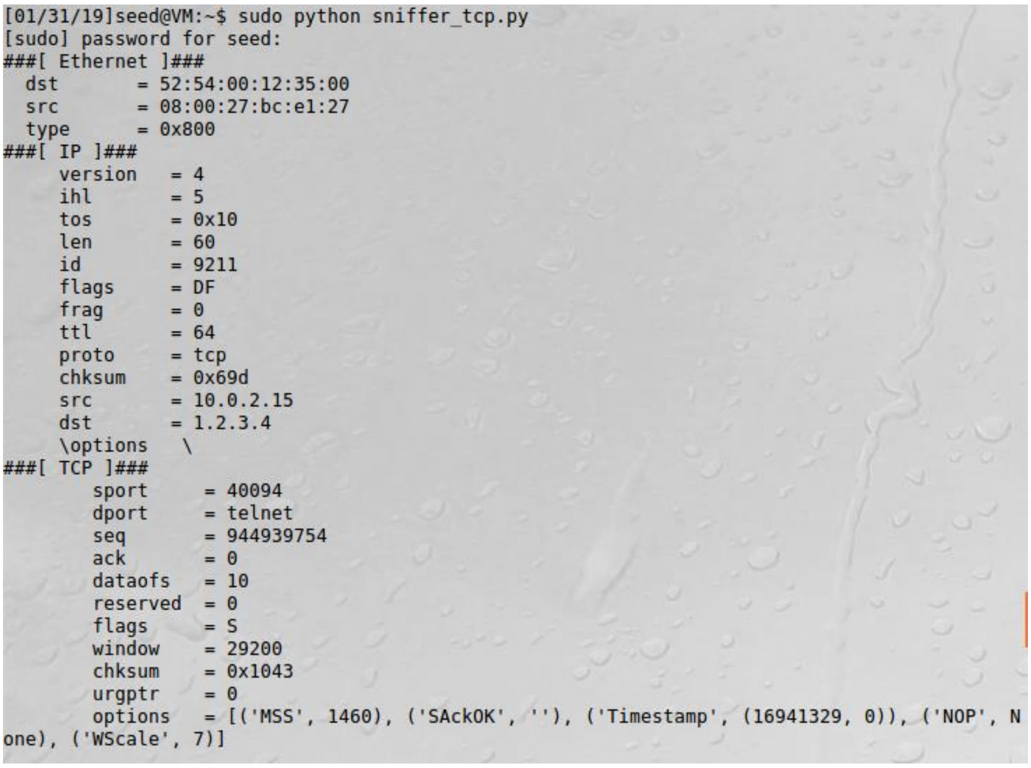
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

```
        pkt.show()


pkt=sniff(filter='src host 10.0.2.15  and tcp dst port 23',prn=print_pkt)
```

Screenshot:



```
[01/31/19]seed@VM:~$ sudo python sniffer_tcp.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:bc:e1:27
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 9211
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x69d
     src       = 10.0.2.15
     dst       = 1.2.3.4
     \options   \
###[ TCP ]###
        sport     = 40094
        dport     = telnet
        seq       = 944939754
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 29200
        chksum    = 0x1043
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', ''), ('Timestamp', (16941329, 0)), ('NOP', N
one), ('WScale', 7)]
```

Command for testing: telnet 1.2.3.4 23 (from the machine with IP mention in the src in the code above)

Observation: Sniffed packet from IP 10.0.2.15 sending request to 1.2.3.4 with destination port 23.

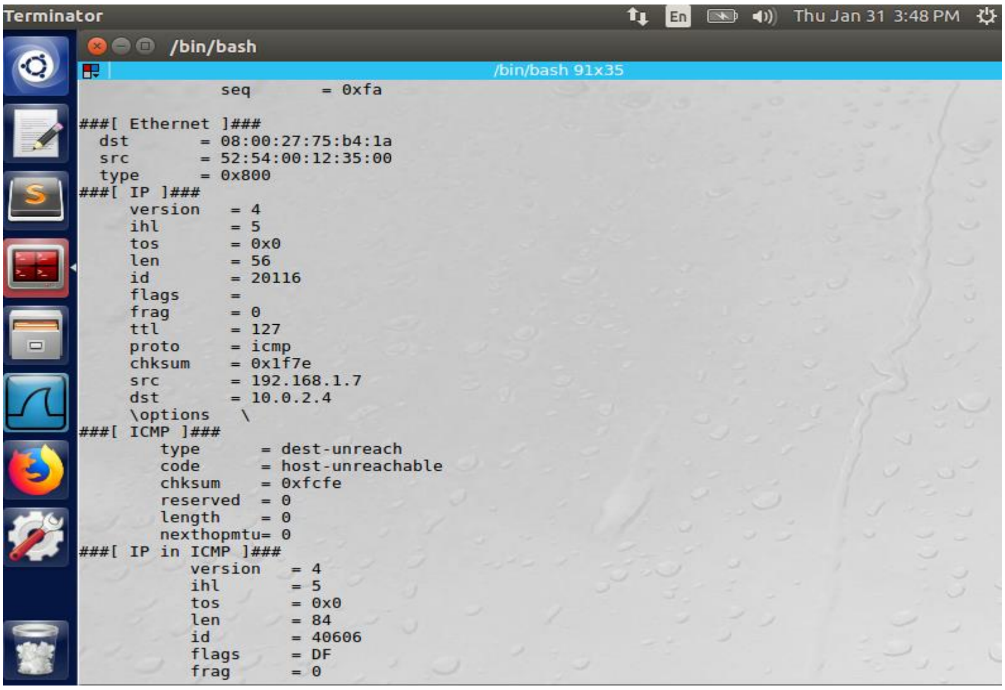
Explanation: I have used single VM (10.0.2.15) for both running the sniffer program and testing. I have used the command mentioned above since telnet uses port 23. From the code mentioned I have changed the filter parameter as 'src host 10.0.2.15 and tcp dst port 23' by referring to the correct BPF syntax. I have sniffed the required packet from the source mentioned which is going to the port 23.

Sniffing through a particular subnet:

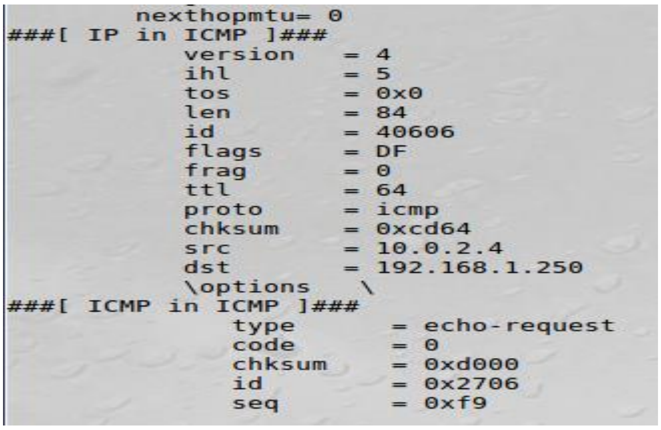
Code:

```
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt=sniff(filter='icmp and net 192.168.1.0/7',prn=print_pkt)
```

Screenshots:



```
             seq       = 0xfa

###[ Ethernet ]###
  dst       = 08:00:27:75:b4:1a
  src       = 52:54:00:12:35:00
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 56
     id        = 20116
     flags     = 
     frag      = 0
     ttl       = 127
     proto     = icmp
     chksum    = 0x1f7e
     src       = 192.168.1.7
     dst       = 10.0.2.4
     \options   \
###[ ICMP ]###
        type      = dest-unreach
        code      = host-unreachable
        chksum    = 0xfcfe
        reserved  = 0
        length    = 0
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 84
           id        = 40606
           flags     = DF
           frag      = 0
```



```
        nexthopmtu= 0
###[ IP in ICMP ]###
           version   = 4
           ihl       = 5
           tos       = 0x0
           len       = 84
           id        = 40606
           flags     = DF
           frag      = 0
           ttl       = 64
           proto     = icmp
           chksum    = 0xcd64
           src       = 10.0.2.4
           dst       = 192.168.1.250
           \options   \
###[ ICMP in ICMP ]###
              type      = echo-request
              code      = 0
              chksum    = 0xd000
              id        = 0x2706
              seq       = 0xf9
```

Command for testing : fping -g 192.168.1.0/24

Observation: Sniffed ICMP packet being sent from source (10.0.2.4) to subnet 192.168.1.7 in 192.168.1.0/24.

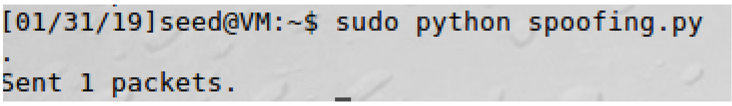
Explanation: For this task I have used 2 VM's. One to test and one to run sniffer code. In the sniffer code I have made changes in the filter as 'icmp and net 192.168.1.0/7' as per BPF syntax. The filter helps us sniff ICMP request packet which is been requested to subnet 192.168.1.7 in 192.168.1.0/24. While testing it captures the routable IP's but sniffs the packet particularly going to single IP 192.168.1.7

1.2) Spoofing

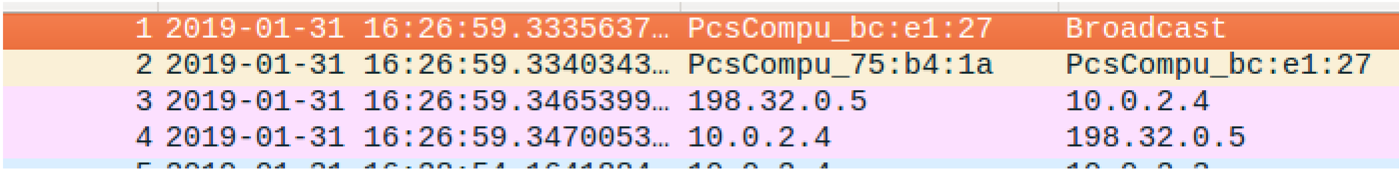
Code:

```
from scapy.all import *

a=IP()
a.src='198.32.0.5'
a.dst='10.0.2.4'
b=ICMP()
p=a/b
send(p)
```

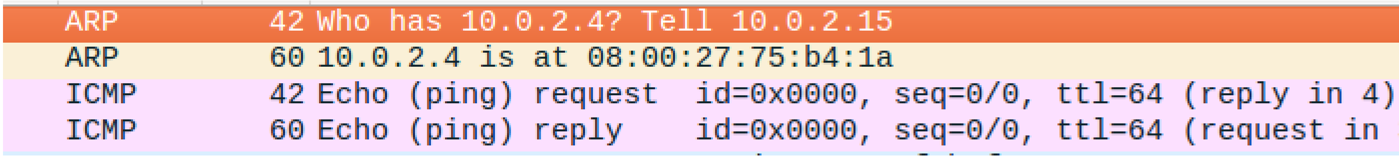
Screenshot:



```
[01/31/19]seed@VM:~$ sudo python spoofing.py
.
Sent 1 packets.
```



```
    1 2019-01-31 16:26:59.3335637… PcsCompu_bc:e1:27     Broadcast
    2 2019-01-31 16:26:59.3340343… PcsCompu_75:b4:1a     PcsCompu_bc:e1:27
    3 2019-01-31 16:26:59.3465399… 198.32.0.5            10.0.2.4
    4 2019-01-31 16:26:59.3470053… 10.0.2.4              198.32.0.5
```



```
ARP        42 Who has 10.0.2.4? Tell 10.0.2.15
ARP        60 10.0.2.4 is at 08:00:27:75:b4:1a
ICMP       42 Echo (ping) request  id=0x0000, seq=0/0, ttl=64 (reply in 4)
ICMP       60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in
```

Observation: Spoofed an ICMP packet to destination 10.0.2.4 from arbitrary source 198.32.0.5. But actual source is 10.0.2.5

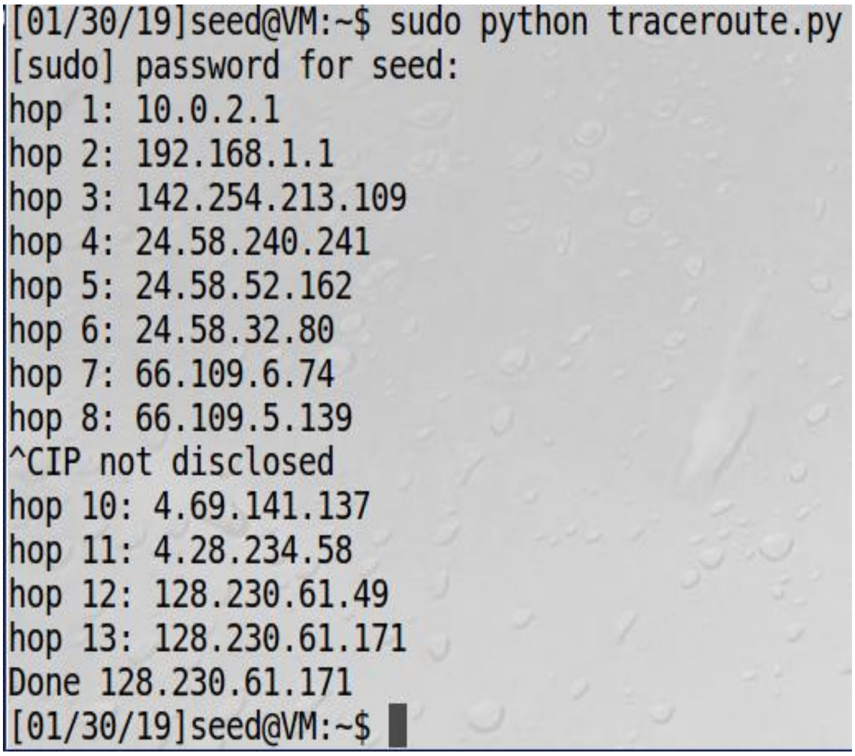
Explanation: From the code given in the question, I included a new line(4th) 'a.src='198.32.0.5' to make it look like a real packet coming from it to the mentioned destination. The actual source is the IP of the second VM that is 10.0.2.5. From the screenshots we can look at the results of the Wireshark, which says the packet has been received by VM(10.0.2.4) from the IP '198.32.0.5'. We can also see an ARP request has been made asking who has '10.0.2.4' and tell it to 10.0.2.15 but the packet was spoofed saying it came from the arbitrary address mentioned above.

1.3) Traceroute

Code:

```
from scapy.all import *

a=IP()
a.dst = '128.230.171.184'
b=ICMP()
for i in range(1,28):
        a.ttl=i
        pkt=a/b
        reply=sr1(pkt,verbose=0)
        if reply is None:
                print "IP not disclosed"
                continue
        elif reply.type==3:
                print "Done",reply.src
                break
        else:
                print "hop %d:" %i,reply.src
```

Screenshot:



```
[01/30/19]seed@VM:~$ sudo python traceroute.py
[sudo] password for seed:
hop 1: 10.0.2.1
hop 2: 192.168.1.1
hop 3: 142.254.213.109
hop 4: 24.58.240.241
hop 5: 24.58.52.162
hop 6: 24.58.32.80
hop 7: 66.109.6.74
hop 8: 66.109.5.139
^CIP not disclosed
hop 10: 4.69.141.137
hop 11: 4.28.234.58
hop 12: 128.230.61.49
hop 13: 128.230.61.171
Done 128.230.61.171
[01/30/19]seed@VM:~$
```

Observation: Retrieved all the router IP information/number of hops along the path from source my VM '10.0.2.15' to destination www.syr.edu IP 128.230.171.184. Also, couldn't get IP information of hop number 9.

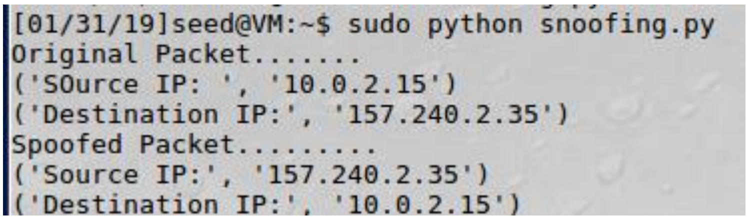
Explanation: I have changed the code given in the question to the one mentioned above. I have created an ICMP packet to be sent to destination '128.230.171.184' hop by hop. I have used for loop with max iterations of 27 hops. I have used sr1() method of scapy which sends packet at layer 3 and waits for a response. In the first if condition it states that if we don't get any reply from a particular IP (in this case hop number 9) we will print a message 'IP not disclosed' and continue with the iteration. In the second if condition we check(by stating if type==3) if our next hop is our destination IP and break from the loop. In the last else statement we just write our current hop's IP information.

1.4) Sniffing and then spoofing

Code:

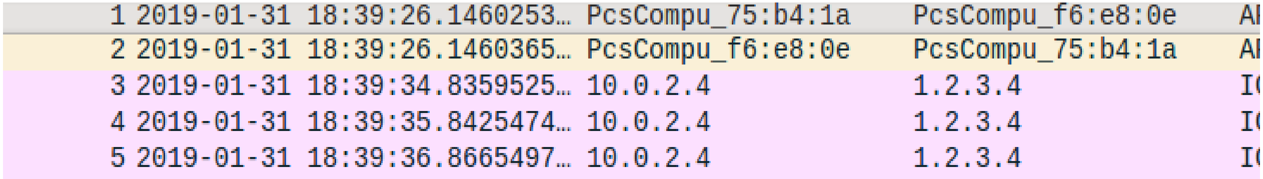
```
#! /usr/bin/python3
from scapy.all import *
def spoof_pkt(pkt):
	if ICMP in pkt and pkt[ICMP].type == 8:
		print ("Original Packet.......")
		print("SOurce IP: ",pkt[IP].src)
		print("Destination IP:", pkt[IP].dst)

		ip=IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
		icmp=ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
		data=pkt[Raw].load
		newpkt=ip/icmp/data

		print("Spoofed Packet.........")
		print("Source IP:",newpkt[IP].src)
		print("Destination IP:",newpkt[IP].dst)

		send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 10.0.2.15', prn=spoof_pkt)
```

Screenshots :

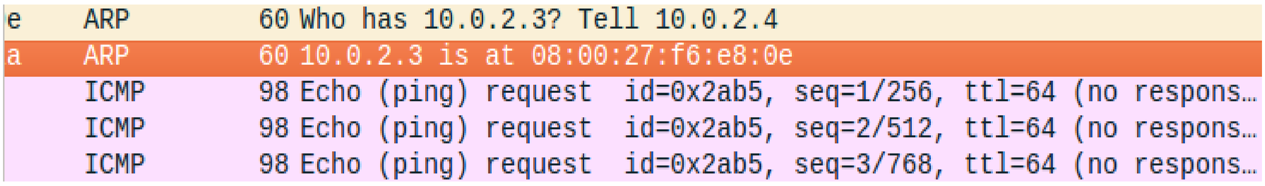


```
[01/31/19]seed@VM:~$ sudo python snoofing.py
Original Packet.......
('SOurce IP: ', '10.0.2.15')
('Destination IP:', '157.240.2.35')
Spoofed Packet.........
('Source IP:', '157.240.2.35')
('Destination IP:', '10.0.2.15')
```

without snoofing:

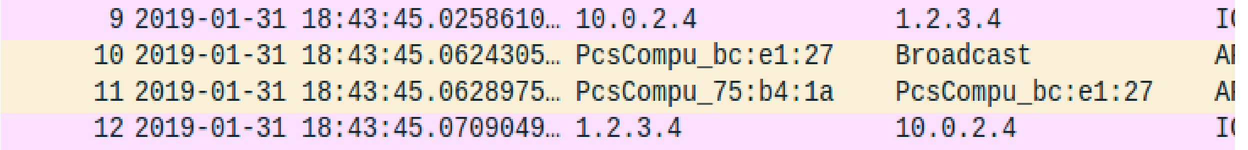


```
      1 2019-01-31 18:39:26.1460253… PcsCompu_75:b4:1a     PcsCompu_f6:e8:0e     AI
      2 2019-01-31 18:39:26.1460365… PcsCompu_f6:e8:0e     PcsCompu_75:b4:1a     AI
      3 2019-01-31 18:39:34.8359525… 10.0.2.4              1.2.3.4               I(
      4 2019-01-31 18:39:35.8425474… 10.0.2.4              1.2.3.4               I(
      5 2019-01-31 18:39:36.8665497… 10.0.2.4              1.2.3.4               I(
```

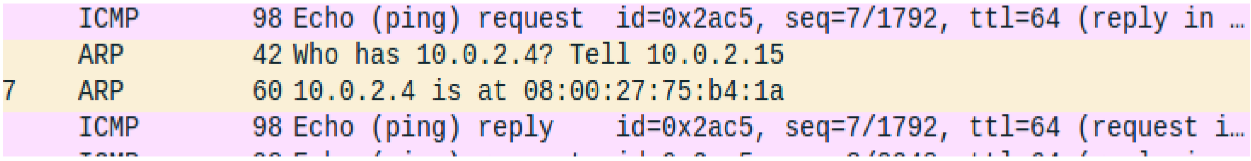


```
e   ARP     60 Who has 10.0.2.3? Tell 10.0.2.4
a   ARP     60 10.0.2.3 is at 08:00:27:f6:e8:0e
    ICMP    98 Echo (ping) request  id=0x2ab5, seq=1/256, ttl=64 (no respons…
    ICMP    98 Echo (ping) request  id=0x2ab5, seq=2/512, ttl=64 (no respons…
    ICMP    98 Echo (ping) request  id=0x2ab5, seq=3/768, ttl=64 (no respons…
```

after snoofing:



```
      9 2019-01-31 18:43:45.0258610… 10.0.2.4              1.2.3.4               I(
     10 2019-01-31 18:43:45.0624305… PcsCompu_bc:e1:27     Broadcast             AI
     11 2019-01-31 18:43:45.0628975… PcsCompu_75:b4:1a     PcsCompu_bc:e1:27     AI
     12 2019-01-31 18:43:45.0709049… 1.2.3.4               10.0.2.4              I(
```



```
    ICMP    98 Echo (ping) request  id=0x2ac5, seq=7/1792, ttl=64 (reply in …
    ARP     42 Who has 10.0.2.4? Tell 10.0.2.15
7   ARP     60 10.0.2.4 is at 08:00:27:75:b4:1a
    ICMP    98 Echo (ping) reply    id=0x2ac5, seq=7/1792, ttl=64 (request i…
```

Observation:
Without Snoofing – When ping request is sent from my VM (10.0.2.4) to a non valid IP (1.2.3.4), there is no request sent back by 1.2.3.4
With Snoofing- When ping request is sent from my VM(10.0.2.4) to a non valid IP(1.2.3.4), snoofing code sniffs the request from the network going from 10.0.2.4 and snoofs a packet acting as a reply from 1.2.3.4 to the source.

Explanation:
Since 1.2.3.4 is not a valid IP we don't receive any reply from it to the source which pinging the request.
When we use snoofing code, the VM(10.0.2.15) lets call its IP1, where the code is running sniffs the request coming from 10.0.2.4(Lets call this IP2) which is request made to target host(1.2.3.4). You can see in the Wireshark result in the 2nd picture(under after snoofing), that IP1 is sending ARP request to IP2. Later IP1 spoofs a packet to IP2 by swapping the src and dest IPs and thus indicating IP2 that the reply came from target host(1.2.3.4).

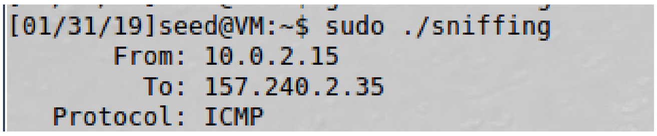
Task 2.1A:

Code:

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

```
#include <net/ethernet.h>



/* Ethernet header */
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 */
};

/* 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
};


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));

    /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
  }
}
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 eth3
  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;
}
```

Screenshot :



```
[01/31/19]seed@VM:~$ sudo ./sniffing
       From: 10.0.2.15
         To: 157.240.2.35
   Protocol: ICMP
```

I tested the above code by running ‘ping facebook.com’ from VM(10.0.215). Got the above result as output has details of source IP, destination IP and protocol.

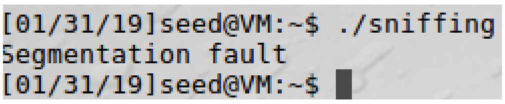
Question 1 ans) The sequence of library calls essential for running sniffer program are:

- **pcap_open_live**(char *dev, int snaplen,int promisc,,char *ebuf)
  This method creates a sniffing session. The arguments used here are **dev** and the device name present in this string is opened by the method; the value present in **snaplen** is the allowed number of bytes be captured by pcap; **promisc** argument holds value 0 or 1, 0

for promiscuous mode off and 1 for promiscuous mode on; argument ebuf holds the error when sniff doesn't work.

- **pcap_compile**(handle, &fp, filter_exp, 0, net)
  This method is used to compile the filter which can be used in the program. The arguments used here are **handle** which holds the session handler created by pcap_open_live(); &fp is a reference that stores the compile version of our filter; filter_exp is a string which holds our actual filter expression; next is an integer argument which holds value 0 or 1, 0 for not optimizing and 1 for optimizing the filter expression; net holds the network mask of the network the filter is applied to.
- **pcap_setfilter**(handle, &fp)
  This method is self-explanatory that it sets the filter. First argument is the session handler and the second one is the reference to the compiled version of the filter.
- **pcap_loop**(handle, -1, got_packet, NULL)
  This method is used as an callback function as in whenever pcap sniffs a packet the program is run again.
- **pcap_close**(handle)
  This method is used to terminate the handler.

Question 2 ans) When we run a sniffer program without root privilege we get bellow error "Segmentation fault". This is because we need root access in Linux to access the network interfaces. Sniffer programs require raw sockets to detect sending packets between applications in network software. To discover NIC we need to be able to create raw sockets.



```
[01/31/19]seed@VM:~$ ./sniffing
Segmentation fault
[01/31/19]seed@VM:~$ 
```

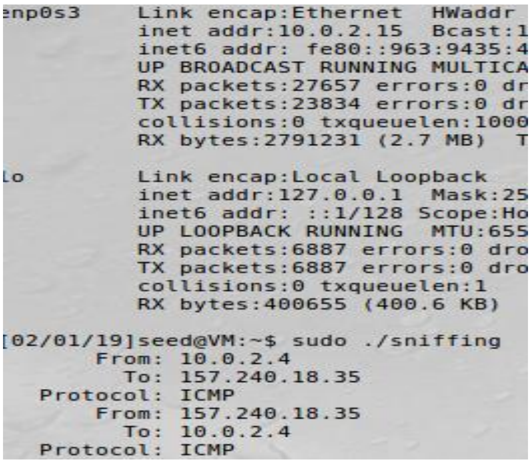
Question 3 ans) Promiscuous mode allows the sniffer code to sniff all the packets coming from systems connected in the same network not just which was intended to receive. If we turn the promiscuous mode off then we can sniff only the packet which is intended to receive.

Code changes :

/*promisc mode on*/
handle = pcap_open_live("enp0s3", BUFSIZ, **1**, 1000, errbuf);

/*promisc mode off*/
handle = pcap_open_live("enp0s3", BUFSIZ, **0**, 1000, errbuf);

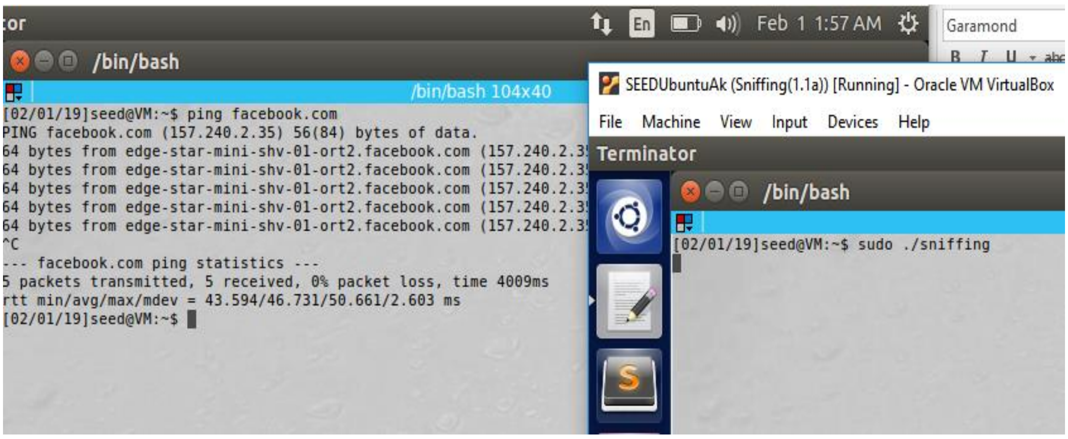
Promiscuous mode ON screenshot:



```
enp0s3    Link encap:Ethernet  HWaddr
          inet addr:10.0.2.15  Bcast:1
          inet6 addr: fe80::963:9435:4
          UP BROADCAST RUNNING MULTICA
          RX packets:27657 errors:0 dr
          TX packets:23834 errors:0 dr
          collisions:0 txqueuelen:1000
          RX bytes:2791231 (2.7 MB)  T

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:25
          inet6 addr: ::1/128 Scope:Ho
          UP LOOPBACK RUNNING  MTU:655
          RX packets:6887 errors:0 dro
          TX packets:6887 errors:0 dro
          collisions:0 txqueuelen:1
          RX bytes:400655 (400.6 KB)

[02/01/19]seed@VM:~$ sudo ./sniffing
       From: 10.0.2.4
         To: 157.240.18.35
   Protocol: ICMP
       From: 157.240.18.35
         To: 10.0.2.4
   Protocol: ICMP
```

Explanation: When sniffer program is made to run with the promiscuous mode ON in VM(10.0.2.15) it sniffed the request packet from src 10.0.2.4 to dest “facebook.com” of ICMP protocol.

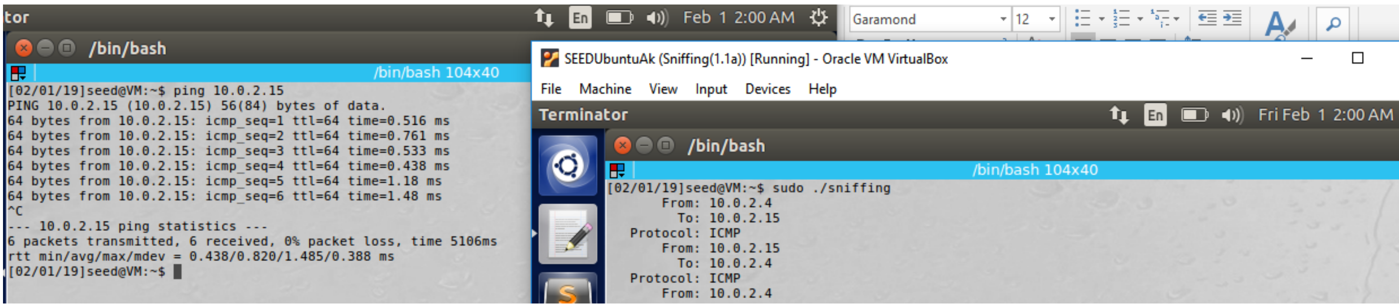
Promiscuous mode OFF screenshots:



```
[02/01/19]seed@VM:~$ ping facebook.com
PING facebook.com (157.240.2.35) 56(84) bytes of data.
64 bytes from edge-star-mini-shv-01-ort2.facebook.com (157.240.2.3
64 bytes from edge-star-mini-shv-01-ort2.facebook.com (157.240.2.3
64 bytes from edge-star-mini-shv-01-ort2.facebook.com (157.240.2.3
64 bytes from edge-star-mini-shv-01-ort2.facebook.com (157.240.2.3
64 bytes from edge-star-mini-shv-01-ort2.facebook.com (157.240.2.3
^C
--- facebook.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4009ms
rtt min/avg/max/mdev = 43.594/46.731/50.661/2.603 ms
[02/01/19]seed@VM:~$
```

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

Explanation: When promiscuous mode is OFF I was not able to sniff packet being sent as a request from 10.0.2.4 to facebook.com.



```
[02/01/19]seed@VM:~$ ping 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.516 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.761 ms
64 bytes from 10.0.2.15: icmp_seq=3 ttl=64 time=0.533 ms
64 bytes from 10.0.2.15: icmp_seq=4 ttl=64 time=0.438 ms
64 bytes from 10.0.2.15: icmp_seq=5 ttl=64 time=1.18 ms
64 bytes from 10.0.2.15: icmp_seq=6 ttl=64 time=1.48 ms
^C
--- 10.0.2.15 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5106ms
rtt min/avg/max/mdev = 0.438/0.820/1.485/0.388 ms
[02/01/19]seed@VM:~$
```

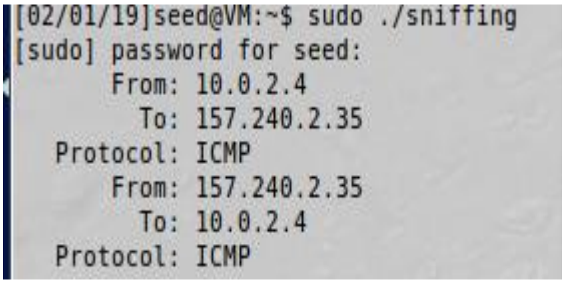
```
[02/01/19]seed@VM:~$ sudo ./sniffing
       From: 10.0.2.4
         To: 10.0.2.15
   Protocol: ICMP
       From: 10.0.2.15
         To: 10.0.2.4
   Protocol: ICMP
       From: 10.0.2.4
```

Explanation: When promiscuous mode is OFF we can sniff packets meant to be received from the same host where the sniffer program is run. Here one VM 10.0.2.4 sends ping 10.0.2.15. The sniffer program is able to sniff the packet since the program is run on target host of the packet being sent.

2.1B)

- **ICMP packets between specific hosts:**
  The two specific hosts I have taken is my VM (10.0.2.4) and facebook.com (157.240.2.35).
  I am testing using VM(10.0.2.15) and command ping facebook.com..

  Screenshot of results:

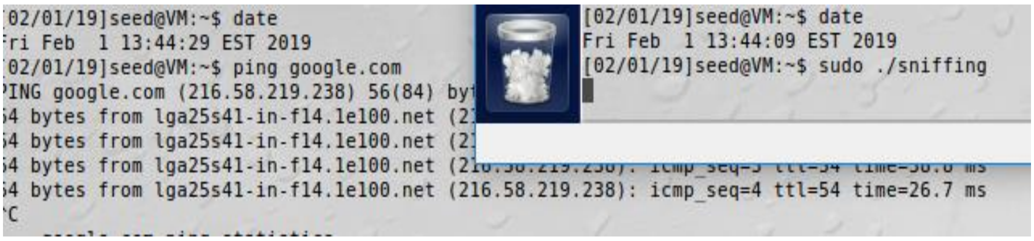


```
[02/01/19]seed@VM:~$ sudo ./sniffing
[sudo] password for seed:
       From: 10.0.2.4
         To: 157.240.2.35
   Protocol: ICMP
       From: 157.240.2.35
         To: 10.0.2.4
   Protocol: ICMP
```

  Observation: Sniffed packets going between hosts 10.0.2.4 and facebook.com by VM(10.0.2.15).

  Explanation: Used the sniffer program in c written above and changed the filter expression to 'char filter_exp[]="icmp and (src host 10.0.2.4 and dst host 157.240.2.35) or (src host 157.240.2.35 and dst host 10.0.2.4)"'. The filter says that the packet sniffed must be of protocol ICMP and the packets must be going between specific hosts 10.0.2.4 and 157.240.2.35.

  The below screenshot is when I tried pinging google.com. Couldn't sniff any packets since google.com IP is not mentioned in the filter.

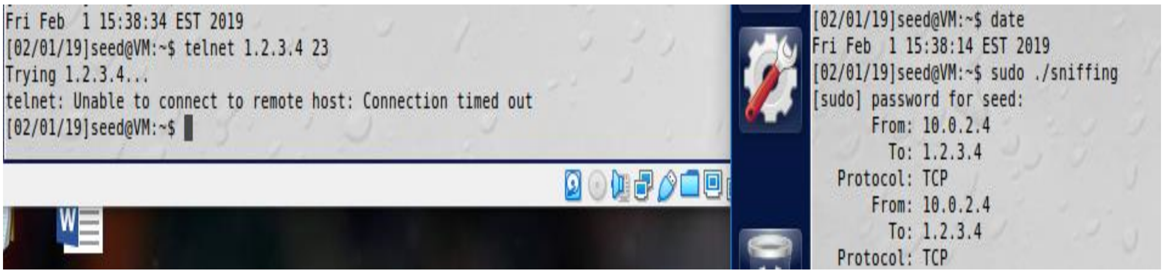


```
02/01/19]seed@VM:~$ date
ri Feb  1 13:44:29 EST 2019
02/01/19]seed@VM:~$ ping google.com
ING google.com (216.58.219.238) 56(84) by
4 bytes from lga25s41-in-f14.1e100.net (2
4 bytes from lga25s41-in-f14.1e100.net (2
4 bytes from lga25s41-in-f14.1e100.net (216.58.219.238): icmp_seq=3 ttl=54 time=30.0 ms
4 bytes from lga25s41-in-f14.1e100.net (216.58.219.238): icmp_seq=4 ttl=54 time=26.7 ms
C
```

```
[02/01/19]seed@VM:~$ date
Fri Feb  1 13:44:09 EST 2019
[02/01/19]seed@VM:~$ sudo ./sniffing
```

- **Capturing TCP packets with a destination port number in the range from 10 to 100**

Screenshots:

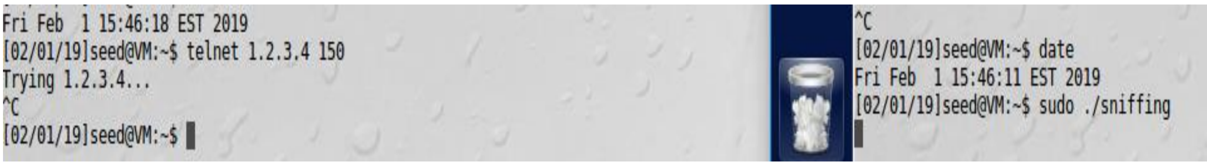


```
Fri Feb  1 15:38:34 EST 2019
[02/01/19]seed@VM:~$ telnet 1.2.3.4 23
Trying 1.2.3.4...
telnet: Unable to connect to remote host: Connection timed out
[02/01/19]seed@VM:~$ 
```

```
[02/01/19]seed@VM:~$ date
Fri Feb  1 15:38:14 EST 2019
[02/01/19]seed@VM:~$ sudo ./sniffing
[sudo] password for seed:
       From: 10.0.2.4
         To: 1.2.3.4
   Protocol: TCP
       From: 10.0.2.4
         To: 1.2.3.4
   Protocol: TCP
```

Observation: Sniffed TCP packet which is been requested from VM(10.0.2.4) to target host 1.2.3.4 at destination port 23.

Explanation: I made changes in the filter expression to 'filter_exp[]="tcp and dst portrange 10-100"' . Thus, the sniffer program sniffs a TCP packet which is going to the destination port between 10-100. In my case the destination port is 23.



```
Fri Feb  1 15:46:18 EST 2019
[02/01/19]seed@VM:~$ telnet 1.2.3.4 150
Trying 1.2.3.4...
^C
[02/01/19]seed@VM:~$ 
```

```
^C
[02/01/19]seed@VM:~$ date
Fri Feb  1 15:46:11 EST 2019
[02/01/19]seed@VM:~$ sudo ./sniffing
```

Observation: Couldn't sniff TCP packet going from 10.0.2.4 to 1.2.3.4 as the destination port is 150 which is not in range between 10-100.

Explanation: Couldn't sniff because the filter conditions doesn't match with the destination port 150.

Task 2.1C: Sniffing Passwords

For this task we want payload to be printed, So I made following changes to sniffer code in C.

- Added tcp header structure
- Added code to extract payload in got_packet() function

/* TCP header */

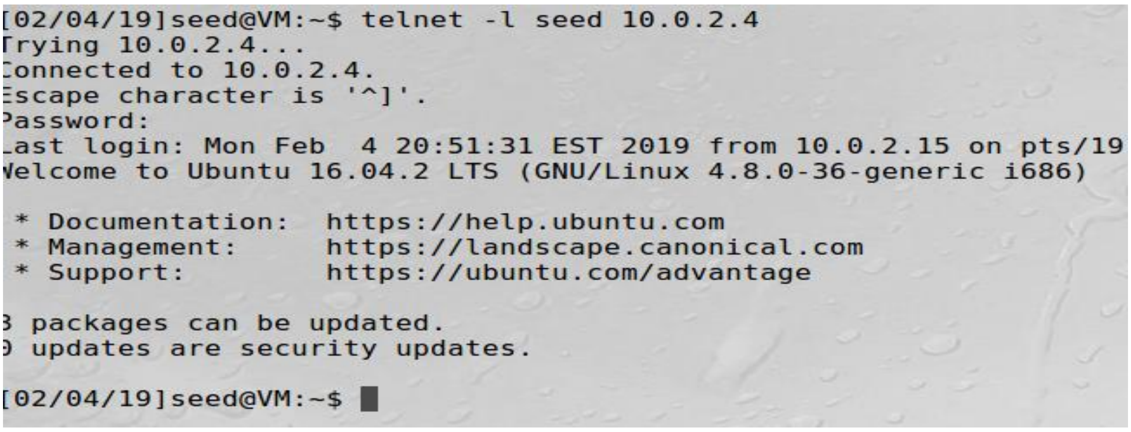
```
struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
            #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
```

```
    #define TH_RST  0x04
    #define TH_PUSH 0x08
    #define TH_ACK  0x10
    #define TH_URG  0x20
    #define TH_ECE  0x40
    #define TH_CWR  0x80
    #define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
    u_short th_win;                /* window */
    u_short th_sum;                /* checksum */
    u_short th_urp;                /* urgent pointer */
};
```

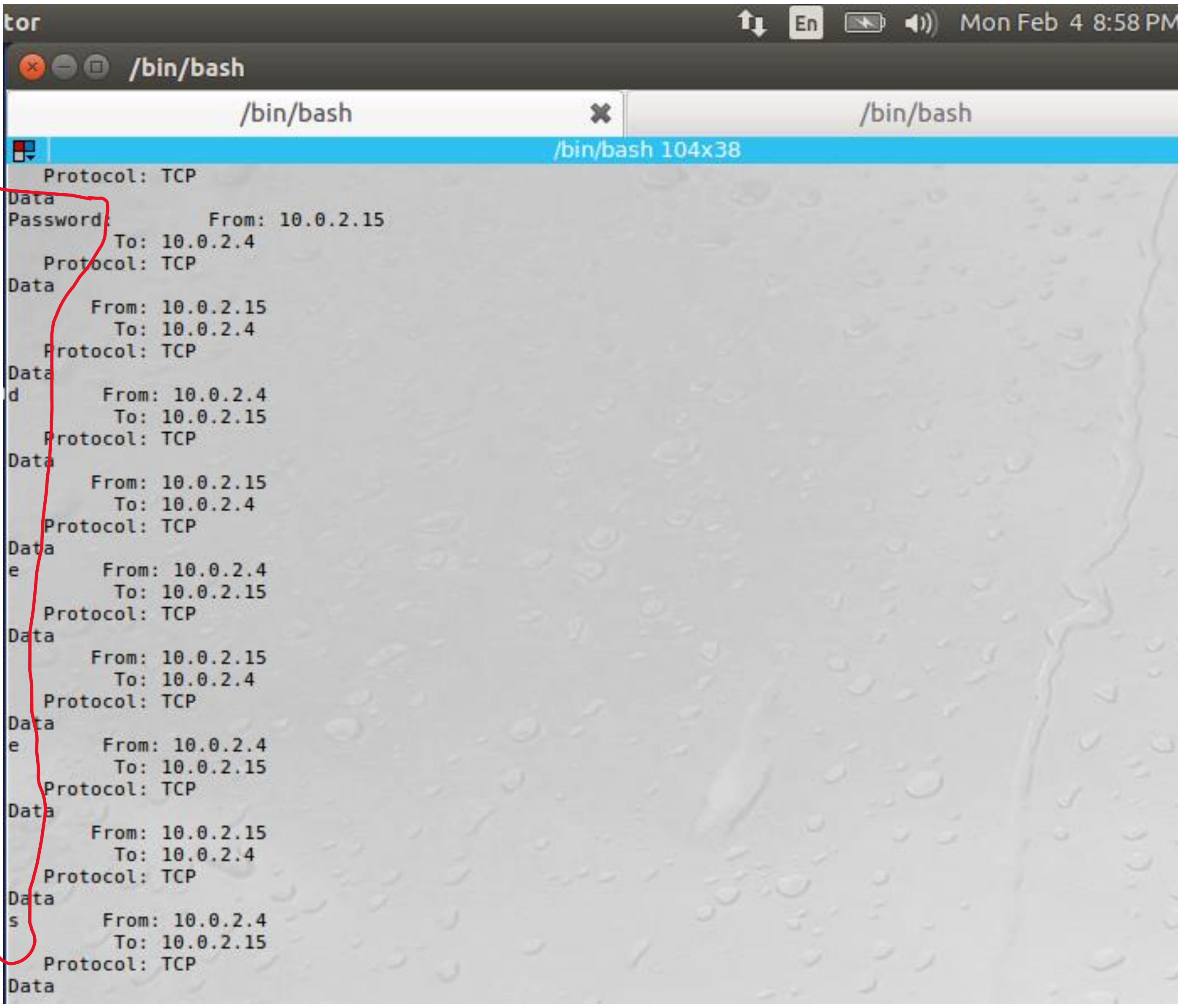
/* capturing payload */

```
  switch(ip->iph_protocol) {
    case IPPROTO_TCP:
      printf("   Protocol: TCP\n");
        struct sniff_tcp *tcp = (struct sniff_tcp *)(packet +  sizeof                    (struct
ethheader) + ip->iph_ihl*4);
        char *payload = (u_char *)(packet +  sizeof(struct                         ethheader)
+ ip->iph_ihl*4 +TH_OFF(tcp)*4);
        int size_payload = ntohs(ip->iph_len) - (ip->iph_ihl*4) - (TH_OFF(tcp)*4);

        printf("Data\n");
        for(int i = 0; i < size_payload; i++) {
            printf("%c", payload[i]);
      }
```

Screenshots:



```
[02/04/19]seed@VM:~$ telnet -l seed 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Password:
Last login: Mon Feb  4 20:51:31 EST 2019 from 10.0.2.15 on pts/19
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.
0 updates are security updates.

[02/04/19]seed@VM:~$ █
```



```
    Protocol: TCP
Data
Password:          From: 10.0.2.15
                 To: 10.0.2.4
    Protocol: TCP
Data
             From: 10.0.2.15
                To: 10.0.2.4
    Protocol: TCP
Data
d             From: 10.0.2.4
                To: 10.0.2.15
    Protocol: TCP
Data
             From: 10.0.2.15
                To: 10.0.2.4
    Protocol: TCP
Data
e             From: 10.0.2.4
                To: 10.0.2.15
    Protocol: TCP
Data
             From: 10.0.2.15
                To: 10.0.2.4
    Protocol: TCP
Data
e             From: 10.0.2.4
                To: 10.0.2.15
    Protocol: TCP
Data
             From: 10.0.2.15
                To: 10.0.2.4
    Protocol: TCP
Data
s             From: 10.0.2.4
                To: 10.0.2.15
    Protocol: TCP
Data
```

Observation: Sniffed tcp packet going from VM(10.0.2.15) to VM(10.0.2.4) with login credentials. Though the payload printed is gibberish I could make out the password.

Explanation: The marked literals in the picture above is the password ”dees”. I pinged telnet -l seed 10.0.2.15 for passing login credentials. I ran the telnet command on 10.0.2.4 to 10.0.2.15.

Task 2.2A: Write a spoofing program

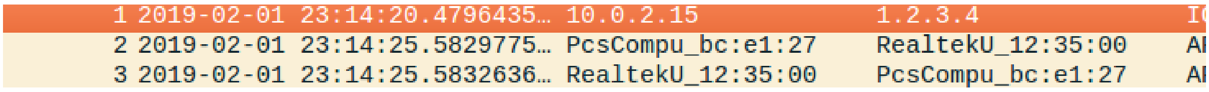
Code:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <pcap.h>
```

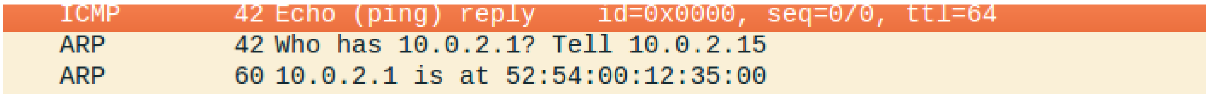
```
#include <stdio.h>
#include <arpa/inet.h>

unsigned short in_cksum (unsigned short *buf, int length)
{
  unsigned short *w = buf;
  int nleft = length;
  int sum = 0;
  unsigned short temp=0;

  /*
   * The algorithm uses a 32 bit accumulator (sum), adds
   * sequential 16 bit words to it, and at the end, folds back all
   * the carry bits from the top 16 bits into the lower 16 bits.
   */
  while (nleft > 1)  {
     sum += *w++;
     nleft -= 2;
  }

  /* treat the odd byte at the end, if any */
  if (nleft == 1) {
      *(u_char *)(&temp) = *(u_char *)w ;
      sum += temp;
  }

  /* add back carry outs from top 16 bits to low 16 bits */
  sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
  sum += (sum >> 16);                  // add carry
  return (unsigned short)(~sum);
}

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
};
```

```
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                   &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
struct icmpheader {
  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
  unsigned short int icmp_chksum; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
  unsigned short int icmp_seq;    //Sequence number
};


/*************************************************************
  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("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.15");
   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;
}
```

Result-Screenshots:



```
1 2019-02-01 23:14:20.4796435… 10.0.2.15          1.2.3.4           IC
2 2019-02-01 23:14:25.5829775… PcsCompu_bc:e1:27  RealtekU_12:35:00 AR
3 2019-02-01 23:14:25.5832636… RealtekU_12:35:00  PcsCompu_bc:e1:27 AR
```

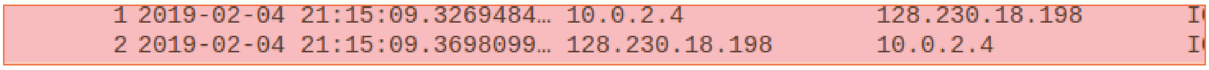


```
ICMP       42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64
ARP        42 Who has 10.0.2.1? Tell 10.0.2.15
ARP        60 10.0.2.1 is at 52:54:00:12:35:00
```

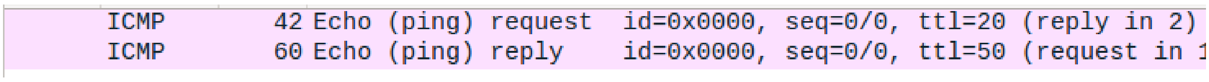
Explanation: Spoofed an ICMP echo-request packet from IP 1.2.3.4 to 10.0.2.15.

Task 2.2B: Spoof an ICMP Echo Request:

Screenshot:



```
1 2019-02-04 21:15:09.3269484… 10.0.2.4          128.230.18.198    IC
2 2019-02-04 21:15:09.3698099… 128.230.18.198    10.0.2.4          IC
```



```
ICMP       42 Echo (ping) request  id=0x0000, seq=0/0, ttl=20 (reply in 2)
ICMP       60 Echo (ping) reply    id=0x0000, seq=0/0, ttl=50 (request in 1
```

Explanation : Ran the same code as above question's by changing source host as my cloned VM and destination host as www.syr.edu. Successfully spoofed the packet to the desired destination.

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

A: Yes, the length of the IP header can be set to arbitrary values. The IP length gets re initiated to its original size, irrespective of the size set in the code.

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

A: Yes, we need to calculate the checksum for the IP header.

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?

A: Raw sockets access gives access to other applications in the network. This allows it to spoof a packet to any application. Since, it raises security concerns we need root privilege to run the program.

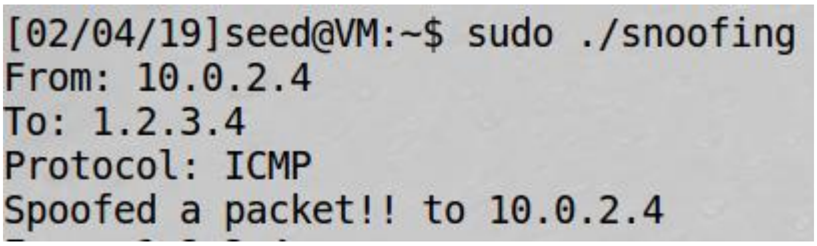
The program fails to run and throws an error indicating raw socket creation was not successful.

3.3 Task 2.3: Sniff and then Spoof.

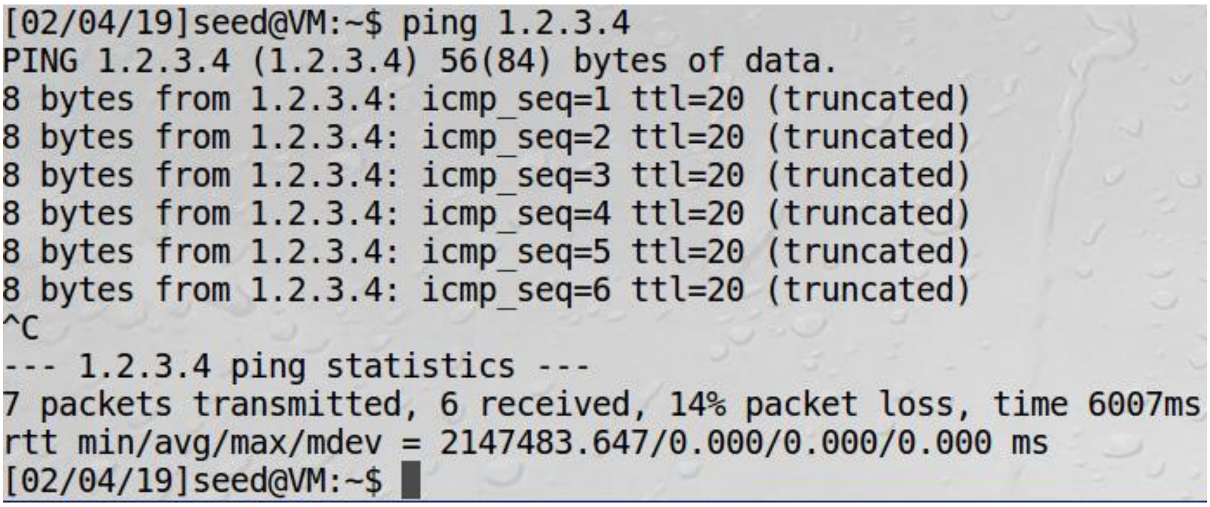
Code: Sniffing code is same as above and the Changes made in got_packet() is:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  char* buffer;
  //Extract headers to retrieve information
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip_src = (struct ipheader *)
                          (packet + sizeof(struct ethheader));
            printf("Received packet from %s ",inet_ntoa(ip_src->iph_sourceip));
            printf("going to %s\n",inet_ntoa(ip_src->iph_destip));
    struct icmpheader *icmp_src = (struct icmpheader *)
                            (packet + sizeof(struct ethheader) + sizeof(struct ipheader));

    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 = 0; //ICMP Type: 8 is request, 0 is reply.
   icmp->icmp_id = icmp_src->icmp_id;
   icmp->icmp_seq = icmp_src->icmp_seq;

   // 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 = ip_src->iph_destip.s_addr;
   ip->iph_destip.s_addr = ip_src->iph_sourceip.s_addr;
   printf("Created sourceip %s ",inet_ntoa(ip->iph_sourceip));
   printf("and destip %s",inet_ntoa(ip->iph_destip));
   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;

}
```

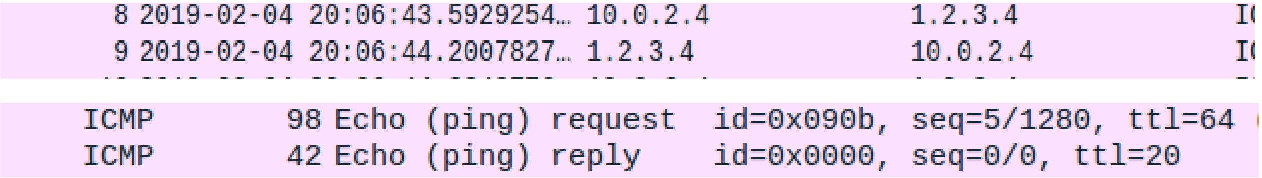
Screenshots:



```
[02/04/19]seed@VM:~$ sudo ./snoofing
From: 10.0.2.4
To: 1.2.3.4
Protocol: ICMP
Spoofed a packet!! to 10.0.2.4
```



```
[02/04/19]seed@VM:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
8 bytes from 1.2.3.4: icmp_seq=1 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=2 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=3 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=4 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=5 ttl=20 (truncated)
8 bytes from 1.2.3.4: icmp_seq=6 ttl=20 (truncated)
^C
--- 1.2.3.4 ping statistics ---
7 packets transmitted, 6 received, 14% packet loss, time 6007ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
[02/04/19]seed@VM:~$ 
```



```
      8 2019-02-04 20:06:43.5929254… 10.0.2.4          1.2.3.4          I
      9 2019-02-04 20:06:44.2007827… 1.2.3.4           10.0.2.4         I

    ICMP       98 Echo (ping) request  id=0x090b, seq=5/1280, ttl=64
    ICMP       42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=20
```

Explanation: I ran the ping command from VM(10.0.2.4) to 1.2.3.4. The code on VM(10.0.2.15) sniffed the packet(echo-request) and spoofed a packet to 10.0.2.4 saying its from 1.2.3.4(echo-reply). We can see in the 2nd picture that VM 10.0.2.4 received the spoofed packets.