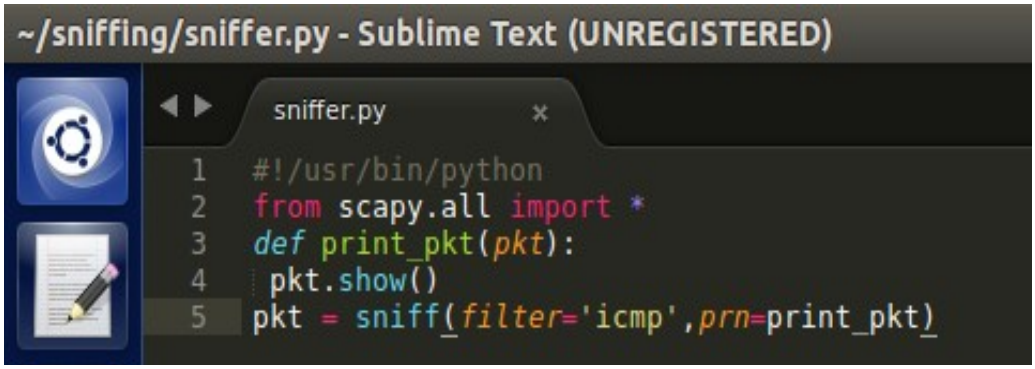
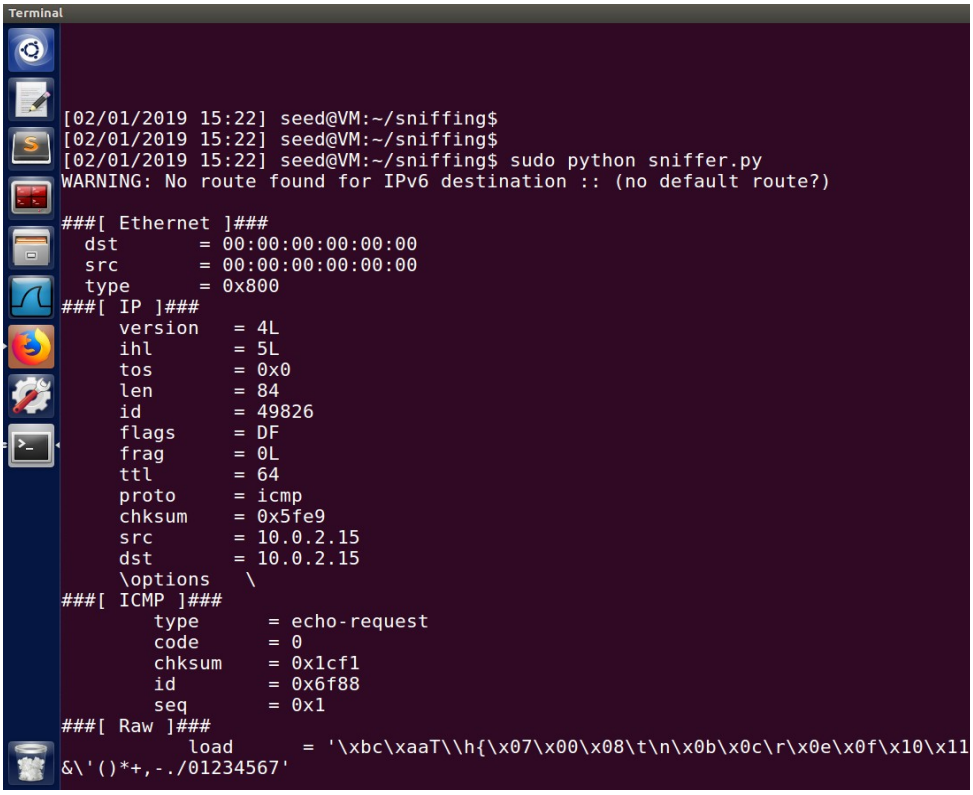
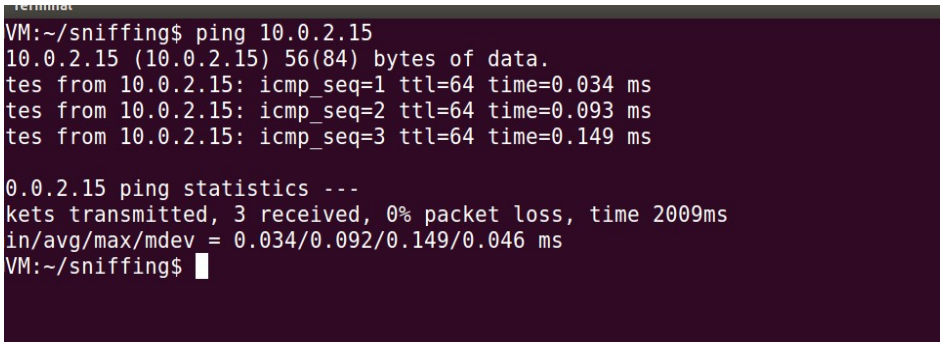
Packet Sniffing and Spoofing Lab

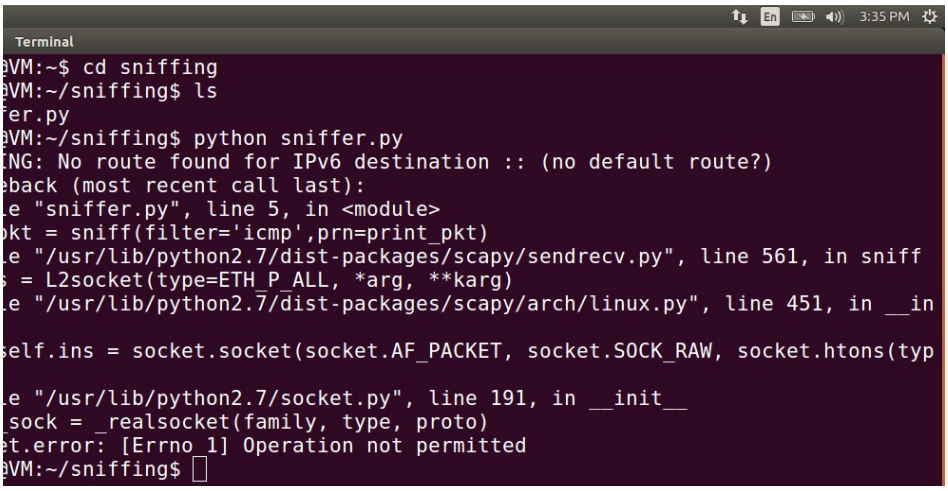
Task 1.1 A: Sniffing Packets







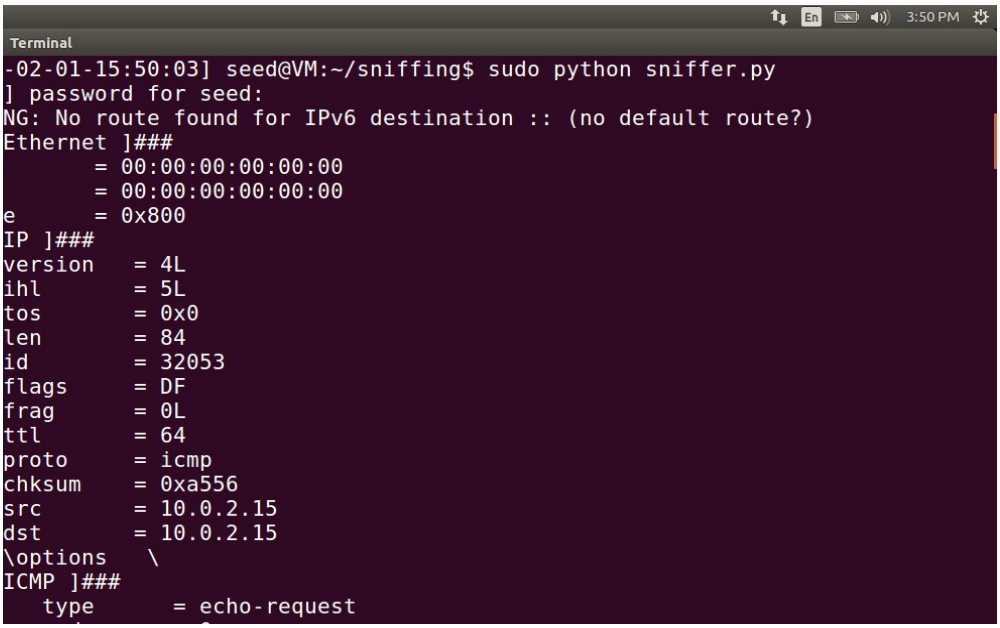
Observation: In task 1.1 A we first copied the code to sniffer.py. After that we executed it. Then we pinged the IP as seen from the above screenshot. We can see that we are able to get the details of the ICMP packets and the statistics of the same. As seen from the above screen shot we need to use sudo in order to run the code



Observation: Similarly we tried to do the same but without using the root privileges and we can see that we are not allowed to execute the code as the operation is not permitted meaning that we need the root privileges to sniff the packets.

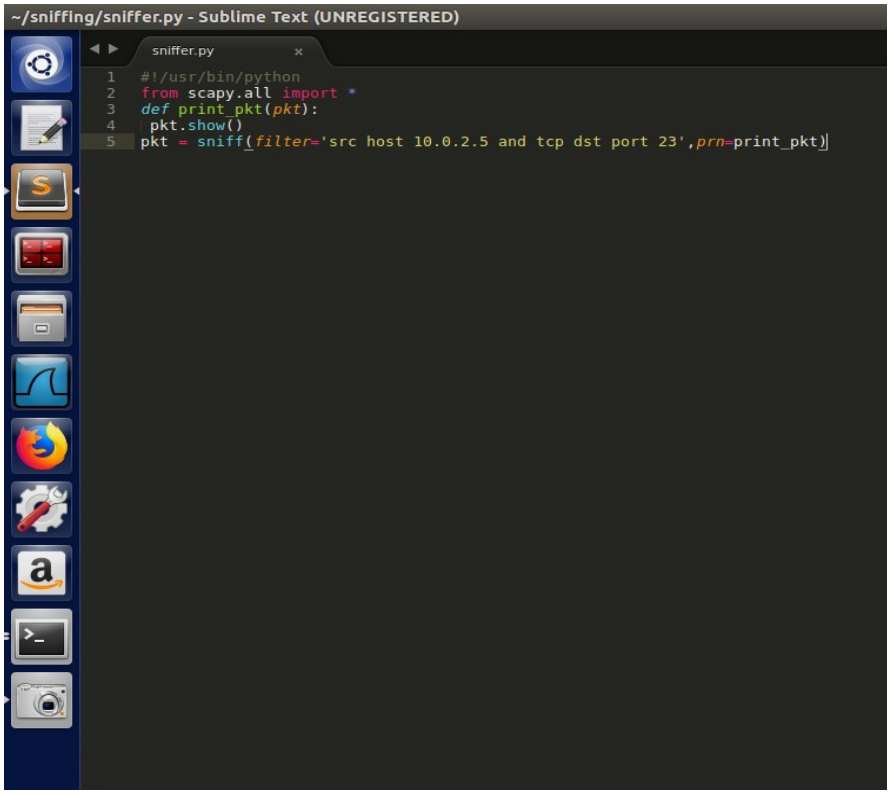
Explanation : As network interface card (NIC) accepts packets & only root can access it and the permission is given to pcap which is used in the code. Therefore, we need root privileges to execute the code.

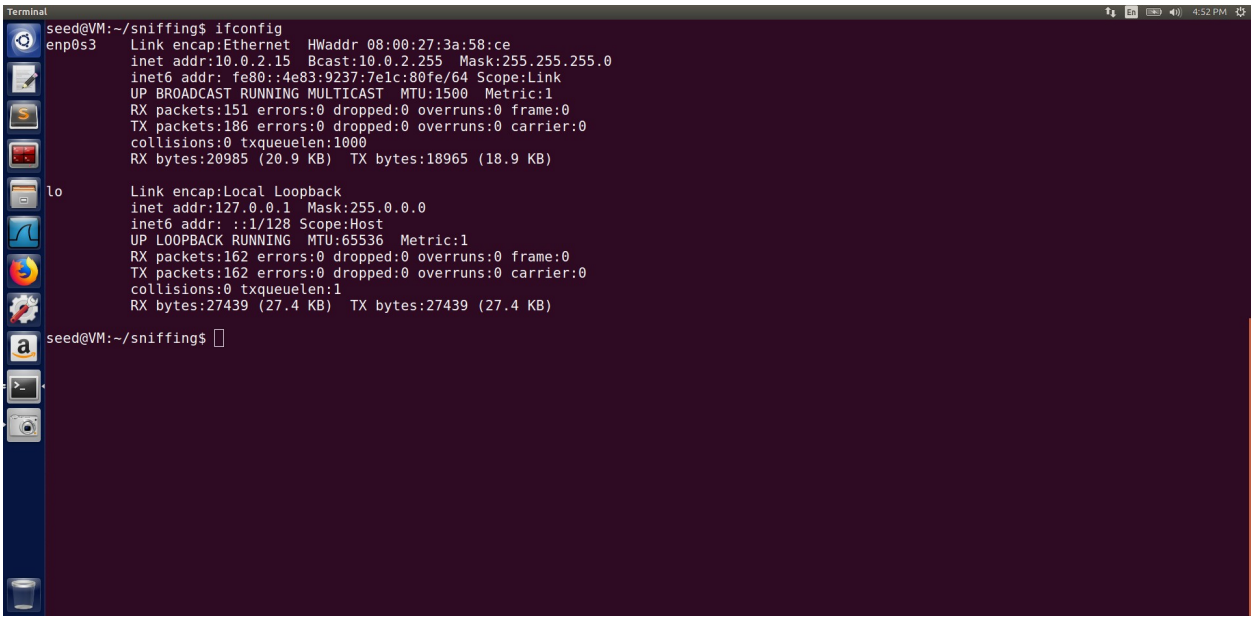
Task 1.1 B - Capture only the ICMP packet

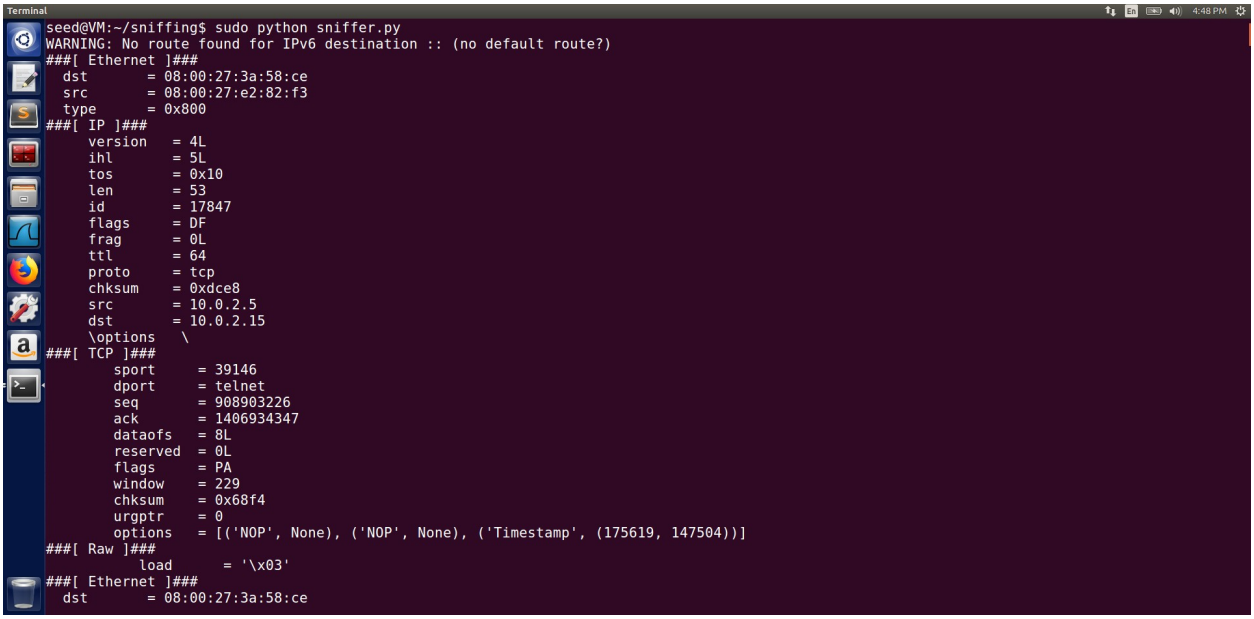


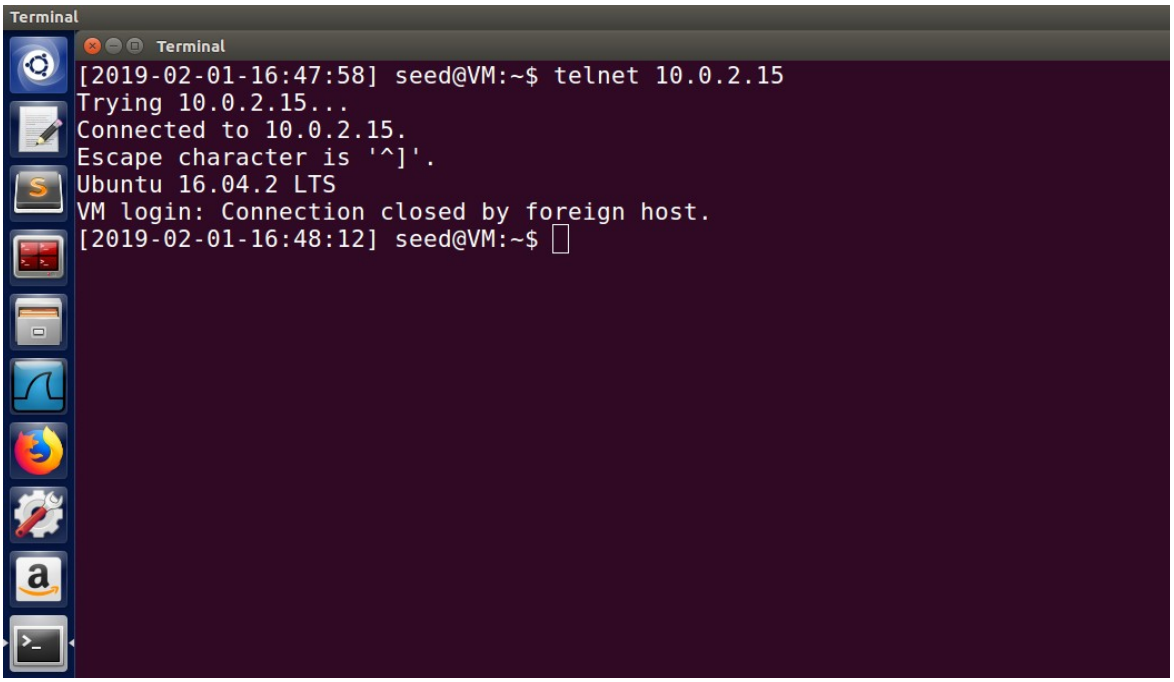
This task is similar to the task 1.1 A where we captured the ICMP packet.

Task 1.1 B -Capture any TCP packet that comes from a particular IP and with a destination port number 23.




Terminal
4:52 PM
seed@VM:~/sniffing$ ifconfig
enp0s3   Link encap:Ethernet HWaddr 08:00:27:3a:58:ce
          inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
          inet6 addr: fe80::4e83:9237:7e1c:80fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:151 errors:0 dropped:0 overruns:0 frame:0
          TX packets:186 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20985 (20.9 KB) TX bytes:18965 (18.9 KB)
lo   Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:162 errors:0 dropped:0 overruns:0 frame:0
          TX packets:162 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:27439 (27.4 KB) TX bytes:27439 (27.4 KB)
seed@VM:~/sniffing$

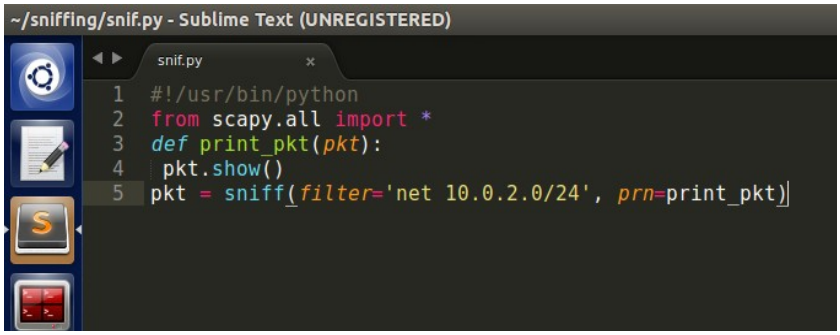

Terminal
4:48 PM
seed@VM:~/sniffing$ sudo python sniffer.py
WARNING: No route found for IPv6 destination :: (no default route?)
###[Ethernet]###
  dst   = 08:00:27:3a:58:ce
  src   = 08:00:27:e2:82:f3
  type   = 0x800
###[IP]###
     version   = 4L
     ihl   = 5L
     tos   = 0x10
     len   = 53
     id   = 17847
     flags   = DF
     frag   = 0L
     ttl   = 64
     proto   = tcp
     chksum   = 0xdce8
     src   = 10.0.2.5
     dst   = 10.0.2.15
     \options   \
###[TCP]###
        sport   = 39146
        dport   = telnet
        seq   = 908903226
        ack   = 1406934347
        dataofs   = 8L
        reserved = 0L
        flags   = PA
        window   = 229
        chksum   = 0x68f4
        urgptr   = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (175619, 147504))]
###[Raw]###
           load   = '\x03'
###[Ethernet]###
  dst   = 08:00:27:3a:58:ce

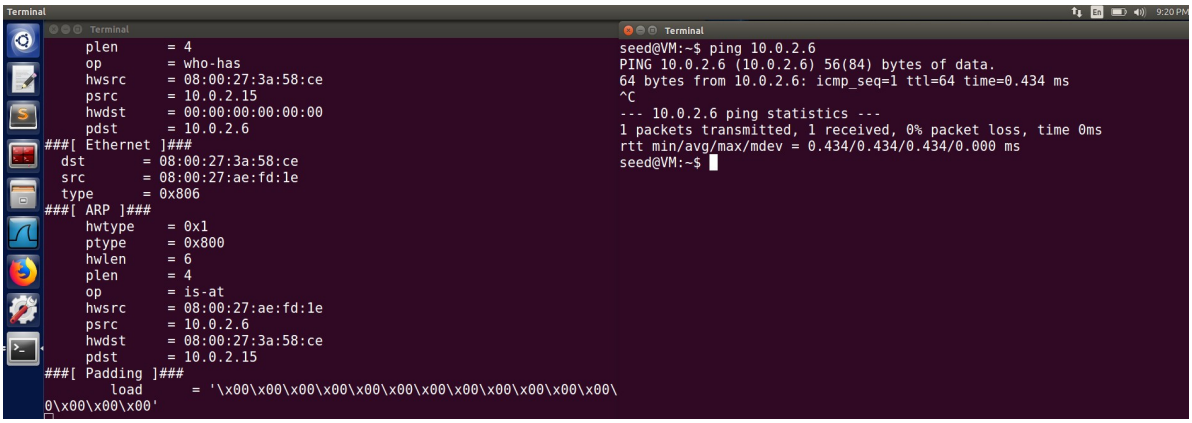


In this task we use the code sniffer.py. In that we filter the source IP to 10.0.2.5 which means that only packets from that IP will be allowed to enter. And we filter the TCP destination port to 23 meaning that packets will only be sent to port 23 of the destination IP. Once, we execute the code we sent the packets from 10.0.2.5 to 10.0.2.15 and we can see that only the TCP packets are seen and no other packets which states that all the TCP packets are being captured.

Explanation: We executed from the IP 10.0.2.5 and connected to IP 10.0.2.15 using telnet. Since, telnet uses the port 23 we can see that we get the details of the packet

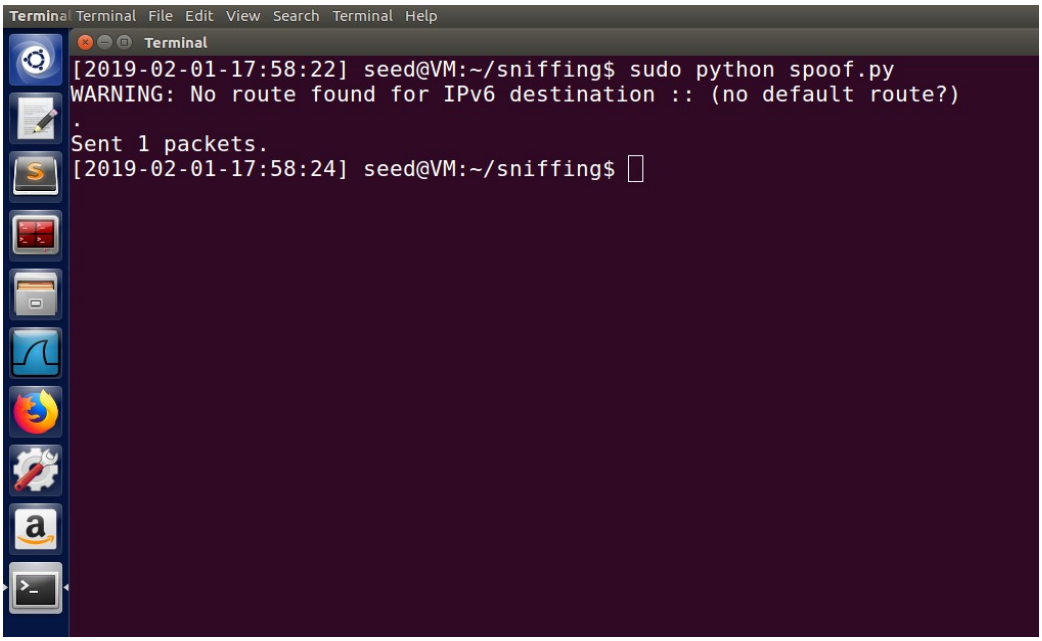
Task 1.1 - Capture packets comes from or to go to a particular subnet.

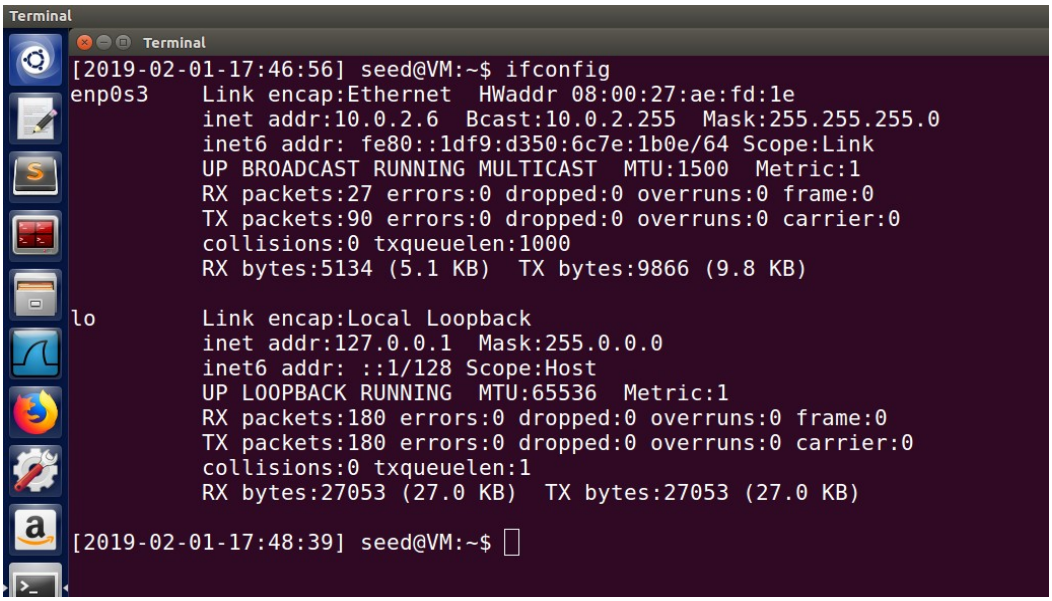


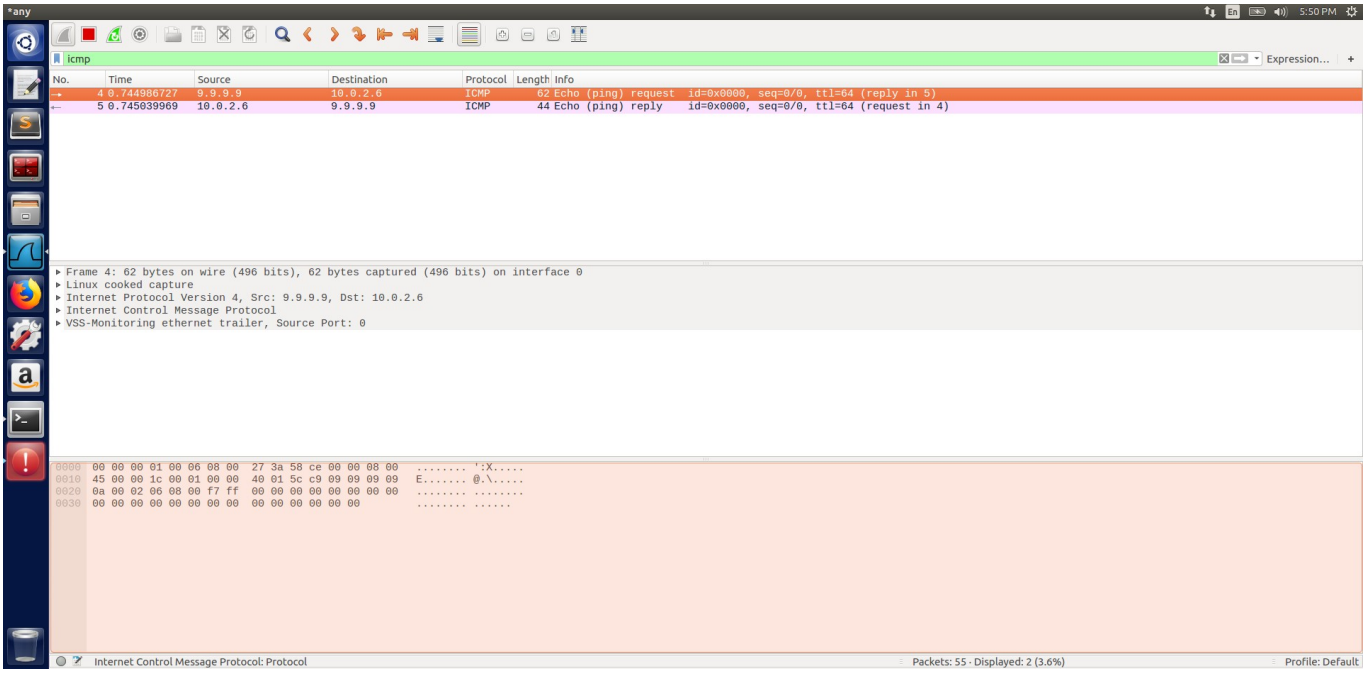


Here we used the filter = ‘net 10.0.2.0/24’

Task 1.2: Spoofing ICMP Packets



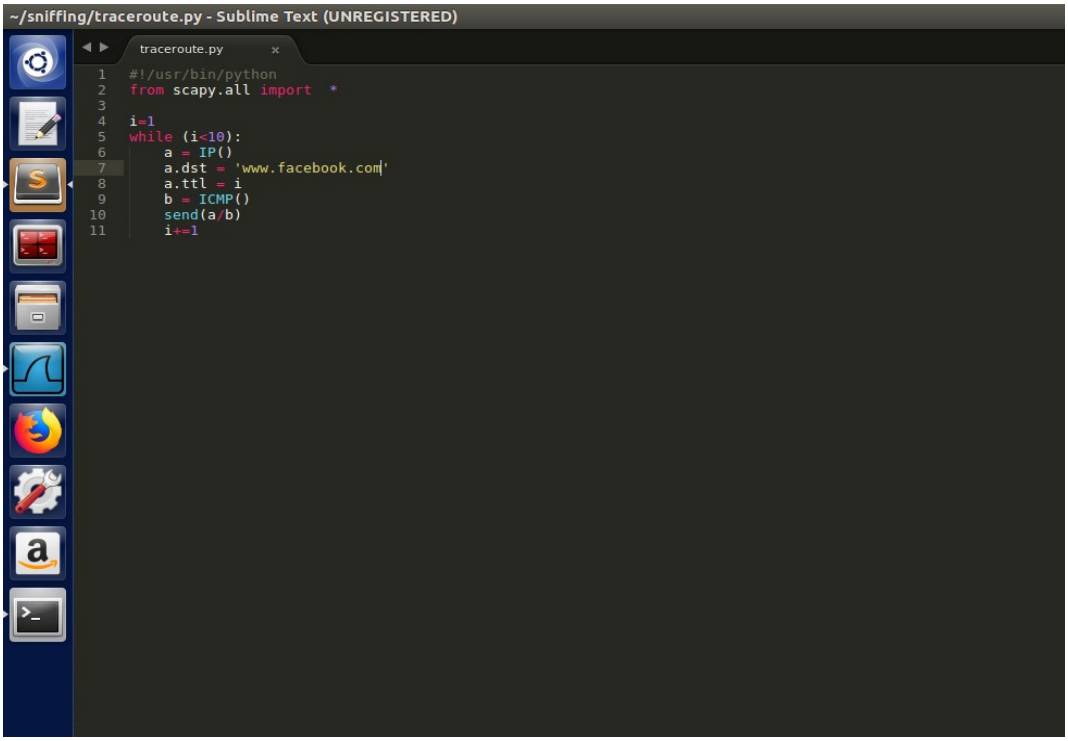


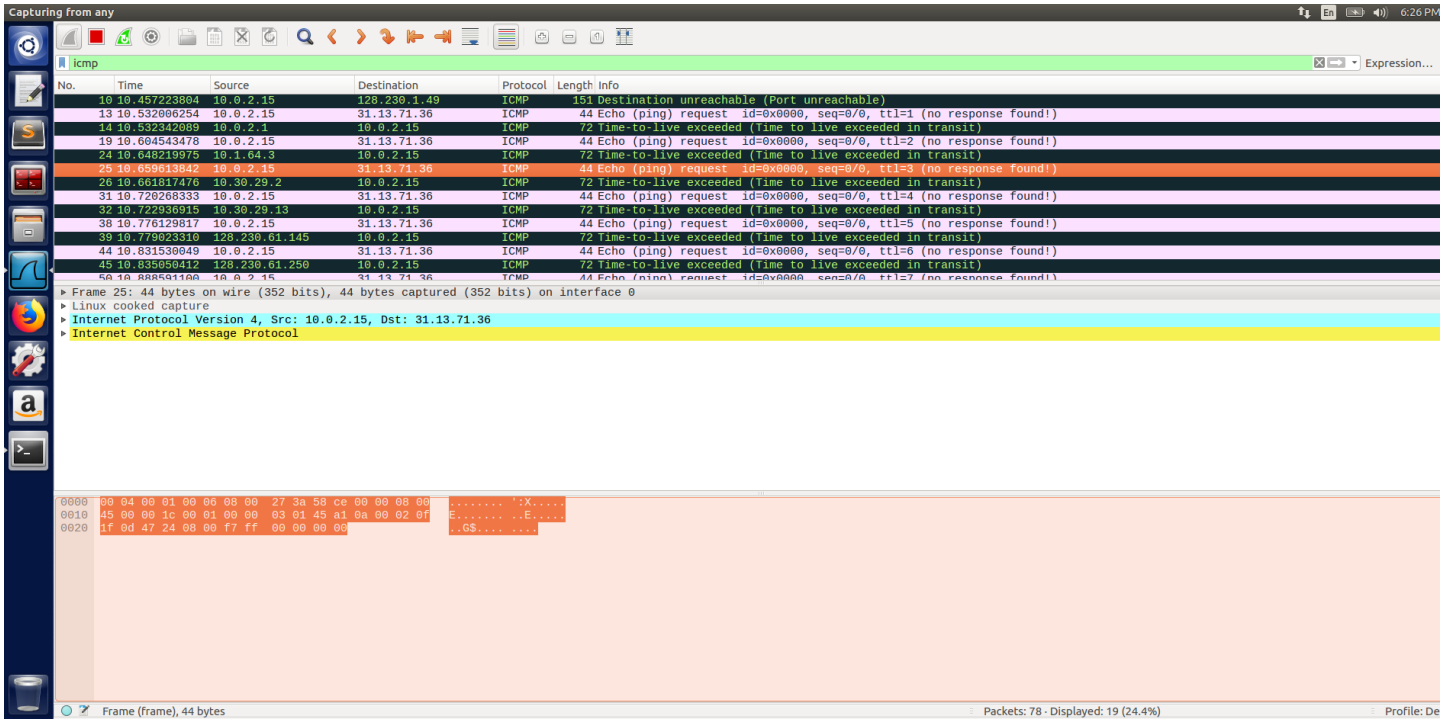


Observation - In this task we keep the source ip as 9.9.9.9 and destination ip as one of the VM's ip that is 10.0.2.6. After that we execute the code and the destination ip assumes that the packets are coming from the source ip 9.9.9.9.

Explanation- We can see on wireshark that we are able to read both the packets as above. We are using arbitrary source ip and keeping one of our VM as destination ip – 10.0.2.6. Wireshark is able to capture the spoofed packets as seen from the screenshot.

Task 1.3: Traceroute

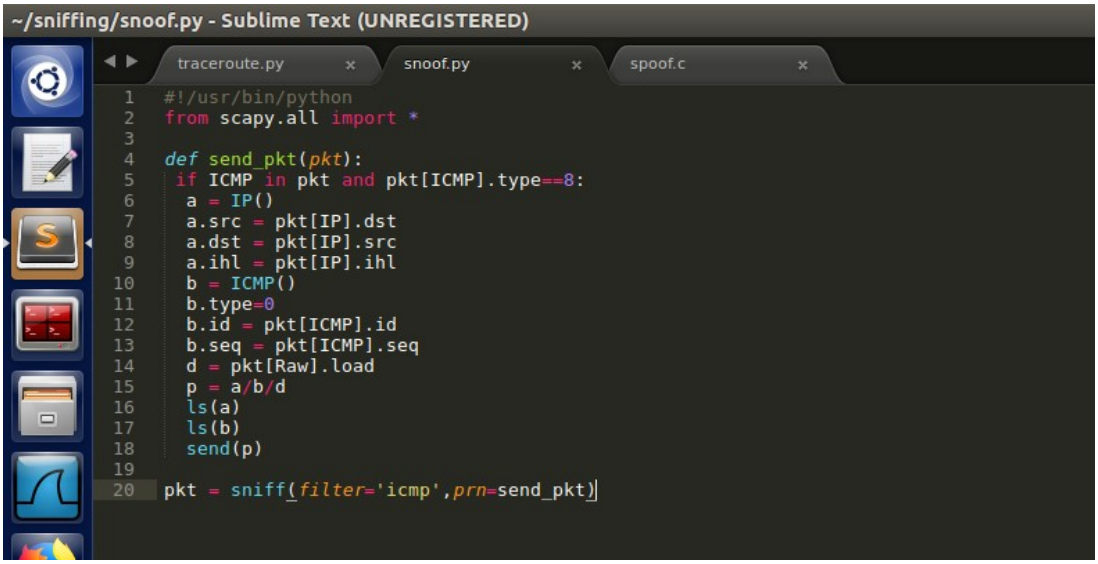


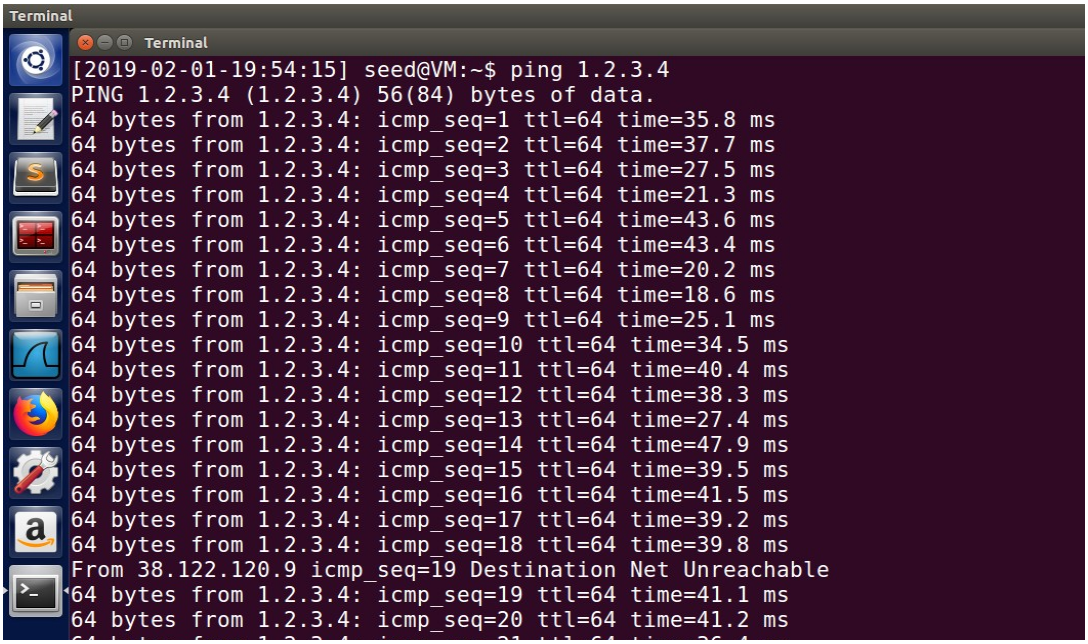


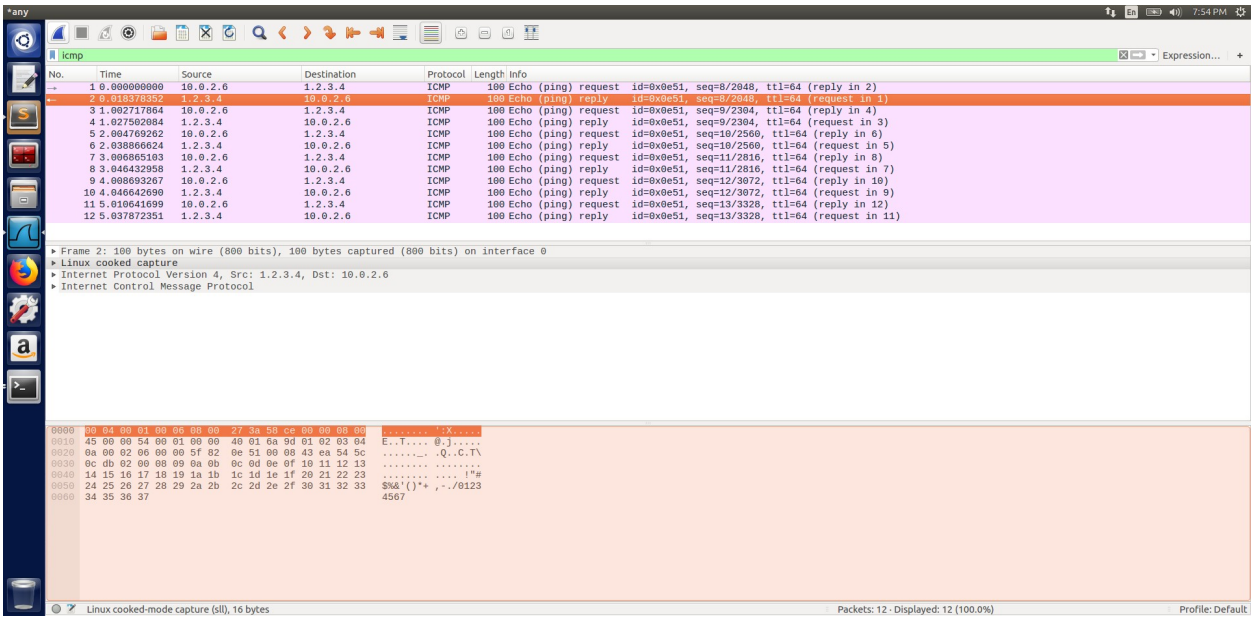
Observation – In this task we compile and execute the code traceroute.py and are pinging to ‘www.facebook.com’ and increase our time to live from 1 to 10.

Explanation - In the wireshark screenshot we can see that if the time to live is short then the packet does not reach the destination and will be dropped by the first router and will send us an ICMP error that will tell us that time to live has exceeded and we will get the IP address of the first router and will keep repeating until the packet reaches the destination. We are able to see numerous IP indicating the hops that are occurring.

Task 1.4: Sniffing and-then Spoofing



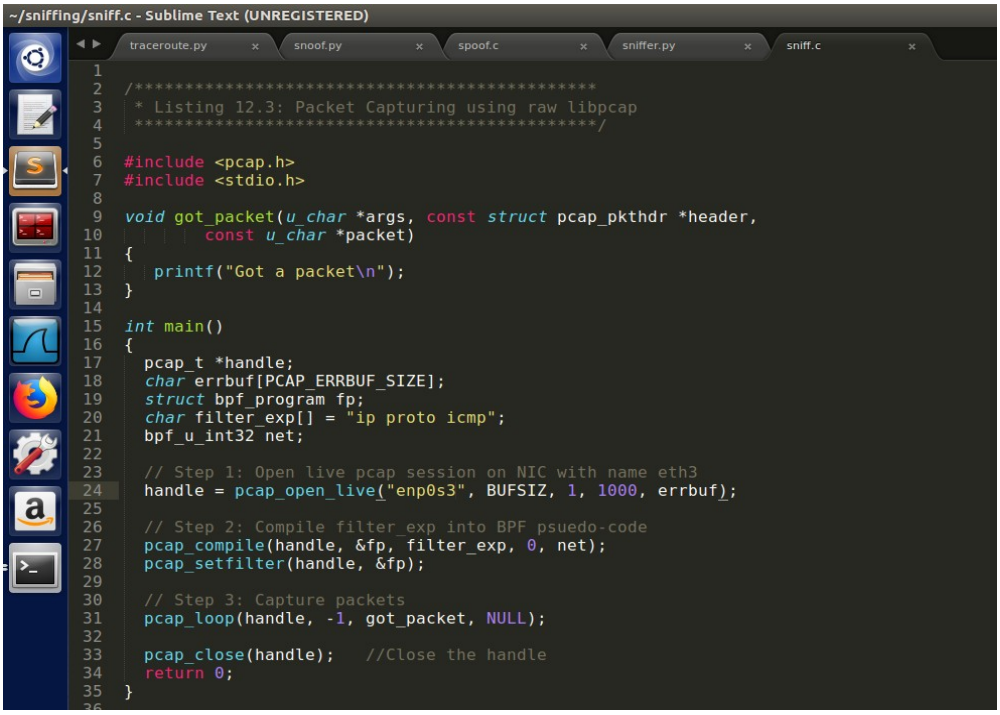


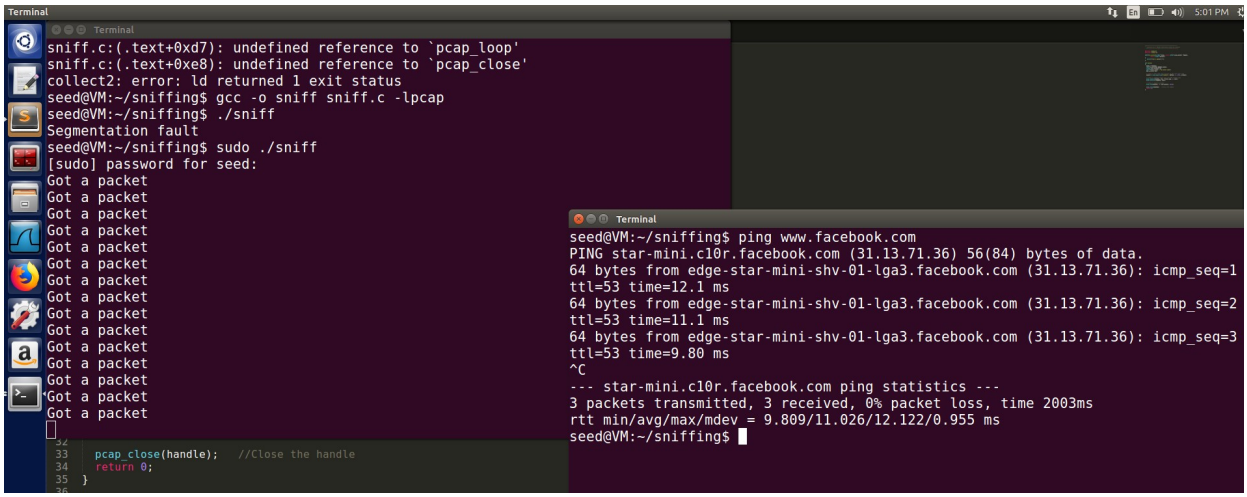


Observation - In this task we compile and execute the snoof.py code. We are pinging any arbitrary ip – 1.2.3.4.

Explanation- Even after that ip not being a valid one it is getting a response from our IP that is 10.0.2.6. Thus, it indicates that whether the destination machine is alive or not our machine sends out the echo reply using the packet spoofing technique.

Task 2.1 A: Writing Packet Sniffing Program





Question 1 : Please use your own words to describe the sequence of the library calls that are essential for sniffer programs

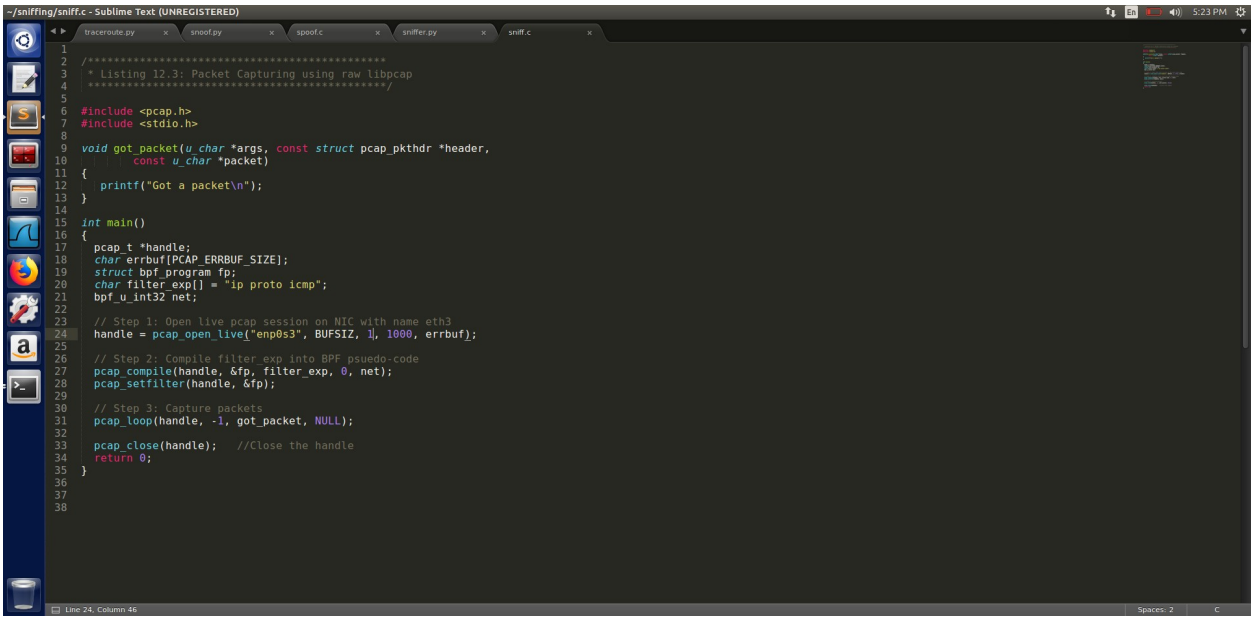
(a) Setting up device : We first have to set up the device which pcap does by itself. If there are any errors then it is saved in the errbuf.
(b) Open sniffing : after the previous setup we open the device for the sniffing using the command – pcap_open_live()
(c) Traffic Filter : pcap_compile() is used to compile the filter and pcap_setfilter() is used to tell what the program will sniff.
(d) Sniffing : We capture packets when it arrives at the NIC.
(e) Close Sniffing : Once all the previous task are done we close the session.

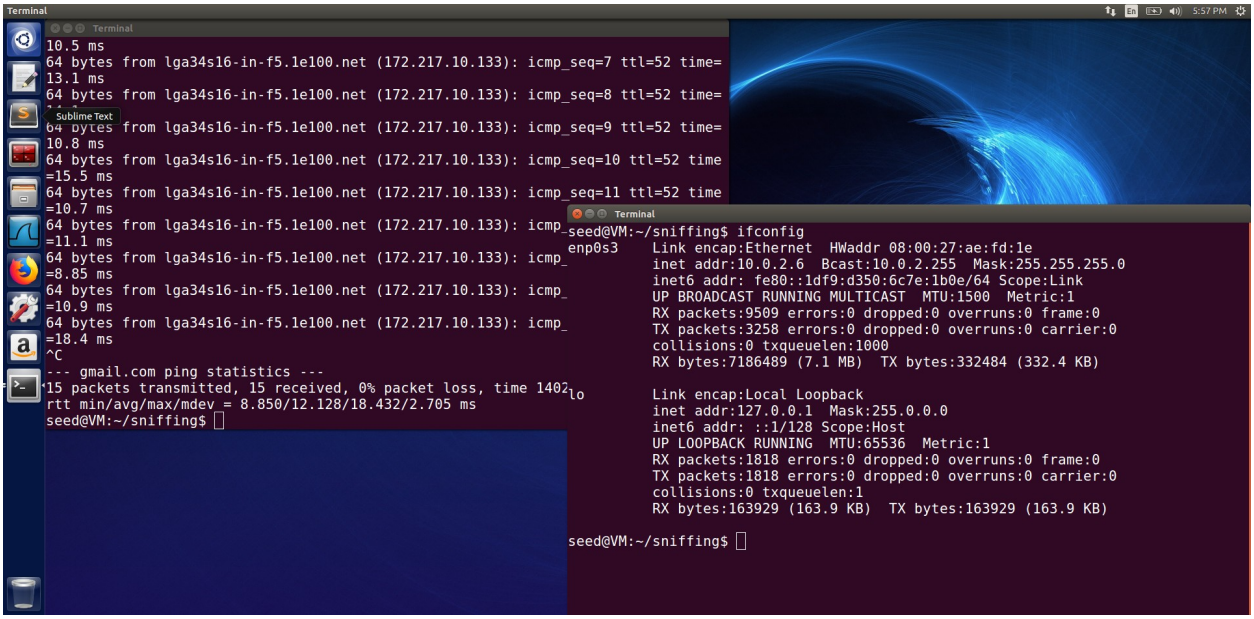
Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

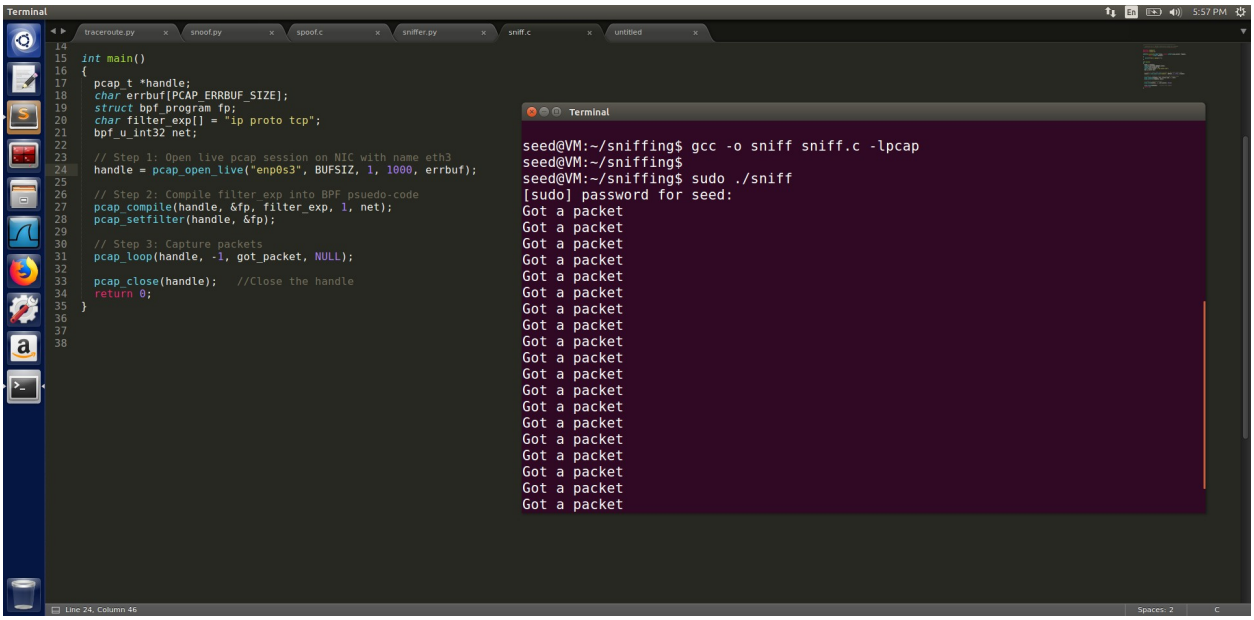
Root permissions are needed in order to use pcap since it can access network interface card that accepts packets that only root can access. The pcap_lookupdev() will fail if the root privileges is not present.

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

Promiscuous Mode

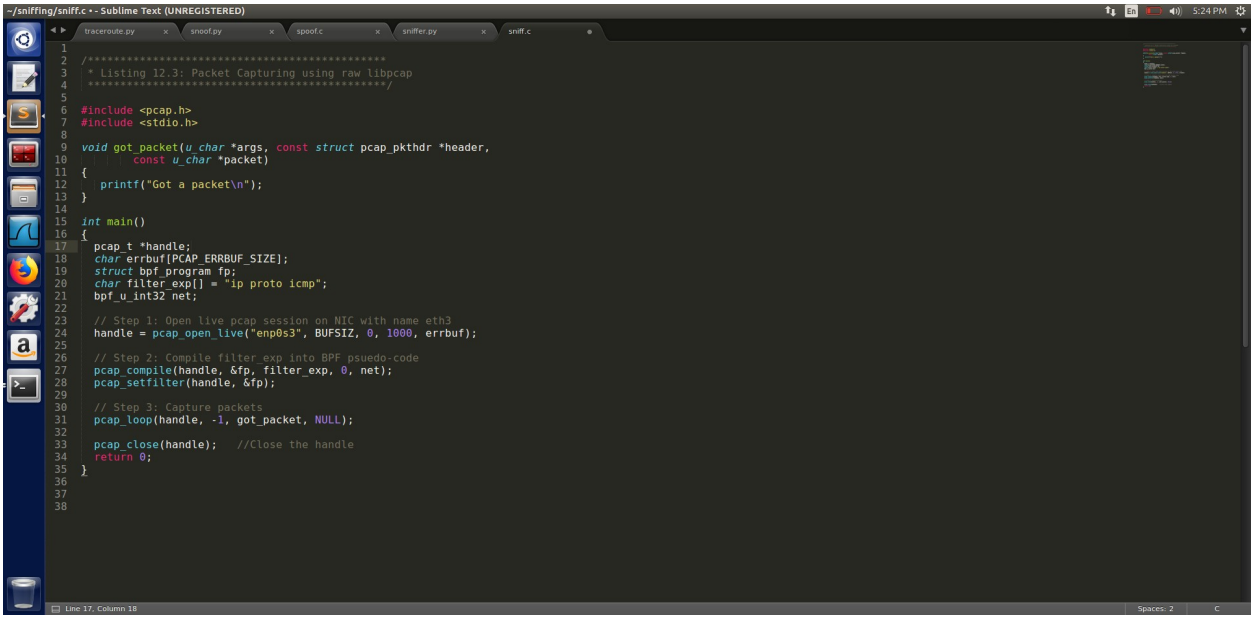

~/sniffing/sniff.c - Sublime Text (UNREGISTERED)
traceroute.py
snoof.py
spoof.c
sniffer.py
sniff.c
/***
* Listing 12.3: Packet Capturing using raw libpcap
***/
#include <pcap.h>
#include <stdio.h>
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet)
{
printf("Got a packet\n");
}
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;
}
Line 24, Column 46
Spaces: 2
C

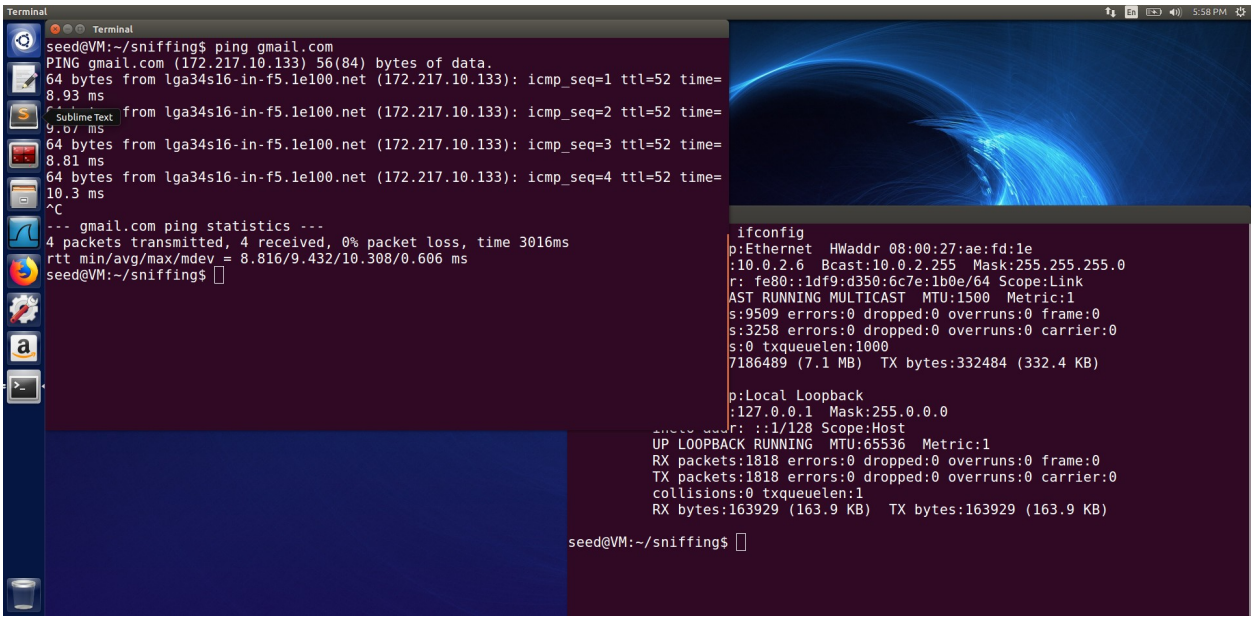

Terminal
Terminal
10.5 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_seq=7 ttl=52 time=
13.1 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_seq=8 ttl=52 time=
Sublime Text
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_seq=9 ttl=52 time=
10.8 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_seq=10 ttl=52 time
=15.5 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_seq=11 ttl=52 time
=10.7 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_
=11.1 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_
=8.85 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_
=10.9 ms
64 bytes from lga34s16-in-f5.1e100.net (172.217.10.133): icmp_
=18.4 ms
^C
--- gmail.com ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 1402
rtt min/avg/max/mdev = 8.850/12.128/18.432/2.705 ms
seed@VM:~/sniffing$
Terminal
seed@VM:~/sniffing$ ifconfig
enp0s3 Link encap:Ethernet HWaddr 08:00:27:ae:fd:1e
inet addr:10.0.2.6 Bcast:10.0.2.255 Mask:255.255.255.0
inet6 addr: fe80::1df9:d350:6c7e:1b0e/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:9509 errors:0 dropped:0 overruns:0 frame:0
TX packets:3258 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:7186489 (7.1 MB) TX bytes:332484 (332.4 KB)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:1818 errors:0 dropped:0 overruns:0 frame:0
TX packets:1818 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:163929 (163.9 KB) TX bytes:163929 (163.9 KB)
seed@VM:~/sniffing$

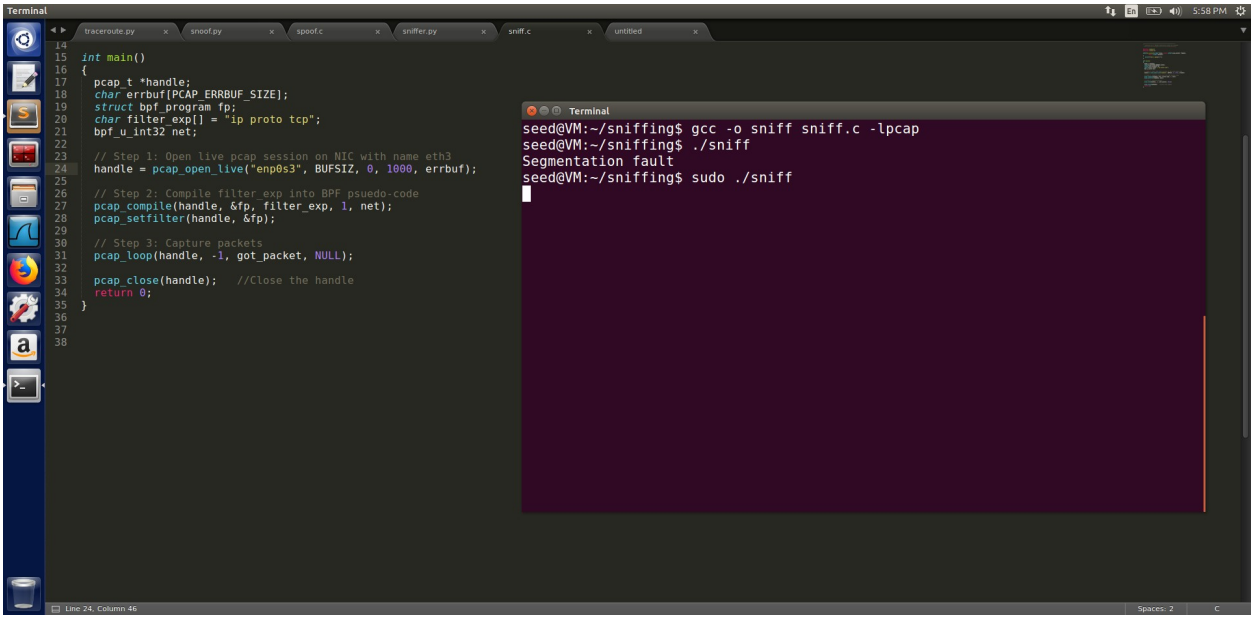


When we turn on the promiscuous mode by changing the value to 1 and then we compile and execute the code after that if we ping gmail.com we can see that the other VM from where we ran the code is able to capture the packets.

Non - Promiscuous Mode



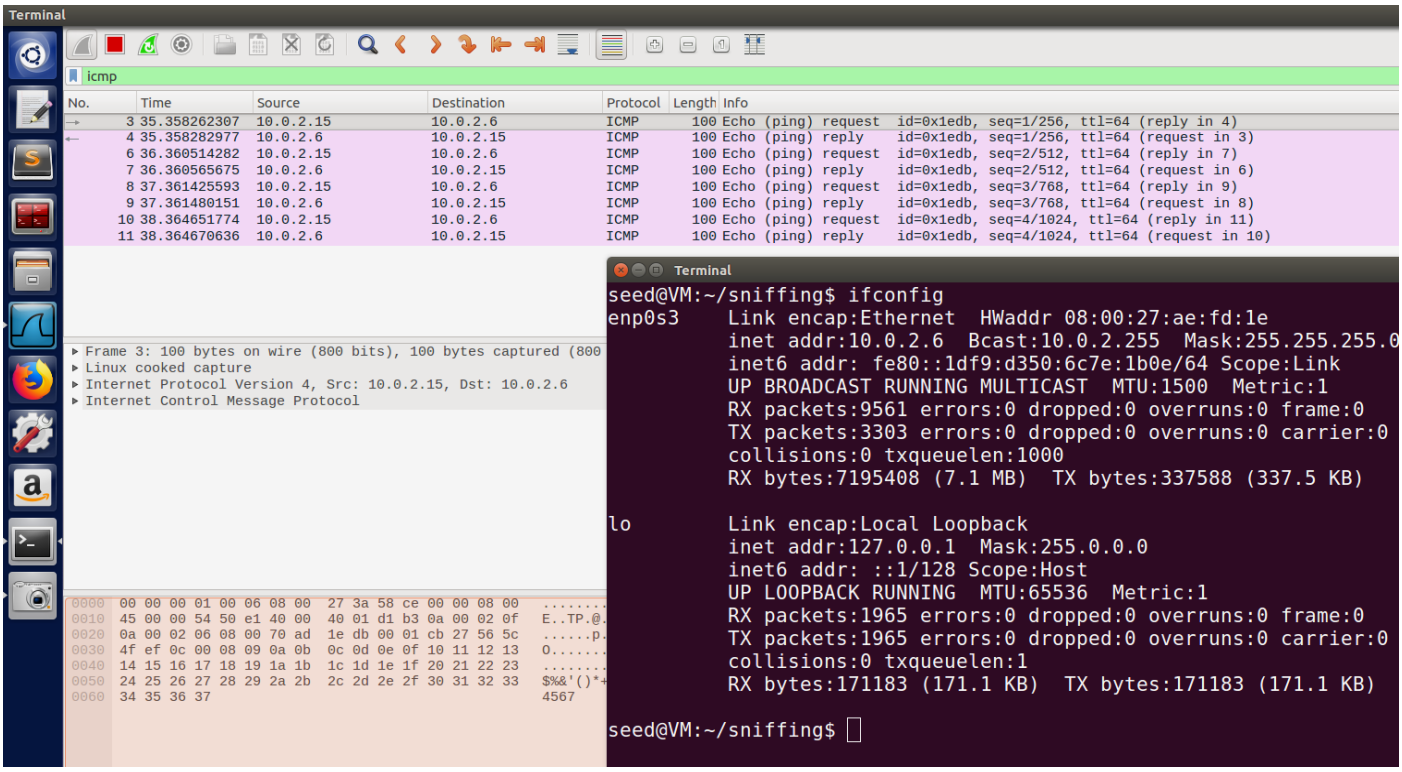


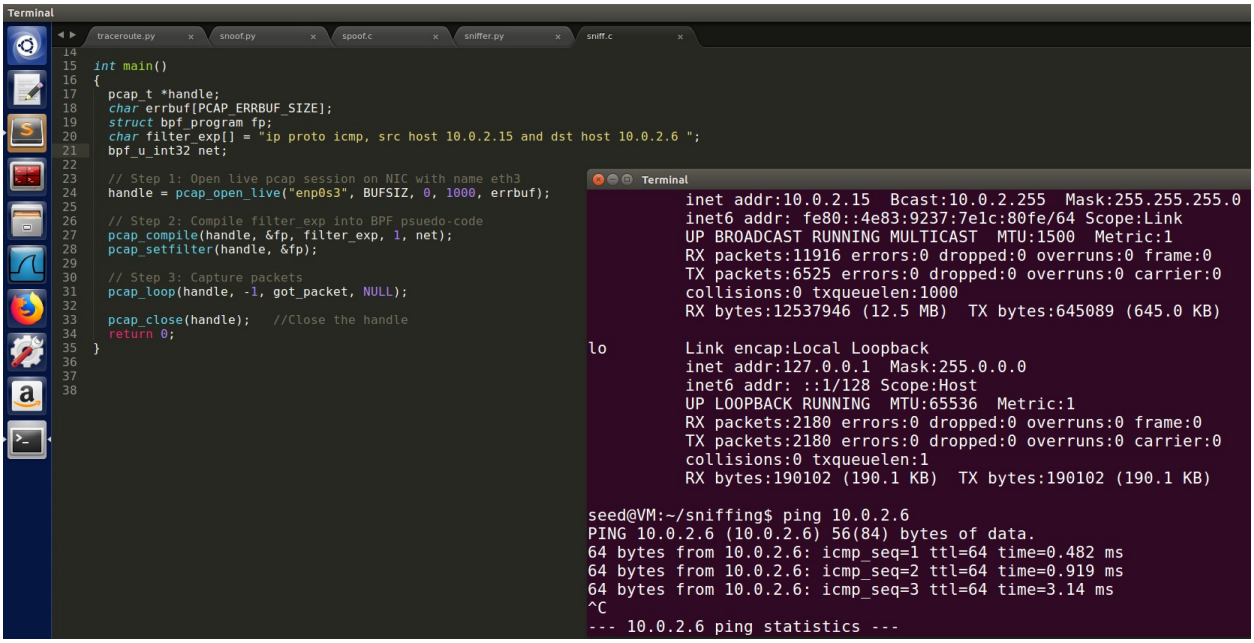


When we turn off the promiscuous mode by changing the value to 0 and then we compile and execute the code after that if we ping gmail.com we can see that the other VM from where we ran the code is not able to capture the packets.

Task 2.1B: Writing Filters

Capture the ICMP packets between two specific hosts.

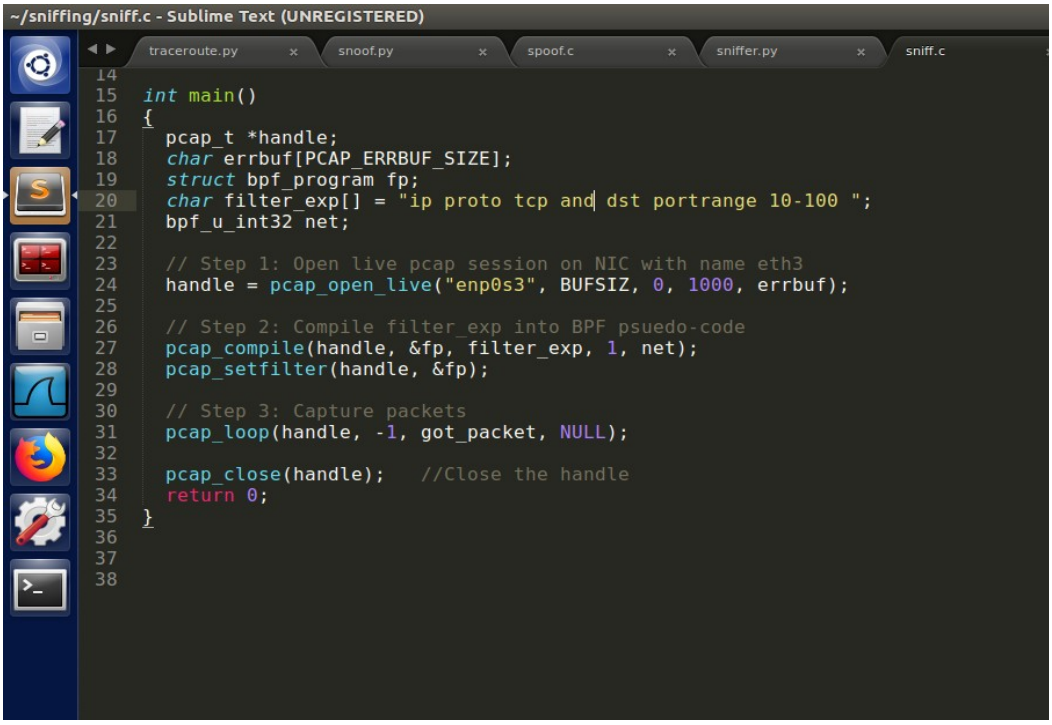


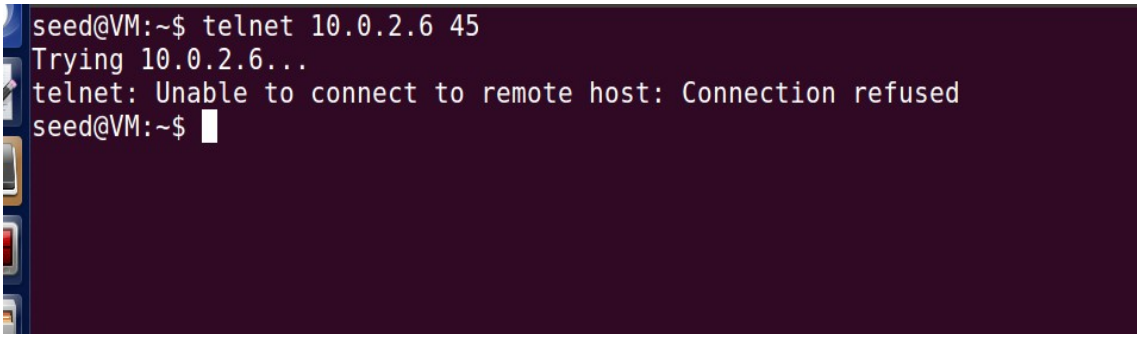


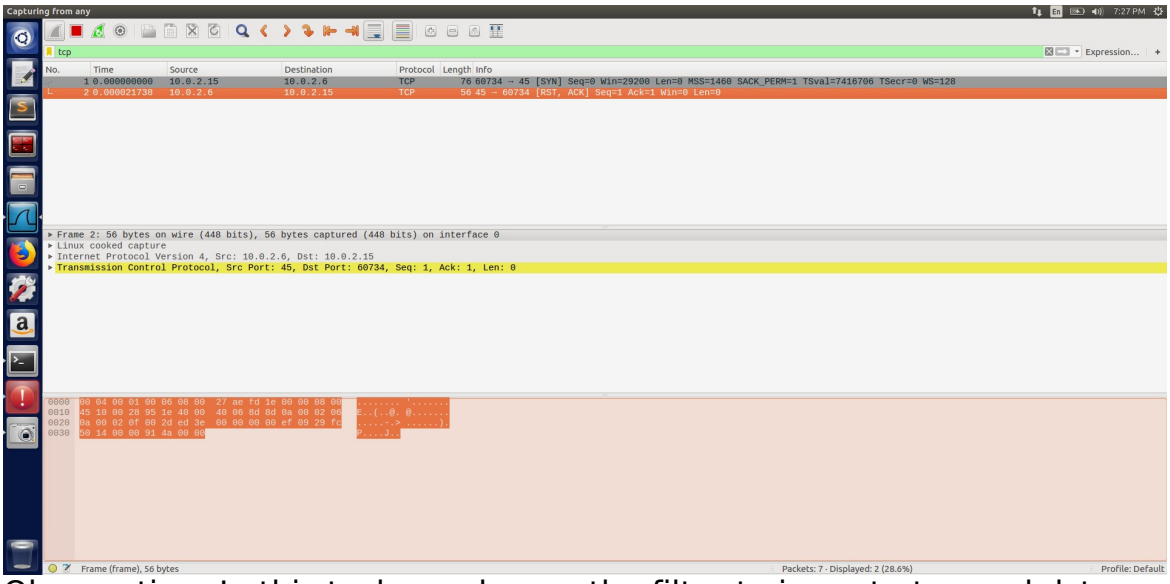
Observation : In the code above we change the filter to ‘ ip proto icmp, src host 10.0.2.15 and dst host 10.0.2.6’

Explanation : The filter that we changed means that only packets that are going from 10.0.2.15 and having its destination 10.0.2.6 are captured which can be seen in the wireshark.

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



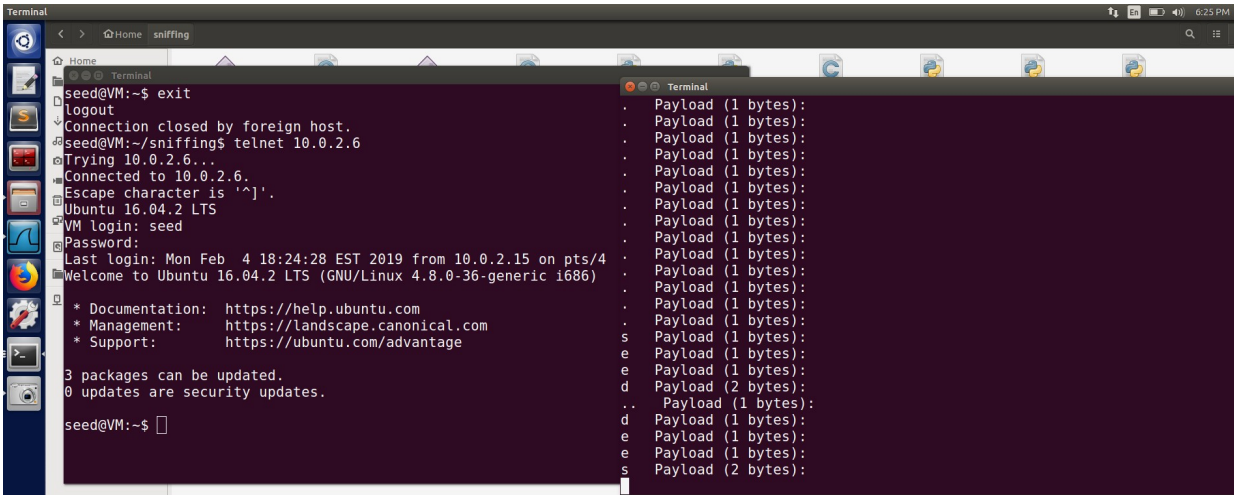


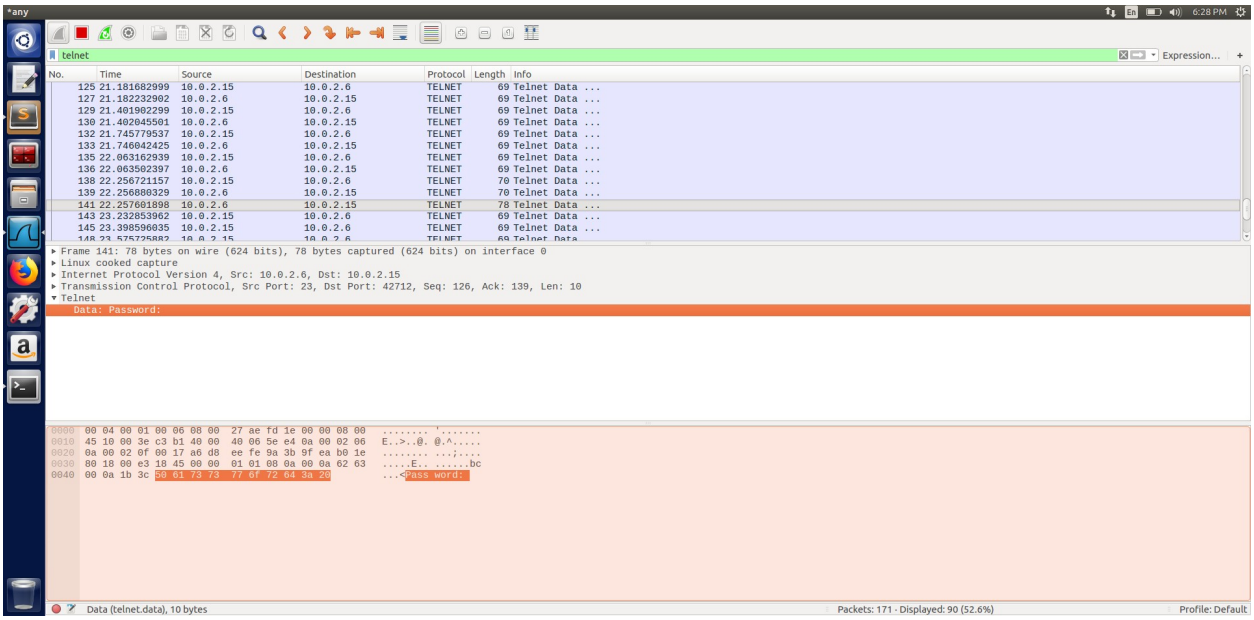


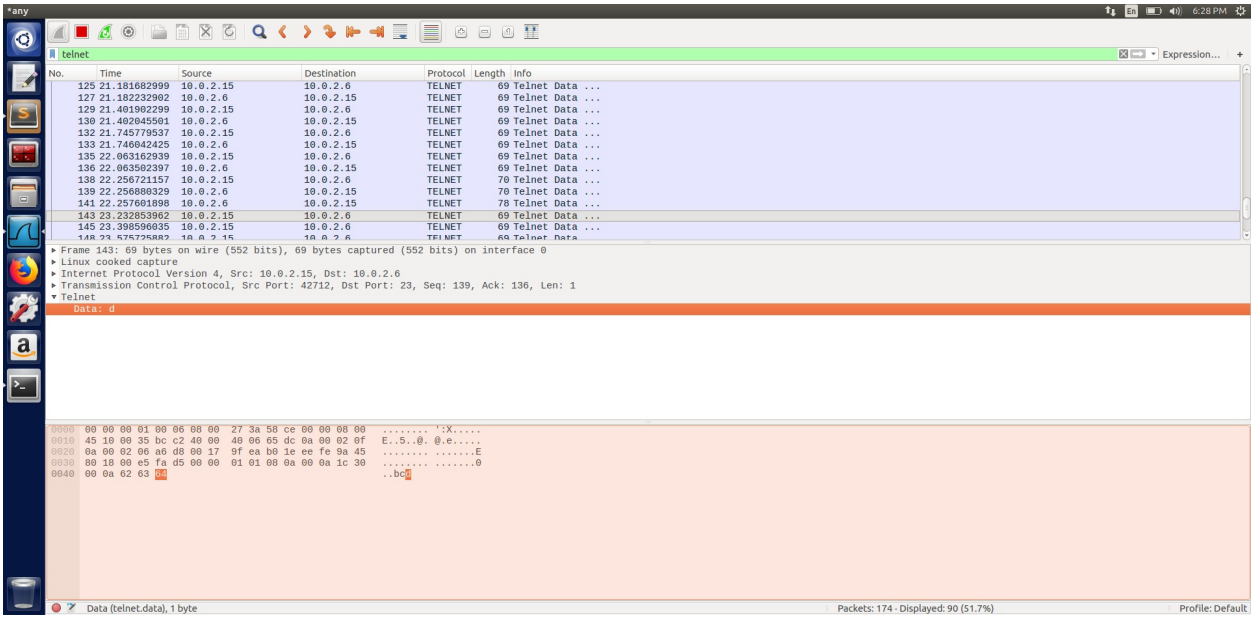
Observation: In this task we change the filter to ip proto tcp and dst portrange 10-100

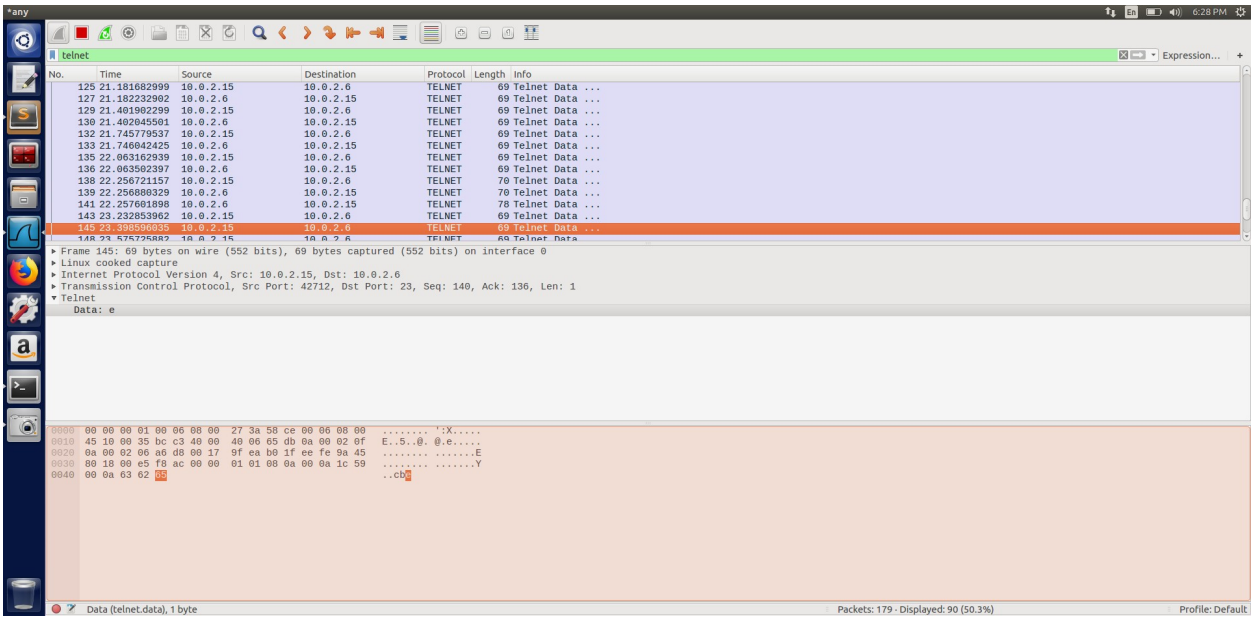
Explanation: we use the command ‘telnet 10.0.2.6 45’ which specifies the ip address and the port number. We can see in wireshark that the packet was sent to port 45

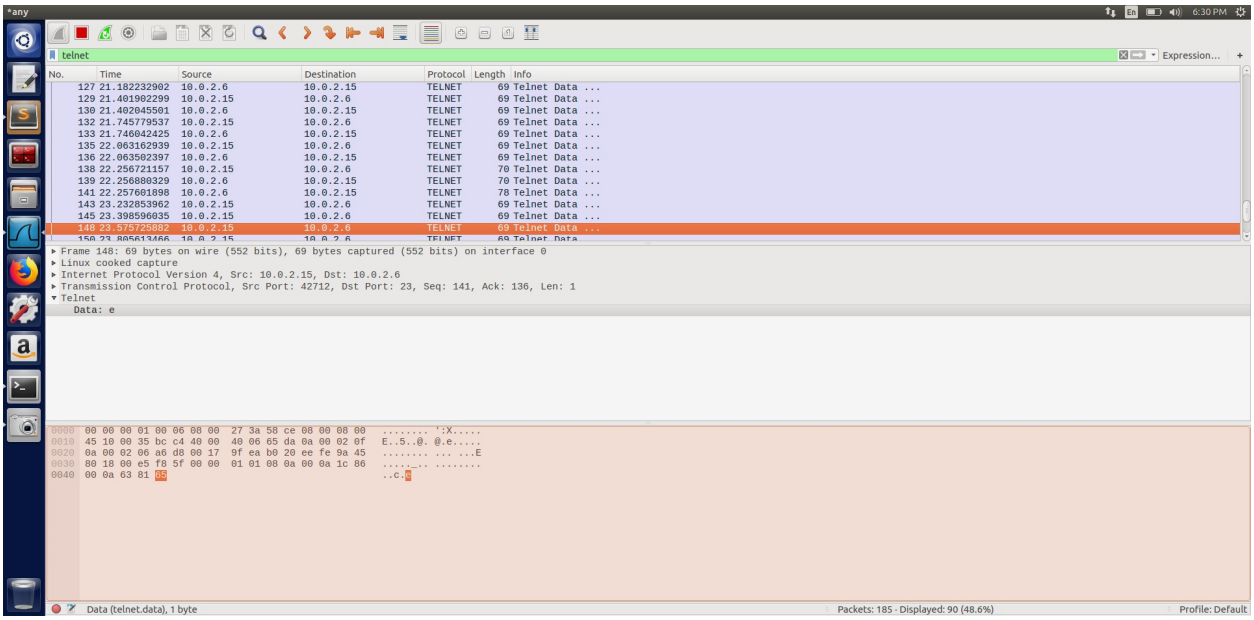
Task 2.1C: Sniffing Passwords

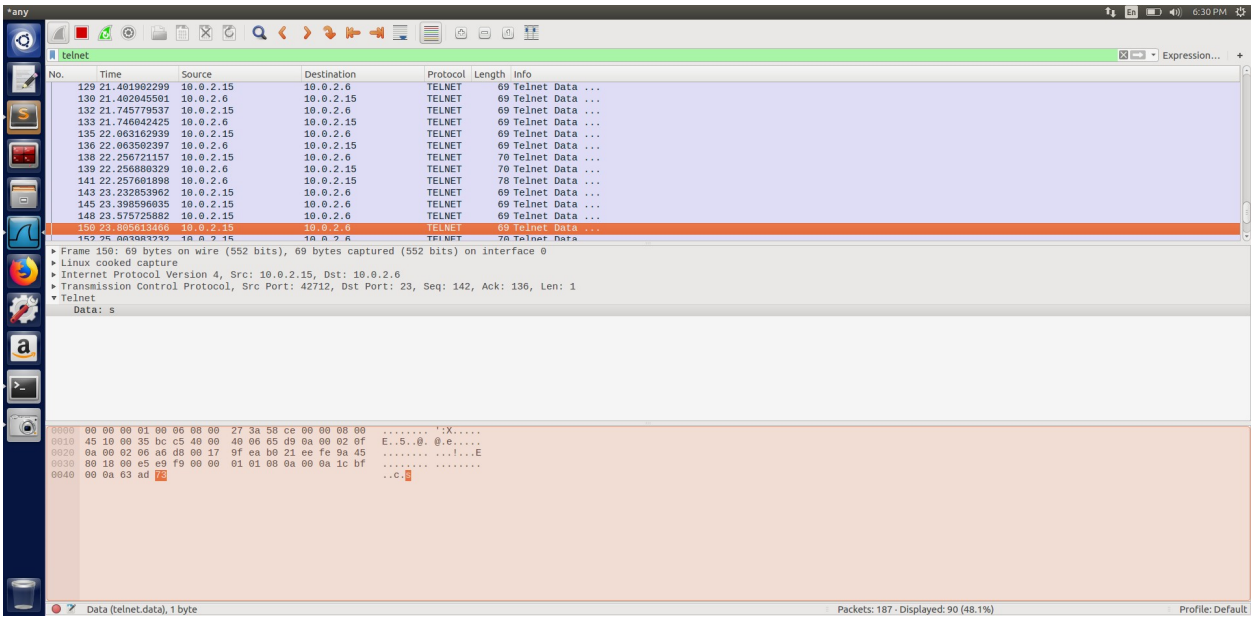



*any
6:28 PM
telnet
Expression...
No. Time Source Destination Protocol Length Info
125 21.181682999 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
127 21.182232902 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
129 21.401902299 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
130 21.402045501 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
132 21.745779537 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
133 21.746042425 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
135 22.063162939 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
136 22.063502397 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
138 22.256721157 10.0.2.15 10.0.2.6 TELNET 70 Telnet Data ...
139 22.256880329 10.0.2.6 10.0.2.15 TELNET 70 Telnet Data ...
141 22.257601898 10.0.2.6 10.0.2.15 TELNET 78 Telnet Data ...
143 23.232853962 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
145 23.398596035 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
Frame 141: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.2.6, Dst: 10.0.2.15
Transmission Control Protocol, Src Port: 23, Dst Port: 42712, Seq: 126, Ack: 139, Len: 10
Telnet
Data: Password:
0000 00 04 00 01 00 06 08 00 27 ae fd 1e 00 00 08 00 .... '.......
0010 45 10 00 3e c3 b1 40 00 40 06 5e e4 0a 00 02 06 E..>..@. @.^.....
0020 0a 00 02 0f 00 17 a6 d8 ee fe 9a 3b 9f ea b0 1e ....;....
0030 80 18 00 e3 18 45 00 00 01 01 08 0a 00 0a 62 63 .....E.. ......bc
0040 00 0a 1b 3c 50 61 73 73 77 6f 72 64 3a 20 ...<Pass word:
Data (telnet.data), 10 bytes
Packets: 171 · Displayed: 90 (52.6%)
Profile: Default


*any
6:28 PM
telnet
Expression...
No. Time Source Destination Protocol Length Info
125 21.181682999 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
127 21.182232902 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
129 21.401902299 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
130 21.402045501 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
132 21.745779537 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
133 21.746042425 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
135 22.063162939 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
136 22.063502397 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
138 22.256721157 10.0.2.15 10.0.2.6 TELNET 70 Telnet Data ...
139 22.256880329 10.0.2.6 10.0.2.15 TELNET 70 Telnet Data ...
141 22.257601898 10.0.2.6 10.0.2.15 TELNET 78 Telnet Data ...
143 23.232853962 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
145 23.398596035 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
Frame 143: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.6
Transmission Control Protocol, Src Port: 42712, Dst Port: 23, Seq: 139, Ack: 136, Len: 1
Telnet
Data: d
0000 00 00 00 01 00 06 08 00 27 3a 58 ce 00 00 08 00 .... ':X.....
0010 45 10 00 35 bc c2 40 00 40 06 65 dc 0a 00 02 0f E..5..@. @.e.....
0020 0a 00 02 06 a6 d8 00 17 9f ea b0 1e ee fe 9a 45 ....E
0030 80 18 00 e5 fa d5 00 00 01 01 08 0a 00 0a 1c 30 0
0040 00 0a 62 63 64 ..bcd
Data (telnet.data), 1 byte
Packets: 174 · Displayed: 90 (51.7%)
Profile: Default

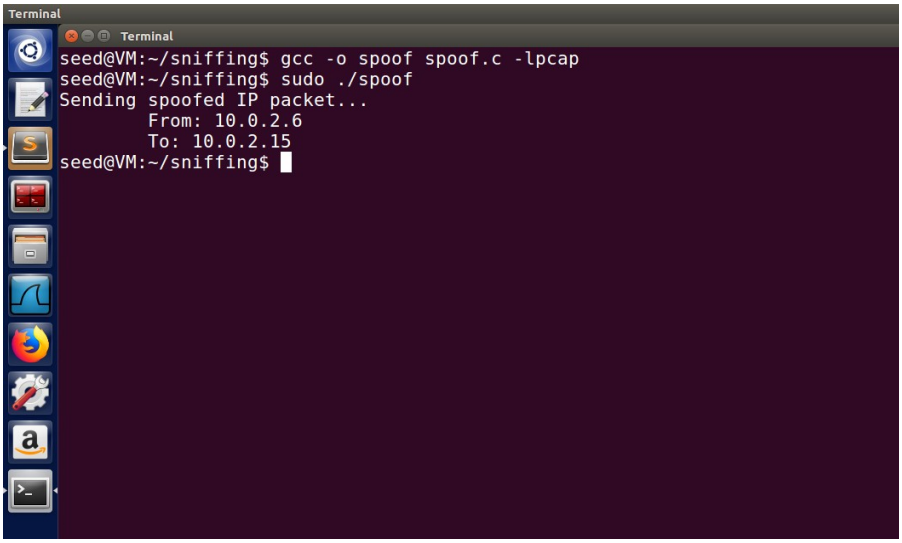

*any
6:28 PM
telnet
Expression...
No. Time Source Destination Protocol Length Info
125 21.181682999 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
127 21.182232902 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
129 21.401902299 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
130 21.402045501 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
132 21.745779537 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
133 21.746042425 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
135 22.063162939 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
136 22.063502397 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
138 22.256721157 10.0.2.15 10.0.2.6 TELNET 70 Telnet Data ...
139 22.256880329 10.0.2.6 10.0.2.15 TELNET 70 Telnet Data ...
141 22.257601898 10.0.2.6 10.0.2.15 TELNET 78 Telnet Data ...
143 23.232853962 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
145 23.398596035 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
Frame 145: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.6
Transmission Control Protocol, Src Port: 42712, Dst Port: 23, Seq: 140, Ack: 136, Len: 1
Telnet
Data: e
0000 00 00 00 01 00 06 08 00 27 3a 58 ce 00 06 08 00 .... ':X.....
0010 45 10 00 35 bc c3 40 00 40 06 65 db 0a 00 02 0f E..5..@. @.e.....
0020 0a 00 02 06 a6 d8 00 17 9f ea b0 1f ee fe 9a 45 ....E
0030 80 18 00 e5 f8 ac 00 00 01 01 08 0a 00 0a 1c 59 ....Y
0040 00 0a 63 62 65 ..cbe
Data (telnet.data), 1 byte
Packets: 179 · Displayed: 90 (50.3%)
Profile: Default

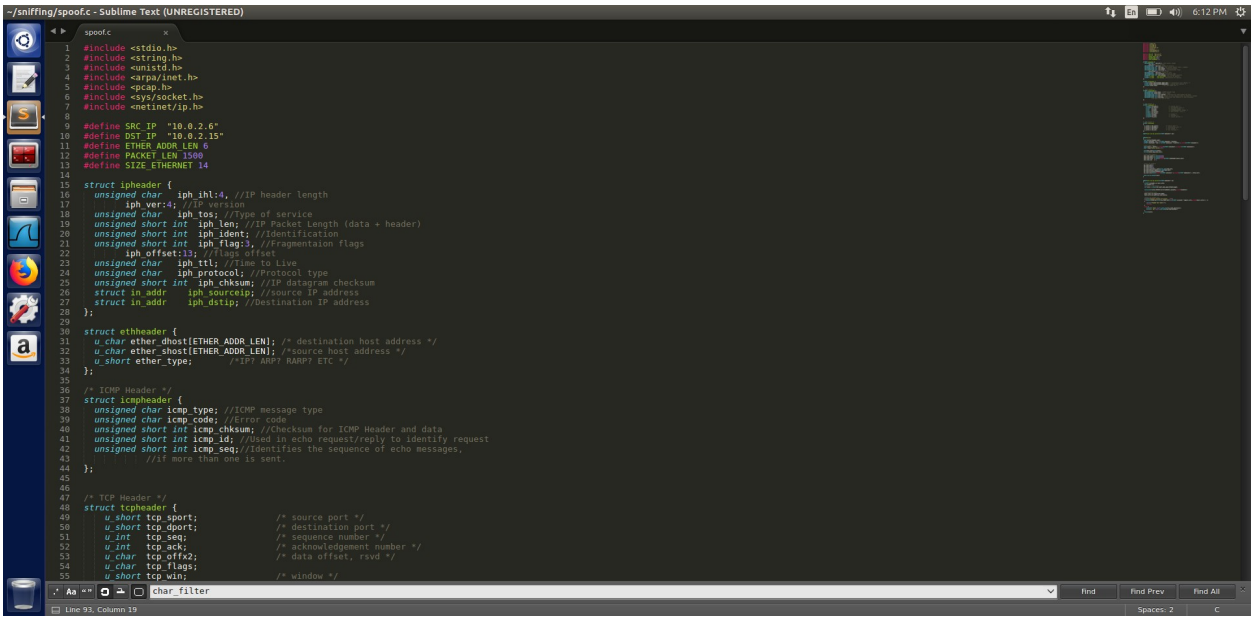

*any
6:30 PM
telnet
Expression...
No. Time Source Destination Protocol Length Info
127 21.182232902 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
129 21.401902299 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
130 21.402045501 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
132 21.745779537 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
133 21.746042425 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
135 22.063162939 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
136 22.063502397 10.0.2.6 10.0.2.15 TELNET 69 Telnet Data ...
138 22.256721157 10.0.2.15 10.0.2.6 TELNET 70 Telnet Data ...
139 22.256880329 10.0.2.6 10.0.2.15 TELNET 70 Telnet Data ...
141 22.257601898 10.0.2.6 10.0.2.15 TELNET 78 Telnet Data ...
143 23.232853962 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
145 23.398596035 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
148 23.575725882 10.0.2.15 10.0.2.6 TELNET 69 Telnet Data ...
Frame 148: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 10.0.2.6
Transmission Control Protocol, Src Port: 42712, Dst Port: 23, Seq: 141, Ack: 136, Len: 1
Telnet
Data: e
0000 00 00 00 01 00 06 08 00 27 3a 58 ce 08 00 08 00 .... ':X.....
0010 45 10 00 35 bc c4 40 00 40 06 65 da 0a 00 02 0f E..5..@. @.e.....
0020 0a 00 02 06 a6 d8 00 17 9f ea b0 20 ee fe 9a 45 ....E
0030 80 18 00 e5 f8 5f 00 00 01 01 08 0a 00 0a 1c 86 ....._.. ....
0040 00 0a 63 81 65 ..c.e
Data (telnet.data), 1 byte
Packets: 185 · Displayed: 90 (48.6%)
Profile: Default

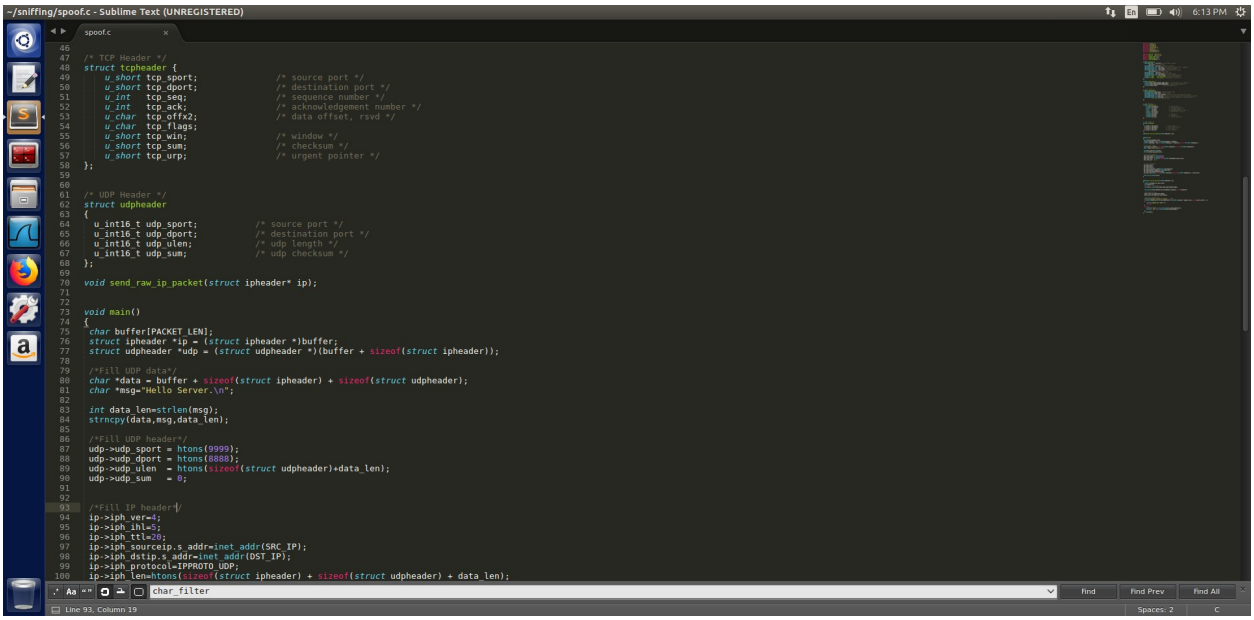


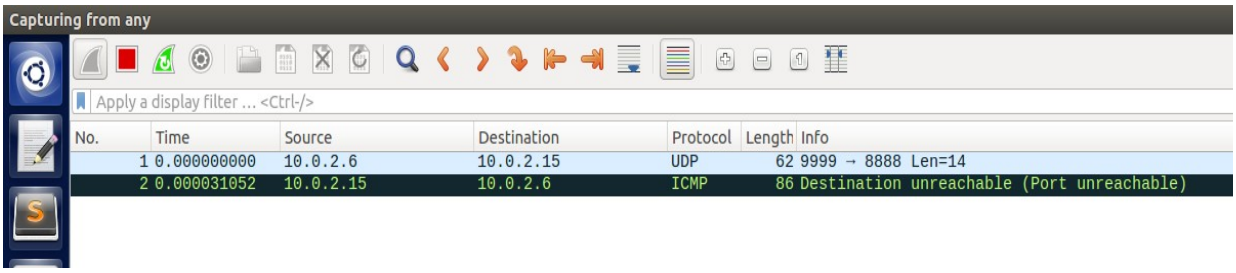
As seen from the above screenshots we are using telnet to connect to 10.0.2.6 and when it asks for username and password we can see from the other VM that we are able to get the password character by character.

Task 2.2A



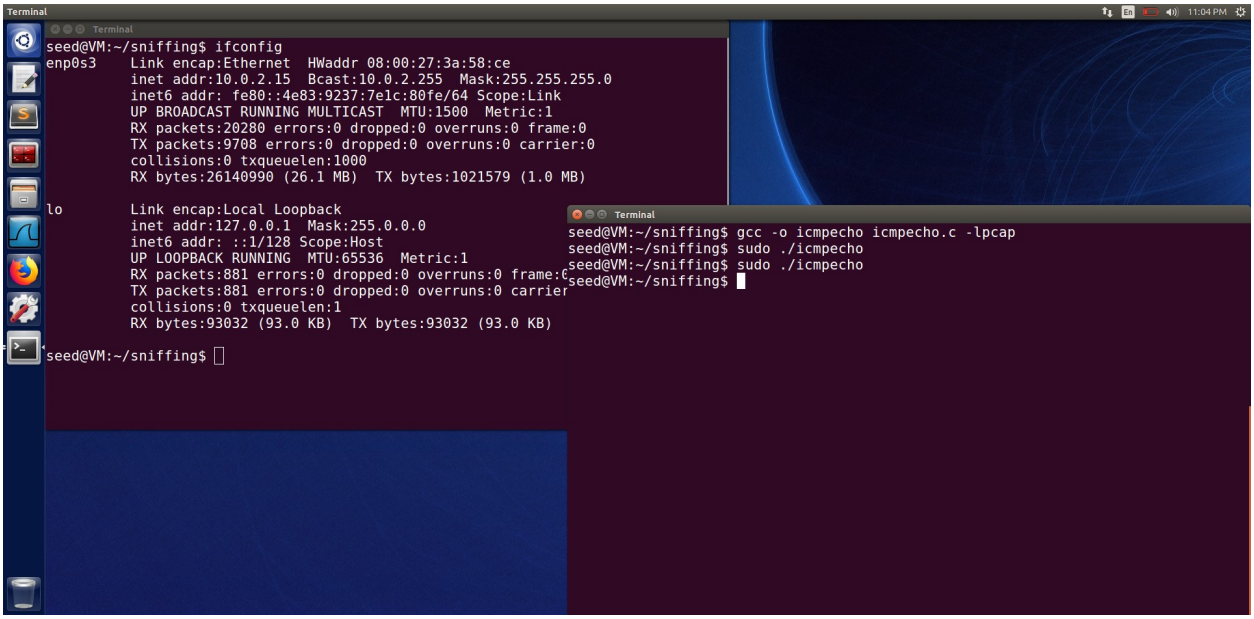


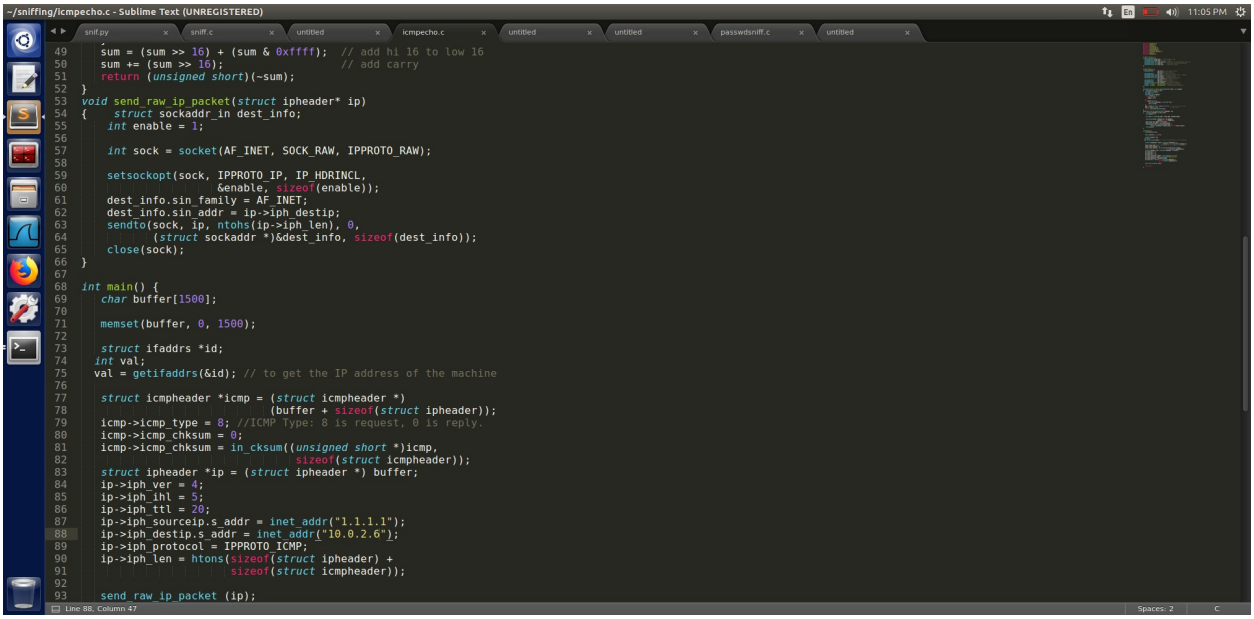


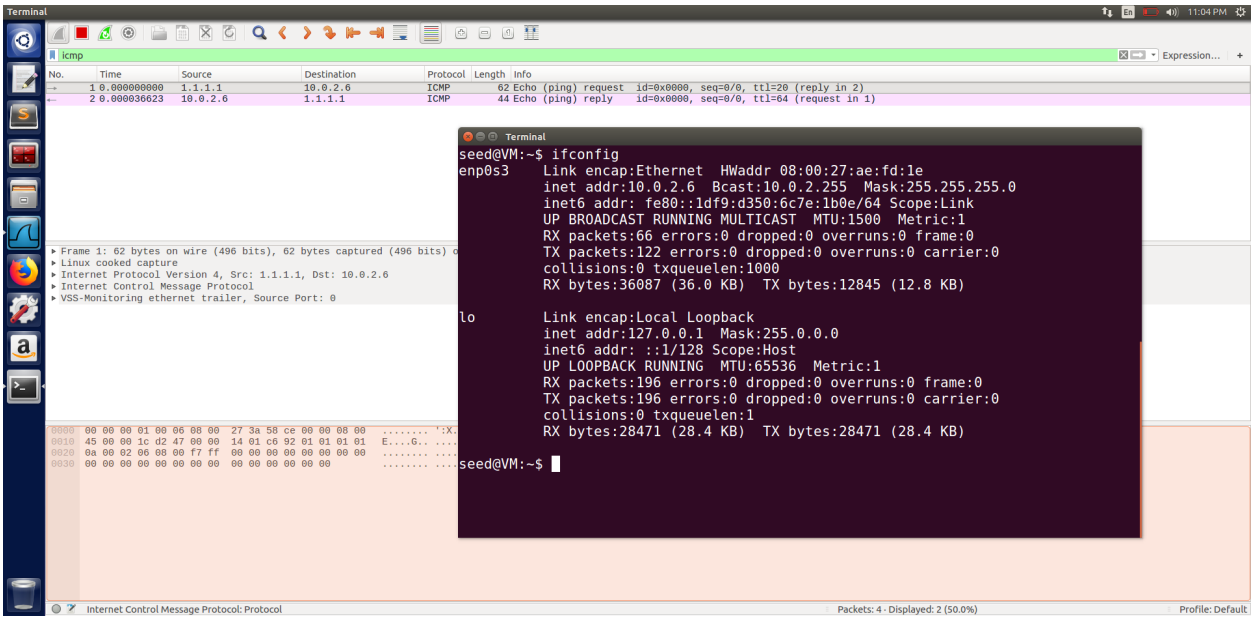


From the above screenshot we can see that we are able to send spoofed packets from 10.0.2.6 to 10.0.2.15

Task 2.2 B


Terminal
seed@VM:~/sniffing$ ifconfig
enp0s3 Link encap:Ethernet HWaddr 08:00:27:3a:58:ce
inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
inet6 addr: fe80::4e83:9237:7e1c:80fe/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:20280 errors:0 dropped:0 overruns:0 frame:0
TX packets:9708 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:26140990 (26.1 MB) TX bytes:1021579 (1.0 MB)
lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:881 errors:0 dropped:0 overruns:0 frame:0
TX packets:881 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:93032 (93.0 KB) TX bytes:93032 (93.0 KB)
seed@VM:~/sniffing$
Terminal
seed@VM:~/sniffing$ gcc -o icmpecho icmpecho.c -lpcap
seed@VM:~/sniffing$ sudo ./icmpecho
seed@VM:~/sniffing$ sudo ./icmpecho
seed@VM:~/sniffing$


~/sniffing/icmpecho.c - Sublime Text (UNREGISTERED)
sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
sum += (sum >> 16); // add carry
return (unsigned short)(~sum);
}
void send_raw_ip_packet(struct ipheader* ip)
{ struct sockaddr_in dest_info;
int enable = 1;
int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
&enable, sizeof(enable));
dest_info.sin_family = AF_INET;
dest_info.sin_addr = ip->iph_destip;
sendto(sock, ip, ntohs(ip->iph_len), 0,
(struct sockaddr *)&dest_info, sizeof(dest_info));
close(sock);
}
int main() {
char buffer[1500];
memset(buffer, 0, 1500);
struct ifaddrs *id;
int val;
val = getifaddrs(&id); // to get the IP address of the machine
struct icmpheader *icmp = (struct icmpheader *)
(buffer + sizeof(struct ipheader));
icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
sizeof(struct icmpheader));
struct ipheader *ip = (struct ipheader *) buffer;
ip->iph_ver = 4;
ip->iph_ihl = 5;
ip->iph_ttl = 20;
ip->iph_sourceip.s_addr = inet_addr("1.1.1.1");
ip->iph_destip.s_addr = inet_addr("10.0.2.6");
ip->iph_protocol = IPPROTO_ICMP;
ip->iph_len = htons(sizeof(struct ipheader) +
sizeof(struct icmpheader));
send_raw_ip_packet (ip);



Observation :As seen from the above screenshot we are able to send spoofed ICMP echo request to 1.1.1.1 and also get ICMP reply

Explanation : We send ICMP echo from 10.0.2.15 and the ICMP echo reply is sent to 10.0.2.6 from where it assumes the request is made. We can see that on Wireshark as seen from the screenshot above

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

If we set the length to any arbitrary value then the IP packet will not be formed properly and when we will send the packet then the size will be truncated because it being too large

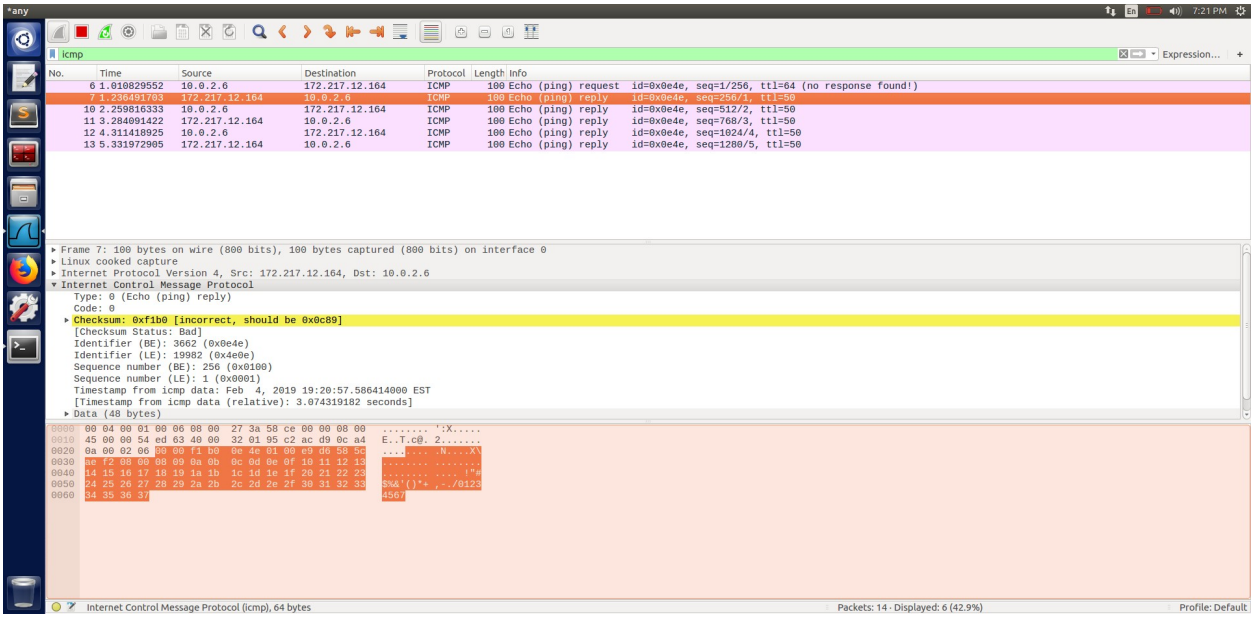
Question 5 Using the raw socket programming, do you have to calculate the checksum for the IP header?

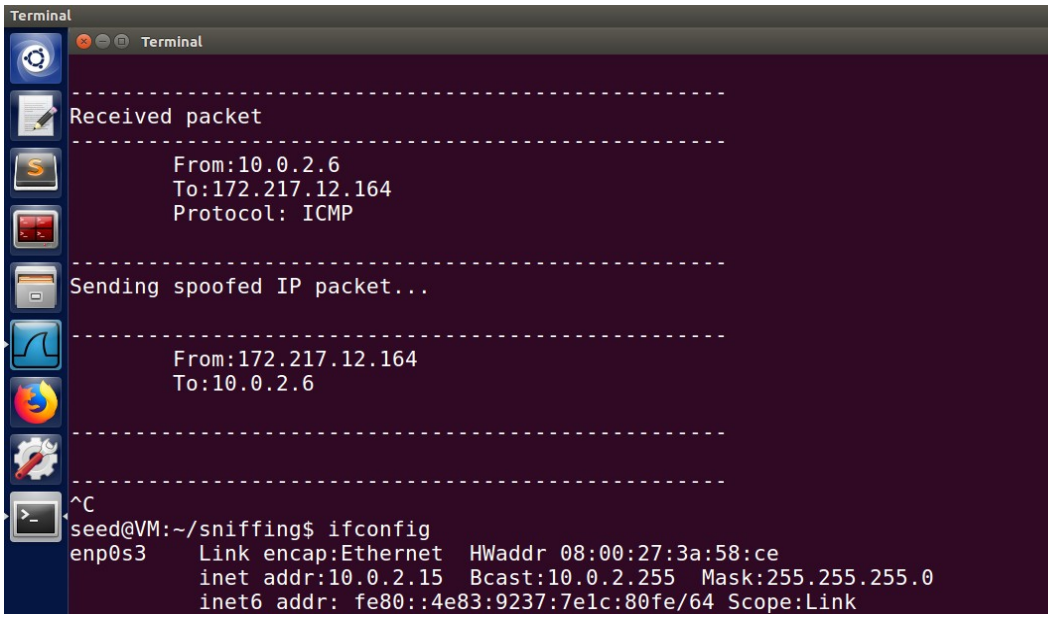
OS calculates the checksum before sending the packet therefore, we don’t need to calculate the checksum

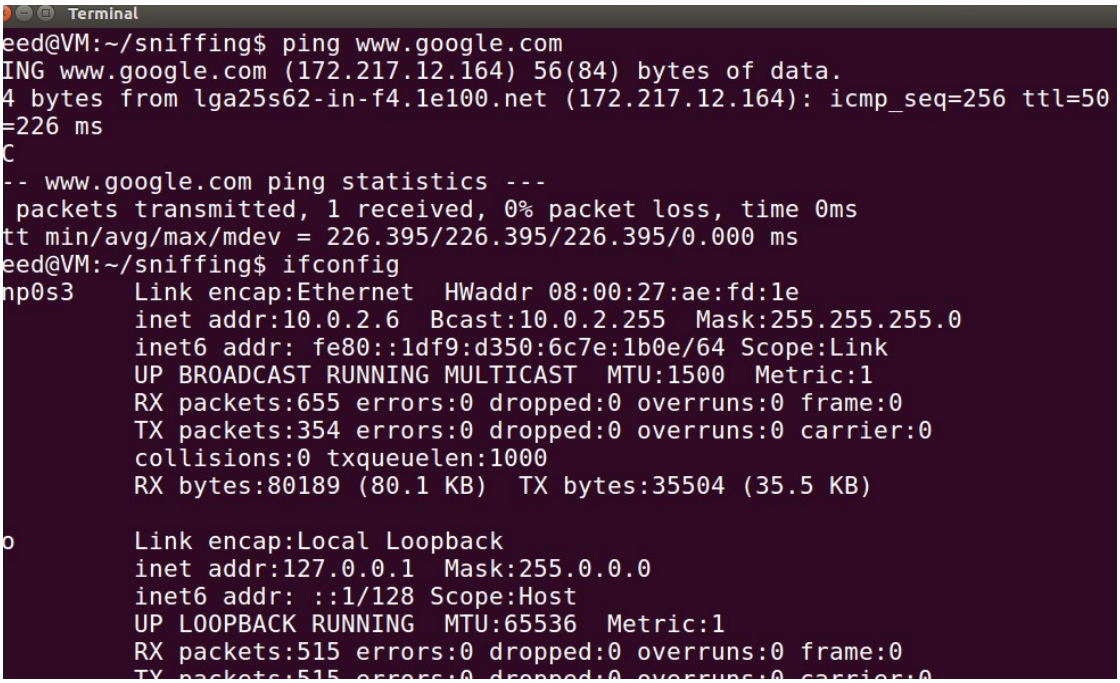
Question 6 Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

If we do not run the program using the root privileges then it will show an error because the program will need to access the NIC and when we use raw socket it gives the user root privileges.

Task 2.3


*any
7:21 PM
icmp
Expression...
No. Time Source Destination Protocol Length Info
6 1.010829552 10.0.2.6 172.217.12.164 ICMP 100 Echo (ping) request id=0x0e4e, seq=1/256, ttl=64 (no response found!)
7 1.236491703 172.217.12.164 10.0.2.6 ICMP 100 Echo (ping) reply id=0x0e4e, seq=256/1, ttl=50
10 2.259816333 10.0.2.6 172.217.12.164 ICMP 100 Echo (ping) request id=0x0e4e, seq=512/2, ttl=64
11 3.284091422 172.217.12.164 10.0.2.6 ICMP 100 Echo (ping) reply id=0x0e4e, seq=768/3, ttl=50
12 4.311418925 10.0.2.6 172.217.12.164 ICMP 100 Echo (ping) request id=0x0e4e, seq=1024/4, ttl=64
13 5.331972905 172.217.12.164 10.0.2.6 ICMP 100 Echo (ping) reply id=0x0e4e, seq=1280/5, ttl=50
Frame 7: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 172.217.12.164, Dst: 10.0.2.6
Internet Control Message Protocol
Type: 0 (Echo (ping) reply)
Code: 0
Checksum: 0xf1b0 [incorrect, should be 0x0c89]
[Checksum Status: Bad]
Identifier (BE): 3662 (0x0e4e)
Identifier (LE): 19982 (0x4e0e)
Sequence number (BE): 256 (0x0100)
Sequence number (LE): 1 (0x0001)
Timestamp from icmp data: Feb 4, 2019 19:20:57.586414000 EST
[Timestamp from icmp data (relative): 3.074319182 seconds]
Data (48 bytes)
0000 00 04 00 01 00 06 08 00 27 3a 58 ce 00 00 08 00 .... ':X.....
0010 45 00 00 54 ed 63 40 00 32 01 95 c2 ac d9 0c a4 E..T.c@. 2.......
0020 0a 00 02 06 00 00 f1 b0 0e 4e 01 00 e9 d6 58 5c .....N....X\
0030 ae f2 08 00 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 ....
0040 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 .... !"#
0050 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 $%&'()*+ ,-./0123
0060 34 35 36 37 4567
Internet Control Message Protocol (icmp), 64 bytes
Packets: 14 · Displayed: 6 (42.9%)
Profile: Default


Terminal
Terminal
Received packet
From:10.0.2.6
To:172.217.12.164
Protocol: ICMP
Sending spoofed IP packet...
From:172.217.12.164
To:10.0.2.6
^C
seed@VM:~/sniffing$ ifconfig
enp0s3 Link encap:Ethernet HWaddr 08:00:27:3a:58:ce
inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0
inet6 addr: fe80::4e83:9237:7e1c:80fe/64 Scope:Link



We can see from the above screenshot that we are able to sniff and spoof i.e., we are able to spoof ICMP echo reply.