

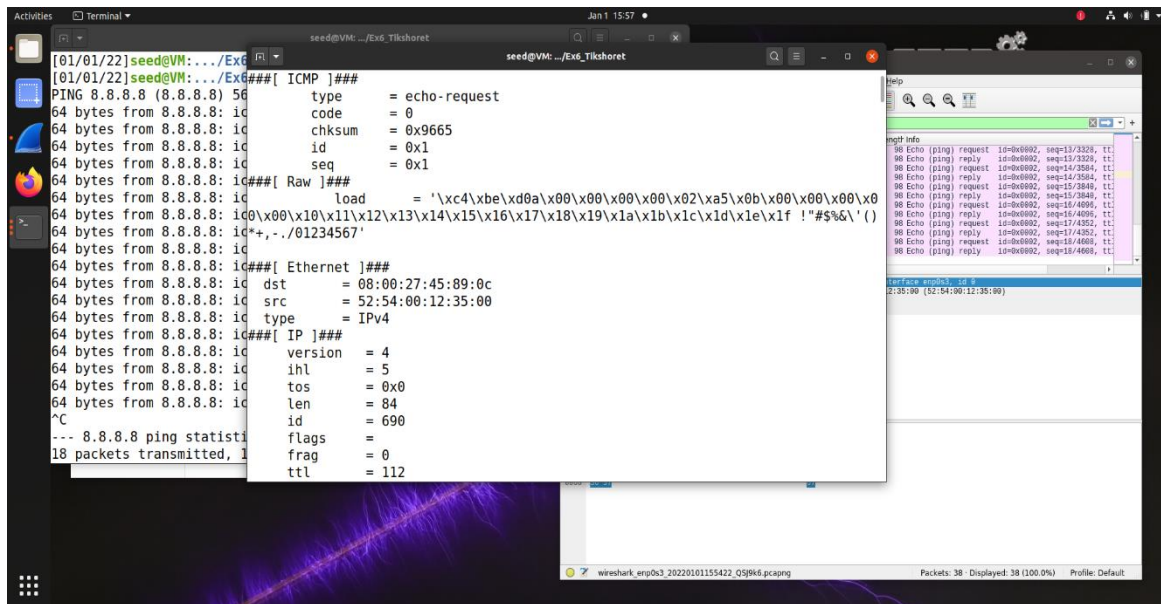
Eden roas 318356599 Noam David 319073235

Task 1.1- Sniffing Packets

We wrote a short plan, we used the scapy library and show() function in order for all the information in the packet to be printed.

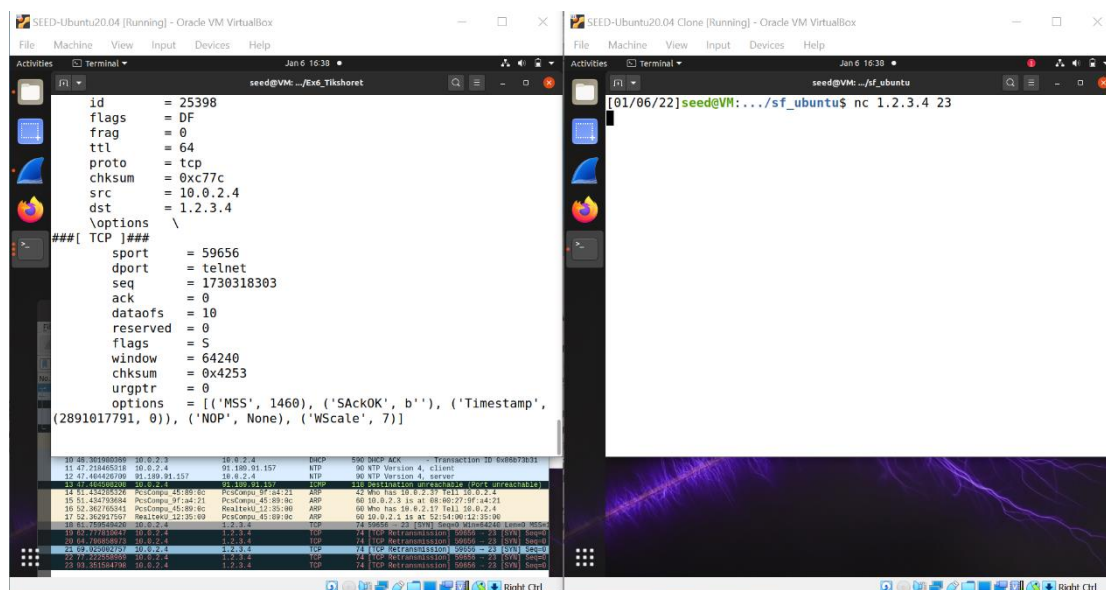
Note-the code was already in the question.

results -

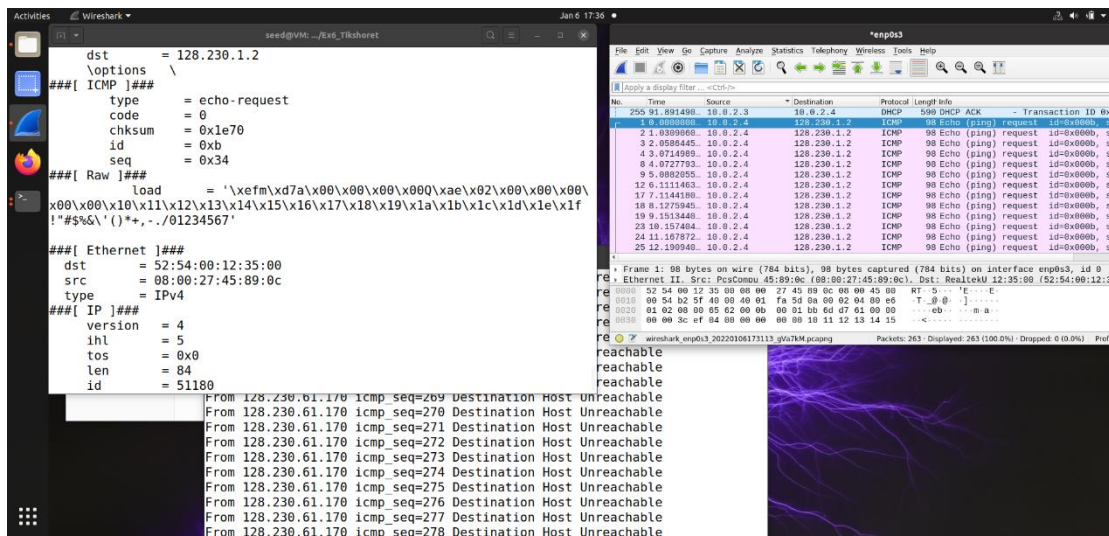


b.1-

we edit the main.py program to use tcp and src host 10.0.2.4 and dst port 23' to filter for only tcp packets coming from host 10.0.2.4 and heading to any IP's port 23. Packets were captured, This means that the filter worked and only TCP packets from 10.0.2.4 being sent to port 23 are captured; the rest were ignored.

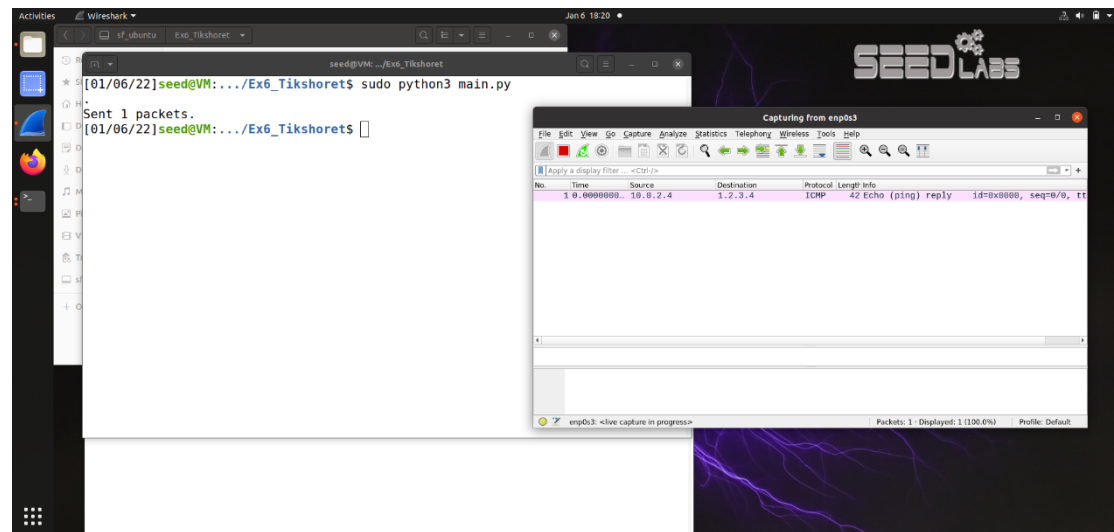


The last filter I need to implement is one that will only capture packets coming from or going to a particular subnet. I will choose 128.115.0.0/16 as the subnet.



Task 1.2-Spoofing ICMP Packets-

We sent packet from IP 1.2.3.4 to our IP-10.0.2.4

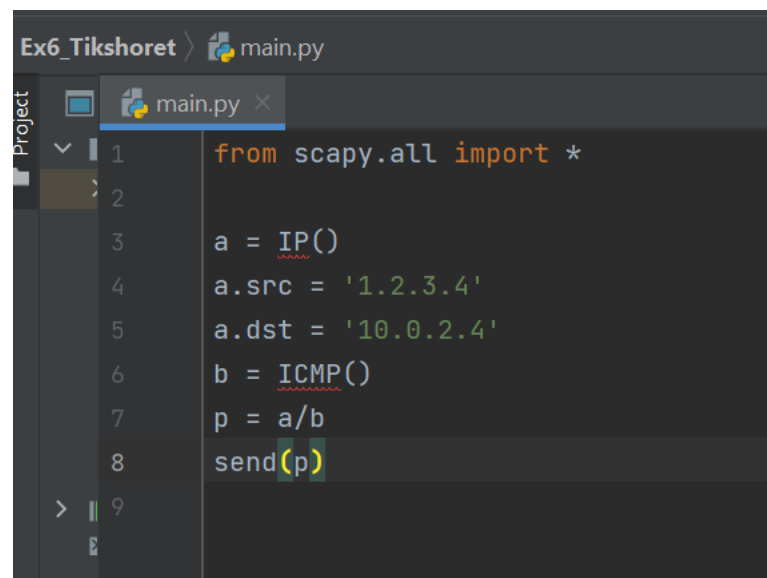


The code:

Line 3- creates an IP object from the IP class.

Line 3 shows how to set the destination IP address field. If a field is not set, a default value will be used.

Line 6 creates an ICMP object. The default type is echo request. In Line, we stack a and b together to form a new object. We can now send out this packet using send() in Line 8.



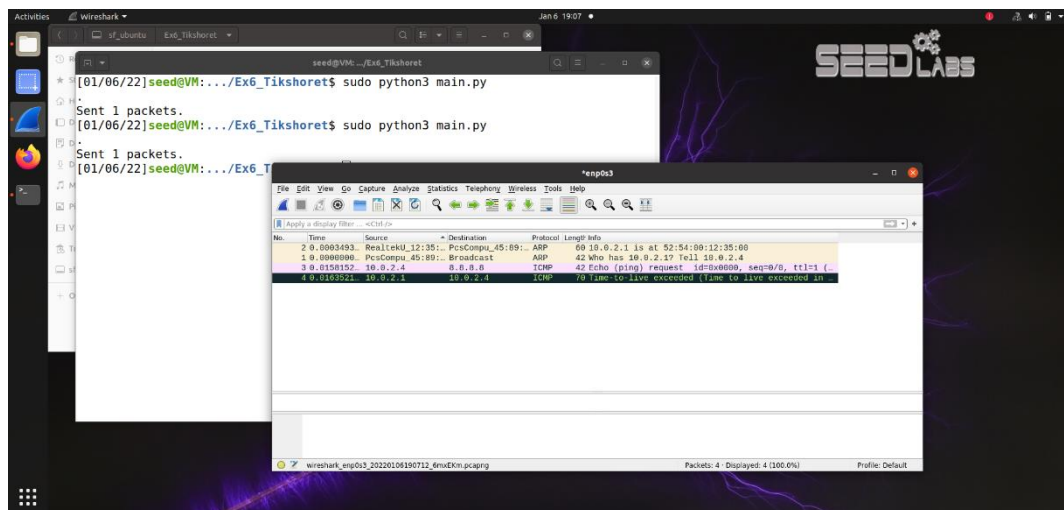
Task 1.3: Traceroute

The goal of this task is to create a version of traceroute using Scapy. The program needs to repeatedly send out packets (I will use ICMP packets) with Time-To-Live (TTL) value starting at 1.

The code-

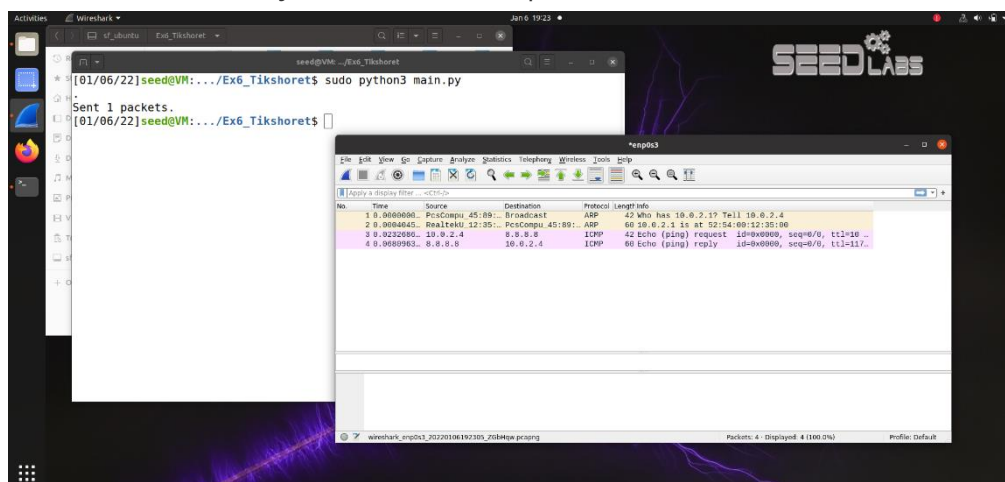
```
main.py x
from scapy.all import *

a = IP()
a.dst = '8.8.8.8'
a.ttl = 1
b = ICMP()
send(a/b)
```



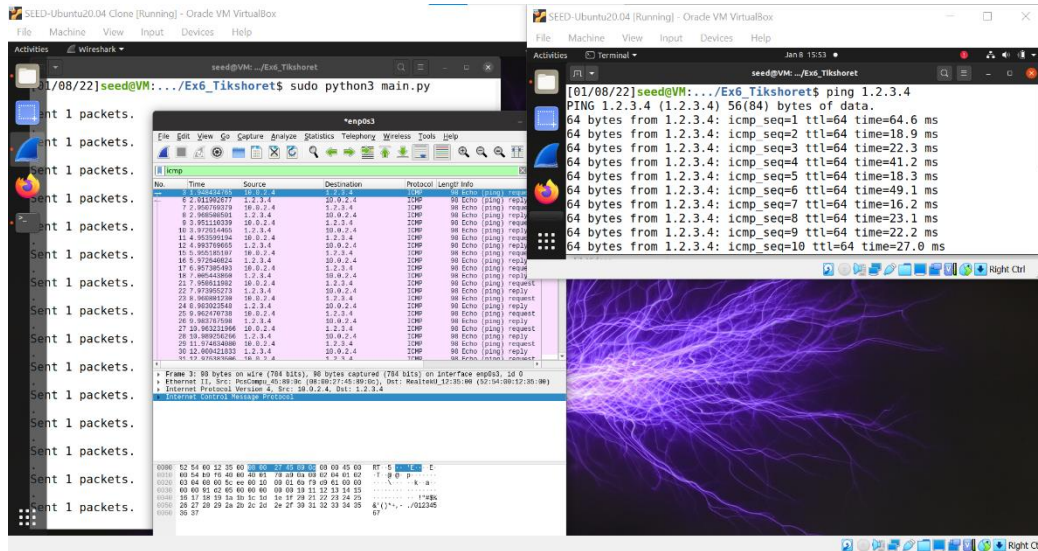
We keep incrementing the TTL value of our packet by 1 and resend it 10 times, until it finally reaches the destination. We will know it successfully reached its destination because the ICMP packet sent back will have a type of 'Echo Reply'. The source IP address of the packet sent back will also match that of the IP address we set as our packet's source IP address.

For more information- you can see the full process at attachments files.



Task 1.4: Sniffing and-then Spoofing

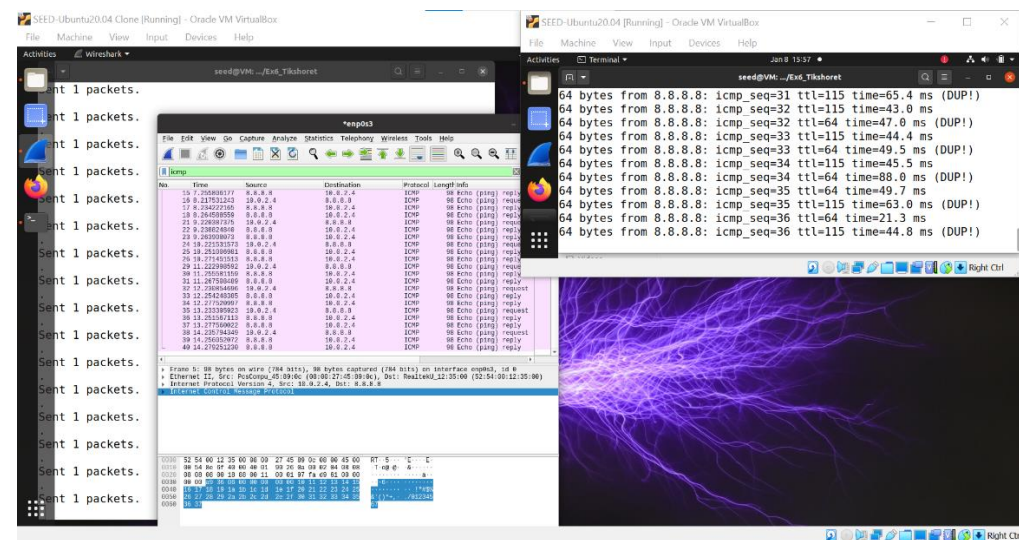
Ping 1.2.3.4-



The screenshot shows a virtual machine window titled "SEED-Ubuntu20.04 Clone [Running] - Oracle VM VirtualBox". The terminal window displays the command `ping 1.2.3.4` and its output. The output shows that the ping command was successful, with a response time of 64.6 ms. The terminal output is as follows:

```
[01/08/22]seed@VM:~/Ex6_Tikshoret$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4): 56(84) bytes of data:
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=64.6 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=18.9 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=22.3 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=41.2 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=18.3 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=49.1 ms
64 bytes from 1.2.3.4: icmp_seq=7 ttl=64 time=16.2 ms
64 bytes from 1.2.3.4: icmp_seq=8 ttl=64 time=23.1 ms
64 bytes from 1.2.3.4: icmp_seq=9 ttl=64 time=22.2 ms
64 bytes from 1.2.3.4: icmp_seq=10 ttl=64 time=27.0 ms
```

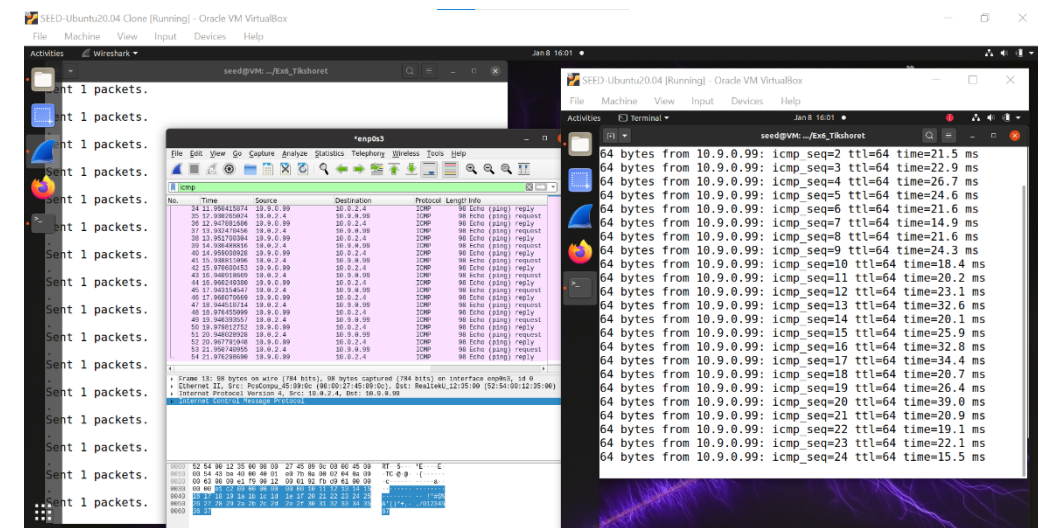
Ping 8.8.8.8-



The screenshot shows a virtual machine window titled "SEED-Ubuntu20.04 Clone [Running] - Oracle VM VirtualBox". The terminal window displays the command `ping 8.8.8.8` and its output. The output shows that the ping command was successful, with a response time of 65.4 ms. The terminal output is as follows:

```
64 bytes from 8.8.8.8: icmp_seq=31 ttl=115 time=65.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=32 ttl=115 time=43.0 ms
64 bytes from 8.8.8.8: icmp_seq=33 ttl=64 time=47.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=33 ttl=115 time=44.4 ms
64 bytes from 8.8.8.8: icmp_seq=33 ttl=64 time=49.5 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=34 ttl=115 time=45.5 ms
64 bytes from 8.8.8.8: icmp_seq=34 ttl=64 time=88.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=35 ttl=64 time=49.7 ms
64 bytes from 8.8.8.8: icmp_seq=35 ttl=115 time=63.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=36 ttl=64 time=21.3 ms
64 bytes from 8.8.8.8: icmp_seq=36 ttl=115 time=44.8 ms (DUP!)
```

Ping 10.9.0.99-

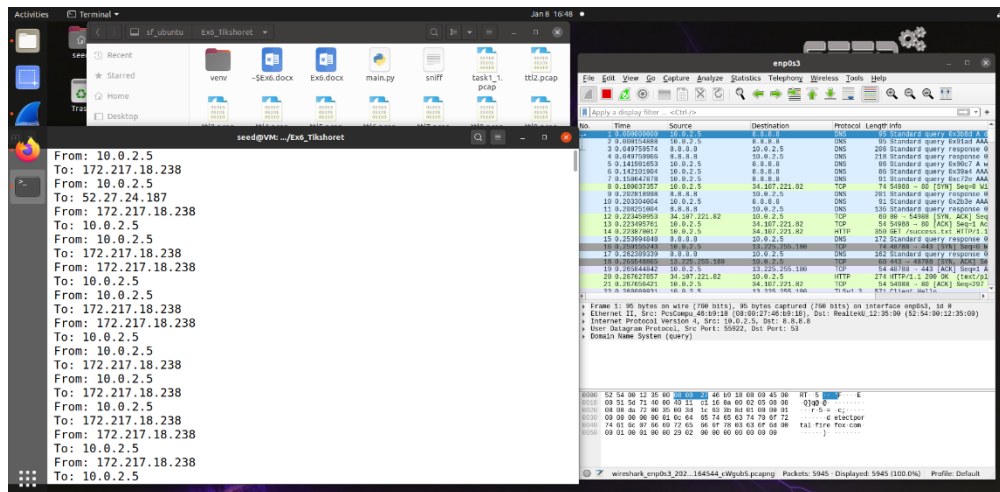


The screenshot shows a virtual machine window titled "SEED-Ubuntu20.04 Clone [Running] - Oracle VM VirtualBox". The terminal window displays the command `ping 10.9.0.99` and its output. The output shows that the ping command was successful, with a response time of 21.5 ms. The terminal output is as follows:

```
64 bytes from 10.9.0.99: icmp_seq=2 ttl=64 time=21.5 ms
64 bytes from 10.9.0.99: icmp_seq=3 ttl=64 time=22.9 ms
64 bytes from 10.9.0.99: icmp_seq=4 ttl=64 time=26.7 ms
64 bytes from 10.9.0.99: icmp_seq=5 ttl=64 time=24.6 ms
64 bytes from 10.9.0.99: icmp_seq=6 ttl=64 time=21.6 ms
64 bytes from 10.9.0.99: icmp_seq=7 ttl=64 time=14.9 ms
64 bytes from 10.9.0.99: icmp_seq=8 ttl=64 time=21.6 ms
64 bytes from 10.9.0.99: icmp_seq=9 ttl=64 time=24.3 ms
64 bytes from 10.9.0.99: icmp_seq=10 ttl=64 time=18.4 ms
64 bytes from 10.9.0.99: icmp_seq=11 ttl=64 time=20.2 ms
64 bytes from 10.9.0.99: icmp_seq=12 ttl=64 time=23.1 ms
64 bytes from 10.9.0.99: icmp_seq=13 ttl=64 time=32.6 ms
64 bytes from 10.9.0.99: icmp_seq=14 ttl=64 time=20.1 ms
64 bytes from 10.9.0.99: icmp_seq=15 ttl=64 time=25.9 ms
64 bytes from 10.9.0.99: icmp_seq=16 ttl=64 time=32.8 ms
64 bytes from 10.9.0.99: icmp_seq=17 ttl=64 time=34.4 ms
64 bytes from 10.9.0.99: icmp_seq=18 ttl=64 time=20.7 ms
64 bytes from 10.9.0.99: icmp_seq=19 ttl=64 time=26.4 ms
64 bytes from 10.9.0.99: icmp_seq=20 ttl=64 time=39.0 ms
64 bytes from 10.9.0.99: icmp_seq=21 ttl=64 time=20.9 ms
64 bytes from 10.9.0.99: icmp_seq=22 ttl=64 time=19.1 ms
64 bytes from 10.9.0.99: icmp_seq=23 ttl=64 time=22.1 ms
64 bytes from 10.9.0.99: icmp_seq=24 ttl=64 time=15.5 ms
```

Task 2: Writing Programs to Sniff and Spoof Packets-

Task 2.1A: Understanding How a Sniffer Works



Question-1

pcap_open_live - this function used to open a listener socket to the network

pcap_compile and Pcap_setfilter - these functions used for making a filter so when we capture packets, we are not capturing all of them, we capture only the packets that we define on the filter like ICMP packets and more.

pcap_loop – this function used for capturing the packets the goes on the network after we define the filter we want.

pcap_close – this function closes the listener socket we opened.

Question 2-

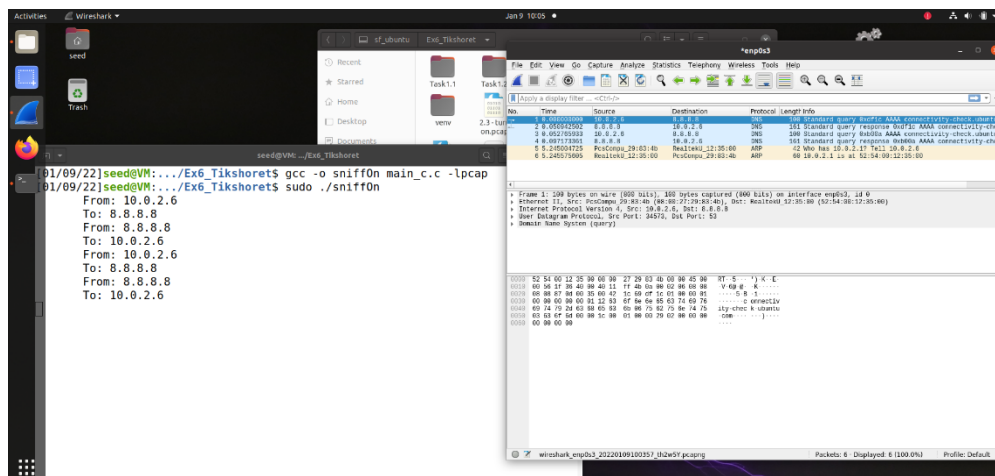
because we are using raw socket the OS is blocking the access to the raw socket we are using to build the packets. The program fails at capturing the packets that goes on the network.

Question 3-

when promiscuous mode is off we can see that the program sniffing packets that my IP was the source (including when someone is answering back to me and then my IP isn't the source IP) at the other hand when promiscuous is on we can see that we capturing packet that going from different computers.

Here some screen shots to show that:

Promiscuous mode on:

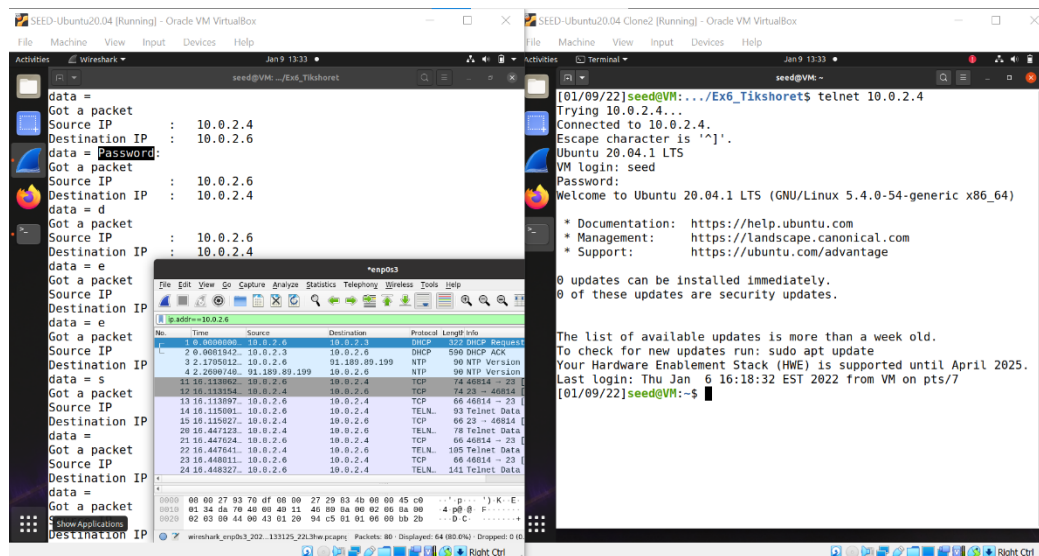


The screenshot displays a Kali Linux desktop environment. In the foreground, a terminal window titled 'seed@VM: .../Ex6_Tikshoret' shows the execution of a C program. The program, named 'main.c', is compiled with 'gcc -o sniffOff main.c -lpcap' and then run with 'sudo ./sniffOff'. The output of the program lists several IP addresses: 10.0.2.6, 224.0.0.251, and 8.8.8.8, each appearing twice. In the background, the Wireshark network traffic analyzer is open, showing a packet capture on the 'eth0' interface. The packet list pane displays two packets, both of type 'DNS'. The first packet is a 'Standard query' from 10.0.2.6 to 224.0.0.251. The second packet is a 'Standard query response' from 224.0.0.251 to 10.0.2.6. The packet details pane shows the structure of the DNS packet, including the 'Standard query' section. The status bar at the bottom of Wireshark indicates 'Packets: 2 (100.0%)' and 'Profile: Default'.

[illegible]

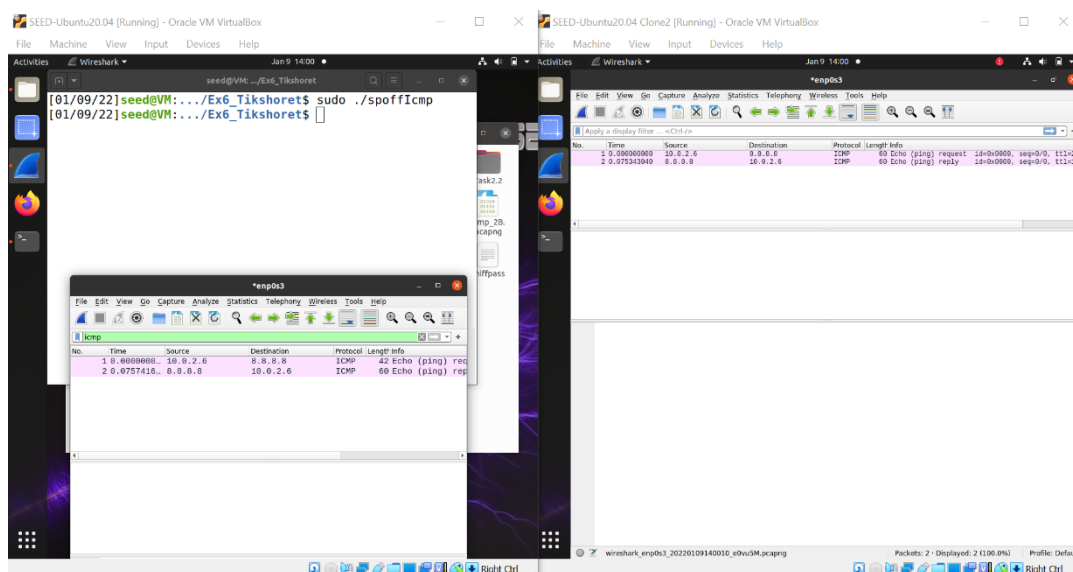
The screenshot displays a Kali Linux desktop environment. On the left, a terminal window shows the execution of a netmap capture using the command `tcpSniff -i eth0 -p 80 -s 10.0.2.15 -d 10.0.2.15`. The output shows a list of captured packets, including a SYN packet from 10.0.2.15 to 10.0.2.15 on port 80. On the right, the Wireshark network protocol analyzer is open, showing the details of the selected packet (No. 1). The packet list shows a SYN packet from 10.0.2.15 to 10.0.2.15 on port 80. The packet details pane shows the Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol (TCP) layers. The TCP layer shows a SYN packet with sequence number 10.0.2.15 and destination port 80. The packet bytes pane shows the raw packet data in hexadecimal and ASCII.

Task 2.1C-Sniffing Passwords-



Task 2.2- Spoofing

Task 2.2A+2.2B- Spoof an ICMP Echo Request:



Question 4-

yes, the IP packet length can be any arbitrary value, although the packet's total length is overwritten to its original size when its sent.

Question5 -

we don't have, the reason is that we can tell the kernal to calculate the checksum for the IP header. In IP header fields we have the parameter "ip_check", if we do "ip_check = 0" that will let the kernal to do the checksum by default unless we change it to different value but then we will have to use a checksum method.

Question 6-

we must have the root privileges and they are necessary to run the program that make use of raw socket. If you use non-privileges user you will not have the permissions to change all the fields in the protocol headers. The root privileges will give the ability to change and set any fields in the protocol headers and access to the socket and put the interface card in promiscuous mode. If we run the program without the root privileges, it will fail at socket setup.

Task 2.3: Sniff and then Spoof-

