Communications and Computing – Final Project:

Alon Firestein – 314984402
Hananel Lazar – 312495161
Due Date: 08/03/21

Introduction:

In this document, we have attached numerous explanations for each task in the project, including screenshots and also for particular tasks also highlighted certain details in the screenshots to make it easier to see the successful results.

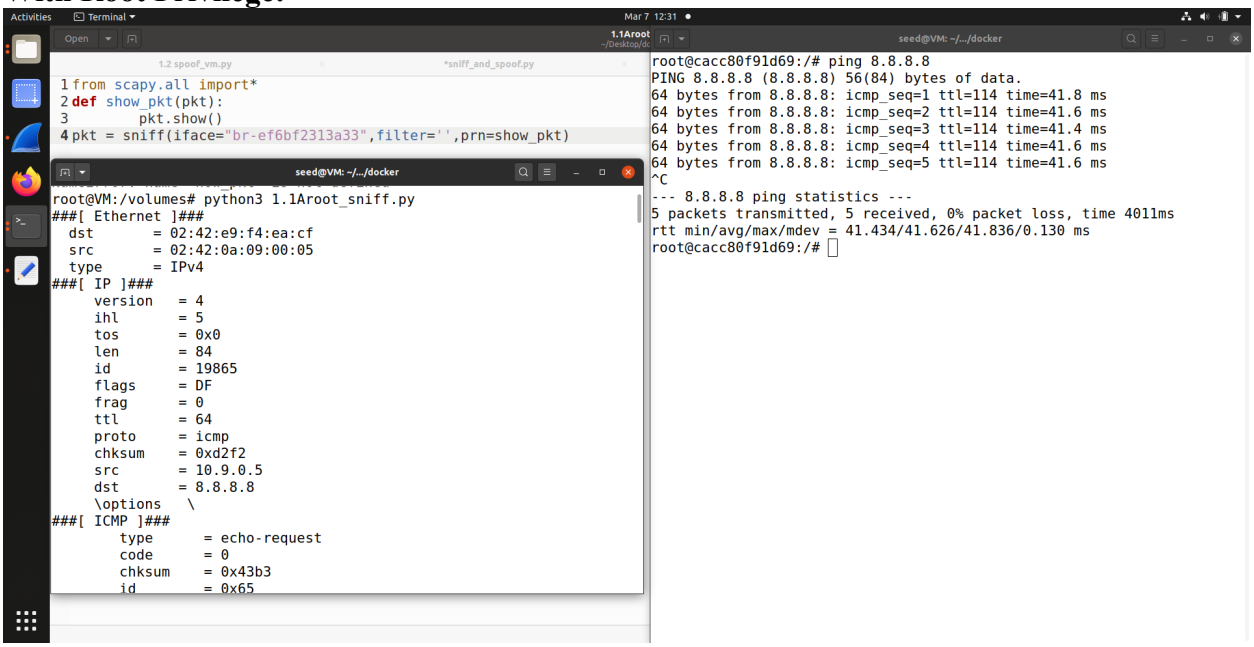
In addition, we have not screenshotted the Wireshark result for *every* task, but we did include all the relevant pcap files located in the "pcap files" folder in the project.
Same thing regarding the code for all the tasks, including the Python and C files.

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

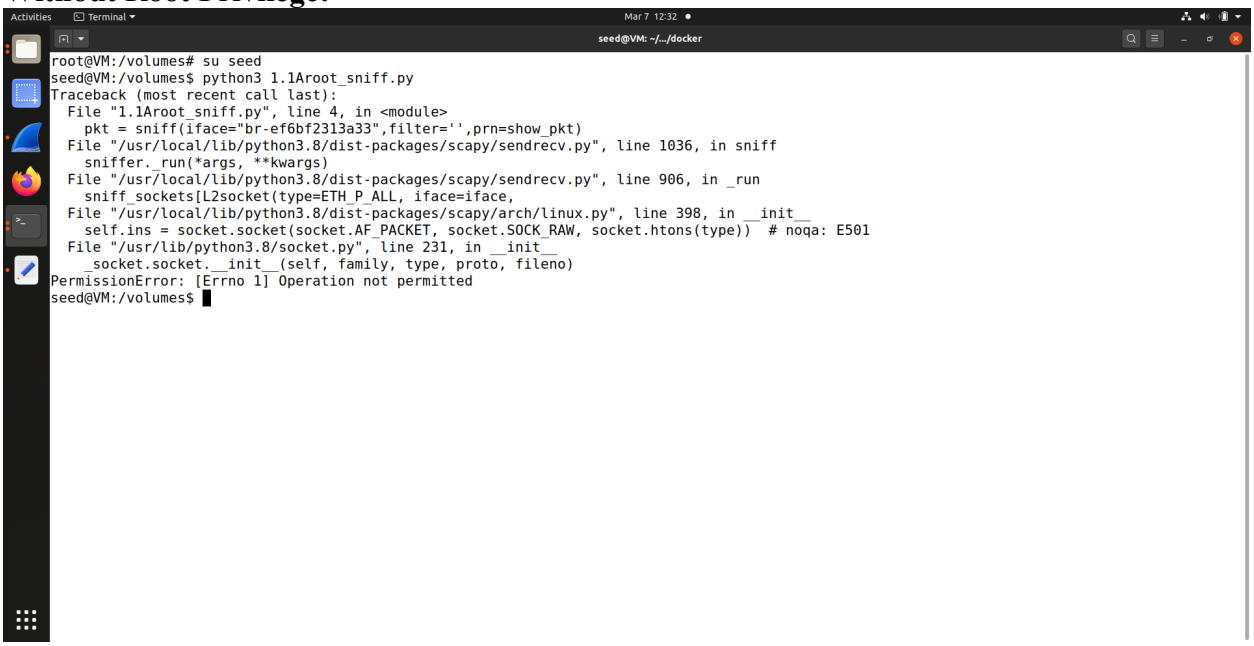
Task 1.1A:

Using Scappy, we build a short python program that sniffs any packet and prints to the terminal all the corresponding information (including Ethernet, IP and RAW information of the packet). We ran the code in two methods, with root privilege and without. Below are screenshots of both methods and the outcome:

With Root Privilege:



Without Root Privilege:

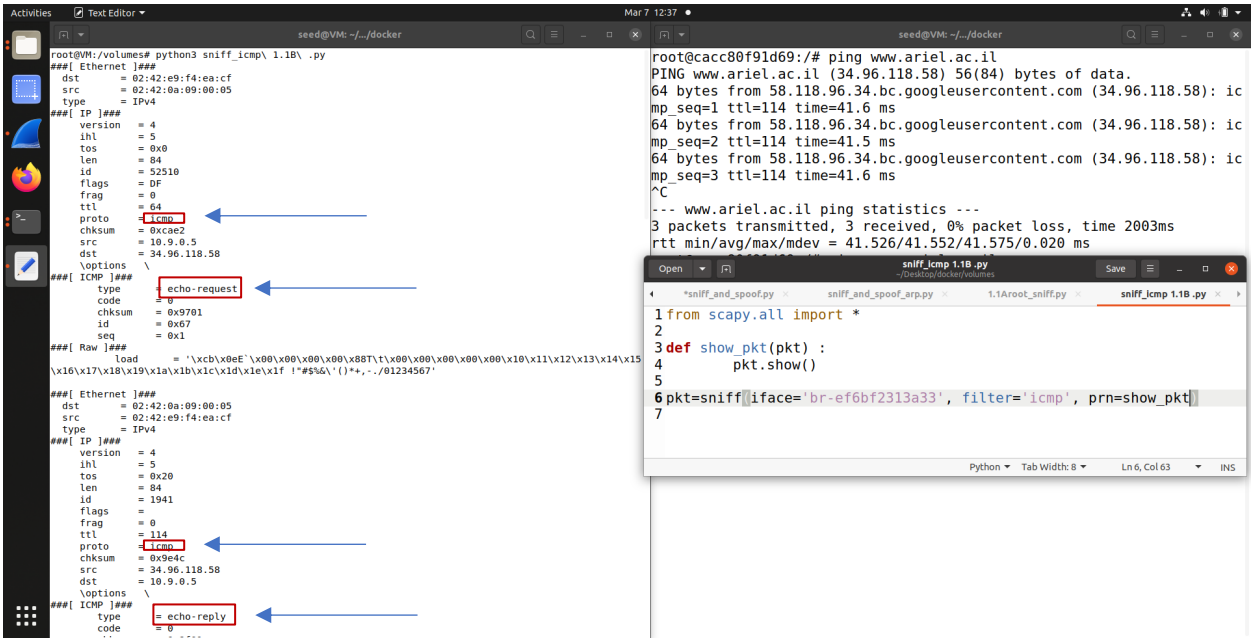


As we can see, when running the program without root privilege we get an error, and the code is not able to sniff for packets. This is because the Scappy library and its functions opens a raw socket which needs root privilege. Opening raw socket has to get root privilege because it allows the changing the modification of packet info/data.

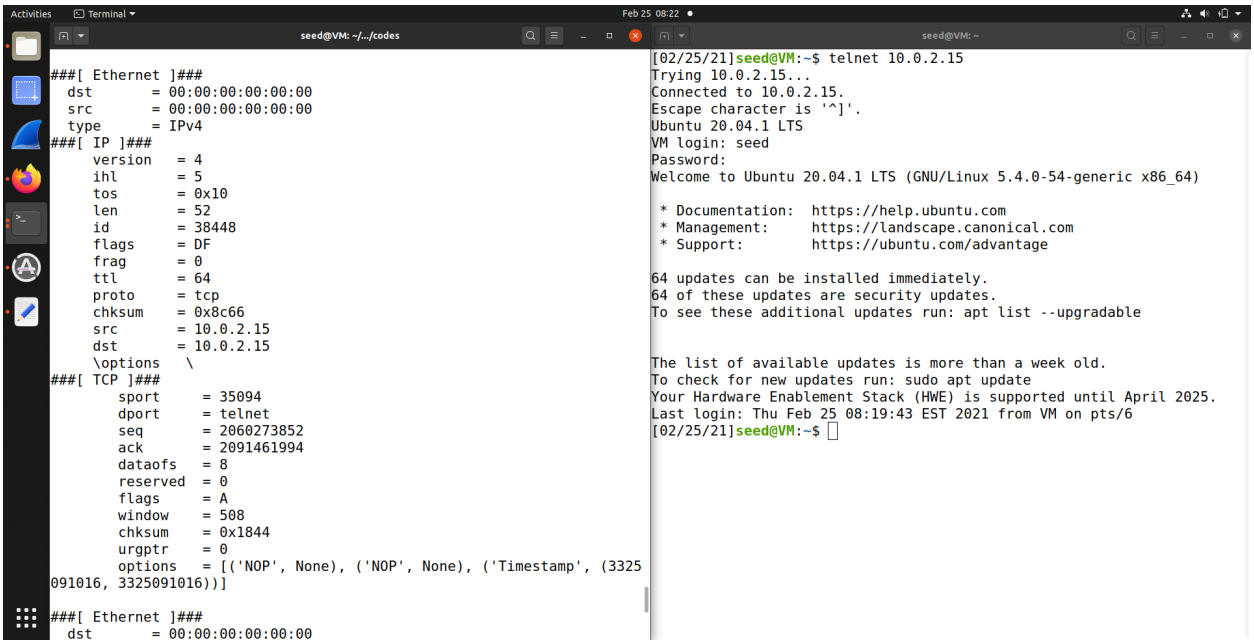
Task 1.1B:

In this task, we used our current python program and modified its sniff function and used its BPF filter parameter. The objectives of this task were to filter out three different things:
For each one, provided is a screenshot.

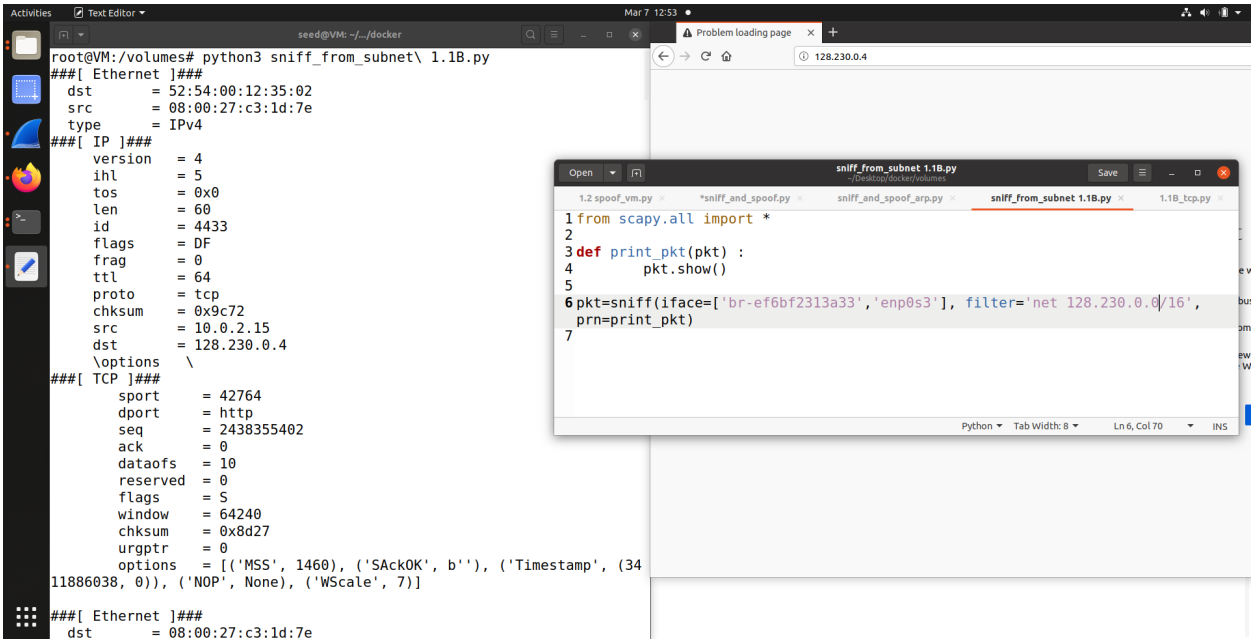
ICMP Packets only:



Capturing any TCP packets that come from a particular IP and with a destination port 23:
(2 photos provided, one showing the outcome of the execution, the second showing the code)



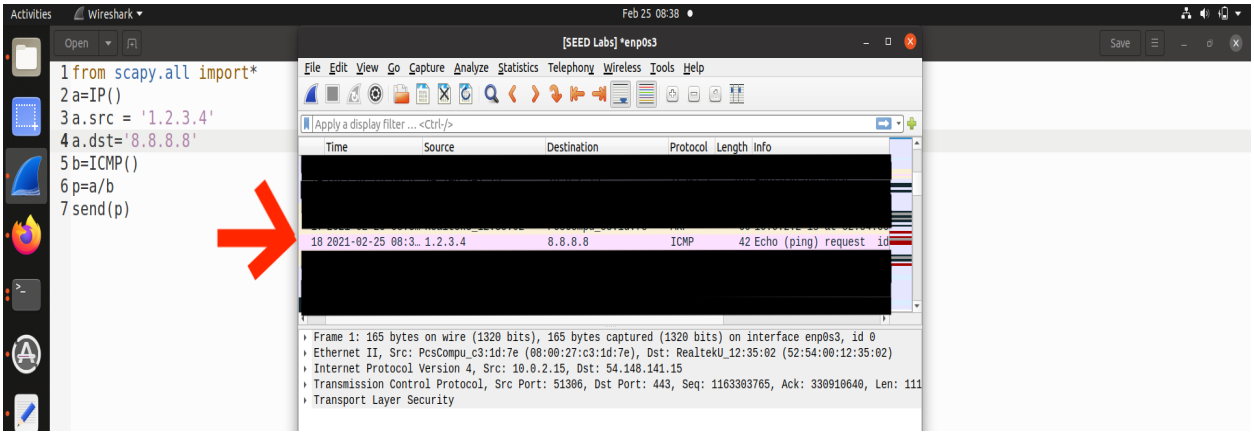
Capturing packets from/to a particular subnet:



Task 1.2:

Using Scappy, we are able to spoof packets and set the fields of the IP packets to arbitrary values. For this task, using Wireshark we were able to see that we successfully spoofed the “IP Source” of the packet and change it so that the Wireshark tool would see that the packet was sent from a different IP address (as requested).

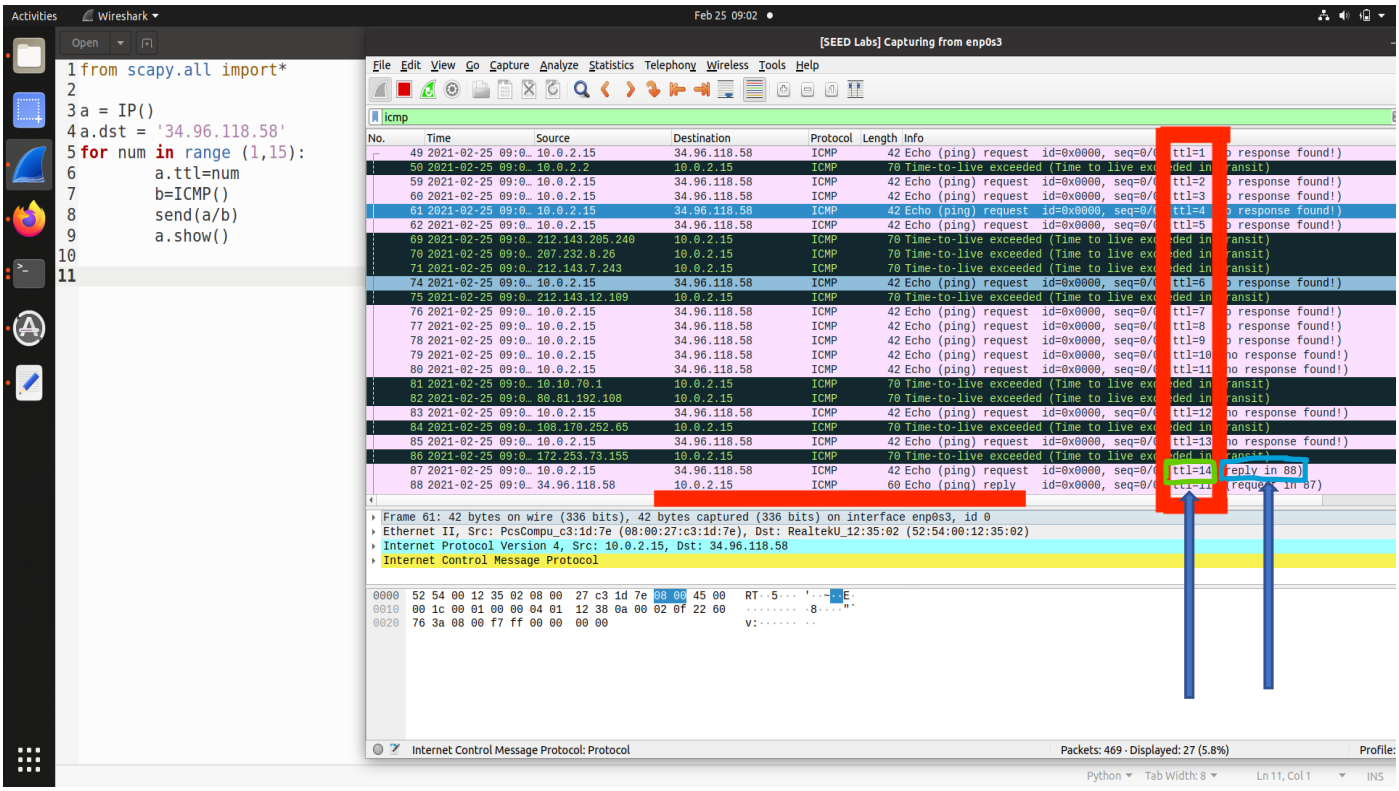
In the screenshot below we can see that Wireshark was able to sniff the ICMP packet sent to Google’s Public DNS (8.8.8.8), but instead of the source IP address being our IP, it read as if the packet was sent from this IP address: 1.2.3.4



Task 1.3:

In this task, to find the distance between our VM and the selected destination (in terms of routers) similar to the traceroute tool, we used Scappy and its TTL field to find how many routers were passed in order to reach the final IP destination. To do this automatically, and not have to change the TTL field each time and increment it by 1, we used a for loop in a particular range and used Wireshark to see which TTL value was exceeded each time and where the ping was successfully transferred to the desired destination.

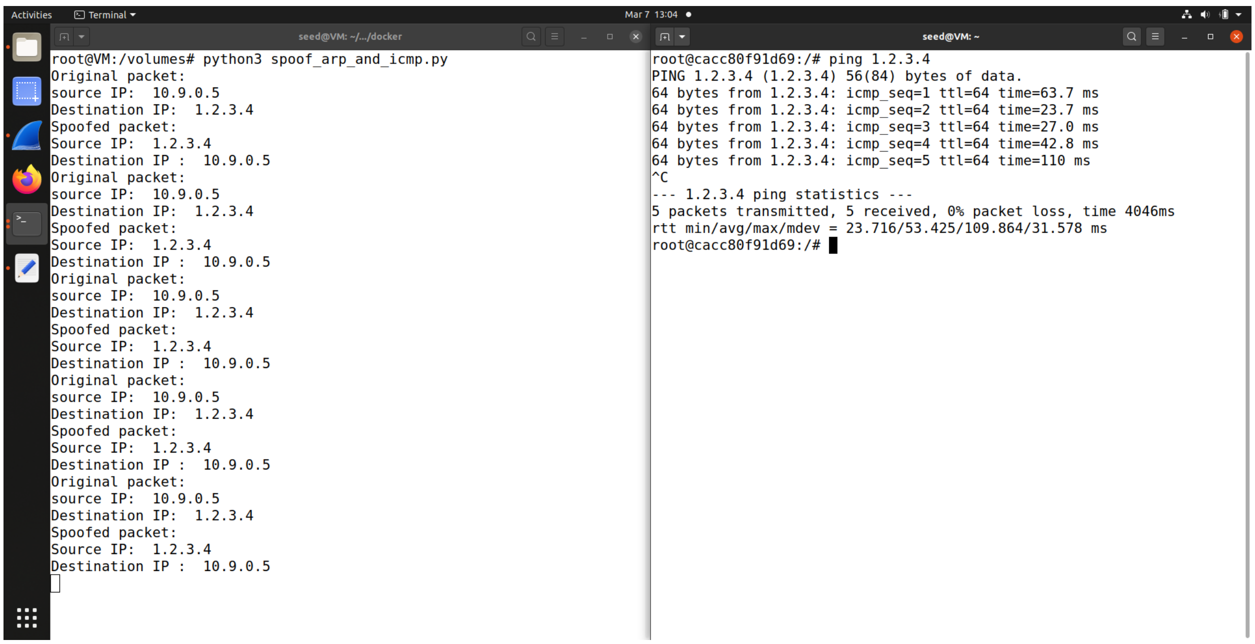
In the screenshot below, we can see the code and the Wireshark program when ran executed our program, and we can see where each ICMP packet failed to receive a reply, and finally where it did receive a reply and there, we can see which TTL value it belongs to.



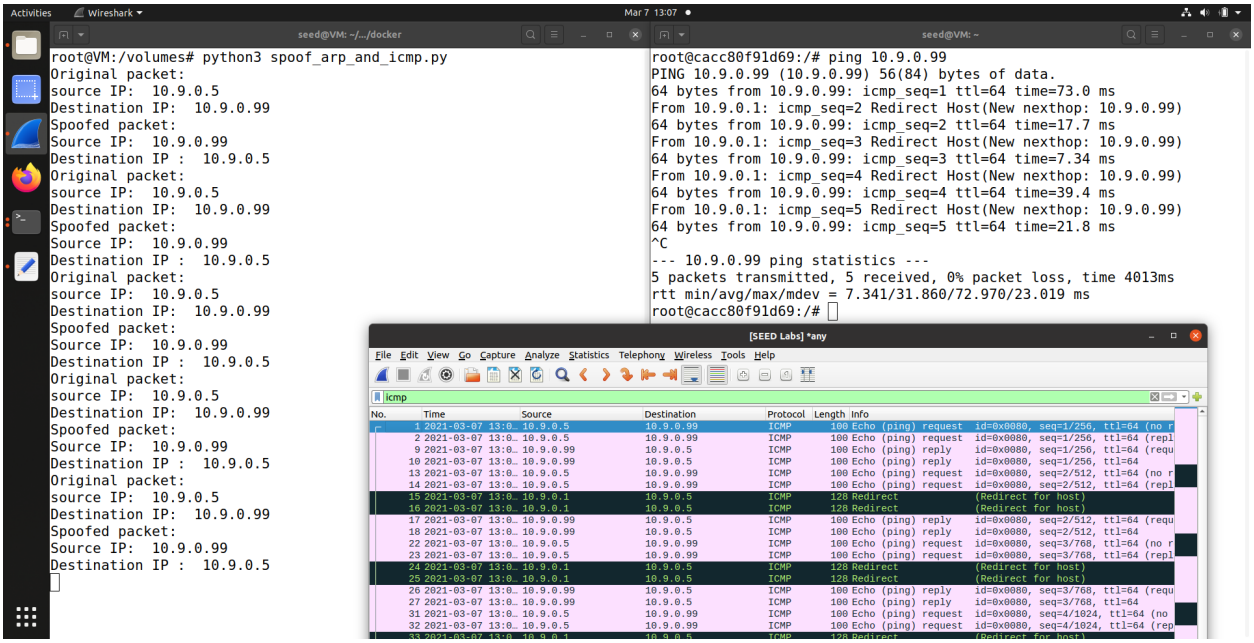
Task 1.4:

In this task, we were able to sniff out pings that generated ICMP echo requests to a certain IP (whether that IP address to a certain machine was alive or not) and then returned a spoof reply to the same machine that sent out the ping.
Therefore, no matter if machine A sent a ping to a non-existing host or even to a machine that is not alive, regardless, machine A received a reply.
Below are 3 screenshots representing the three different IP's we pinged on our host machine, and running the program on our attacker container we were able to sniff and spoof accordingly.
Above each screenshot is written the IP that the host machine pinged.

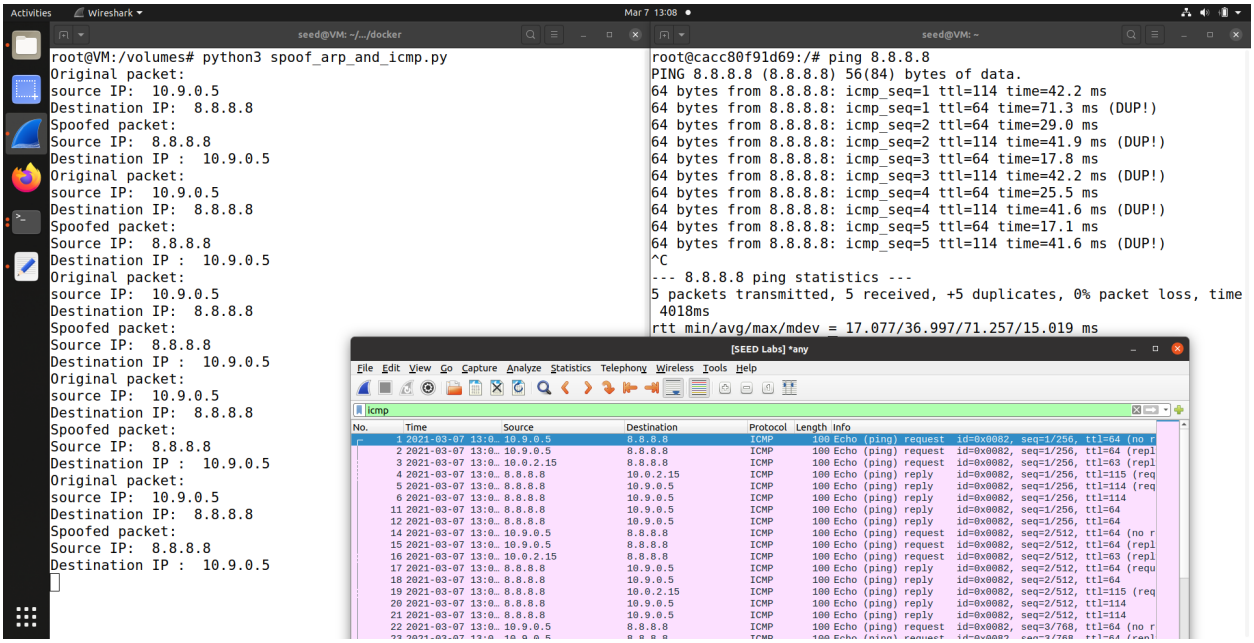
1.2.3.4: A non-exisiting host on the Internet



10.9.0.99: A non-existing host on the LAN



8.8.8.8: An existing host on the Internet

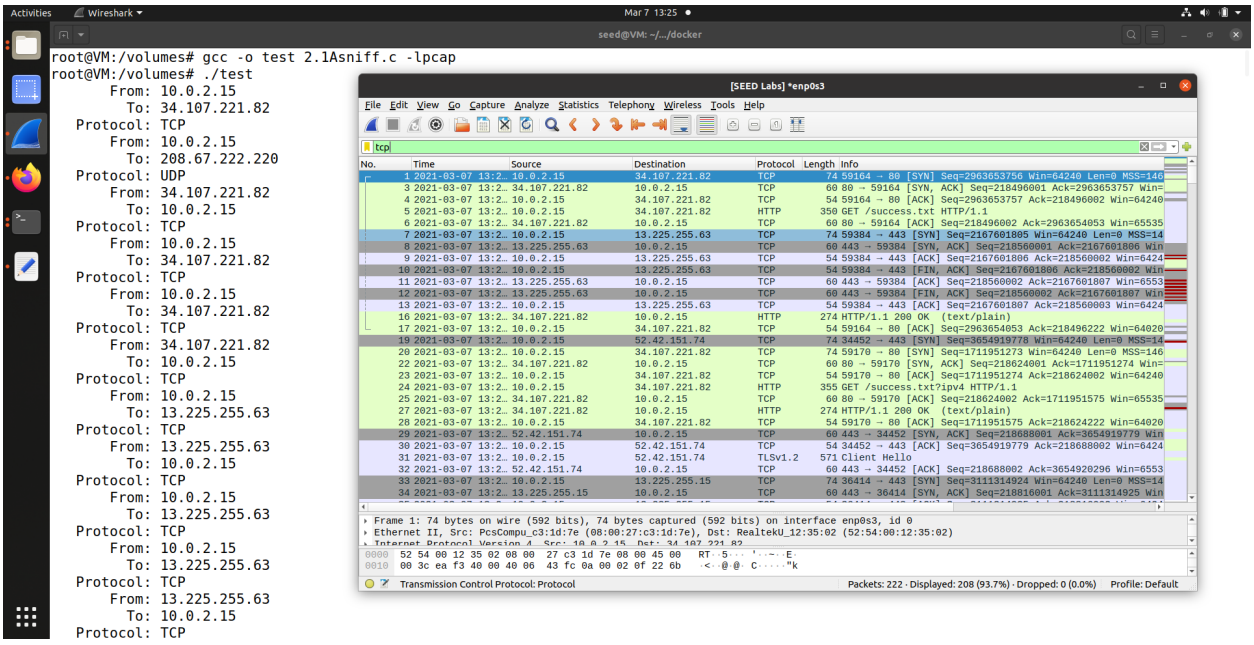


Lab Task Set 2: Writing Programs to Sniff and Spoof Packets:

Task 2.1A:

We built a sniffer program in C that sniffs packets and prints out information of the captured packet, including the source and destination IP addresses, protocol type, source port… etc. We wrote this program using the pcap library, and through that the sniffer became a simple sequence of procedures. In addition, all details of the packet capturing are handled by the pcap library.

Below is a screenshot as requested of the execution of the sniffer program:



Question 1:

There are a sequence of library calls that are essential for sniffer programs. The fundamental function calls that are used for such sniffer programs include a couple steps:

- Setting up the device and determining which ethernet interface that the program will utilize and to start capturing with.
- After setting up the device, occurs the initialization of the pcap to create a session and tells it to sniff on a particular device. Typically, there is only one session per device to be sniffed.
- The traffic filtering rules occurs, and this ensures that the type of traffic sniffed on an interface is the type that we were looking for (filtering). It is also possible to use a blank filter but, in that case, it will be sniffing all packets and will be analyzing all the fields.
- Now the actual execution of the sniff occurs.
- The session ends/is terminated, and the program stops sniffing.

Question 2:

Similar to what I explained in task 1.1A, when running the scappy sniffer program without root privilege. Running the sniffer program that uses pcap needs root access because it wants to access network interfaces and it is impossible to do so without having root access in Linux. Otherwise the program will fail at the "pcap_open_live" stage. Sniffer programs need raw sockets that allow the direct sending of packets by the applications bypassing all applications in network software of the operating system. And in order to create raw sockets, we need to be a root user because we can't discover the NIC (network interface card) until we are root.

Question 3:

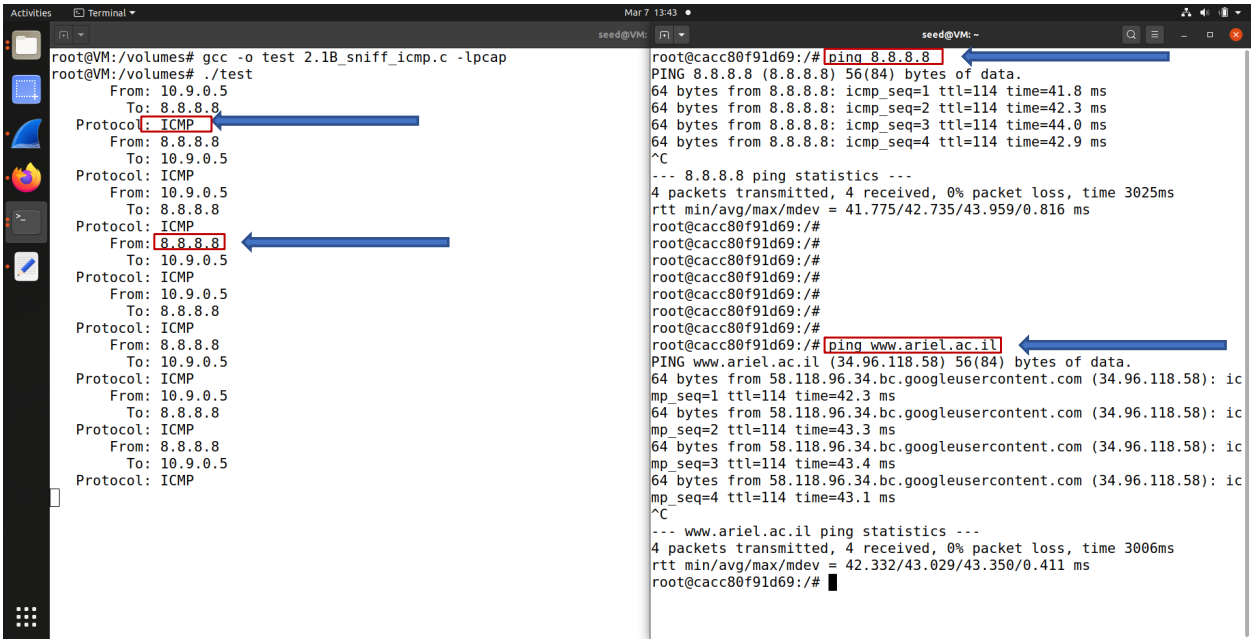
As requested, to demonstrate this for example, let's consider we have two VM's, with the sniffer program running on machine the first machine (1), and the second machine tries to ping a random IP address. Now in non-promiscuous mode I found that no packet is sniffed at machine 1 other than those with the destinations address of the first machine.
In other words, we can simply say that promiscuous mode is one in which all the packets are sent to a computer or sniffed by a sniffer and not only those which are addressed to it.
Whereas when we are in a non-promiscuous mode, only those packets are sent to the computer or sniffed by a sniffer which are specifically addressed to it.
We do this using the "pcap_open_live" function and enter "1" as the third parameter of the function to turn on promiscuous mode, and enter "0", to turn it off. Therefore, we will listen to the traffic that passes only in the specific virtual machine on which we ran the program, and not on all the network traffic.

Task 2.1B:

Sniffer programs using the pcap library, we can easily filter out specific data we want from the packets using pcap filters. The filter expression consists of one or more primitives. Primitives usually consist of an ID (name or number) preceded by one or more qualifiers. There are three different kinds of qualifiers (type, dir, and proto).

Below are two screenshots of the execution of the sniffer program consisting of two different filters. The first is to capture ICMP packets between two specific hosts, and the second is filtered to capture the TCP packets with a destination port number in the range from 10 to 100.

Capturing ICMP packets between two specific hosts:



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

```
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44478
       Destination Port: 80
   Protocol: TCP
    Payload (20 bytes):
.......
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44478
       Destination Port: 80
   Protocol: TCP
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44478
       Destination Port: 80
   Protocol: TCP
    Payload (296 bytes):
........
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44478
       Destination Port: 80
   Protocol: TCP
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44486
       Destination Port: 80
   Protocol: TCP
    Payload (20 bytes):
.......
       From: 10.0.2.15
         To: 34.107.221.82
       Source Port: 44486
```

Open ▾ | sniff.c ~/Desktop/project | Save

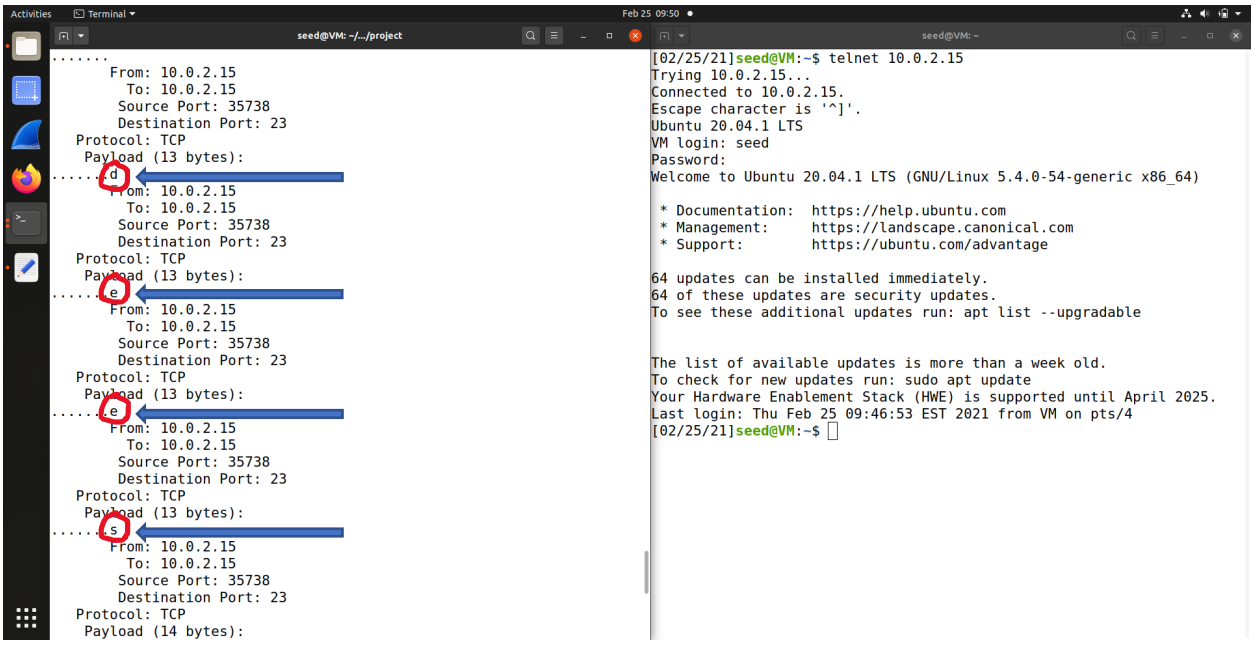
```
84                data+=12;
85                printf(".......%c\n", *data);
86                //for(int i = 0; i < size_data; i++) {
87                //      if (isprint(*data)) printf("%c", *data);
88                //      else printf(".");
89                //      data++;
90                //}
91        }
92
93  }
94}
95
96int main() {
97
98        pcap_t *handle;
99        char errbuf[PCAP_ERRBUF_SIZE];
100       struct bpf_program fp;
101       char filter_exp[] = "proto TCP and dst portrange 10-100";
102       bpf_u_int32 net;
103
104       // step 1: open live pcap session on NIC with interface name
105       handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
106
107       // step 2: compile filter_exp into BPF pseudo-code
108       pcap_compile(handle, &fp, filter_exp, 0, net);
109       pcap_setfilter(handle, &fp);
110
111       // step 3: capture packets
112       pcap_loop(handle, -1, got_packet, NULL);
113
114       pcap_close(handle); // close the handle
```

C ▾ | Tab Width: 8 ▾ | Ln 106, Col 1 ▾ | INS

Task 2.1C:

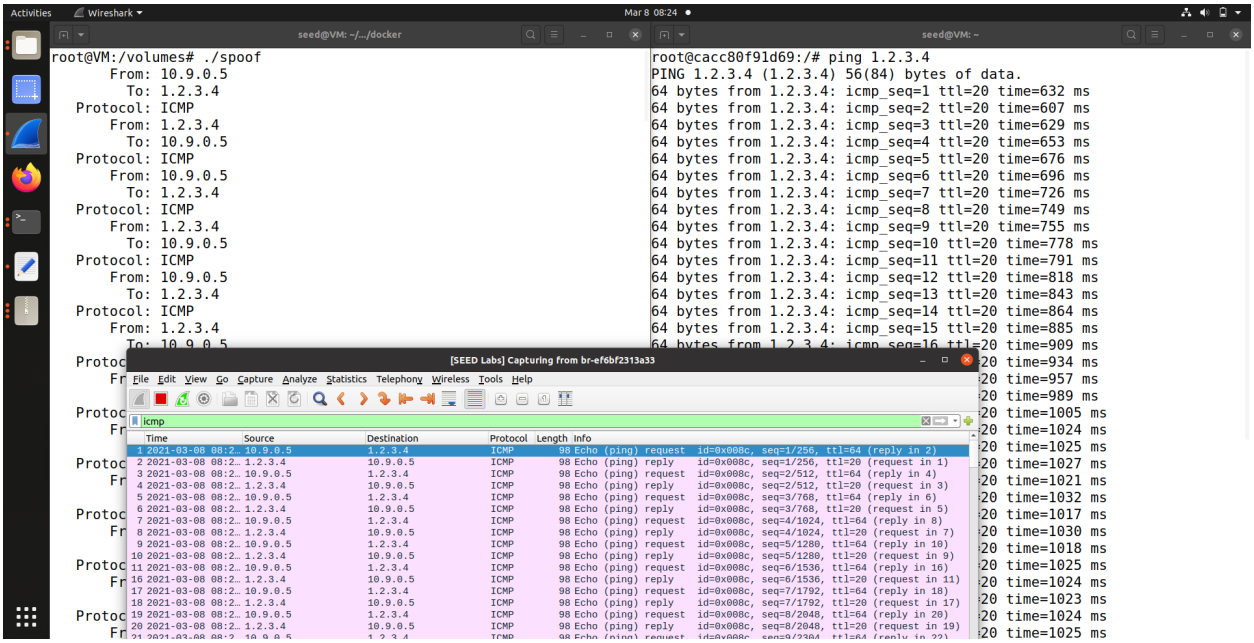
In this program we used our sniffer program to capture the password when using telnet on the network we are monitoring. Given that we ran this program on the SEED VM, when using telnet we logged in to the seed account and entered the password in the terminal and at the same time ran our sniffer program to sniff out the password.

In the screenshot below we can see that we successfully were able to sniff out the password of the host machine. The password was: “dees”.



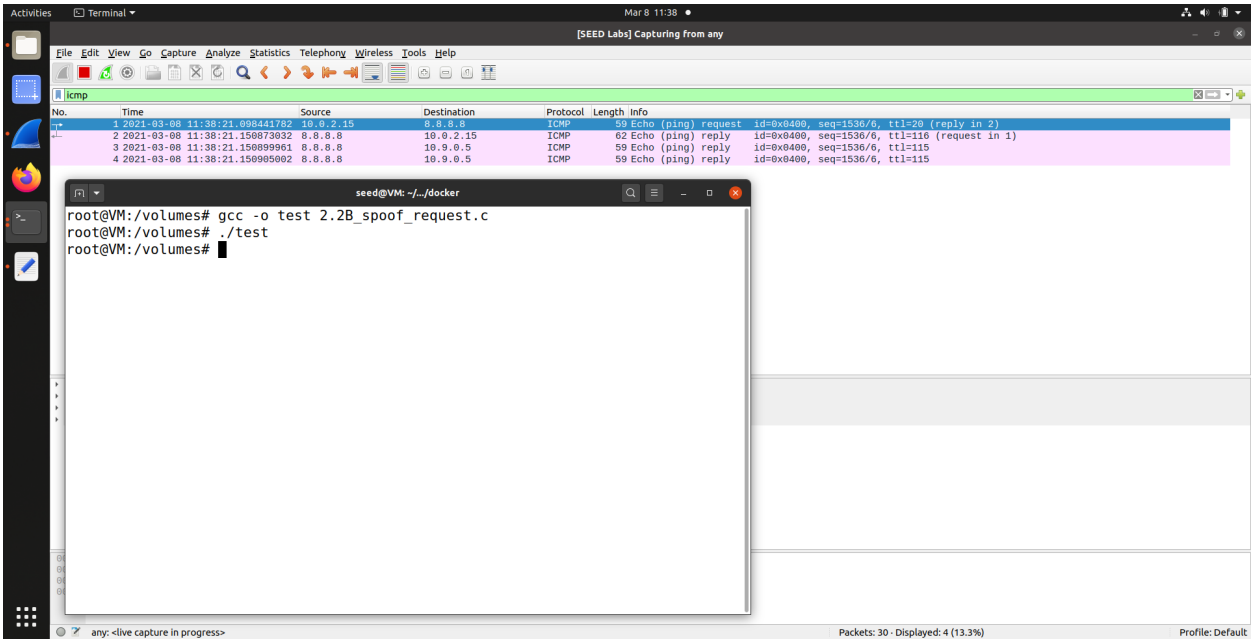
Task 2.2A:

Below is a screenshot of our packet spoofing program in action. In addition, added to the project located in the Wireshark pcap files folder, is the Wireshark packet trace that proves that the packet was successfully spoofed.



Task 2.2B:

Below is a screenshot of our packet spoofing program in action that spoofed an ICMP echo request on behalf of another machine. In addition, added to the project located in the Wireshark pcap files folder, is the Wireshark packet trace that proves that the packet was successfully spoofed, and we can see the echo reply coming back from the remote machine.



Question 4:

Yes, the length of the IP header can be set to arbitrary values, but it won't change, and it will return to its original size. The length of the IP header can be set to arbitrary values regardless of how big the actual packet is. This is because when we send a packet larger than its actual size, the additional data in the payload is a chunk of zeros. And that is in fact what we see on the captured packets when we inspect them using a program such as Wireshark. The payload will manually increase until completing the total length of the packet.

Question 5:

No, for IP headers the computer/operating system automatically does this, or rather fills it in by default.

Question 6:

In order to run programs that use raw sockets, similarly to what we explained in question 2 (for the sniffer program), we need root privilege because we can spoof packets, which may interfere with inbound traffic. In other words, opening a raw socket allows to read anything that is received in a given interface, so, basically, we can read any packet that is directed to any application - even if that application is owned by another user. That basically means that the user with this capability is able to read any and all communications of all users on the machine.

Task 2.3:

Below is a screenshot of our sniffing and spoofing program in action that sniffs ICMP echoes of another machine and spoofs out a reply regardless of the target IP address, and regardless of whether the machine is alive or not.

Therefore, the ping program will always receive a reply.

Here we pinged 1.2.3.4 – a non-existing IP on the Internet and received a reply.