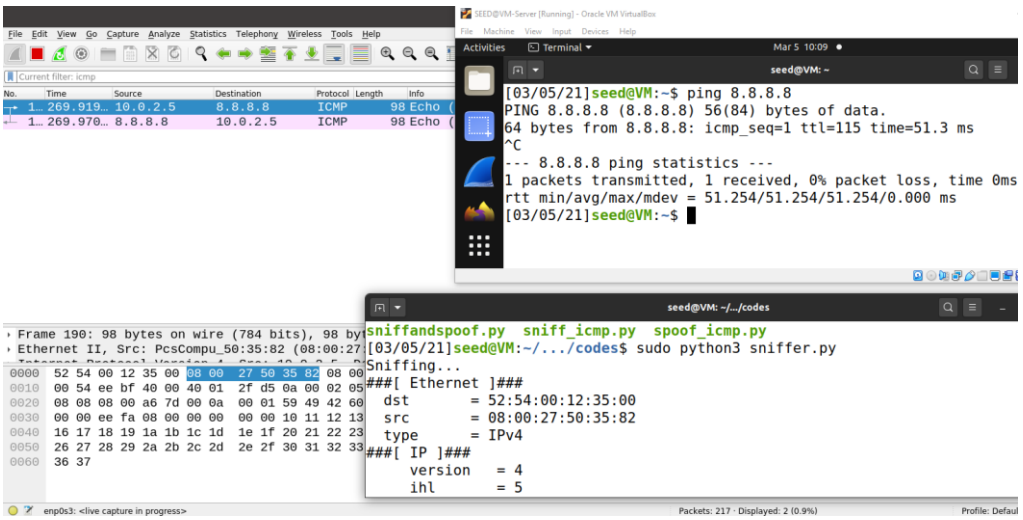
Sniffing and Spoofing

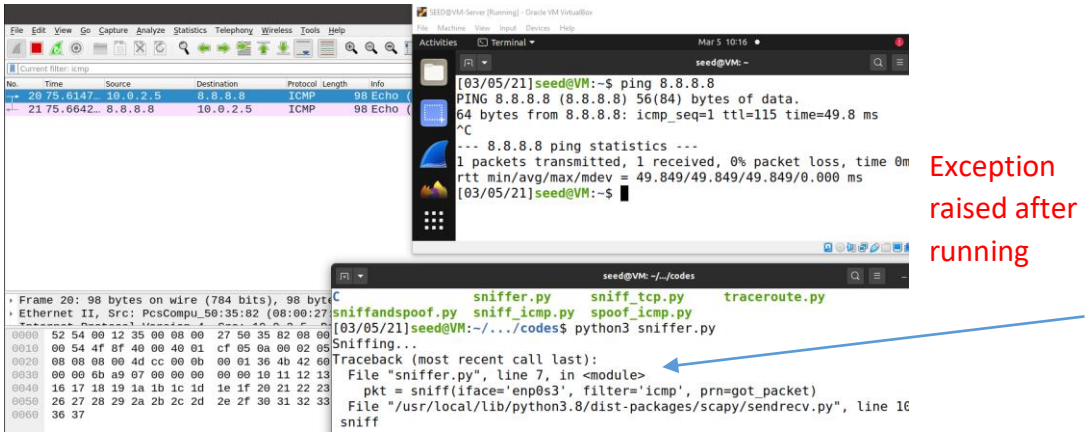
Task 1:

1A:



The image above describes running the python code 'sniffer.py',

with root privileges, which sniffs packets on the machine's default network interface.



Exception raised after running

The image above describes running the same code (sniffer.py),

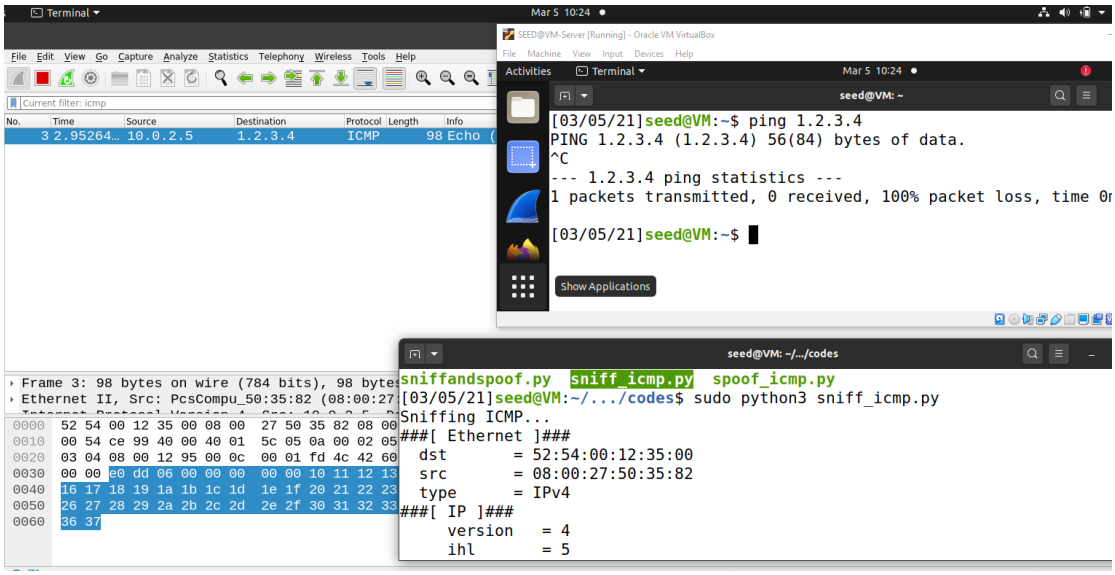
Without root privileges. Which will not succeed to run the sniffing program.

Sniffing/Spoofing programs require root privileges in order to receive packets from our network interface device.

1B: ICMP:

As you can see, our ICMP sniffing program successfully sniffed an ICMP-Echo request to 1.2.3.4, which got no response.

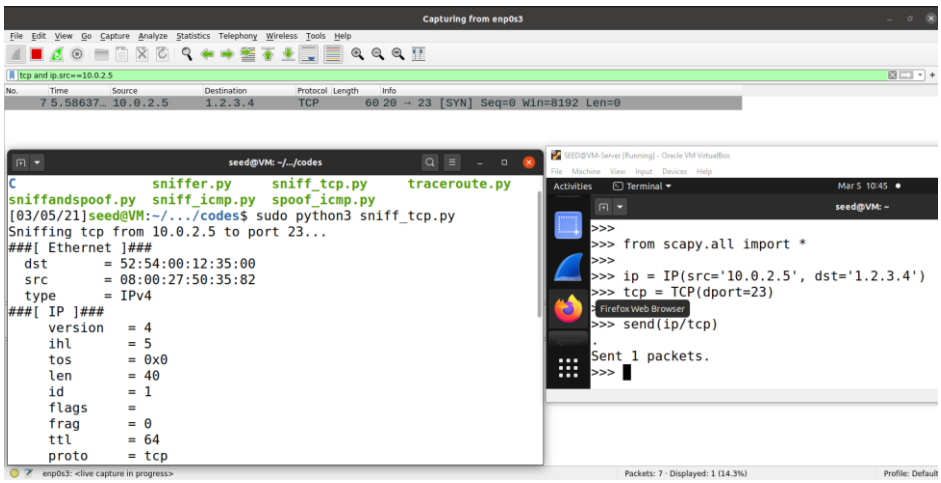
The message was sent from our server (10.0.2.5), and the code which was running on our attacker's VM terminal (10.0.2.4) printed out the packet.



TCP:

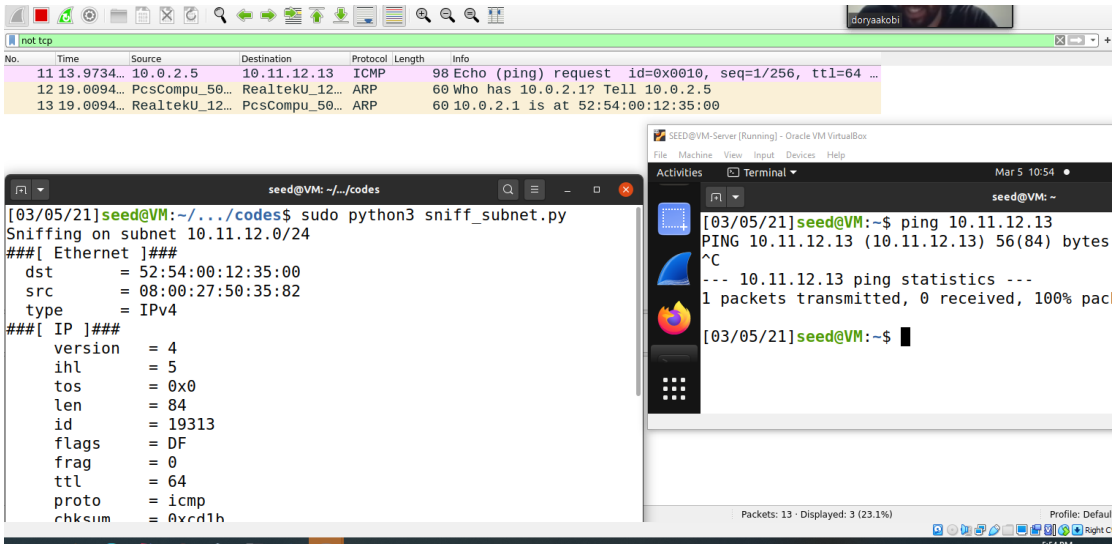
Our TCP program successfully sniffed 2 TCP packets that arrived at our Attacker's VM (10.0.2.4) at port 23.

The IP/TCP packet was forged using scapy and sent through our Server VM (Could have been sent to any other destination as well out-side of 10.0.2.0/24. **Port 23 is reserved for the Telnet protocol,** which is used to open connections unencrypted between machines.



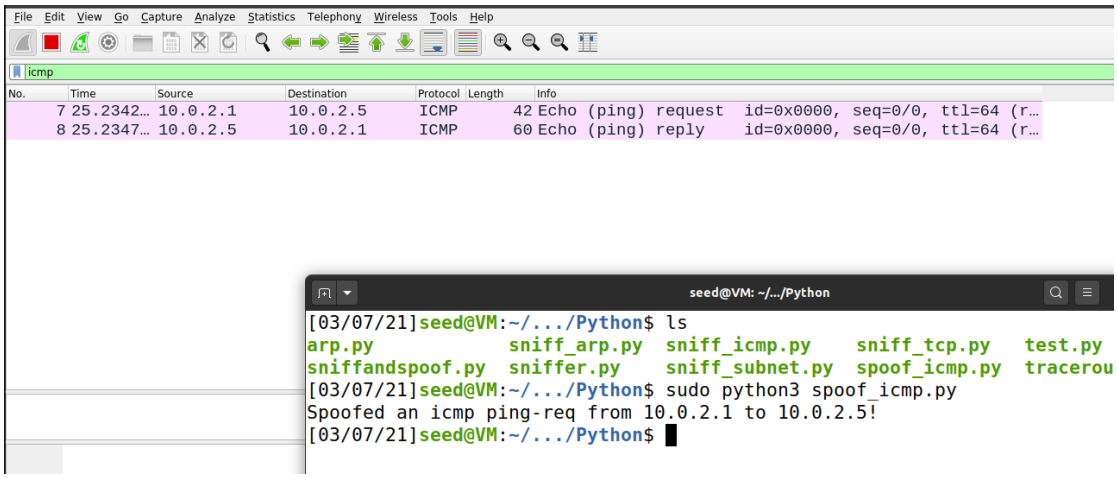
Specific subnets:

Our specific subnet sniffer successfully sniffs packets with IP destination: 10.11.12.0/24 (Any client in the subnet 10.11.12.X).



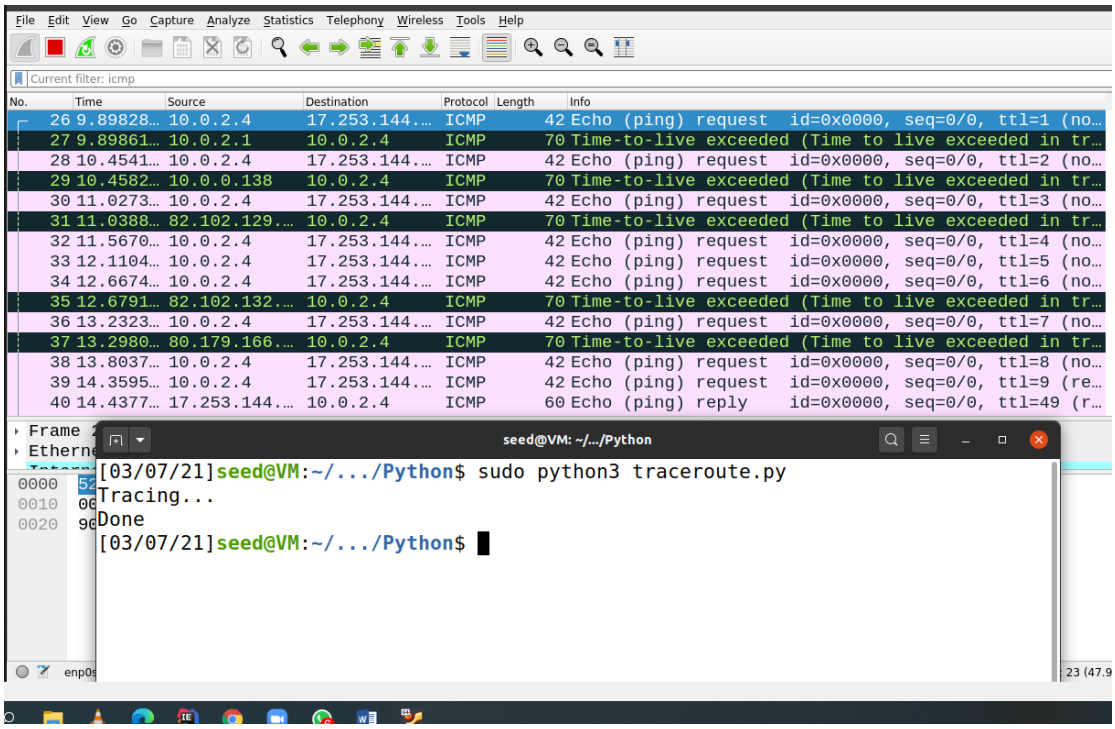
1.2:

Spoofing ICMP:



1.3:

Traceroute:



The image above displays how we use tracerouting to establish the connection with 'apple.com'.
In our code we generate ICMP packets with TTL in range of 1-13, sent to the 'apple.com' website holding IP address '17.253.144.10'.
Each router that the packet pass through decreases the packets TTL by 1, the first router which returns the TTL exceeded response is our gateway (i.e. 10.0.2.1), which connects us with the outer network.

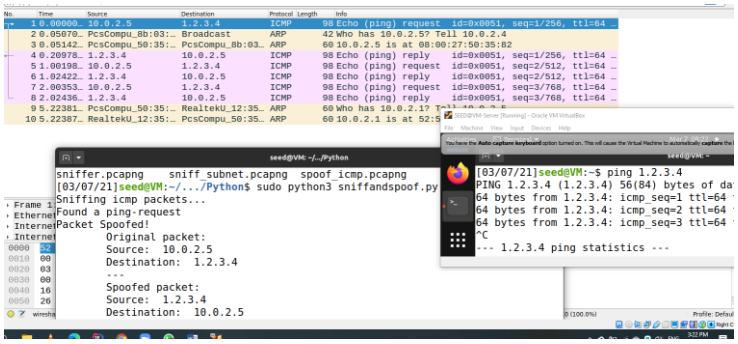
Eventually, we send out a packet with a sufficient amount of TTL to reach its final destination. That happens with a TTL sum of 5 for this case (a total amount of 5 routers), and setting TTL to 8 or higher, allows us to communicate and receive replies from the destination server.

1.4:

Sniff and Spoof:

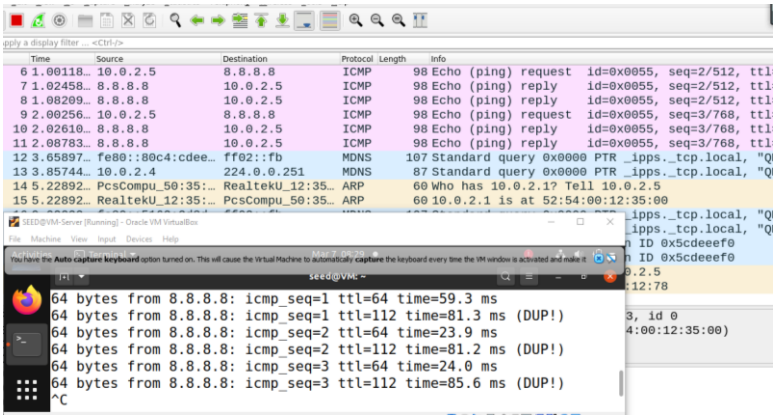
The images below demonstrate our sniffing and spoofing tool, which monitors our network interface (enp0s3) and awaits ICMP Echo-Request packets.

The first image below shows that our server (10.0.2.5) pings a Non-Existing host on the Internet, and successfully spoofs an ICMP reply packet, which keeps the PING program running and receiving responses, although the host 1.2.3.4 does not exist.



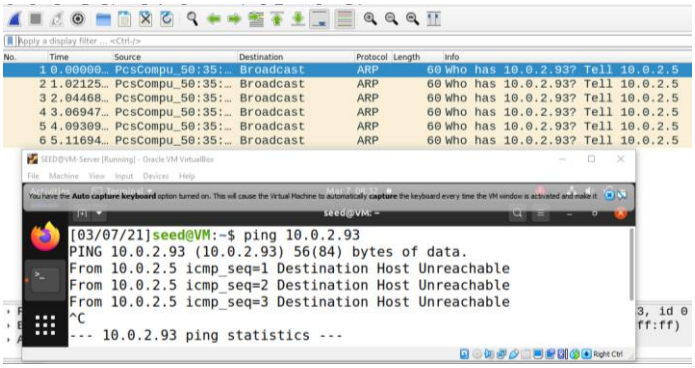
The second image here shows our server pinging an existing host on the internet (outside of our LAN), and successfully spoofing replies, although the remote server is also sending replies.

This causes a Duplicate ICMP reply packet for the pinging server.



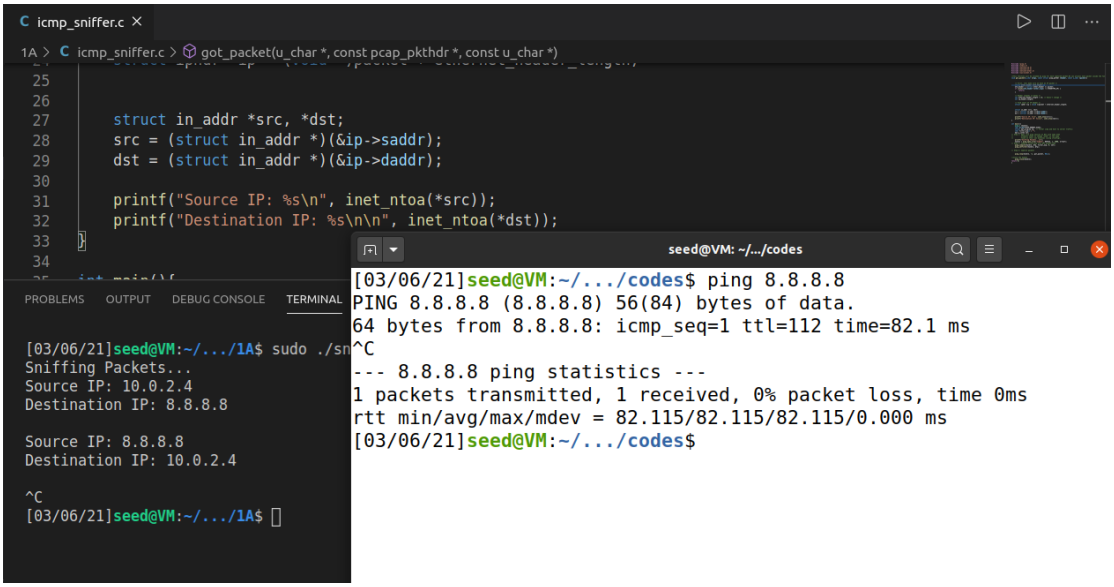
(Non-Existing host on our LAN)

The third image is an example of pinging a non-existing host in our LAN resulting an ARP spam that asks: **Who has the non-existing host?**, which no one replies to. Thus no packet is sent and cannot be sniffed and then spoofed.



C:

2.1A:



The image above shows how running 'sudo ./sniffer' starts sniffing all kinds of packets, as long as they have an IP header, with a source and destination.

In the terminal on the right we have pinged google (8.8.8.8) and got a reply. Both of the packets are displayed on the left terminal, with their source and destination printed.

Follow-up Questions:

1. A. The initial call for the function pcap_open_live() determines to which interface we are currently listening to, and capture mode (promiscuous, etc…).

   B. Later calls to: pcap_set_promisc();

   pcap_setfilter();

   pcap_compile();

   etc…

   are used to compile the string to filter the packets we're interested in, and insert more sniffing options.

   C. the function pcap_loop() allows up to continue live listening in a loop on the specified network interface.
   D. the function pcap_close() releases all resources and close the program.

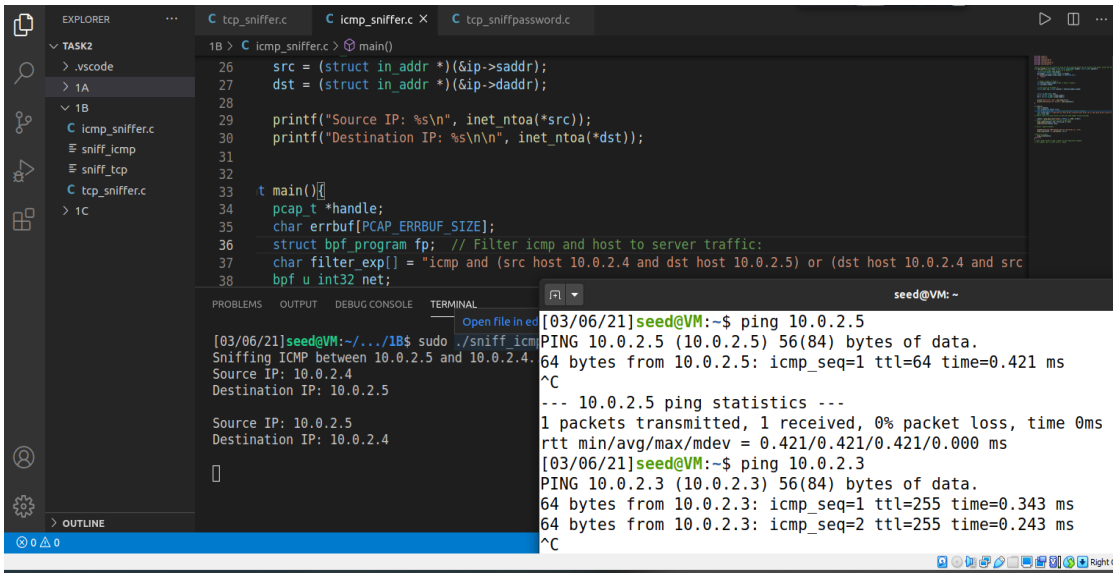
2. While running the program, we have to use the ROOT privileges, in-order to take the data from our network interface, which receives and sends network data (packets).

   Running without the ROOT privileges will deny access to the network interface and will not allow creating a Raw Socket.

   The program will fail executing the **pcap_compile()** function (Because it is called before creating a socket), because the network interface passed to pcap_open_live() needs to be compiled before running, and notes that root privileges were not given.

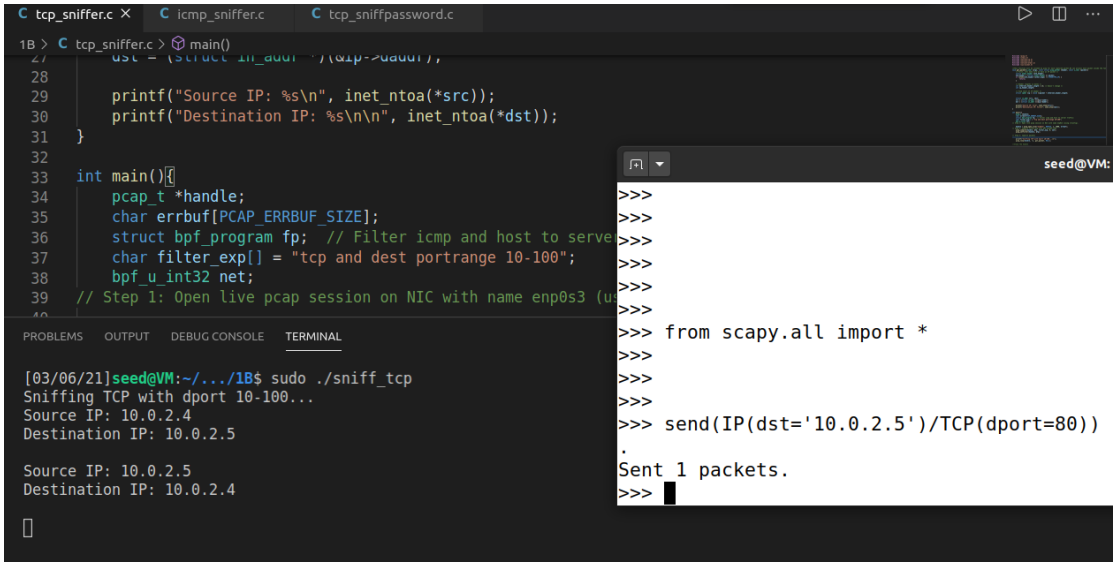
3. We have tried **turning off** the promiscuous mode in-order to receive only packets that are destined to our Attacker's VM (10.0.2.4), and **turning on** the promiscuous mode in-order to **receive all packets** that are passing through our network interface.
   This could be described by spoofing an ICMP packet to another machine from a spoofed src address(Ex: 10.0.2.3 to 10.0.2.5), and if the promisc mode is **OFF**, then the packets should not show up. If promisc mode is **ON**, then the packets should show up.

2.1B:



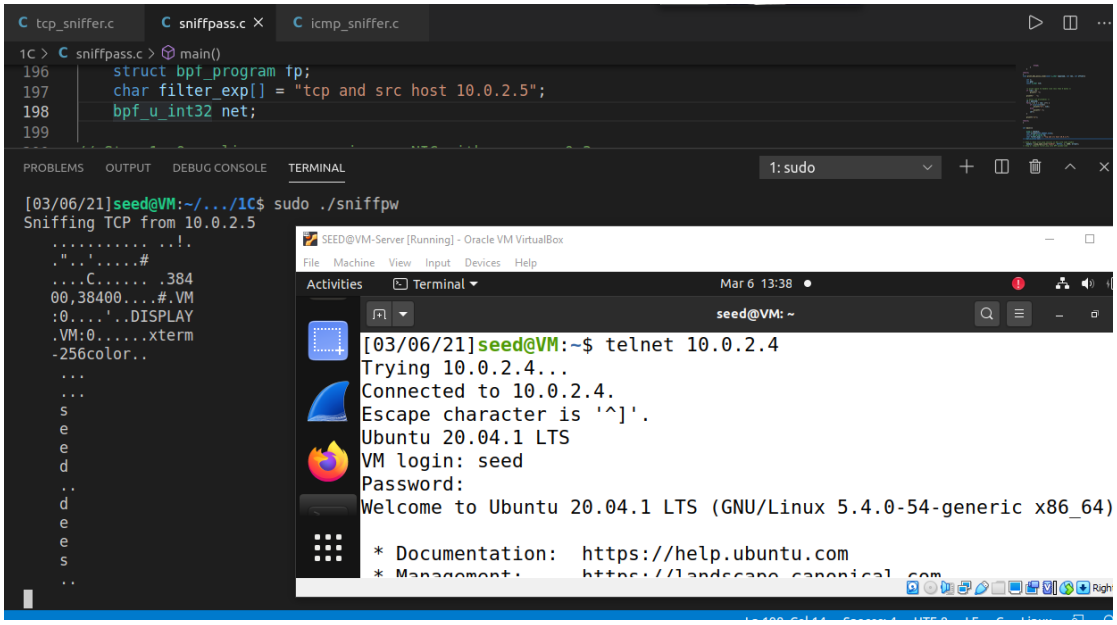
In the above image you can see our sniffer program 'icmp_sniffer.c' captures and prints packets that are from 10.0.2.4 to 10.0.2.5 and the other way around, and are of type ICMP.

Note: that the second ping is from 10.0.2.4 to 10.0.2.3, and our sniffing program did not capture these packets!!! (which means the filtering works!)



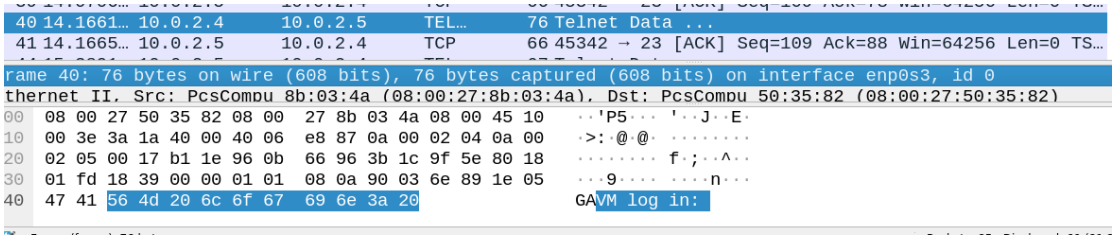
In the above image you can see our sniffer program 'tcp_sniffer.c' successfully captures and prints packets that have a destination port in range 10-100.

2.1C:



The image above shows how we have successfully sniffed out all the TCP traffic from our server VM (10.0.2.5) to our Attacker's VM (10.0.2.4) through a telnet (port 23) connection (**Unencrypted**).

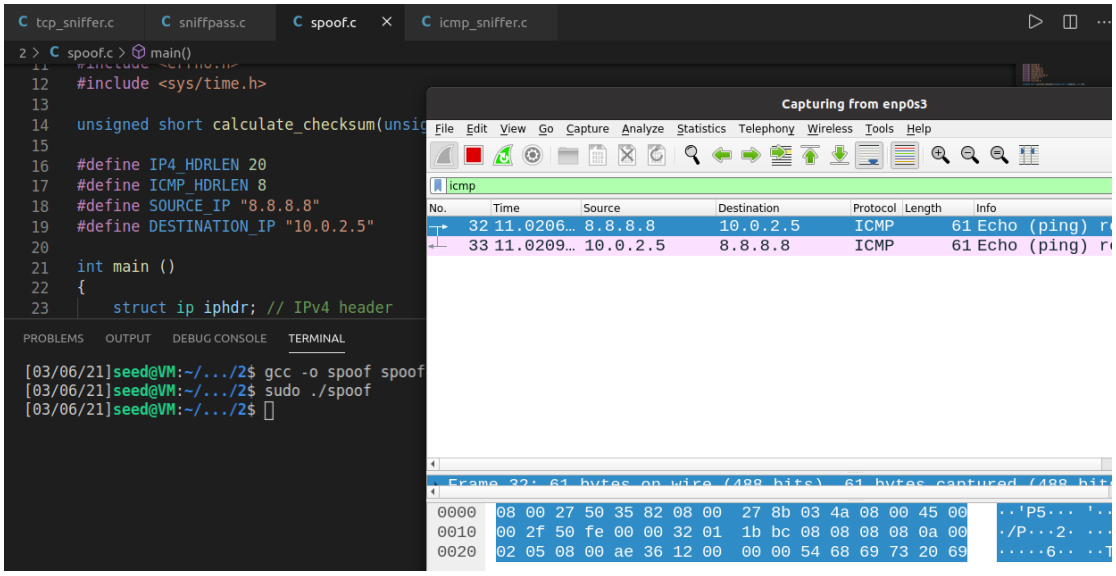
On the left terminal, you can see the stream of characters that match the VM Login field('seed'), and after 1 line, the stream that matches the password field('dees').



In the above image you can see host VM (10.0.2.4) requires VM Login info from the other connection (10.0.2.5).

In the above image you can see a Wireshark tool, called TCP -> Stream follower, which Follows the unencrypted TCP data stream, and extracts the Login and password details.

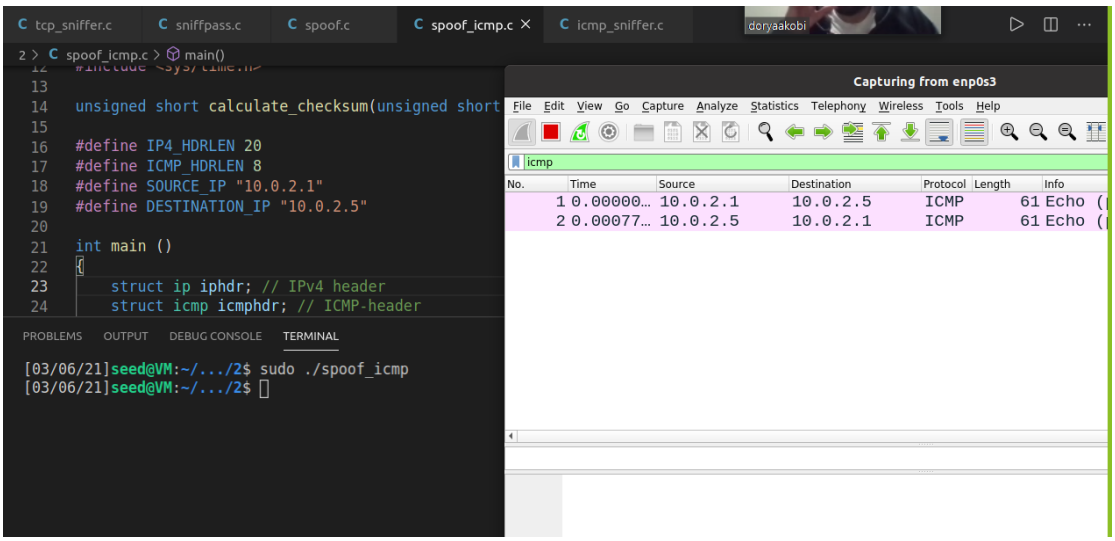
2.2A:



In the above image, you can see how running the spoof.c program successfully spoofs and ICMP packet.

Successful ICMP spoofing causes the destination host to reply (if it is alive) to the spoofed source (Result achieved).

2.2B:



In the above image, you can see how running the spoof_icmp.c program successfully spoofs an ICMP packet.

Successful ICMP spoofing causes the destination host to reply (if it is alive) to the spoofed source (Result achieved).

Follow-up Questions:

4. Yes, we can set the IP packet length field to an arbitrary value, Causing the payload to become a chunk of zeroes (0's) because the memory location of the payload is lost.

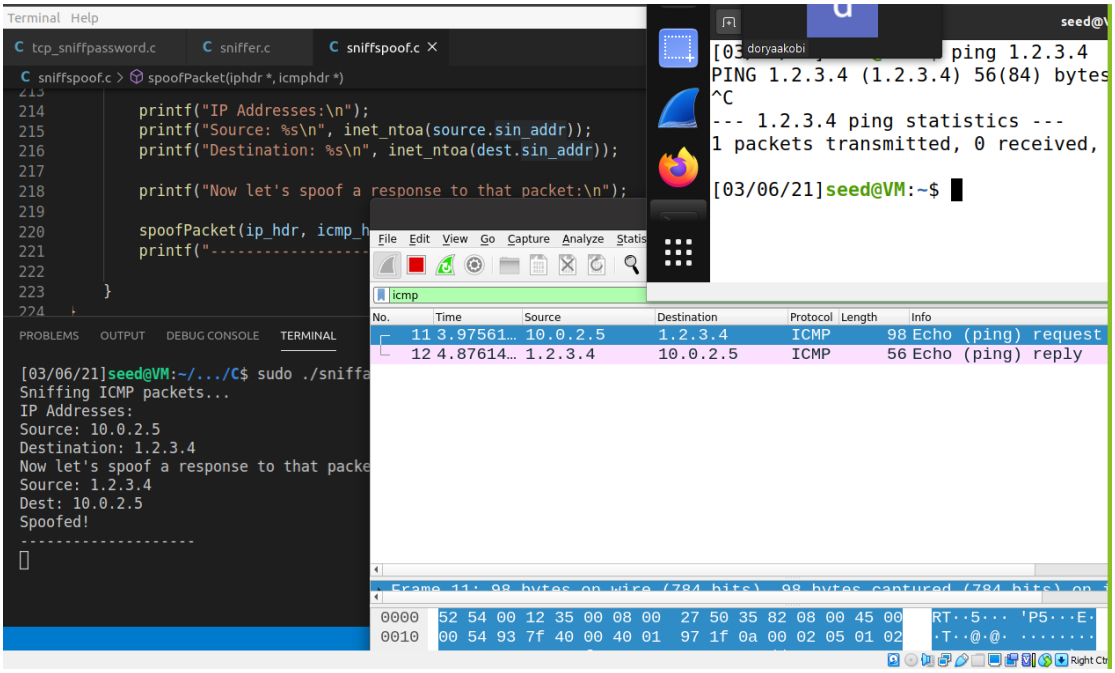
5. No, we do not have to calculate the checksum field for **IP** headers, because the raw socket functionality does that for us.

6. You have to provide root privileges in order to use Raw Sockets, because raw sockets enable us to set all the fields of a packet as we would like, and disable the OS's Kernel from intervening with adding the headers it-self. By doing so, it is possible to construct our own SPOOFED packet and send it. If no root privileges are required -> then it was very easy to forge packets and send them around the world and create kind of a chaos.

   Our program still fails in pcap_compile() function, because we are compiling and running pcap before creating a socket.

   If we were to create a **Raw Socket** before compiling pcap, the program will fail in the socket creation (sock = socket(…)).

2.3 Sniff and Spoof:



In this sniff and spoof program, we have accomplished spoofing sniffed ICMP Echo-request packets, with all the needed packet fields, except for the payload (Which probably causes the ping program to not receive a reply).

In this task we were able to ping all required addresses (i.e. Existing on the LAN, Non-Existing on the Internet and Existing on the Internet Addresses), only in the Wireshark packet trace.

If we were to copy the packets payload successfully into our new **Spoofed** packet, then the PING program should work the same as 1.4(python sniff and spoof).

In the images above and below you can see in our Wireshark trace, that we have spoofed 1 Echo-Reply for 1 Echo-Request packet.