**Packet Sniffing and Spoofing Lab Report**

**1. Overview**

To comprehend the concepts of packet sniffing and spoofing in detail, the following activities were accomplished as part of this lab

* Setting up working environment via Docker containers. But, the primary tool
* Utilizing tools used for sniffing and spoofing e.g., Scapy together with Python and C programming.

This report covers the steps, commands, and observations made during the lab. I have also attached relevant screenshots to show the successful execution of each task.

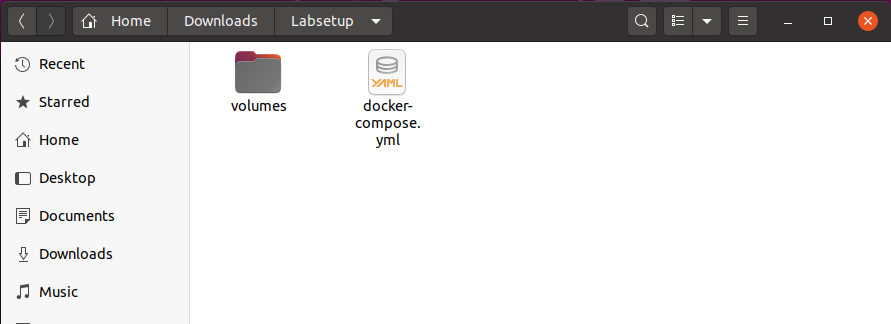
**2. Environment Setup**

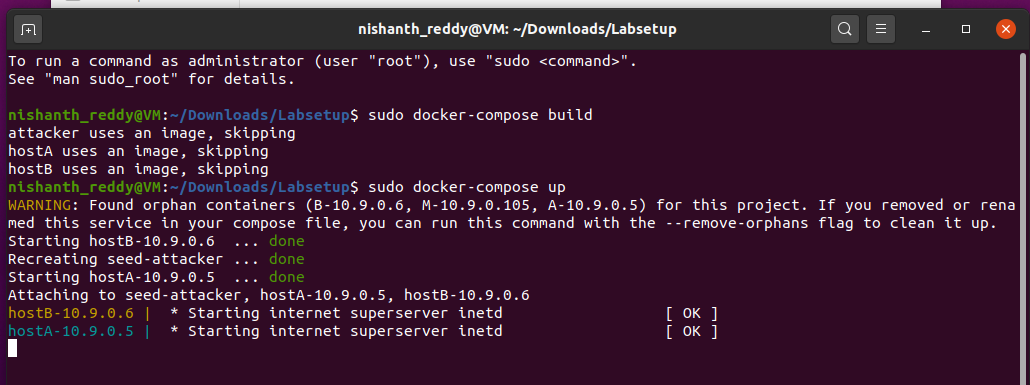
**2.1. Download and Setup Containers**

To set up the lab environment, I first downloaded and unzipped the **Labsetup.zip** file from the SEED website. I then used Docker Compose to set up the containers as instructed.

**sudo docker-compose build**

**sudo docker-compose up**

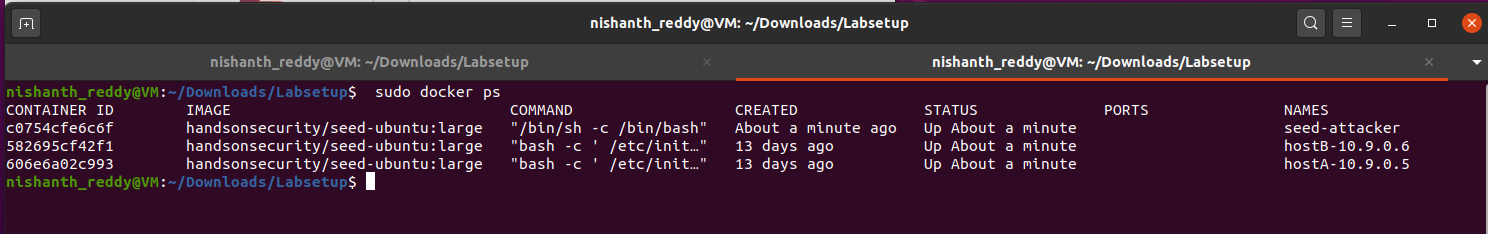
****

****

All containers ran in the background. I used the following command to verify the container status:

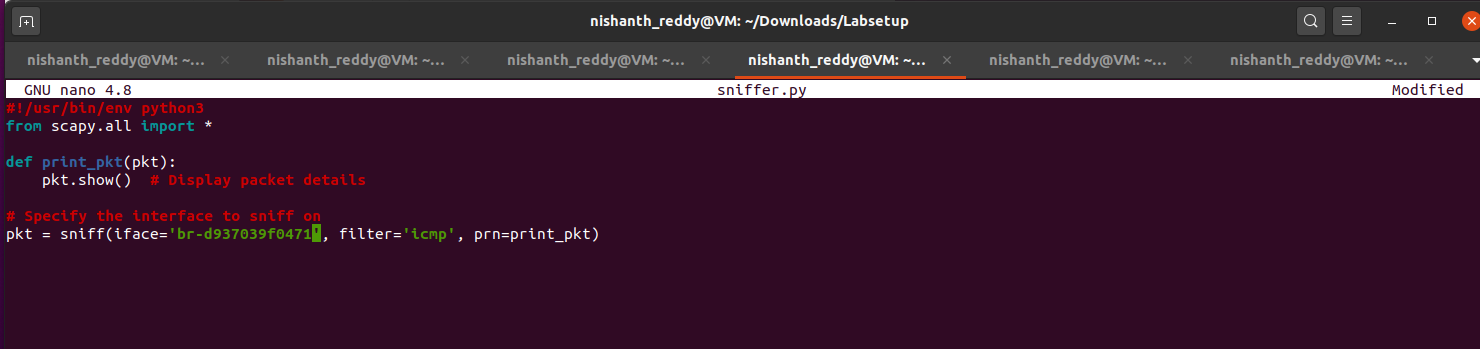
sudo docker ps

This command listed all running containers. Here is an example output showing the containers:



**3. Packet Sniffing**

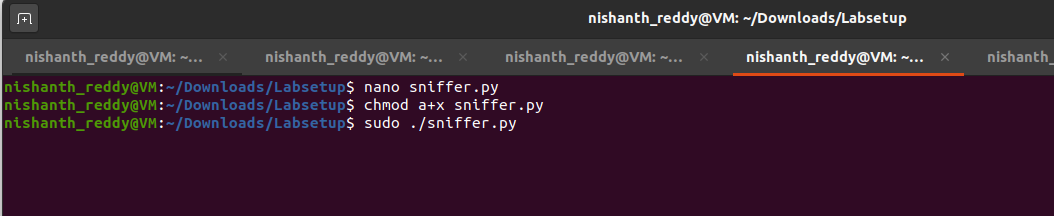
**3.1 Task 1.1A: Basic Packet Sniffing**

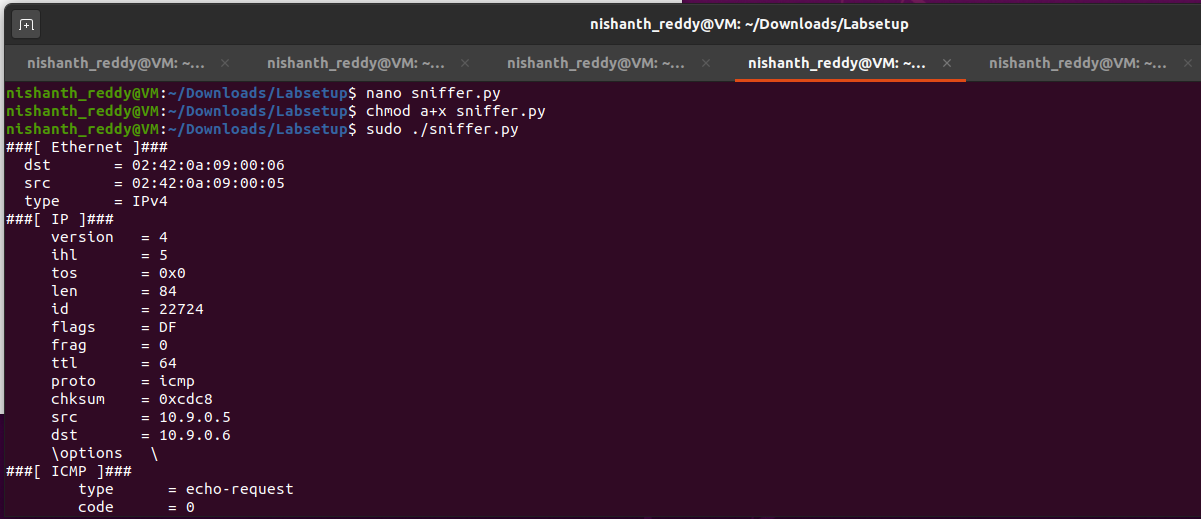
I started by creating a new Python script named “sniffer.py”. I opened my terminal and ran the following command:

I pinged host B, from host A i.e ‘**ping 10.9.0.6’** then I executed this script with root privileges:

**chmod a+x sniffer.py**

**./sniffer.py**

****

****

When I ran the program without root privileges, I observed that packet capture failed, which is expected because raw socket operations need root privileges.

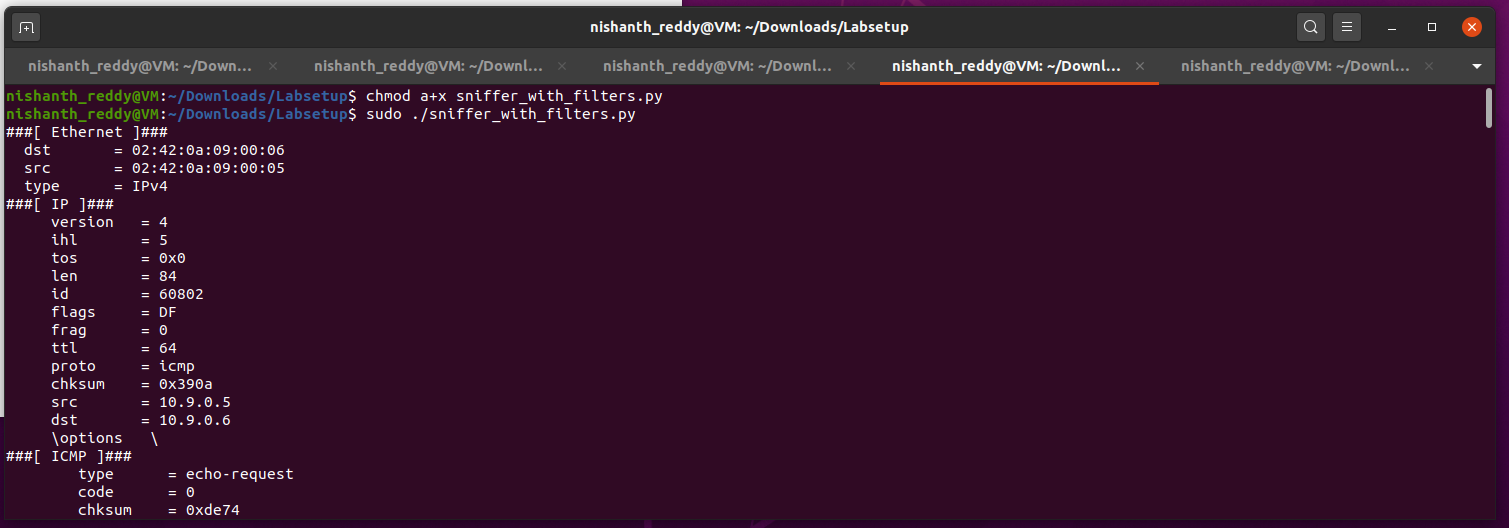
**Task 1.1B: Setting Filters for Sniffing**

**Step 1: Create the Sniffer Script**

1. I created a new script file named sniffer\_with\_filters.py using a text editor then I ran:

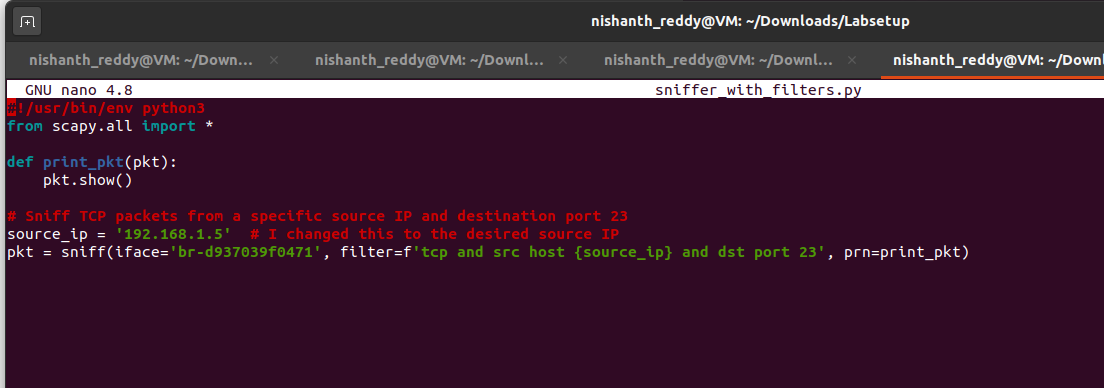
nano sniffer\_with\_filters.py

**Observe the Output:** As packets arrived, I observed that the details of the captured ICMP packets were displayed in the terminal.



**Step 2: Modify the Script for TCP Packets**

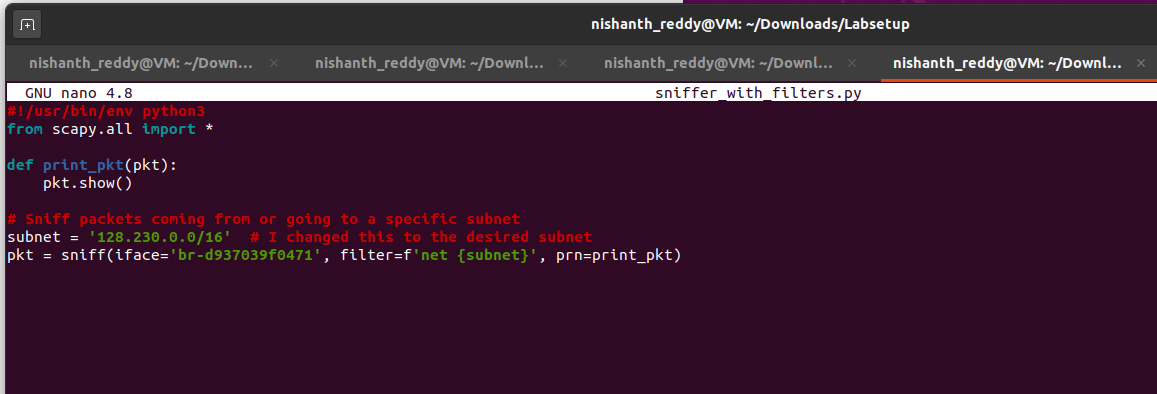
I modified the script to capture TCP packets from a specific source IP with a destination port of 23. I updated the code as follows:

 I saved the changes and ran the script again.

**Check for TCP Packets:** I monitored the output to ensure that only the specified TCP packets were captured.

**Step 3: Capture Packets from a Specific Subnet**

I updated the script once more to capture packets coming from or going to a specific subnet (e.g., 128.230.0.0/16):



1. **Observe the Output:** I checked the output to see that packet from or to the specified subnet were being captured successfully.

**3.2 Task 1.2: Spoofing ICMP Packets**

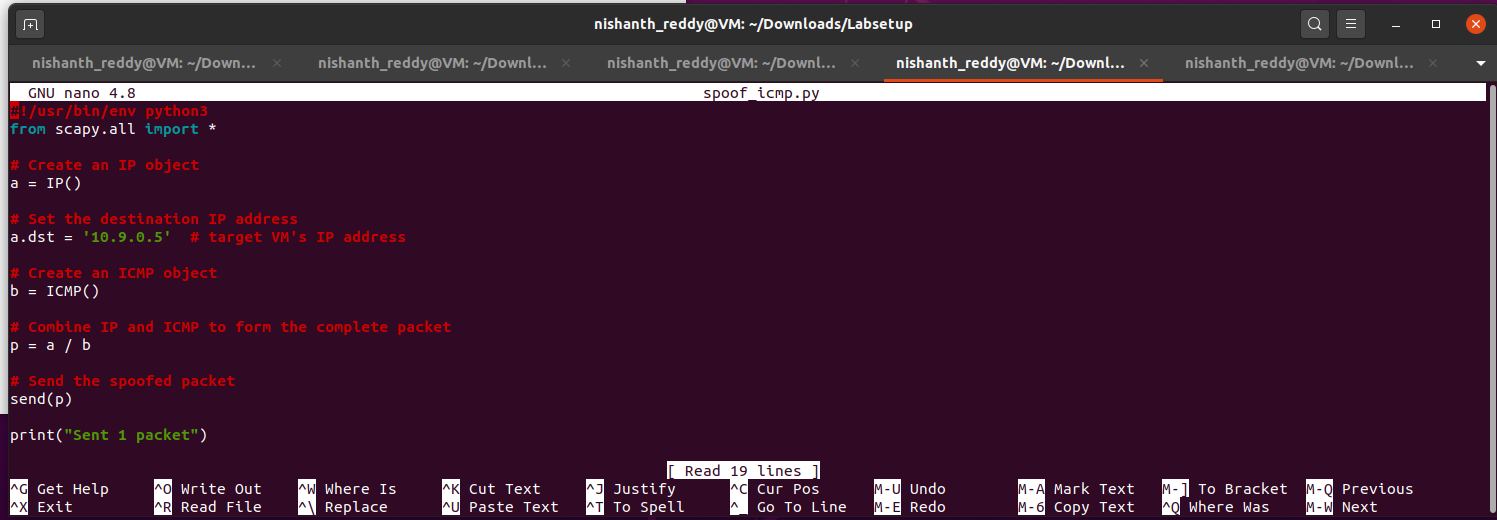
In this task, I learned to use Scapy as a packet spoofing tool that allows me to set the fields of IP packets to arbitrary values. The objective was to spoof IP packets with an arbitrary source IP address by creating ICMP echo request packets and sending them to another VM on the same network. I used Wireshark to observe whether my spoofed request would be accepted by the receiver. If accepted, an echo reply packet would be sent to the spoofed IP address.

**Step 1: Setting Up the Environment**

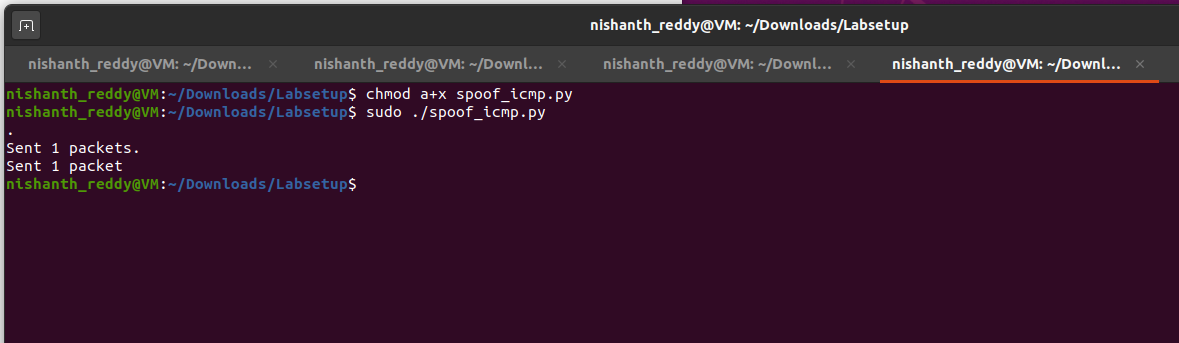
1. **Open a Terminal:** I began by opening a terminal in my attacker container where Scapy was installed.

**Step 2: Write the Spoofing Script**

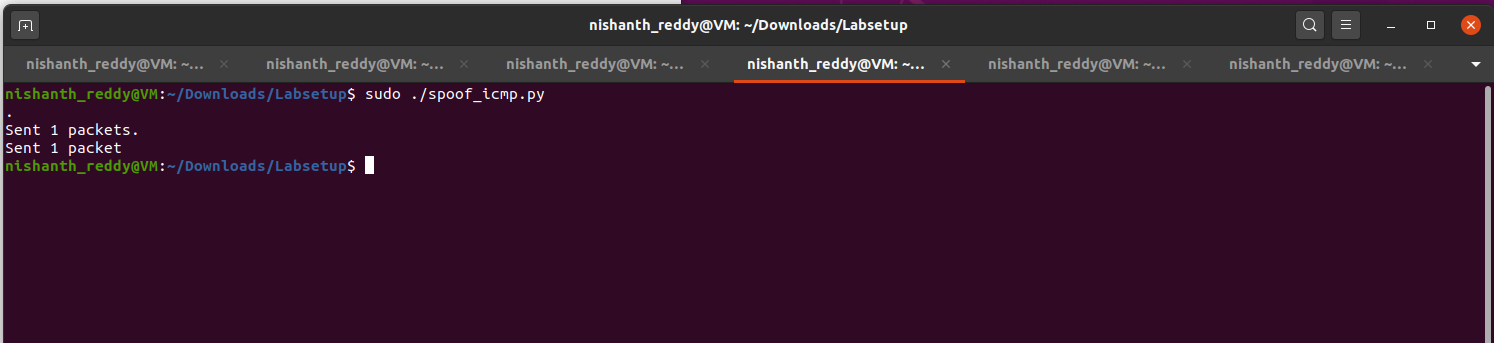
I created a new Python script named spoof\_icmp.py using a text editor



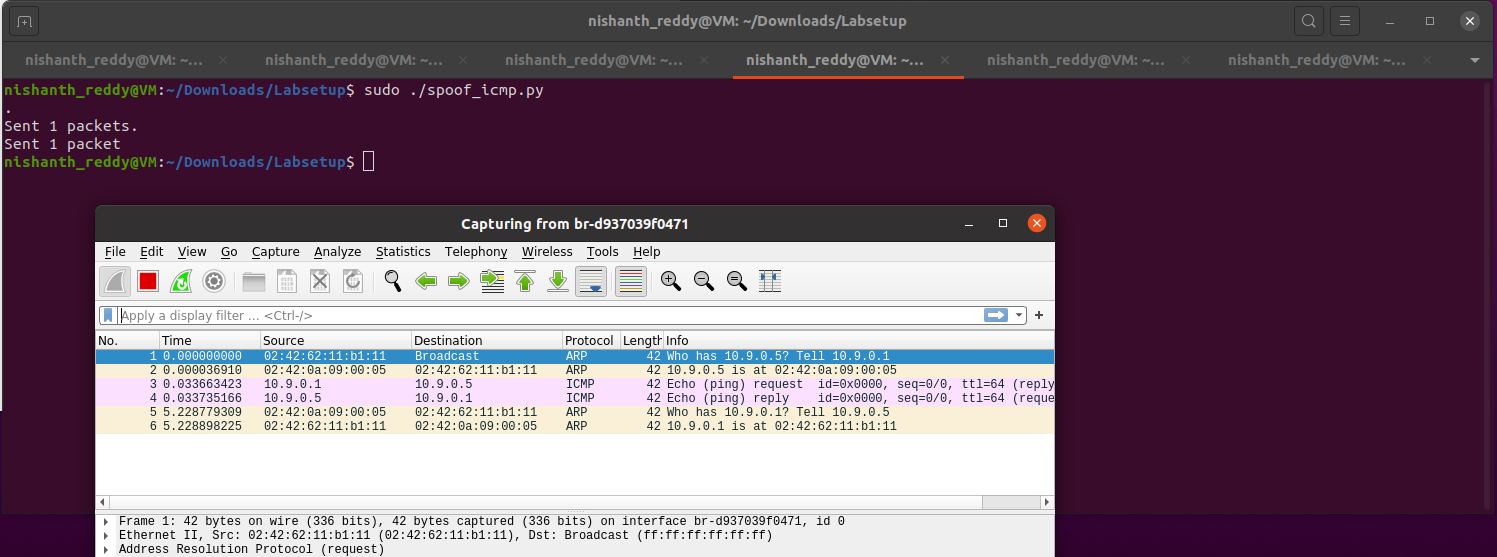
I then run the code



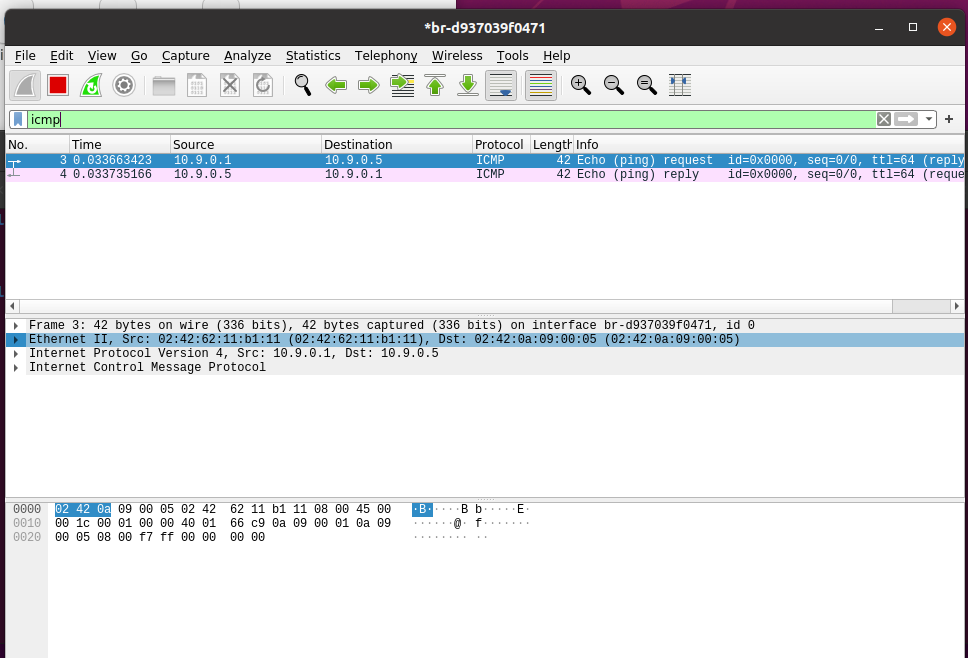
The terminal confirmed that 1 packet was sent. I ensured that Wireshark was running to capture and observe the network traffic.



I opened Wireshark to capture network packets and set the appropriate interface (e.g., br-d937039f0471) to monitor the traffic.



I applied a filter in Wireshark for ICMP packets:



1. **Check for Echo Replies:** After sending the spoofed packet, I looked for echo reply packets coming back to the spoofed source IP. This indicated that the target VM had received my request and responded.

**Observations:**

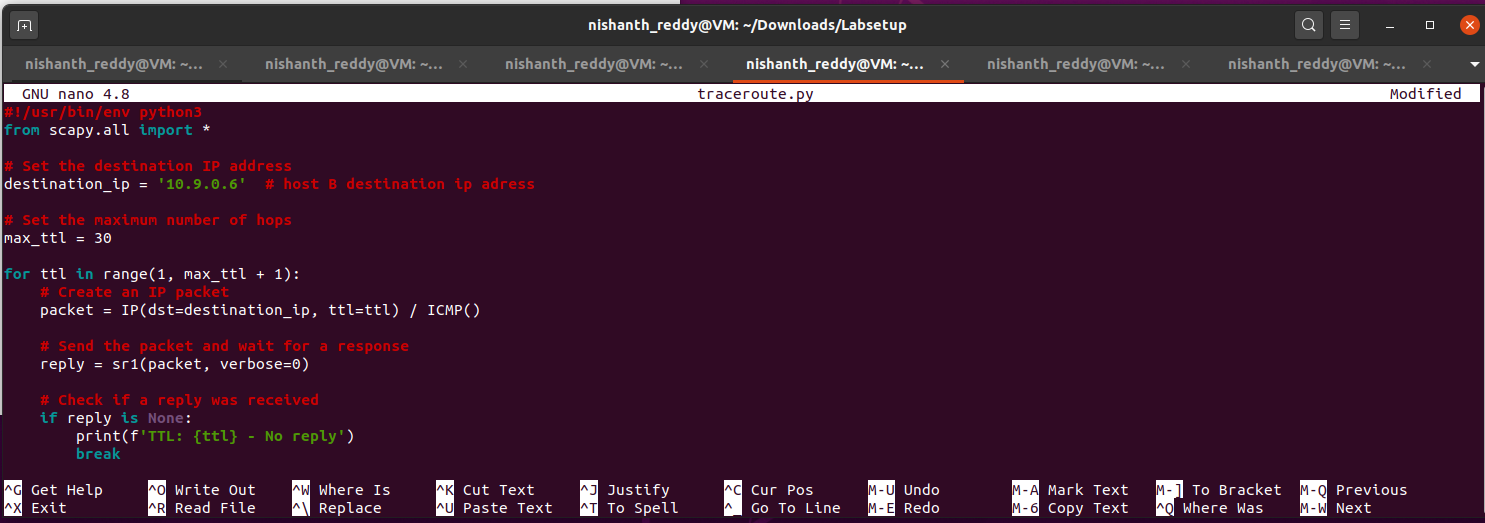
* When I sent the spoofed ICMP echo request, I observed in Wireshark that the packet appeared in the capture list.
* The target VM responded with an ICMP echo reply, which confirmed that the spoofed request was accepted.
* The echo reply packet was sent back to the spoofed source IP address I specified in the script.
* The response time for the echo reply was consistent with the expected latency for packets sent on the local network.

**3.3 Task 1.3: Traceroute**

The objective of this task was to use Scapy to estimate the distance, in terms of the number of routers, between my VM and a selected destination. This is essentially what the traceroute tool does. I wrote my own tool for this purpose. The idea was straightforward: I needed to send a packet (any type) to the destination, starting with its Time-To-Live (TTL) field set to 1. This packet would be dropped by the first router, which would send me an ICMP error message, indicating that the time-to-live had been exceeded. That is how I could get the IP address of the first router. I would then increase the TTL field to 2, send out another packet, and get the IP address of the second router. I repeated this procedure until my packet finally reached the destination.

**Step 1: Setting Up the Environment**

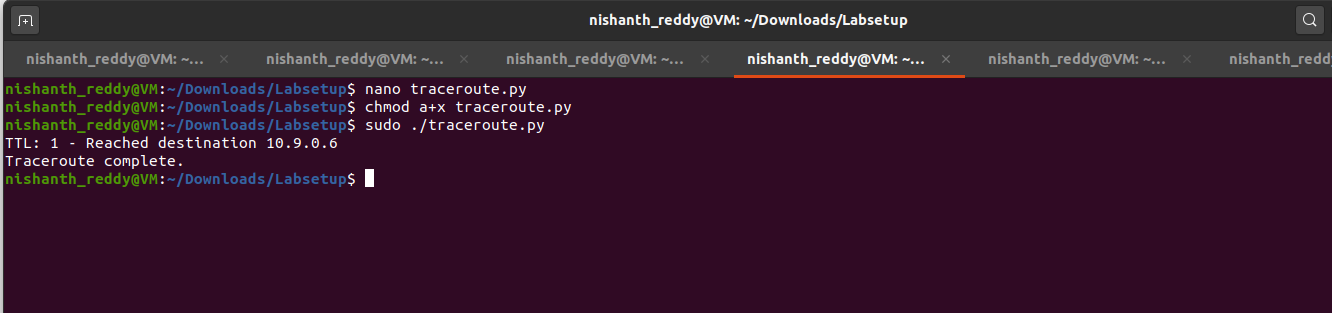
I created a new Python script named traceroute.py using a text editor:



I ran the script with root privileges to perform the traceroute:

chmod a+x traceroute.py

sudo ./traceroute.py



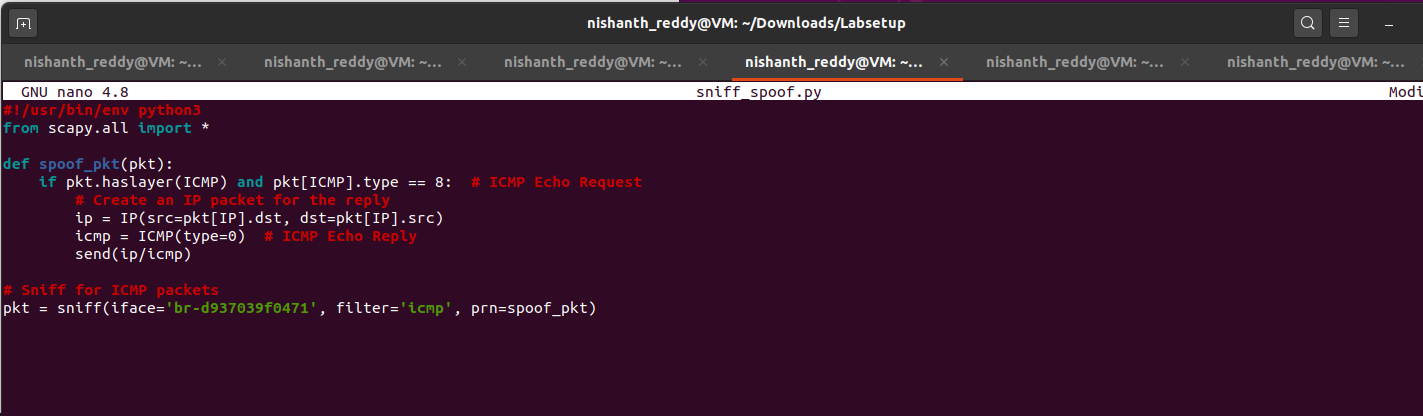
As the script executed, I monitored the output, which displayed the TTL value and the IP address of each router along the path. If I reached the destination, the script indicated that I had successfully completed the traceroute. I verified that the output displayed each hop’s IP address until the final destination was reached, demonstrating the path taken by the packets.

**Observations:**

* When the TTL was set to 1, I received an ICMP "Time Exceeded" message from the first router, which provided its IP address.
* As I incremented the TTL, each subsequent packet revealed the IP addresses of the routers along the path until the destination was reached.
* The final output indicated that I successfully traced the route to the destination, showing the total number of hops.

**3.4 Task 1.4: Sniffing and then Spoofing**

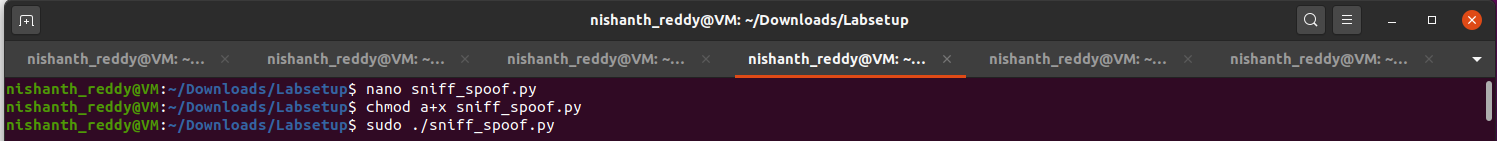
I created a new Python script named sniff\_spoof.py using a text editor:



I ran the script with root privileges to start monitoring the LAN for ICMP echo requests:

chmod a+x sniff\_spoof.py

sudo ./sniff\_spoof.py

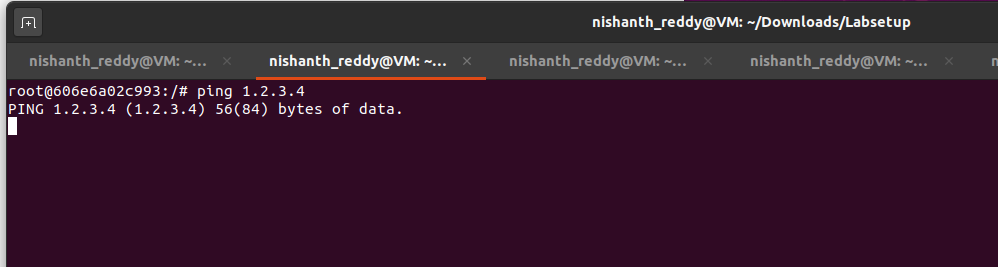


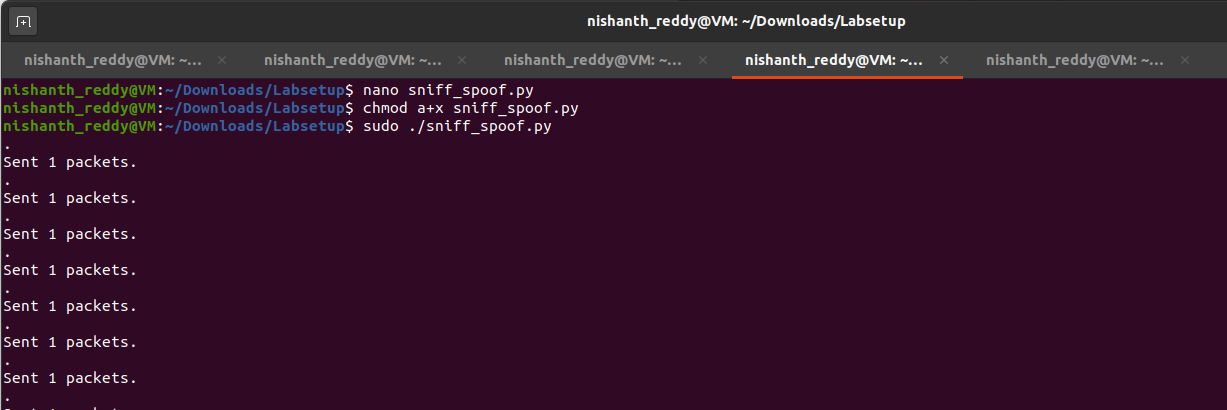
**Step 5: Test the Program by Pinging Different IPs**

Now, I would test the functionality by pinging different IP addresses from the user container.

I pinged the IP address 1.2.3.4, which is a non-existing host on the Internet:

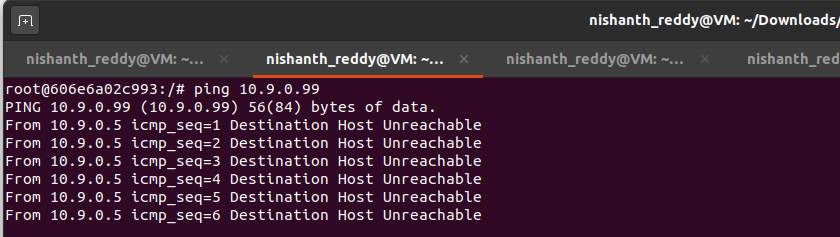
ping 1.2.3.4





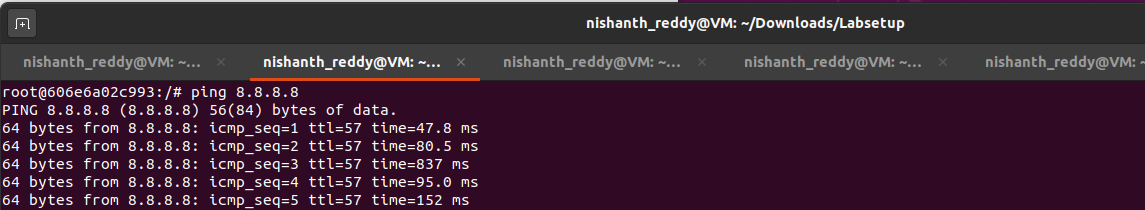
Next, I pinged the IP address 10.9.0.99, which is also a non-existing host on the LAN:

ping 10.9.0.99



Finally, I pinged the well-known IP address 8.8.8.8, which is an existing host on the Internet:

ping 8.8.8.8



**Step 6: Observations and Explanation of Results**

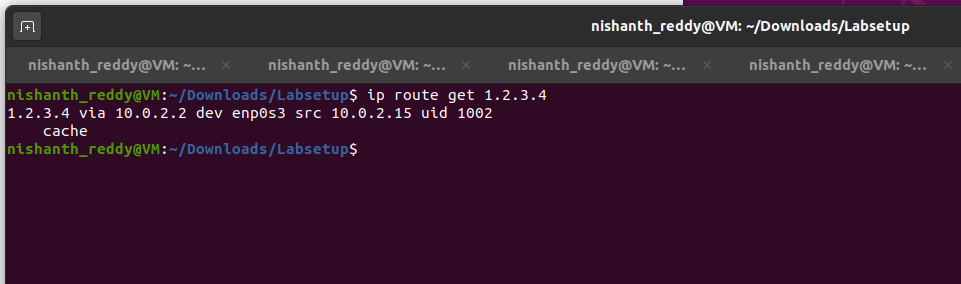
**Observations:**

* **Ping to 1.2.3.4:**  
  When I pinged this non-existing host, my program immediately responded with a spoofed ICMP echo reply. The ping command on the user container reported that the host was "alive."
* **Ping to 10.9.0.99:**  
  Similar to the first test, when I pinged this non-existing host on the LAN, my sniff-and-spoof program generated an echo reply, and the ping command indicated that the host was reachable.
* **Ping to 8.8.8.8:**  
  In this case, when I pinged the existing host, I also received a spoofed echo reply from my program. However, since 8.8.8.8 is a valid address, I also received the legitimate reply from Google’s DNS servers.

**Explanation of Results:**

* My program successfully responded to ICMP echo requests with spoofed replies for all three tests. This demonstrated that regardless of whether the target host was alive, the ping program always received a reply.
* Understanding how the ARP protocol works was important here. Since my program operated on the same LAN, it could intercept ICMP requests directed at any IP address and send replies.
* The use of routing was also significant. To confirm the route to a specific destination, I could use the command: **“ip route get 1.2.3.4”**

This command showed me how my packets would be routed toward the destination IP, confirming that all replies were generated locally on the network.

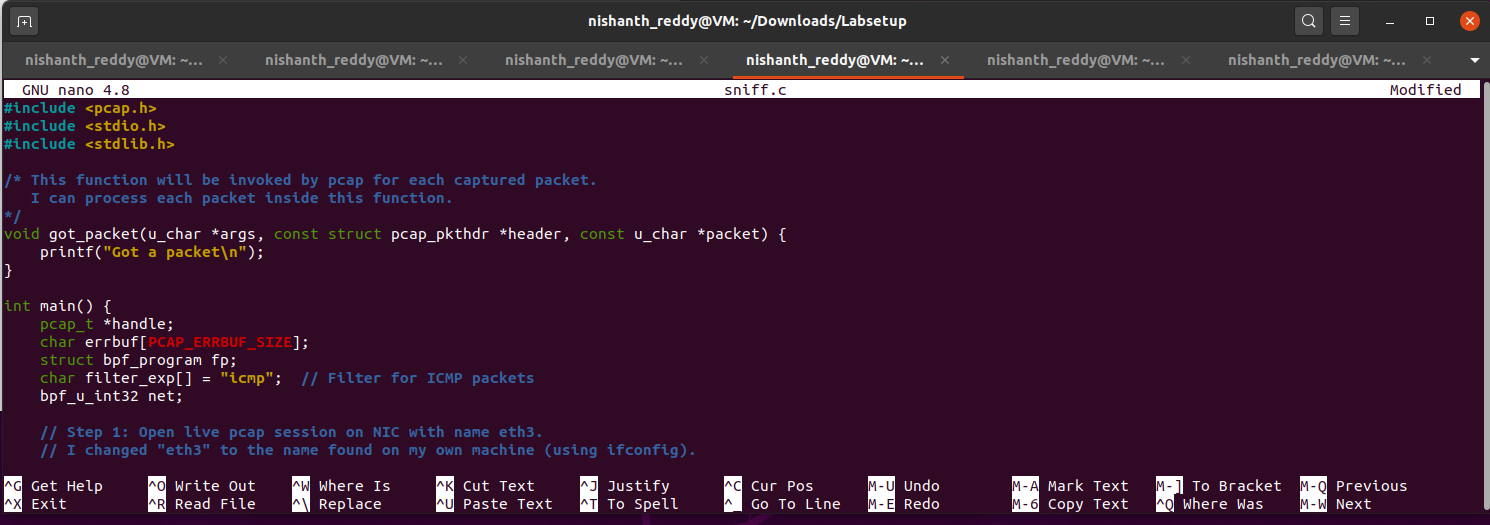


**4 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets**

For this set of tasks, I compiled the C code inside the host VM and then ran the code inside the container. I used the docker cp command to copy files from the host VM to the container. Here’s how I did it:

**4.1 Task 2.1: Writing a Packet Sniffing Program**

Sniffer programs can be easily written using the pcap library. With pcap, the task of sniffers becomes invoking a simple sequence of procedures in the library. The following sample code demonstrates how to write a simple sniffer program using pcap:

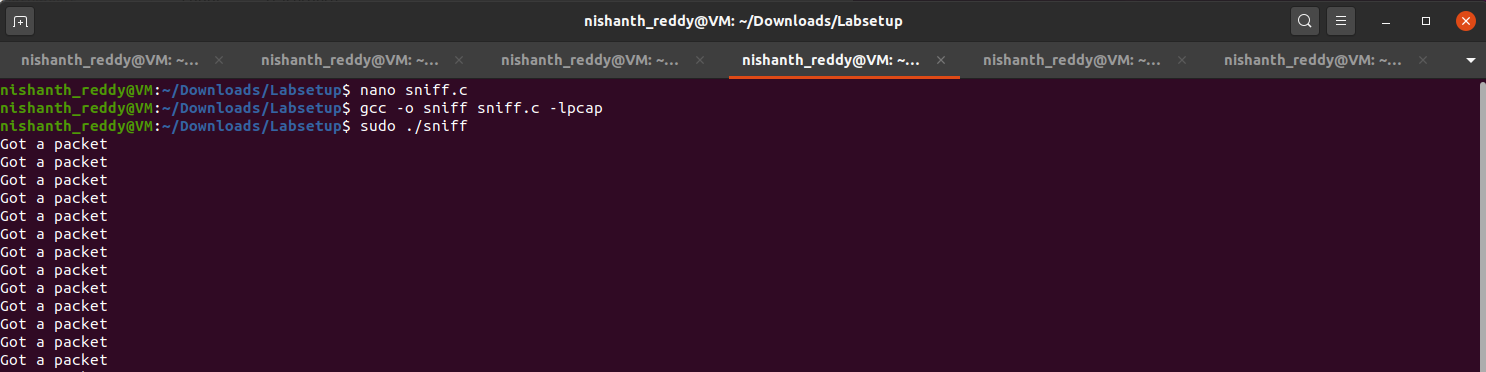


**Compiling the Sniffer Program**

To compile and run the program, I used the following command, ensuring to link against the pcap library:

gcc -o sniff sniff.c -lpcap

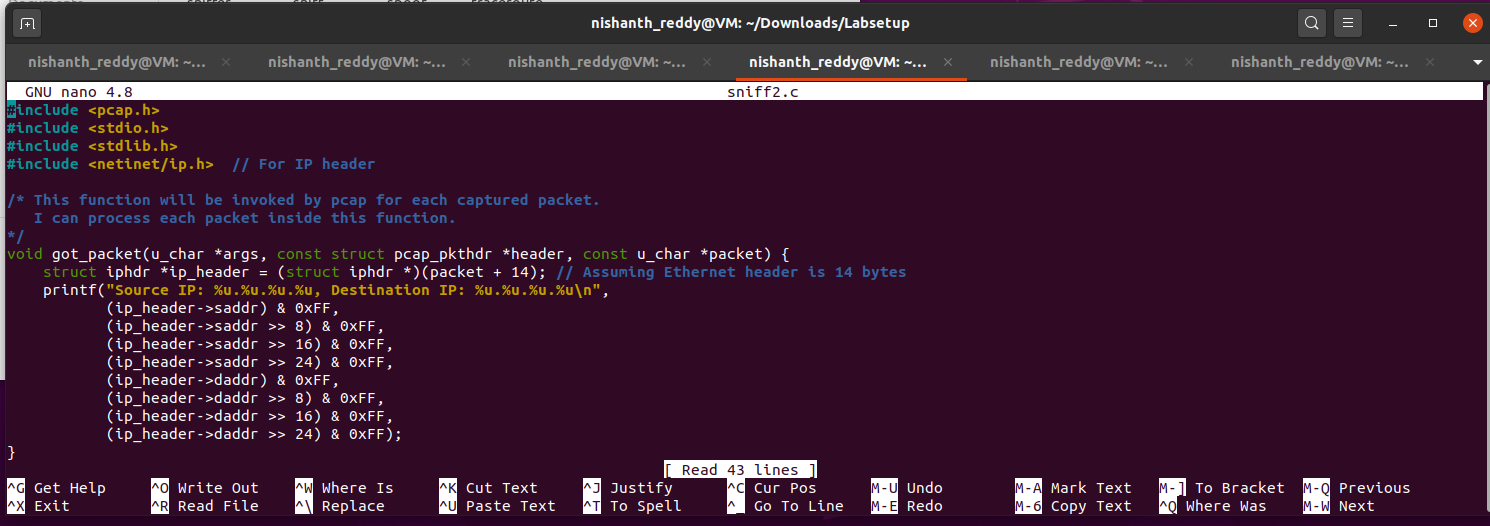
sudo ./sniff

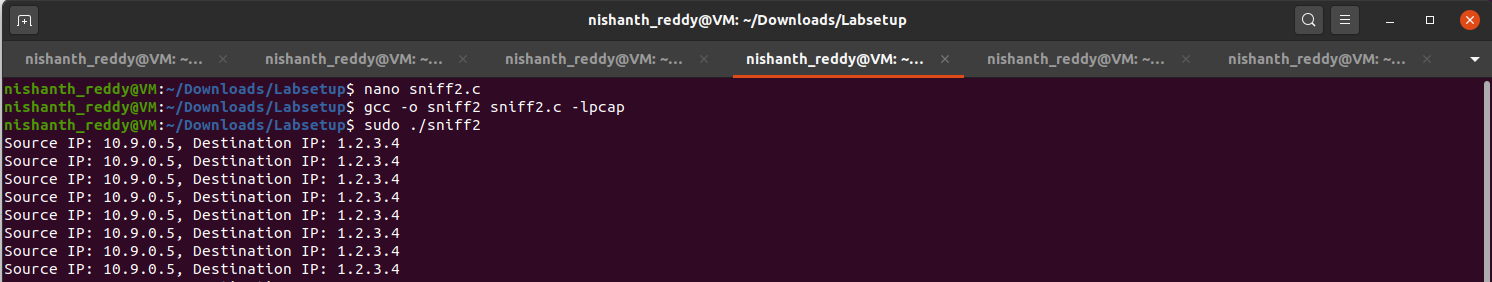


**Task 2.1A: Understanding How a Sniffer Works**

**Step 1: Writing the Sniffer Program**

I created a new file named sniff2.c.

I compiled the program using “gcc -o sniff2 sniff2.c -lpcap” then executed the sniffer program to capture packets “sudo ./sniff2”



**Questions and Answers**

1. **Question 1:** Describe the sequence of the library calls that are essential for sniffer programs.

**Answer:** The essential sequence of library calls for sniffer programs using pcap typically involves:

* + **Opening a live pcap session** using pcap\_open\_live(), specifying the network interface, buffer size, and timeout.
  + **Compiling a filter expression** into BPF pseudo-code with pcap\_compile().
  + **Setting the filter** on the opened session using pcap\_setfilter().
  + **Starting the packet capture loop** using pcap\_loop(), which invokes a callback function for each captured packet.
  + **Closing the session** with pcap\_close() when done.

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

**Answer:** Root privileges are required to run a sniffer program because capturing packets directly from the network interface involves raw socket access, which is restricted to prevent unauthorized access to network traffic. If the program is executed without root privileges, it fails at the pcap\_open\_live() function, returning an error indicating insufficient permissions to access the network interface.

1. **Question 3:** Turn on and turn off the promiscuous mode in your sniffer program.

**Answer:** In my sniffer program, I turned on promiscuous mode by setting the third parameter of pcap\_open\_live() to 1, which allows the program to capture all packets on the network, not just those addressed to it:

**handle = pcap\_open\_live("br-d937039f0471", BUFSIZ, 1, 1000, errbuf);**

To turn off promiscuous mode, I set this parameter to 0:

**handle = pcap\_open\_live("br-d937039f0471", BUFSIZ, 0, 1000, errbuf**);

I demonstrated the difference by running the sniffer program in both modes. When promiscuous mode was ON, I could see all packets on the network, while in OFF mode, I only saw packets addressed to my interface.

To check whether an interface’s promiscuous mode is on or off, I used the following command:

**ip -d link show dev br-f2478ef59744**

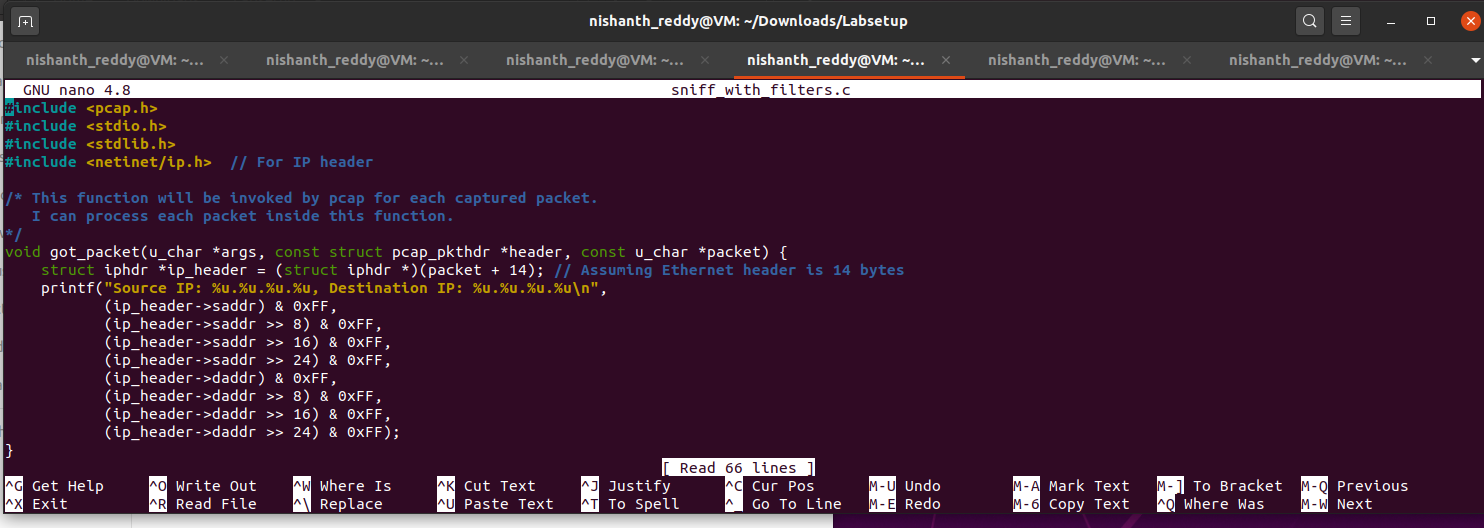
The output indicated whether promiscuity was set to 1 (ON) or 0 (OFF).

**Task 2.1B: Writing Filters**

In this task, I wrote filter expressions for my sniffer program using the pcap library to capture specific types of packets. Below are the steps I followed, along with the necessary code snippets and commands.

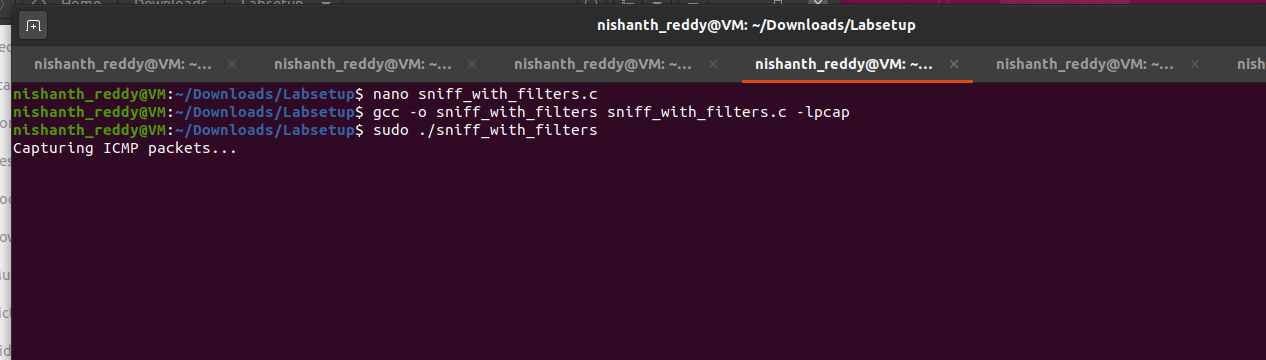
**Step 1: Writing the Sniffer Program**

I started by creating a new C source file named sniff\_with\_filters.c using a text editor:

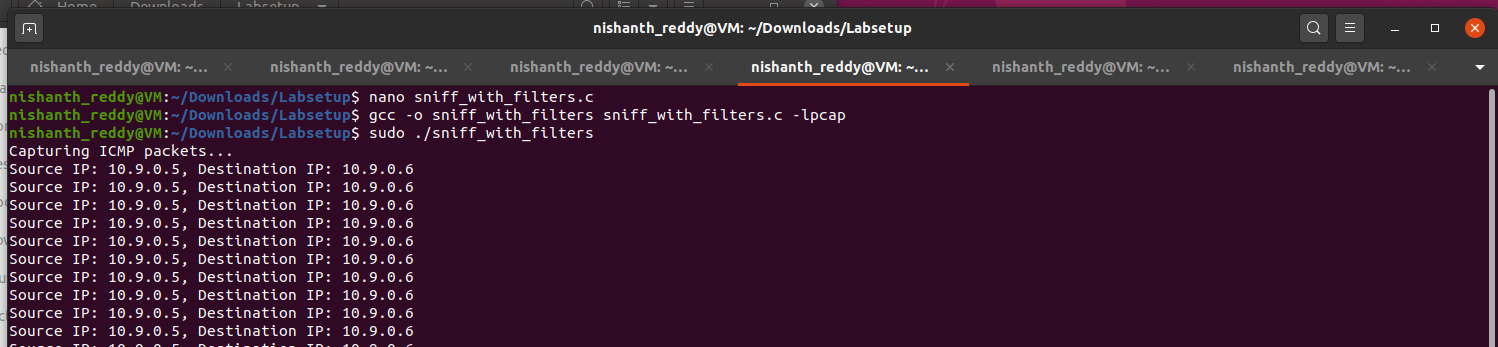


I compiled the program using the command **“gcc -o sniff\_with\_filters sniff\_with\_filters.c -lpcap”**

then executed the sniffer program to capture ICMP packets between the specified hosts with the command **“sudo ./sniff\_with\_filters”**



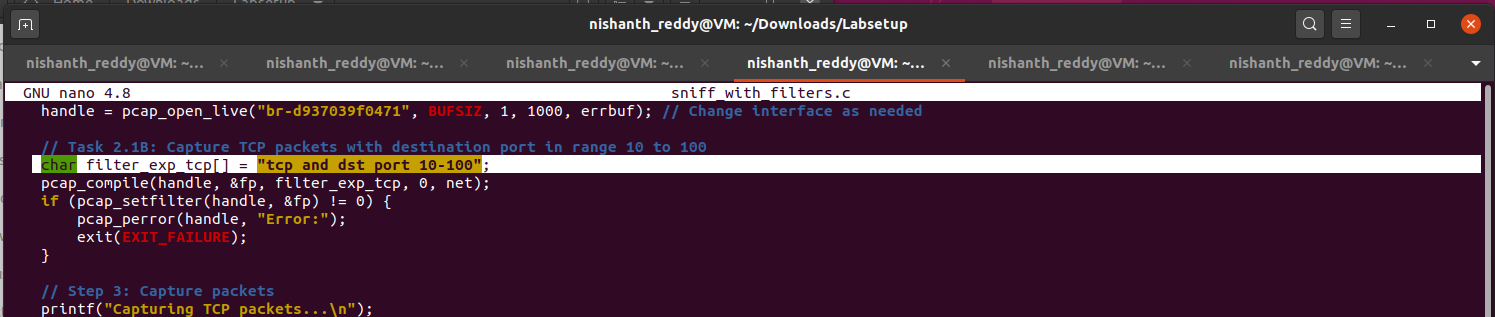
While the program was running, I pinged 10.9.0.6 from 10.9.0.5 to generate ICMP packets.



The above screenshot demonstrates that the program captured the ICMP packets and displayed their source and destination IP addresses.

**Running the Sniffer Program for TCP Packets**

After stopping the previous instance, I ran the program again to capture TCP packets with a destination port number in the range from 10 to 100:

****

I created TCP traffic to ensure packets were sent within the specified destination port range.

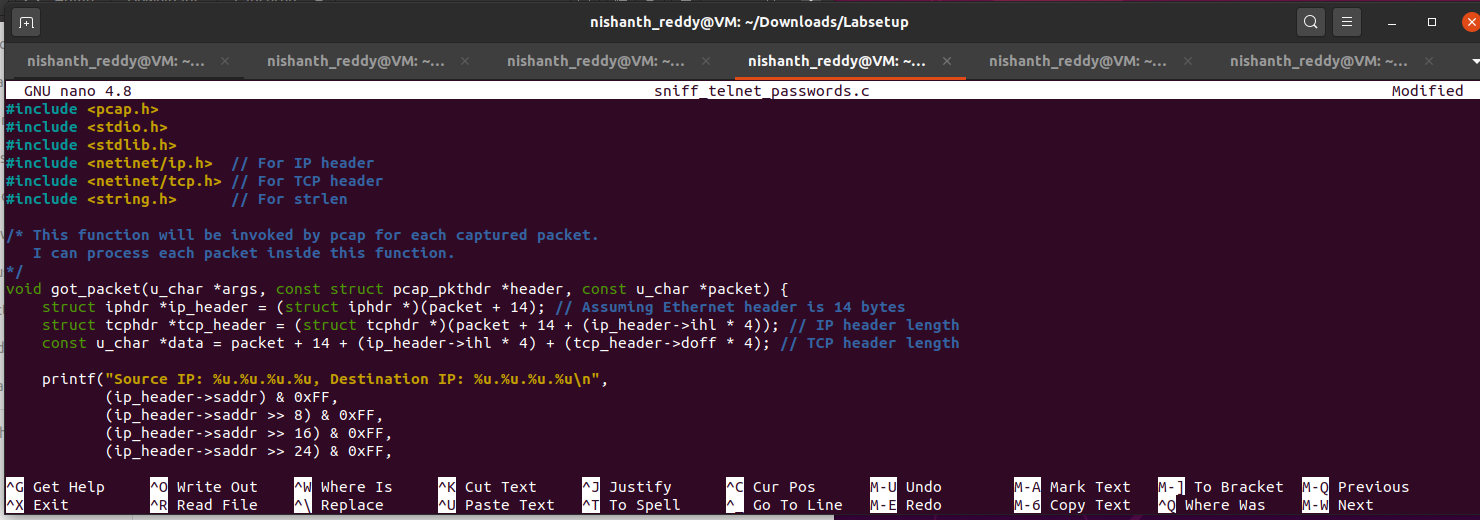
**Observations**

Through these steps, I wrote filter expressions to capture specific packets, compiled my sniffer program, and successfully demonstrated the capturing of both ICMP and TCP packets based on the filters applied.

**Task 2.1C: Sniffing Passwords**

In this task, I modified my sniffer program to capture passwords when someone is using Telnet on the monitored network. I named the code sniff\_telnet\_passwords.c. Below are the steps I followed, along with the complete code.

I started by creating a new C source file named sniff\_telnet\_passwords.c using a text editor:

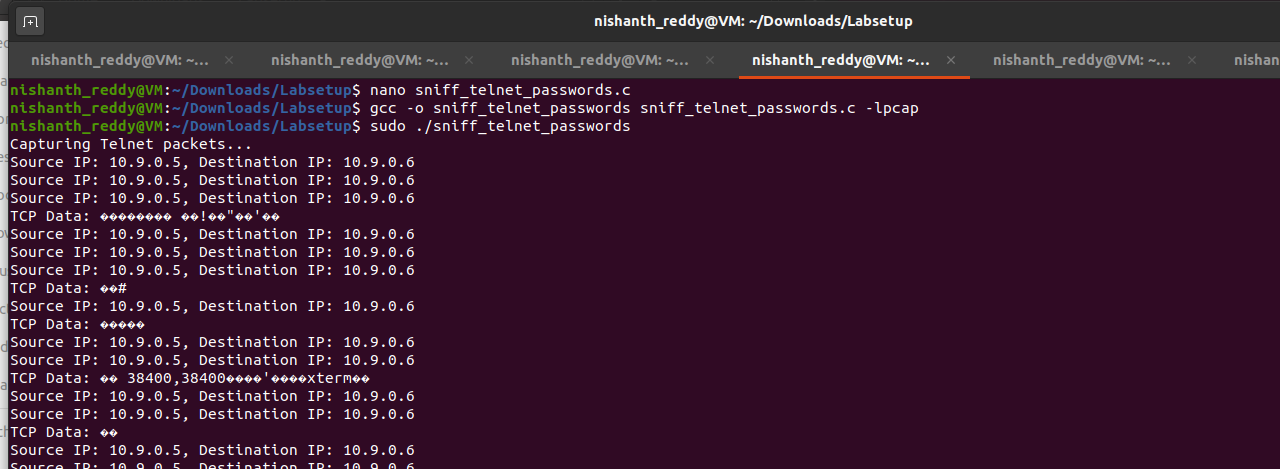


I compiled the program using the following command:

**gcc -o sniff\_telnet\_passwords sniff\_telnet\_passwords.c -lpcap**

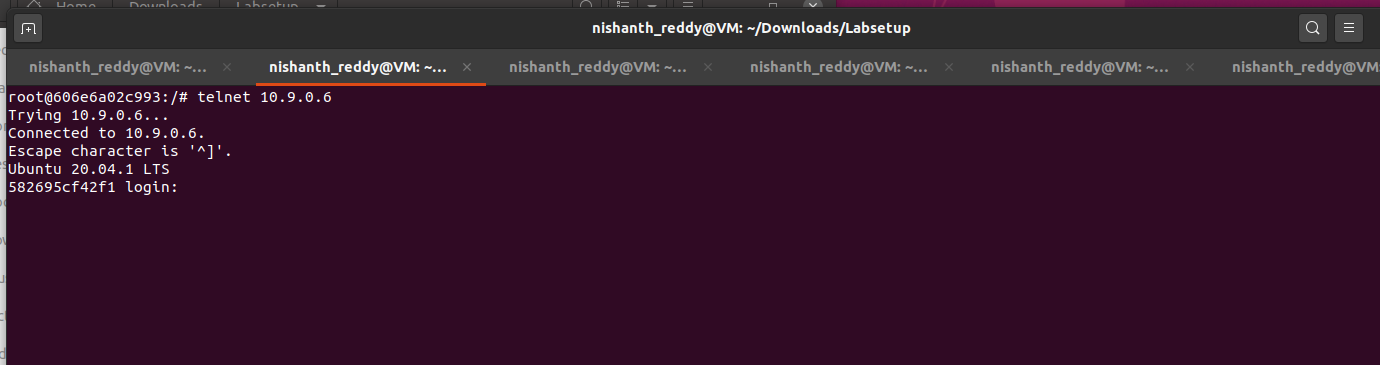
I executed the sniffer program to capture Telnet packets, which may include passwords:

**sudo ./sniff\_telnet\_passwords**

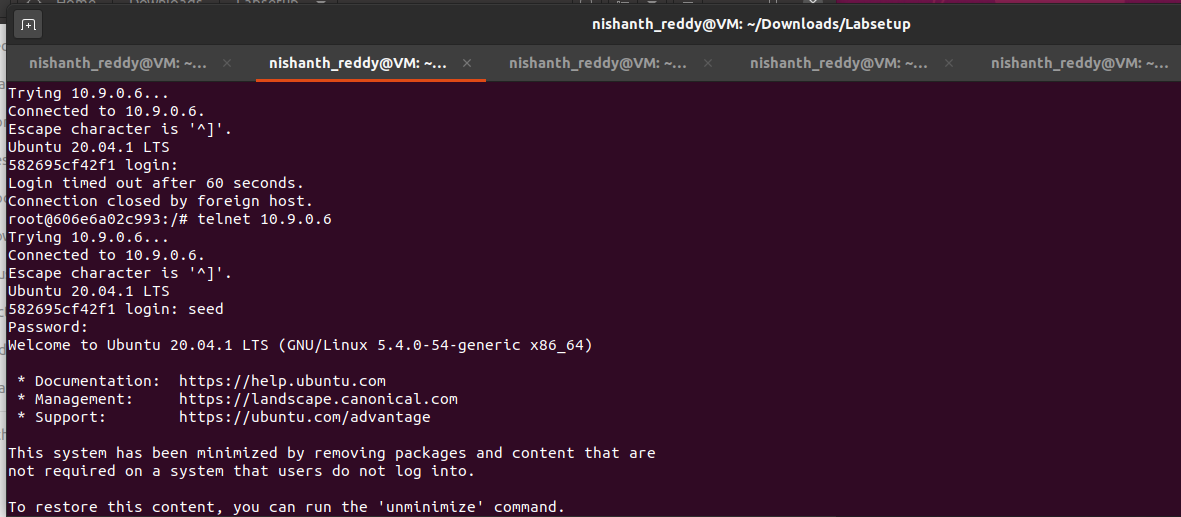


**Establishing a Telnet Session:** On another terminal or machine, I opened a Telnet session to a target machine. For example:

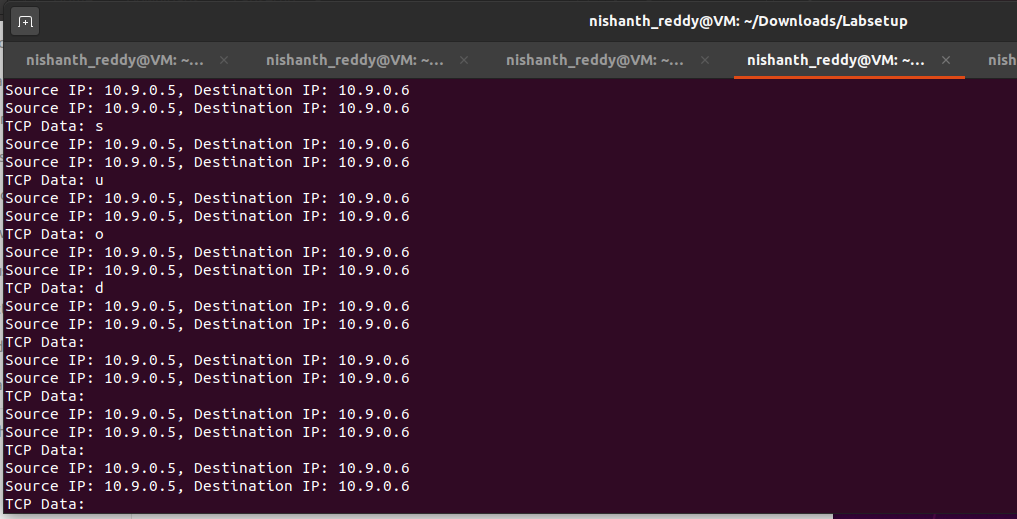
telnet 10.9.0.6



**Logging In:** I logged in with a username and password.



As I entered the password, the sniffer program captured the packets, and I observed the output in the terminal where the sniffer was running.



**Analyzing Captured Data**

**Reviewing Captured TCP Data:** After the Telnet session, I examined the output from the sniffer. I manually marked the section of the output where the password was visible in the captured TCP data.

**Observation**

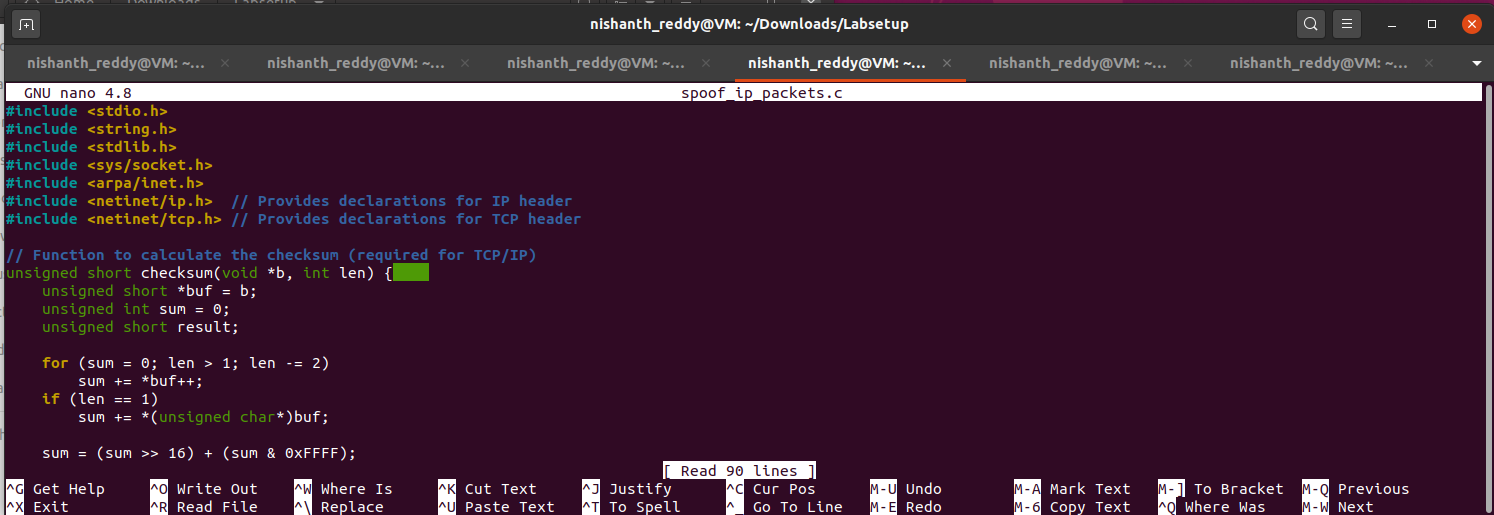
By following these steps, I successfully modified my sniffer program to capture and display the data part of TCP packets, specifically focusing on Telnet traffic to sniff passwords. I provided screenshots of the output showing the captured data, including the passwords.

################################

**Task 2.2A: Writing a Spoofing Program**

In this task, I wrote a packet spoofing program in C using raw sockets to create and send spoofed IP packets. The program allows me to set arbitrary fields in the packet headers.

I started by creating a new C source file named spoof\_ip\_packets.c using a text editor:

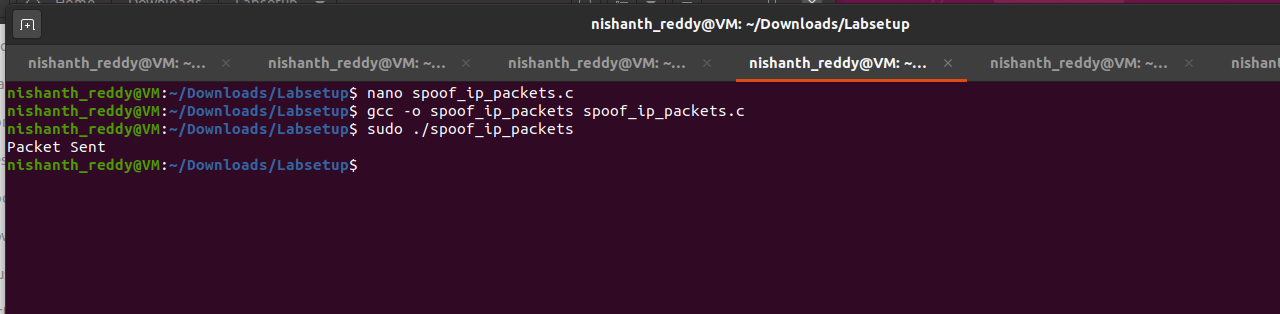


I compiled the program using the following command:

**gcc -o spoof\_ip\_packets spoof\_ip\_packets.c**

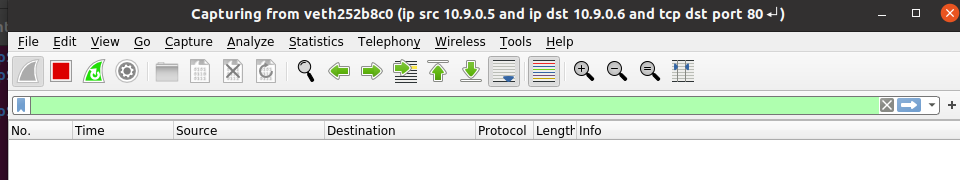
I executed the spoofing program with root privileges to send out the spoofed IP packet:

**sudo ./spoof\_ip\_packets**



I opened Wireshark to monitor the network traffic and verify that my spoofed packet was sent successfully. I applied a filter to capture only packets related to the spoofed source and destination IPs:

ip.src == 10.9.0.5 && ip.dst == 10.9.0.6



After running the spoofing program, I checked Wireshark for the captured packet and ensured that it showed the correct source and destination IP addresses, confirming the success of the spoofing attempt.

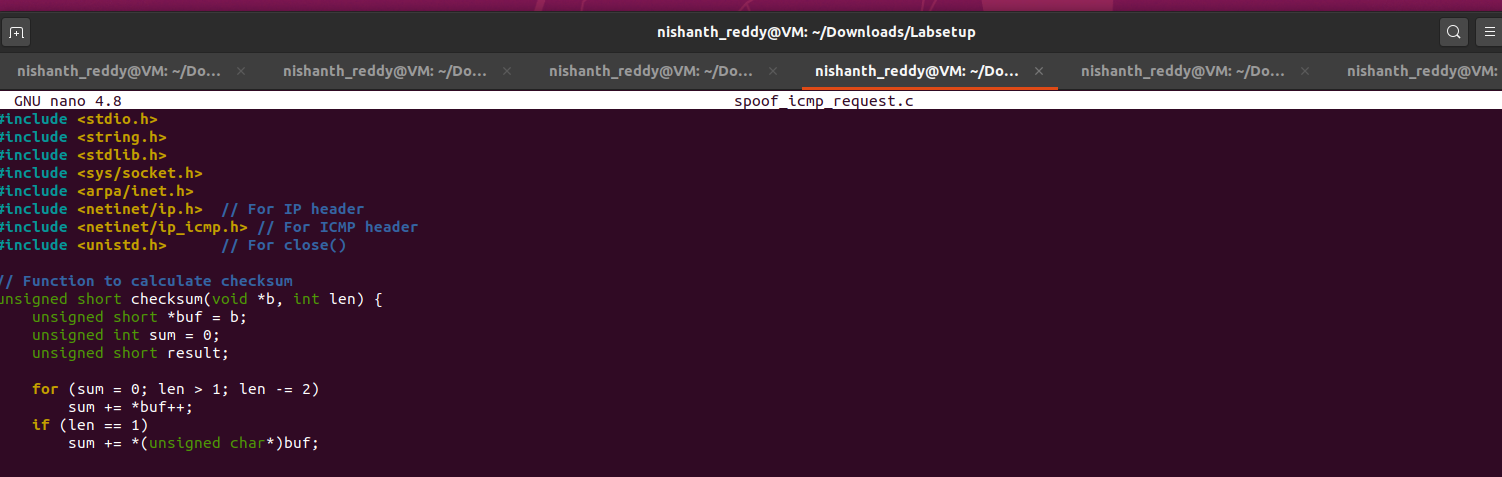
**Conclusion**

By following these steps, I successfully wrote a packet spoofing program that constructs and sends a spoofed IP packet using raw sockets. I provided evidence using Wireshark to demonstrate that my program successfully sent out spoofed IP packets.

**Task 2.2B: Spoof an ICMP Echo Request**

In this task, I created a program to spoof an ICMP echo request packet using another machine’s IP address as the source IP. The spoofed packet was sent to a live remote machine on the Internet, and Wireshark was used to capture the echo reply.

I named the file spoof\_icmp\_request.c and used a text editor to create the program:



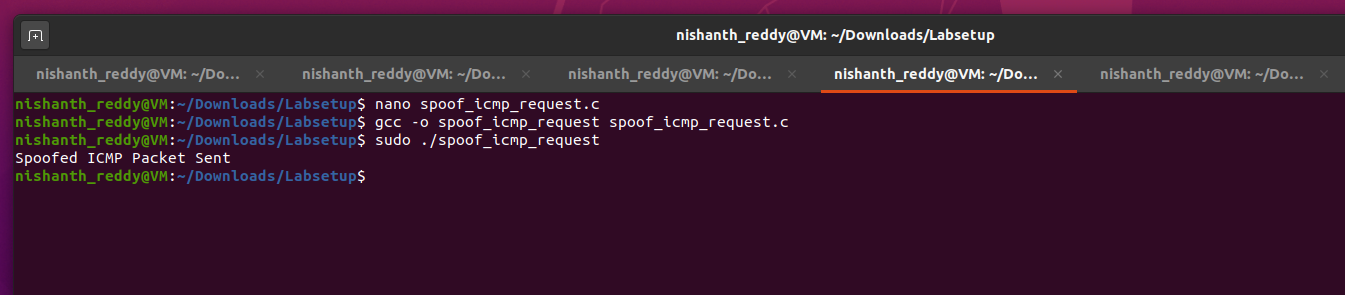
**Step 2: Compile the Program**

I used the following command to compile the ICMP spoofing program:

**gcc -o spoof\_icmp\_request spoof\_icmp\_request.c**

I executed the program with root privileges to send the spoofed ICMP echo request packet:

**sudo ./spoof\_icmp\_request**

****

**Step 4: Verifying the Spoofed Packet with Wireshark**

I opened Wireshark and started capturing packets on the network interface. I applied the following **capture filter** to see only the relevant packets:

icmp and ip.src == 8.8.8.8

This filter captures the ICMP echo reply coming from the remote machine (8.8.8.8) back to my machine.

**Check for ICMP Echo Reply:** If the spoofing was successful, I would see an **ICMP Echo Reply** from 8.8.8.8 (the remote machine) to the spoofed source IP (10.9.0.5).

**Questions**

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

* **Answer:** No, the IP packet length field should accurately reflect the actual size of the packet, including headers and payload. If you set it to an arbitrary value that doesn’t match the packet size, the receiver will either drop the packet or experience issues while processing it.

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

* **Answer:** Yes, when using raw sockets, you must manually calculate the checksum for the IP header. The operating system will not do this for you. In the program, I used a function to calculate and set the IP header 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?**

* **Answer:** Raw socket operations require root privileges because they provide direct access to the network, allowing the user to craft and send arbitrary packets. Without root privileges, the program will fail at the socket() call with a permission error (socket() error: Operation not permitted). This is to prevent unprivileged users from sending potentially harmful or spoofed packets.

**Task 2.3: Sniff and then Spoof**

In this task, I wrote a **Sniff and then Spoof** program, which listens for ICMP echo requests on the network and immediately spoofs an ICMP echo reply. This makes the requesting machine believe that the target machine (IP X) is alive, regardless of whether it is.

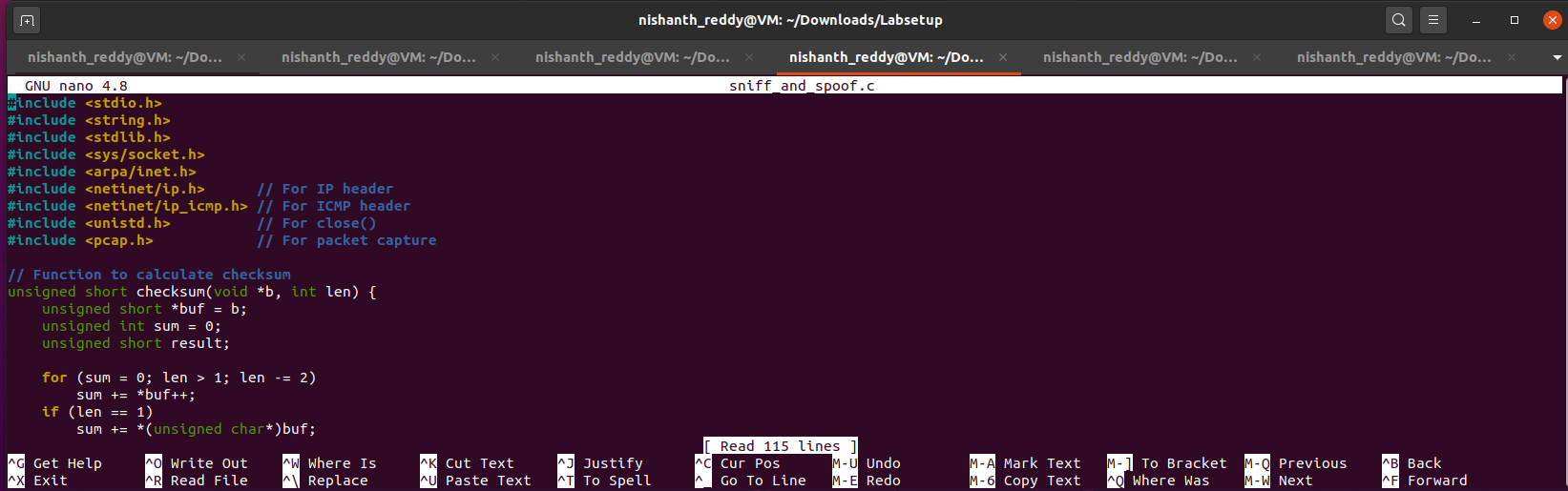
**Step 1: Understanding the Sniff-and-Spoof Program**

The program combines two main functionalities:

1. **Sniffing**: It captures ICMP echo request packets from the network.
2. **Spoofing**: Once it captures a request, it sends a spoofed ICMP echo reply back to the sender.

This needs to be run on an attacker machine, while another machine on the same LAN sends ICMP echo requests (using ping).

I wrote the following code to sniff ICMP echo requests and immediately send a spoofed ICMP echo reply.



I compiled the program using the following command:

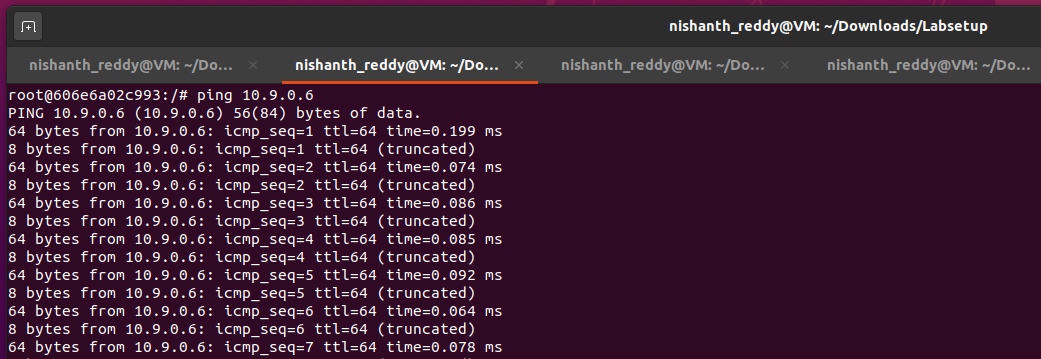
gcc -o sniff\_and\_spoof sniff\_and\_spoof.c -lpcap

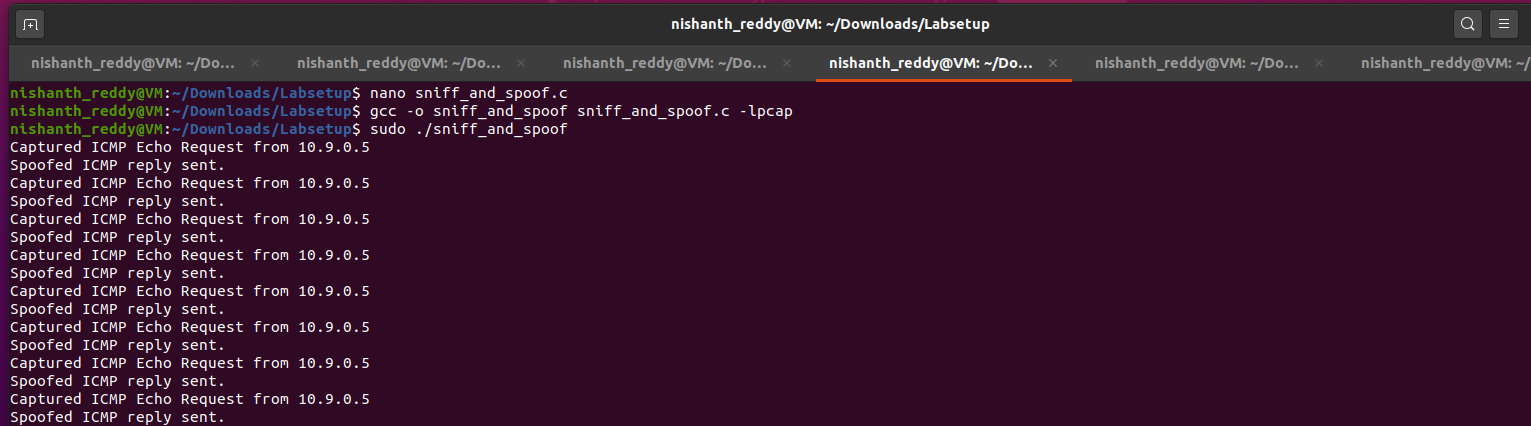
I executed the program with root privileges:

sudo ./sniff\_and\_spoof

I pinged an arbitrary IP address (10.9.0.6) from host A machine on the LAN:

ping 10.9.0.6



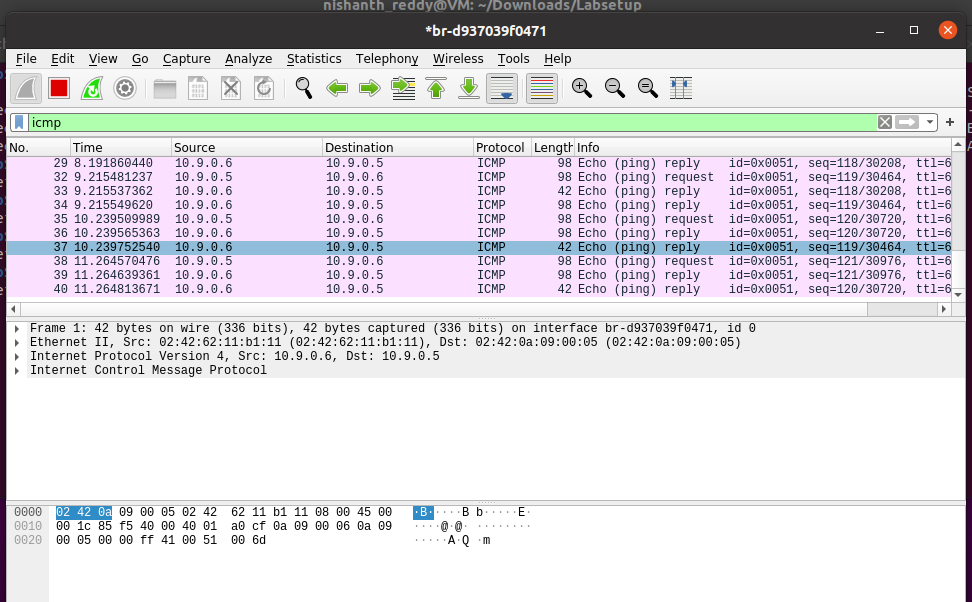


**Step 5: Verifying with Wireshark**

1. **Monitor Traffic:** I opened Wireshark and applied the following capture filter to monitor ICMP echo request and reply packets:

**icmp**

I could see the ICMP echo requests being sent by the ping command on **Machine A** and the spoofed ICMP replies sent by my program.



**Observation**

The sniff-and-spoof program successfully captured ICMP echo requests and immediately sent spoofed replies. This made **Machine A** believe that the target IP was alive, regardless of its actual status.

**Conclusion**

In this lab, I learnt how to engage in packet sniffing and spoofing using raw sockets, which are very important in the communication networks, use of raw sockets allowed me to build packets and send the packets with the headers, I was also able to manipulate protocols such as the ICMP, TCP, UDP directly on the network layer. I learned how to sniff network traffic and how to forge IP addresses and answer to ICMP-echo, making open networks are an easy target. Furthermore, I enhanced my insights about network and host byte orders, about how operating systems handle raw sockets. In conclusion, all activities of this lab helped in gaining insight about network security methods and threats related to packet manipulation and, therefore, underscored the importance of making networks safe and closely observing their status.