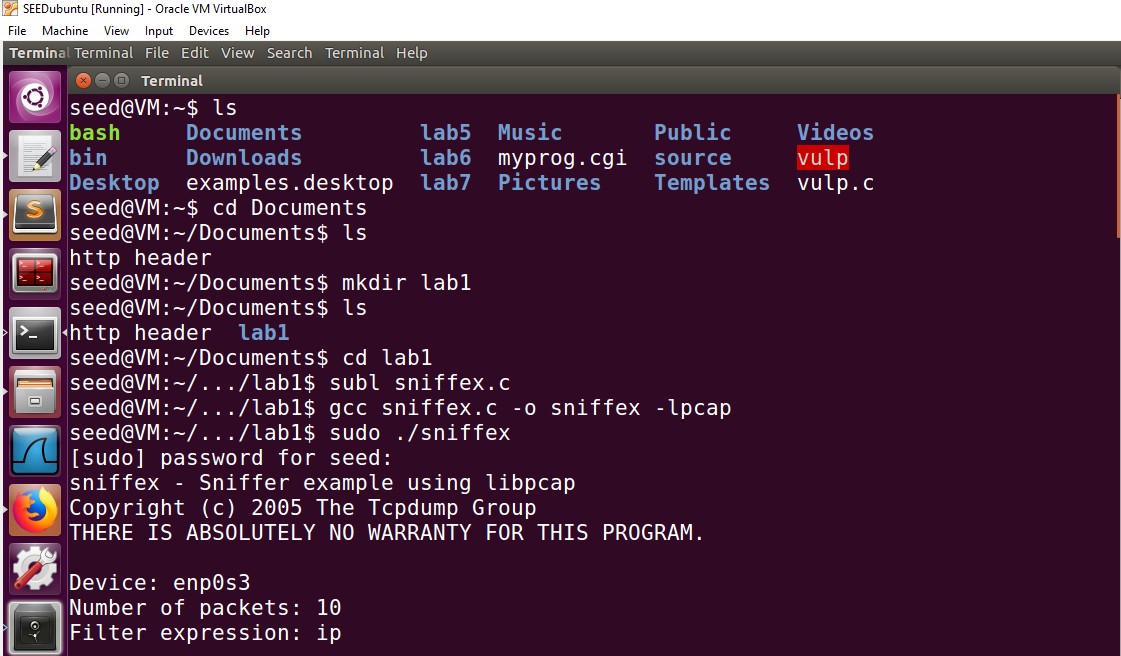
**SEED Labs – Packet Sniffing and Spoofing Lab :-**

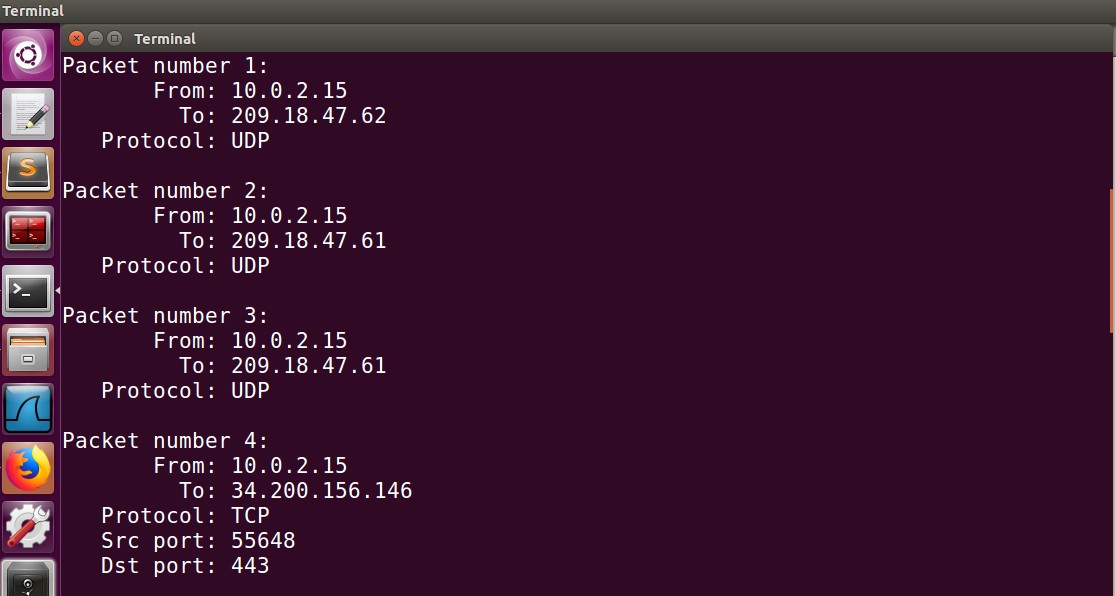
Contributors: 206696163 Qusai + 317310779 Eli

**Task 1: Writing Packet Sniffing Program**

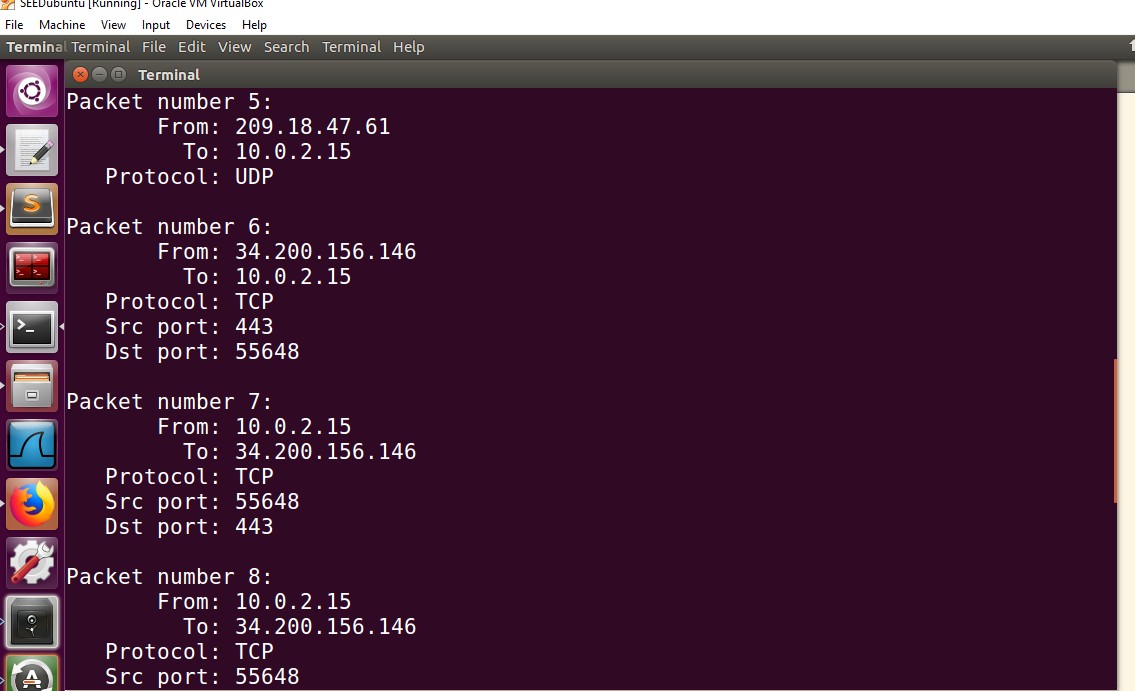
**Task 1.a: Understanding Sniffex**

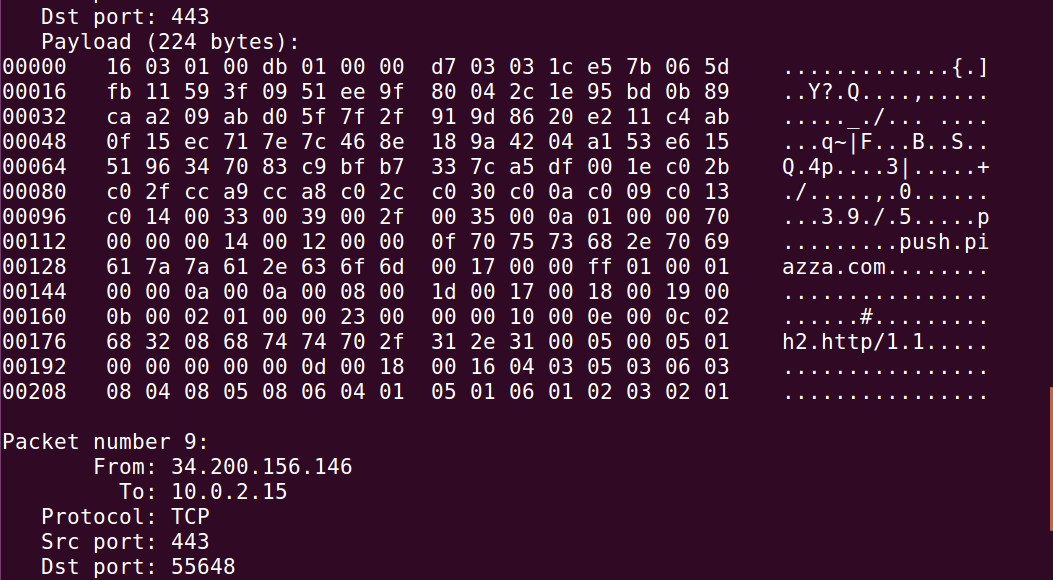


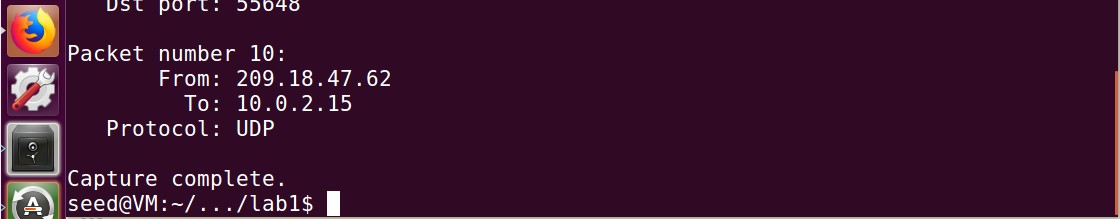
**1**



**2**







**3**

First let try to run our sniffex code with root privilege. We successfully capture 10 packets after compilation.

# Problem 1: Here are the steps to the sequence of library calls essential for sniffer programs

# Setting up Device:

**pcap sets the device on its own. If this fails, it saves the error message into errbuf. pcap\_lookupdev(errbuf) can be used to find a device to sniff on.**

# Opening the device for sniffing:

pcap uses pcap\_open\_live() to open session on a device we will be sniffing on. The format of the statement is as follows:

# pcap\_t \*pcap\_open\_live(char \*device, int snaplen, int promisc, int to\_ms, char

**\*ebuf)**

* + char \*device: specifies the device we are sniffing on.
  + snaplen: specifies max number of bytes to be captured by pcap.
  + promisc: specifies if Promiscous mode is on or not.
  + to\_ms: this value is non-zero as this is the read time out in milliseconds.
  + char \*ebuf: stores error messages.

Note: Promiscous Mode is used to sniff all network traffic and not just the traffic to, from, or routed through a specific host.

# Filtering Traffic:

We perform filtering using two functions in pcap library:

pcap\_compile() is used to compile the filter expression stored in a regular string. pcap\_setfilter() is used to set the compiled filter to determine what the program sniffs.

Here’s the prototype for them:

# int pcap\_compile(pcap\_t \*p, struct bpf\_program \*fp, char \*str, int optimize, bpf\_u\_int32 netmask)

* + Pcap\_t \*p: specifies session handle.
  + struct bpf\_program \*fp: specifies reference to the place we will store the compiled version of our filter.
  + char \*str: specifies expression in a regular string format.
  + int optimize: integer that decides if the expression should be "optimized" or not.
  + bpf\_u\_int32 netmask: specifies the network mask of the network the filter applies to.

# int pcap\_setfilter(pcap\_t \*p, struct bpf\_program \*fp)

* + pcap\_t \*p: session handler.
  + struct bpf\_program \*fp: specifies reference to the compiled version of the expression.

# Sniffing:

**u\_char \*pcap\_next(pcap\_t \*p, struct pcap\_pkthdr \*h)** is used to capture a single packet at a time.

* + pcap\_t \*p: session handler
  + struct pcap\_pkthdr \*h: a pointer to a structure that holds general information about the packet
  + The function returns a u\_char pointer to the packet that is described by this structure

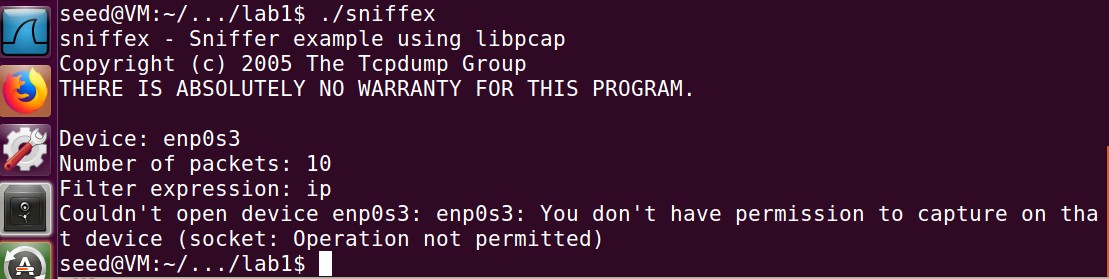
**int pcap\_loop(pcap\_t \*p, int cnt, pcap\_handler callback, u\_char \*user)** is used to enter a loop that waits for n number of packets to be sniffed before being done.

* pcap\_t \*p: session handle
* int cnt: specifies how many packets it should sniff for before returning. Negative value means it should sniff until an error occurs.
* pcap\_handler callback: the name of the callback function
* u\_char \*user: useful in some applications, NULL for many situations.

# Close the Sniffing Session:

pcap\_close() is used to close the sniffing session.

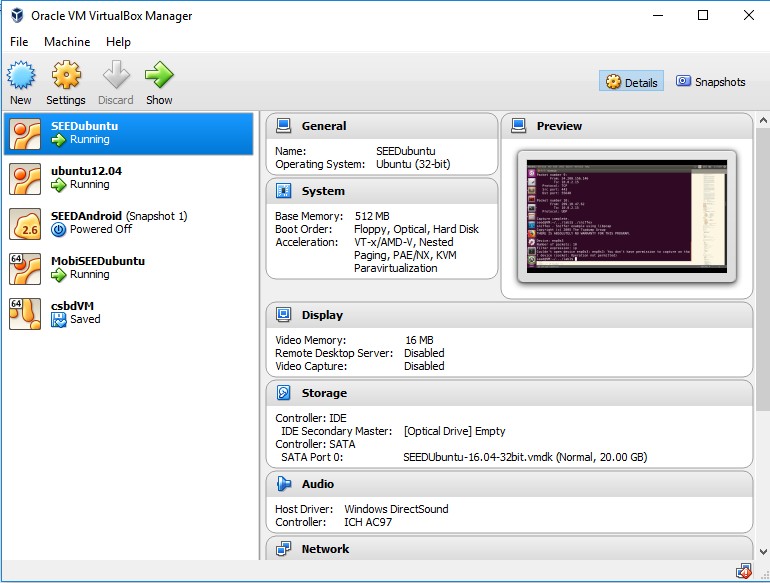
# Problem 2:

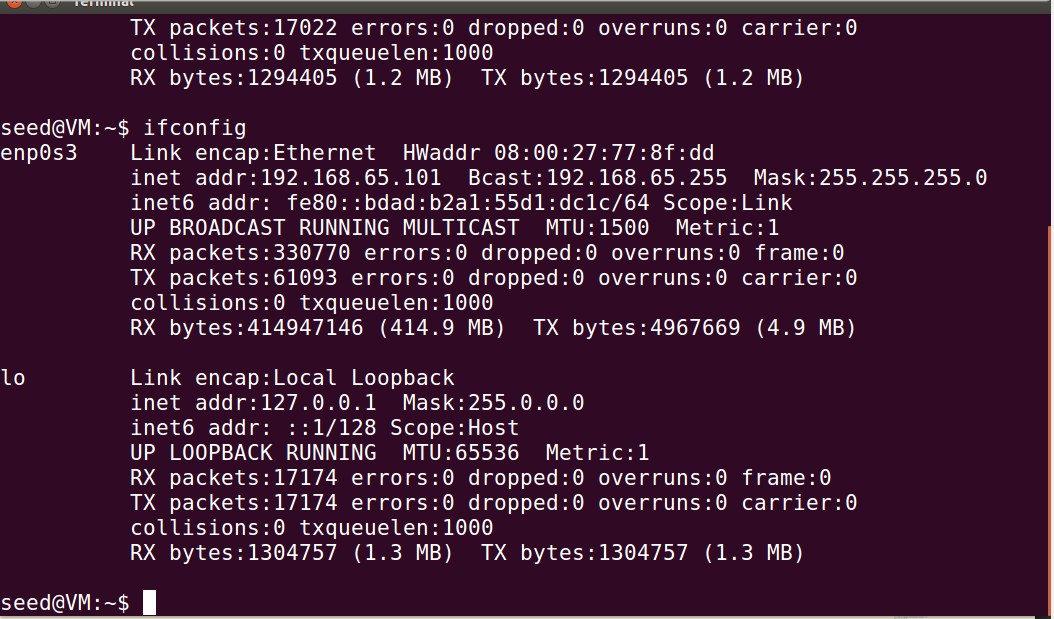


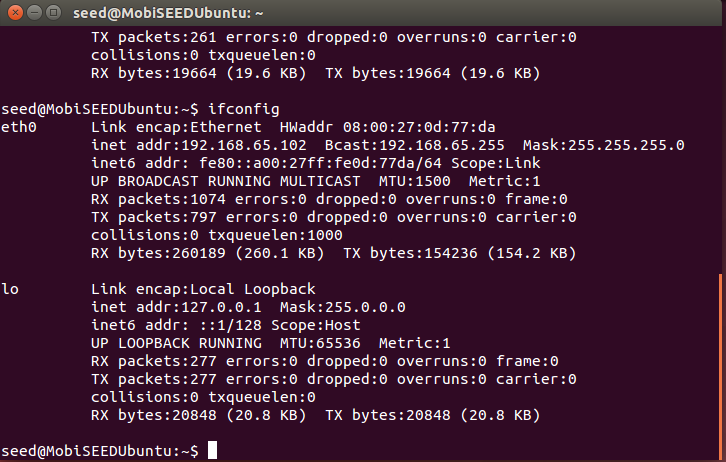
When sniffex is run without root privileges, it says that we don’t have permission to capture on that device.

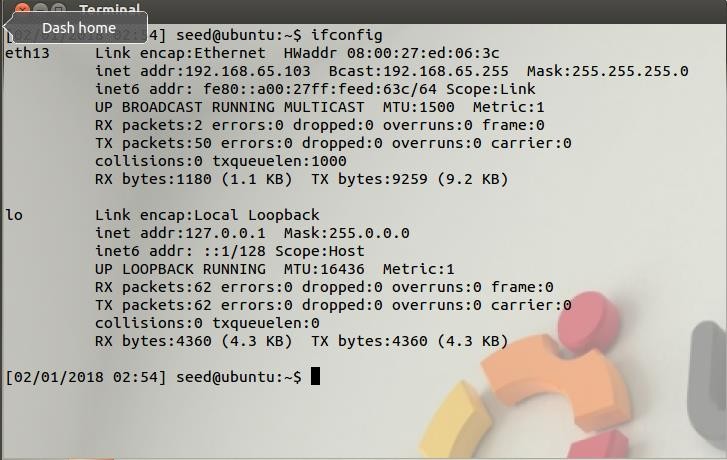
**Explanation:** Pcap needs root permissions to run sniffex because it has to access the network interface card. Network interface card is the physical device that accepts the packets into the system and only root can access it. The pcap\_lookupdev() in the sniffex program fails as it is looking for the interface to sniff on and this requires root access. But since root privilege is not given, it fails.

# Problem 3:









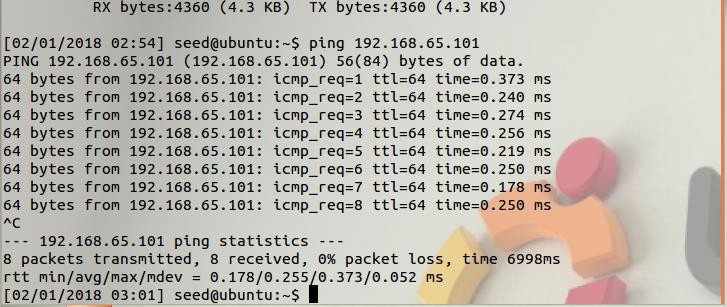
For this task we setup 3 machines. We change the networks of the machines from NatNetwork to Host only Adapter so that they work under the same network.

**IP Addresses: For convenience,**

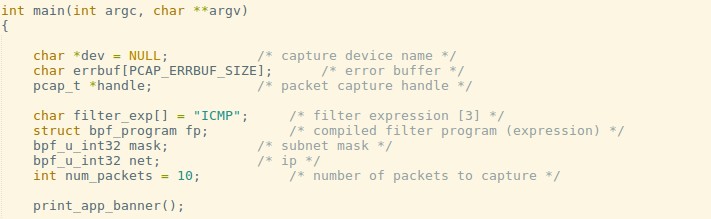
# M1: 192.168.65.101

**M2: 192.168.65.102**

**M3: 192.168.65.103**

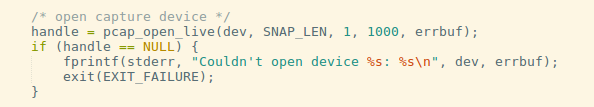


We are verifying if ping works and M3 is used to ping M1 (192.168.65.101) and we find that this is successful.



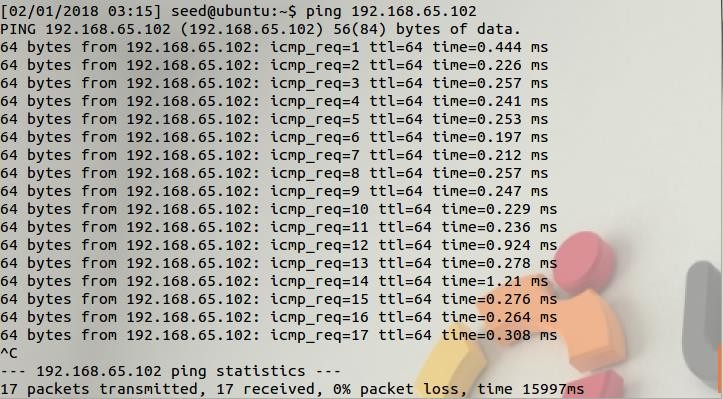
We set our filter to capture ICMP packets.

A



We turn on Promiscous mode by setting 3rd argument of pcap\_open\_live as 1.

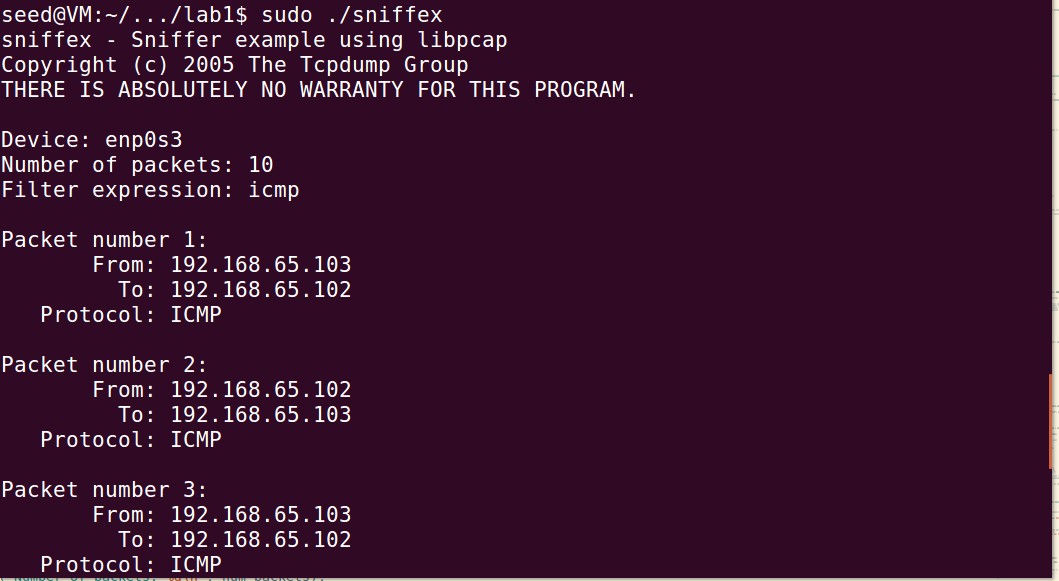
B



**We create a ping from M3 to M2 now.**

**C**

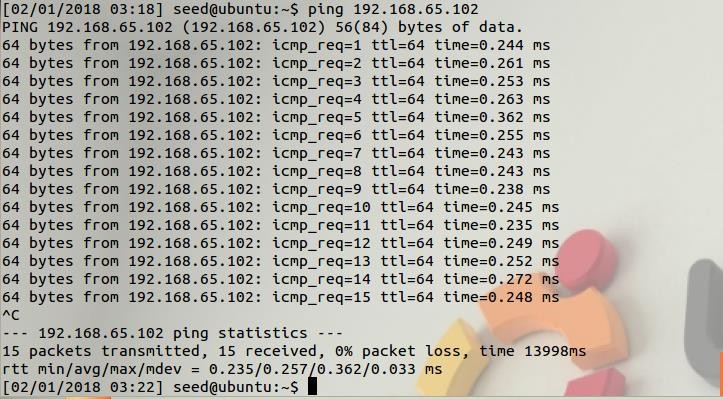
**D**



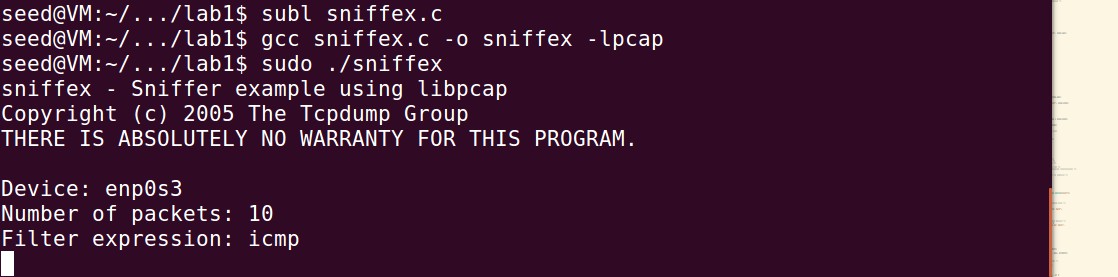
When promiscuous mode is turned on, the user sitting on M1 can observe this connection by running the sniffer program.

**Explanation:**

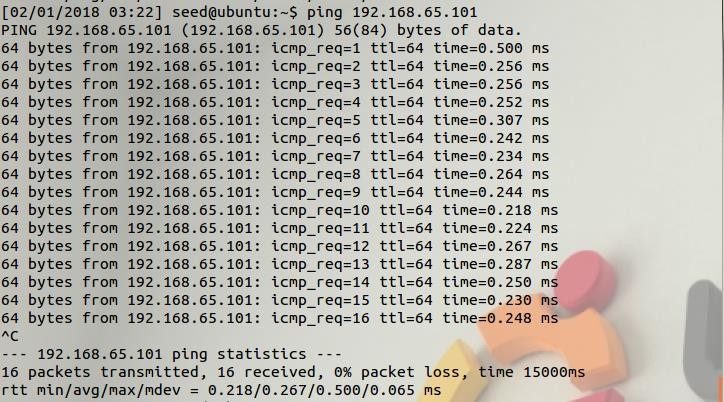
Promiscuous mode bit is set in the pcap\_open\_live() function. The 3rd bit parameter is set to 1, indicating that promiscuous mode is on. When promiscuous mode is on, sniffer program can capture all the packets in the same network regardless of the destination IP.

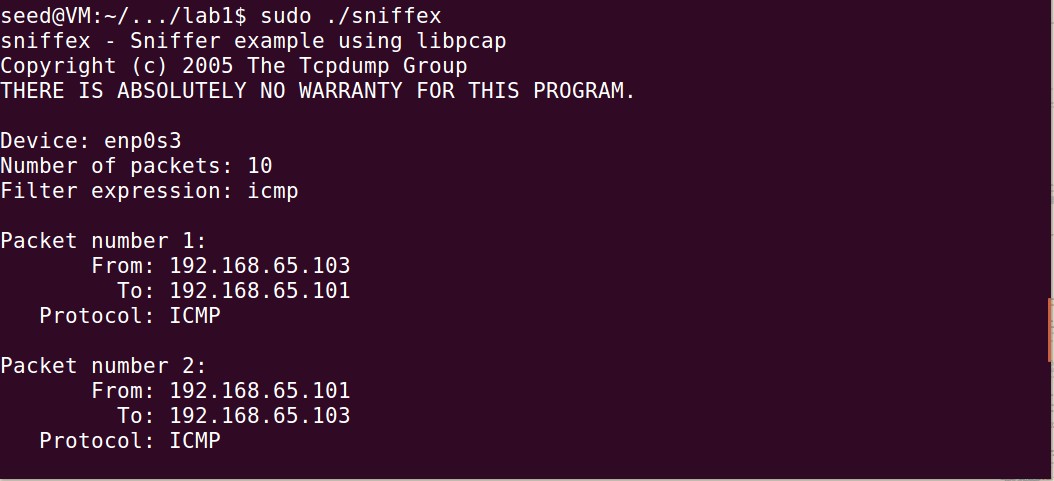


**And Now, we turn off the promiscuous mode and perform the same ping operation.**



**We observe that M1 cannot capture packets of communication between M3 and M2.**





**M3 pings M1 when Promiscous mode is turned off. We are able to sniff this.**

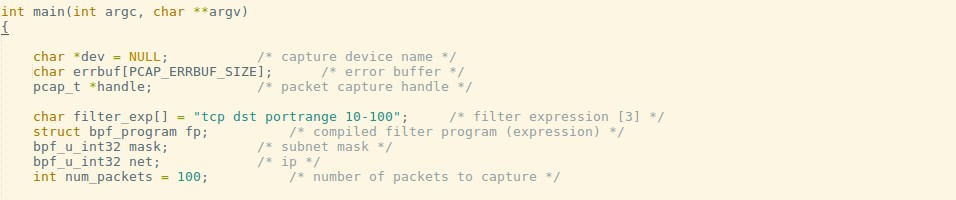
**Explanation: Promiscuous mode bit is set in the pcap\_open\_live() function. The 3rd bit parameter is set to 0, indicating that promiscuous mode is off. When promiscuous mode is off, sniffer program cannot capture all the packets in the same network, it can only capture packets whose destination IP is the IP of the sniffer’s system.**

# Task 1.b: Writing Filters

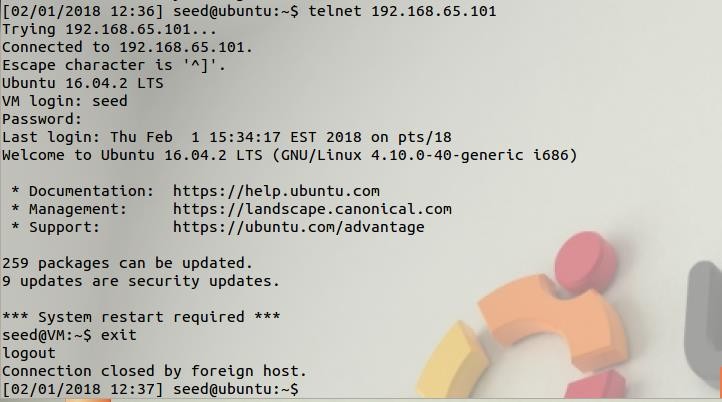
* **Capturing ICMP Packets**

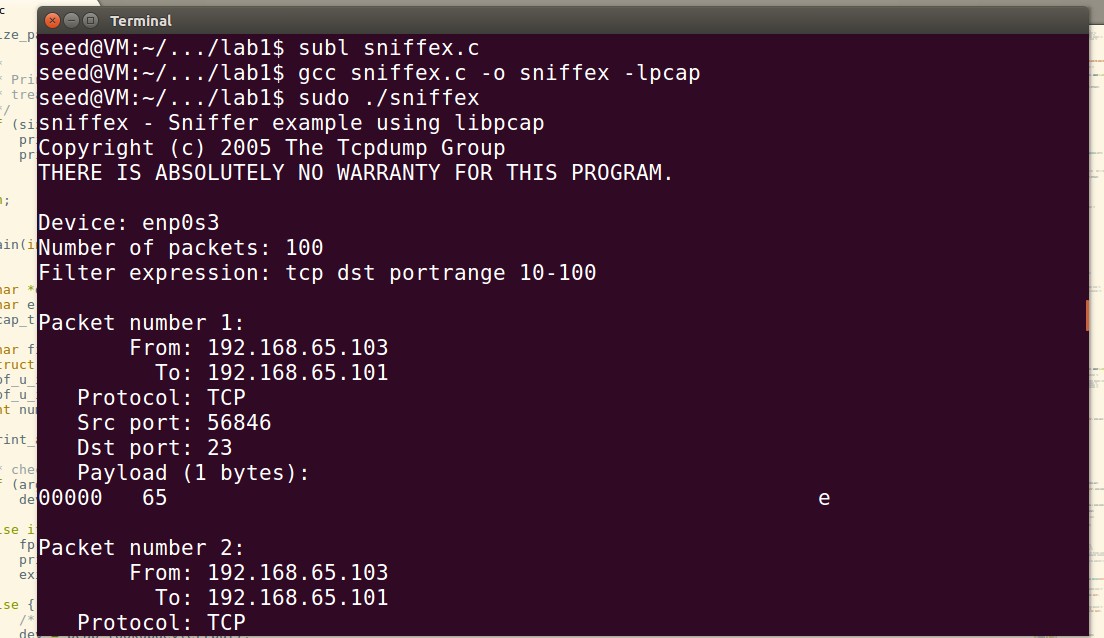
The answer refer to our , Screenshot at Pictures A+B+C+D

# Capture the TCP packets that have a destination port range from to port 23



**We** **set our filter expression to TCP packets that have a destination port 23.**

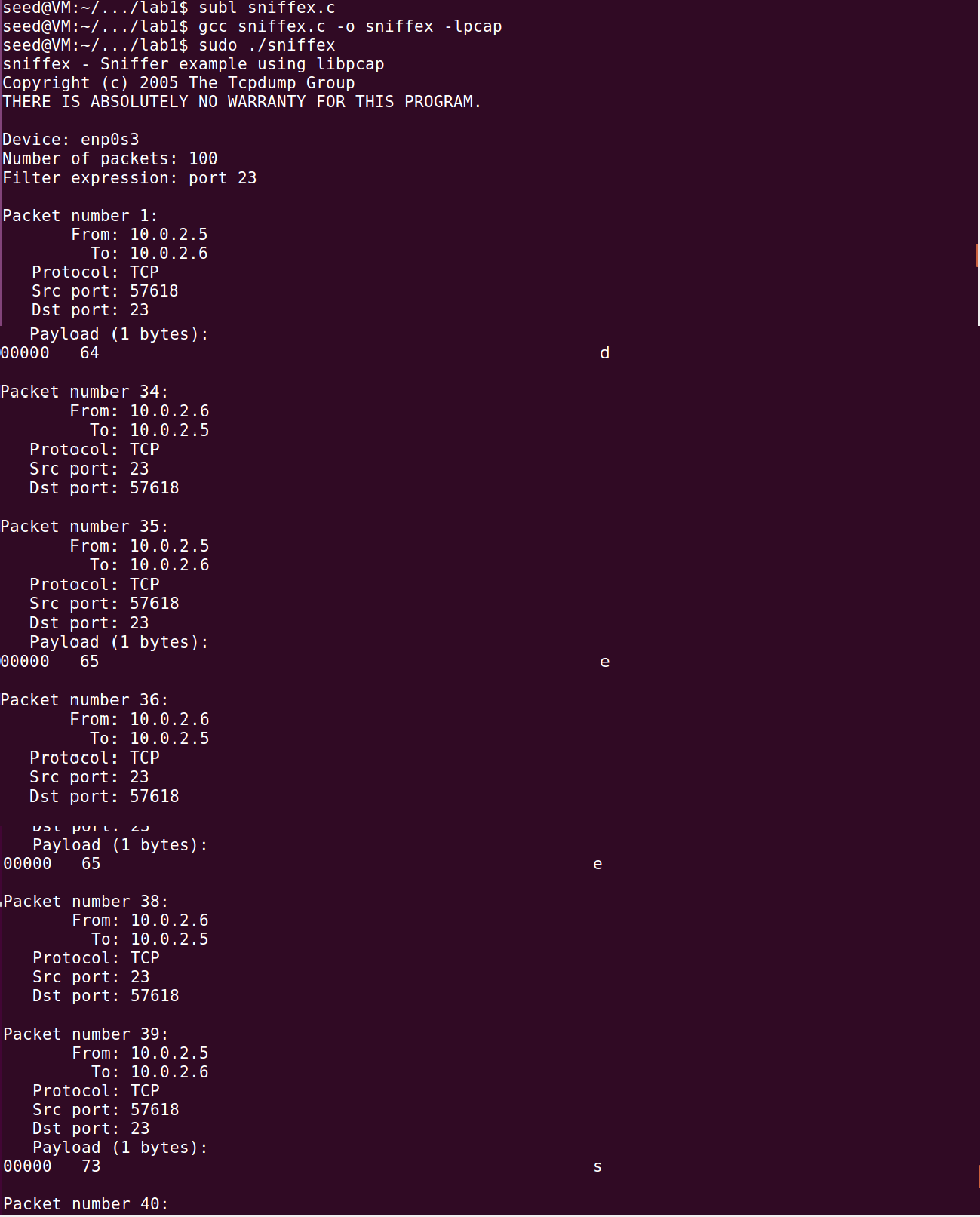




**The above screenshots depicts that TCP packets are captured that have a destination port number in the range of 10 to 100. The filter expression is shown in the code. When the user establishes a telnet connection, the destination port is 23, which falls in the range, so the sniffer captures those packets.**

**Explanation:**

**Filters are used to capture specific traffic. In the above case we capture TCP packets whose destination port number is between 10 and 100. To apply filter, we first need to create a rule set to filter the traffic, then we need to compile the rule set because the filter has to be understood by pcap. We then need to apply the filter using pcap\_setfilter(). This makes pcap only receive packets based on the filter applied.**



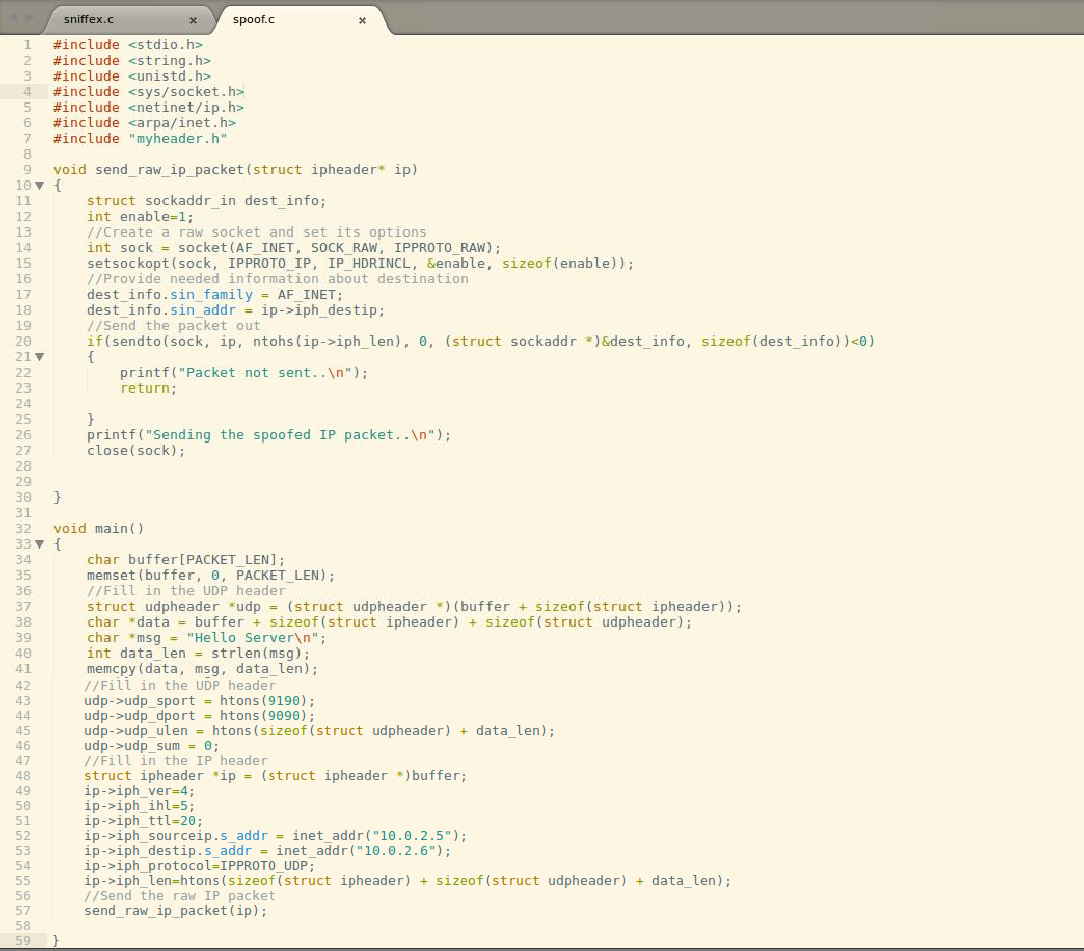
User establishes a telnet connection to host 10.0.2.6. The credentials for the host are entered by the user and this is seen in plaintext in the attacker’s terminal because he is running the sniffer program with filter set to port 23.

**Explanation:**

Telnet connection runs on port 23. When we sniff telnet connections, the entire traffic is displayed in plaintext including the username and password.

**Task 2: Spoofing**

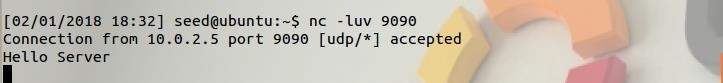
**Task 2.a: Write a spoofing program**

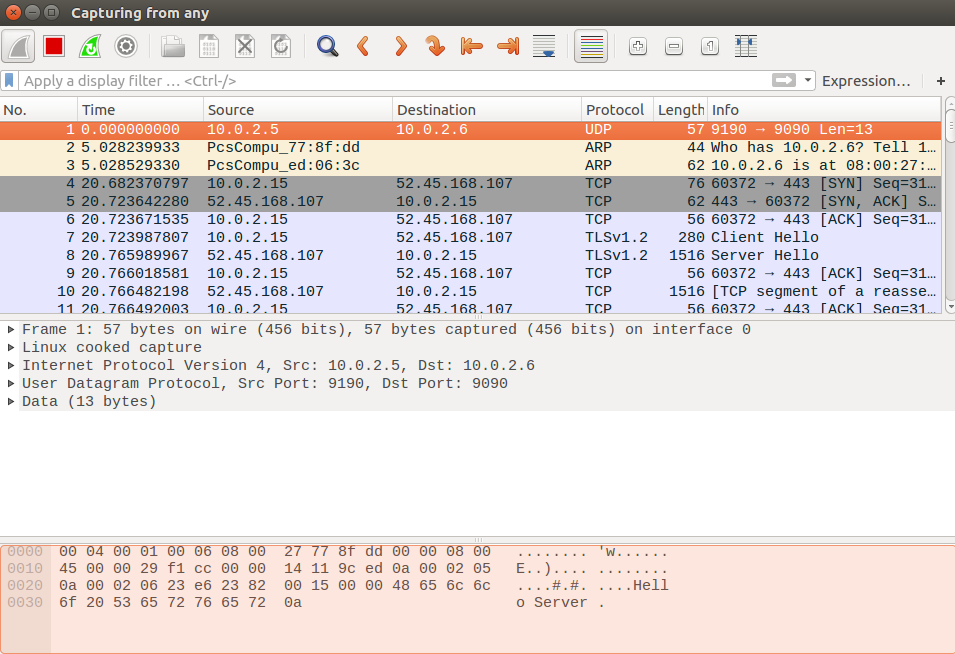


The above screenshot shows our spoofing program.



The above screenshot compiles our spoof program.

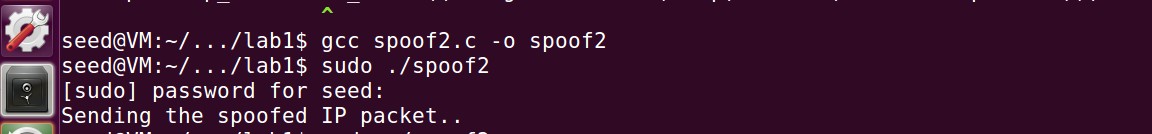


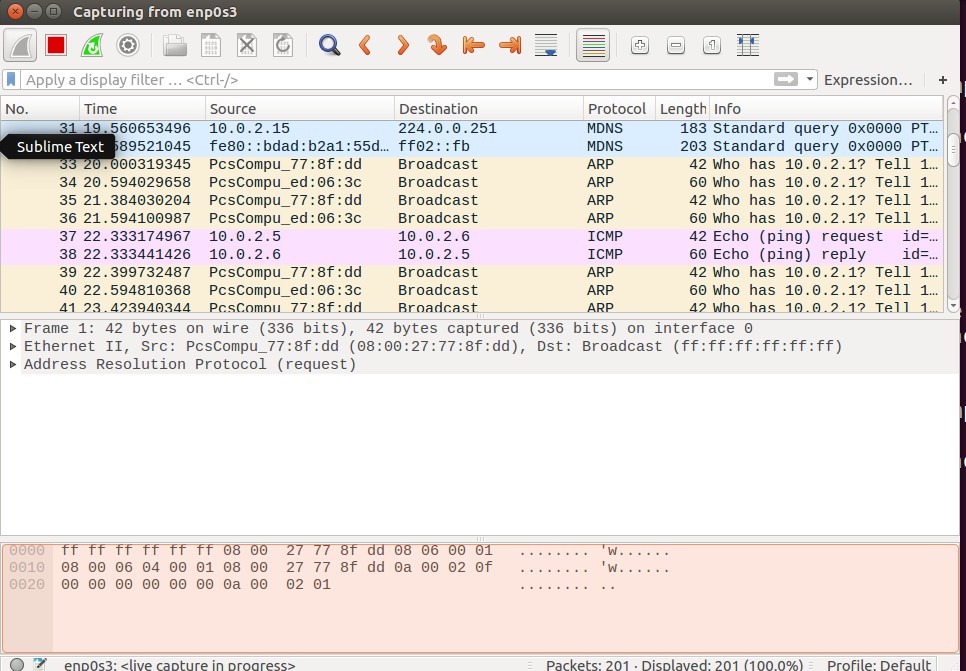


Attacker sends spoofed UDP packet with a message to server who is listening. This is confirmed by the wireshark capture that the source IP of the packet is different from that of the attacker’s.

The attacker on 10.0.2.15 sends to spoofed UDP packet with the message “Hello Server” to 10.0.2.6 with source IP as 10.0.2.5. The source udp port is 9190 and destination udp port is 9090. The server on 10.0.2.15 is listening to incoming connections using netcat on port 9090. The wireshark capture shows the proof that the source ip of the packet is 10.0.2.5 and the destination ip of the packet is 10.0.2.6.

# Task 2.b: Spoof an ICMP Echo Request







In the screenshots above, we can see that the attacker sends a spoofed ICMP request to a host and the host sends back an ICMP reply which is evident in the Wireshark capture.

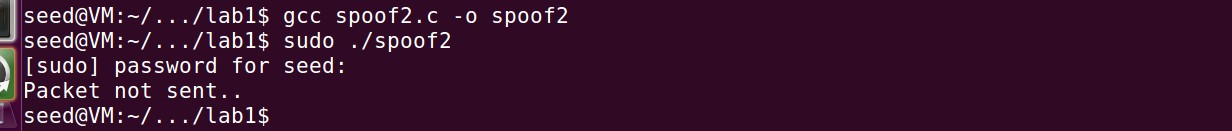
**Explanation**:

The attacker on 10.0.2.15 creates an ICMP packet with source address as

10.0.2.5 and sends the request to 10.0.2.6. The host at 10.0.2.6 receives the ICMP packet

and then sends the reply to 10.0.2.5. This is captured by wireshark as shown in the screenshot. The attacker creates the ICMP packet by specifying the contents in ICMP header and the IP header. The packet is sent using raw socket.

# Problem 4:





The IP packet length field is set to an arbitrary value of 100. The packet is not sent and truncated as seen in the screenshots.

**Explanation:**

The IP packet will not be formed properly if we set the length to some random value. When the packet is sent, it will be truncated because it is too big and is dropped. The length should actually be the sum of size of ipheader and the size of icmp header.

# Problem 5:

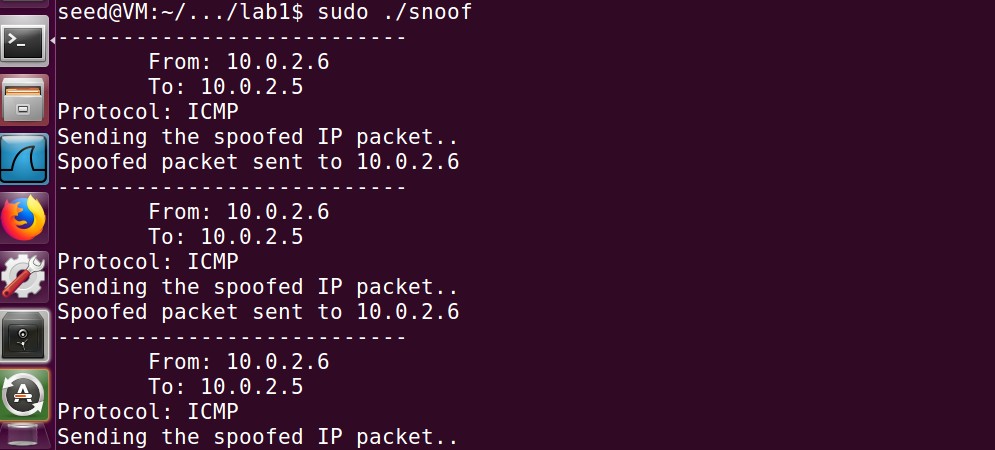
**Explanation:** The checksum for the IP header is calculated by the OS before transmitting the packet over the network, so regardless of the value specified, the OS calculates and then transmits it.

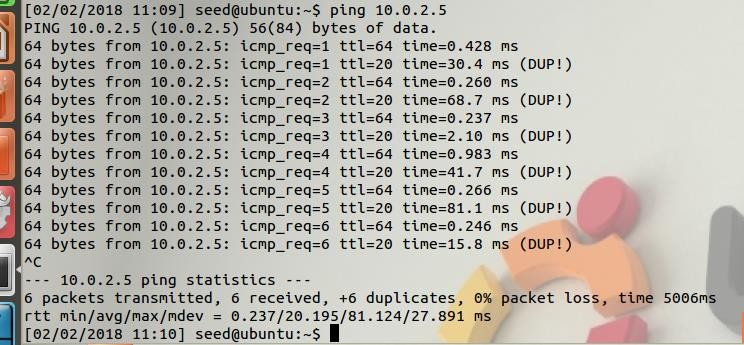
# Problem 6:



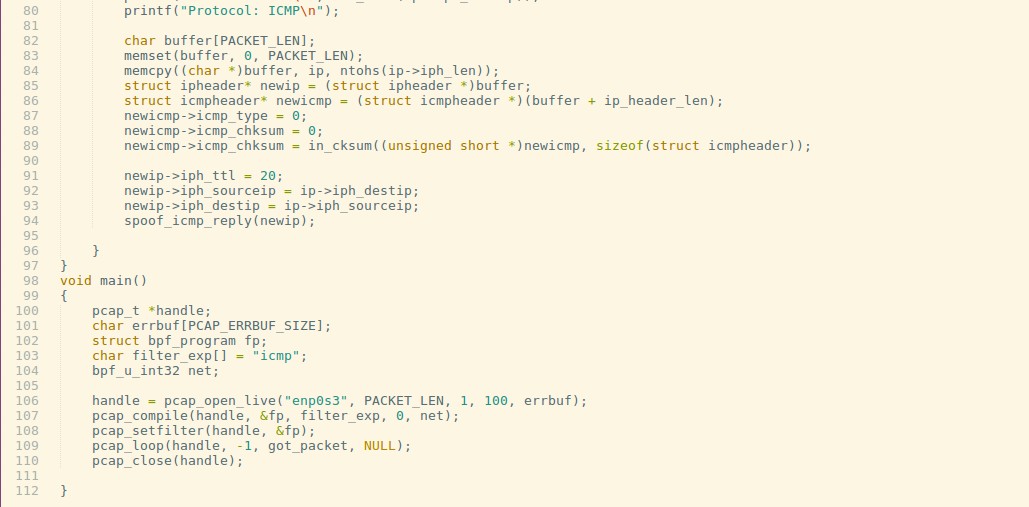
**Explanation:** When the spoof program is run without root privileges, it throws an error because to send a packet, the program needs to access the Network Interface Card (NIC). Raw sockets gives the user the privilege to spoof a packet and set arbitrary values to any field in the packet headers. So when raw sockets are used, it is necessary to have root privileges to perform these tasks.

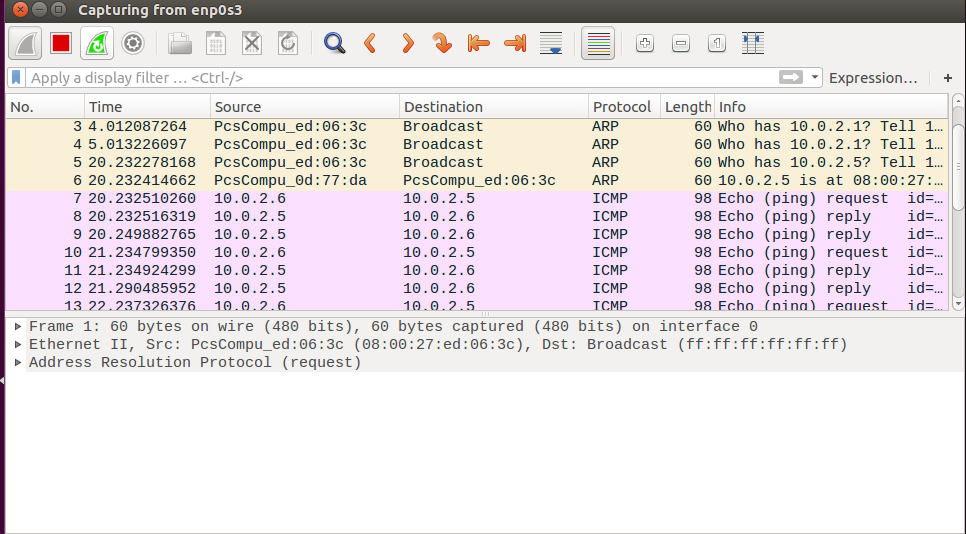
**Task 3: Sniffing and then Spoofing (Snoofing)**











User pings a host 10.0.2.5 on the network, the attacker sniffs the ICMP request, immediately spoofs the ICMP reply to the source of the ICMP request. The user receives the ICMP reply from the attacker as shown in the wireshark capture.

**Explanation:**

Snoofing is sniffing for the request and immediately sending the reply. The user pings a host 10.0.2.5, the attacker on 10.0.2.6 receives the ICMP packet using pcap which listens to traffic (promiscuous mode on), spoofs an ICMP reply using raw socket

by replacing the source ip as the destination ip and the destination ip as the source ip. The fields in the ip header and the icmp header are spoofed by the attacker. When the reply is sent to the User, it seems like he gets a normal reply from the host he pings to. The wireshark capture proves our results.