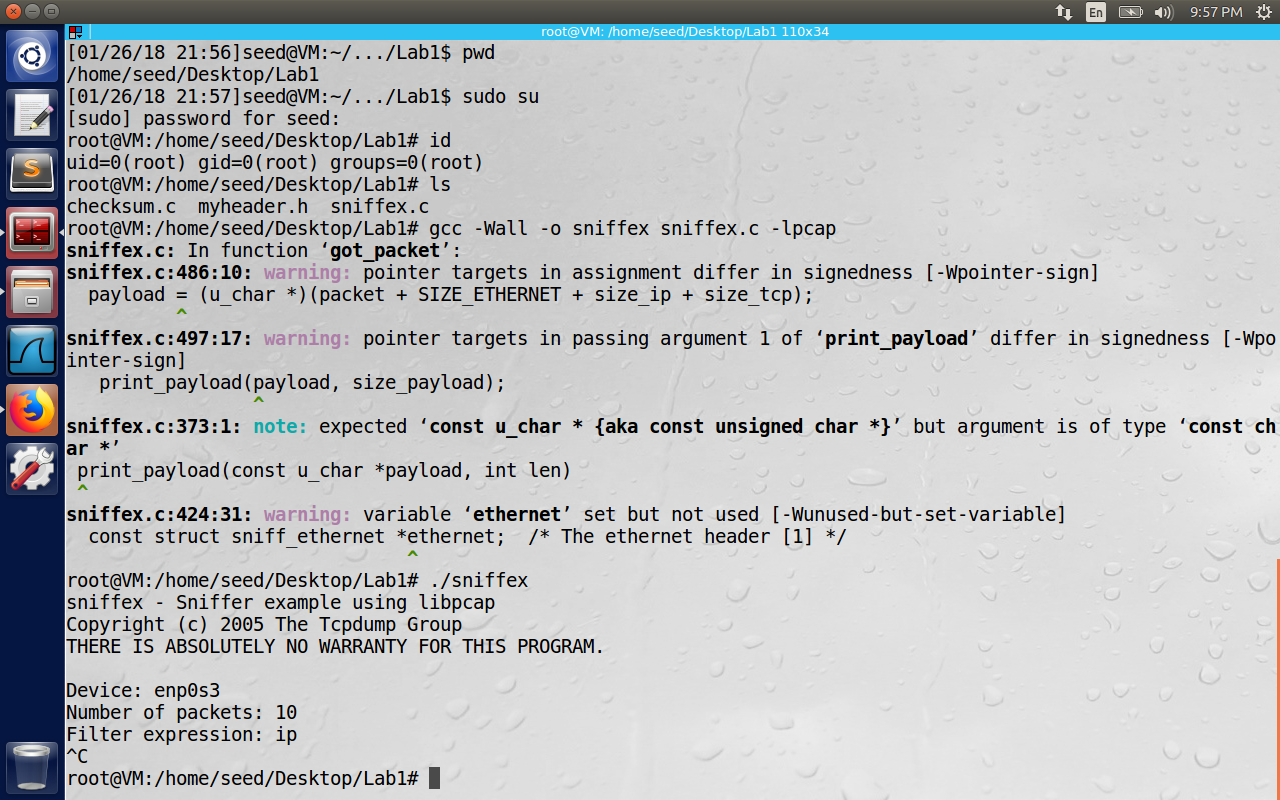
Packet Sniffing and Spoofing Lab

Name: JASHWANTH REDDY GANGULA

SUID: 646254141

Task1: Writing Packet Sniffing Program

Task1.a: Understanding sniffex:



The sniffex.c is downloaded from tcpdump.org and compiled successfully (with some warnings). Under the root privileges, the program was running successfully as shown in the above screenshot.

**Problem1:** Please use your own words to describe the sequence of library calls that are essential for sniffer programs.

**Opening the device:**

Find a capture device if not specified on the command line argument. Get network number and mask associated with capture device and open the capture device. Make sure we are capturing on the Ethernet device. Not all the devices at the link-layer provide the Ethernet header, for example loopback and wi-fi interfaces maynot provide Ethernet header in the monitor mode. pcap\_datalink() function returns the type of link-layer headers.

**Filtering traffic:**

We can implement the filtering at the application level/hardware level. Instead of wasting CPU cycles with if/else statements at the application layer, we can use Berkley Packet Filter by BPF driver do the filtering directly. Before applying the filter, we compile it using

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

Here p is the session handle, fp is where we place the compiled filter program. It will later be injected into kernel, to apply on every incoming packet, str is the filter string(for example if the string is “ip host 10.1.2.3” we want to filter all the IP traffic to and from host 10.1.2.3), optimize is for 1 or 0 saying if the filter expression should be optimized or not, netmask is the network mask of the network filter applies to. Next pcap\_setfilter(pcap\_t \*p, struct bpf\_program \*fp) sets the compiled filter to the hardware.

**The Actual Sniffing:**We can either capture a single packet, or wait in a loop for n packets to be sniffed at once.

u\_char \*pcap\_next(pcap\_t \*p, struct pcap\_pkthdr \*h) can be used to capture a single packet returning a pointer to the packet. Here p is our session handle, h is packet header that contains the time when the packet is sniffed, its length etc.

int pcap\_loop(pcap\_t \*p, int cnt, pcap\_handler callback, u\_char \*user) captures the cnt argument number of packets, user specifies any extra arguments to be sent to the callback function to process or understand the packets. The callback should use typecasting method to understand the fields of the packet. For example

ethernet = (struct sniff\_ethernet\*)(packet);

ip = (struct sniff\_ip\*)(packet + SIZE\_ETHERNET);

tcp = (struct sniff\_tcp\*)(packet + SIZE\_ETHERNET + size\_ip);

size\_tcp = TH\_OFF(tcp)\*4;

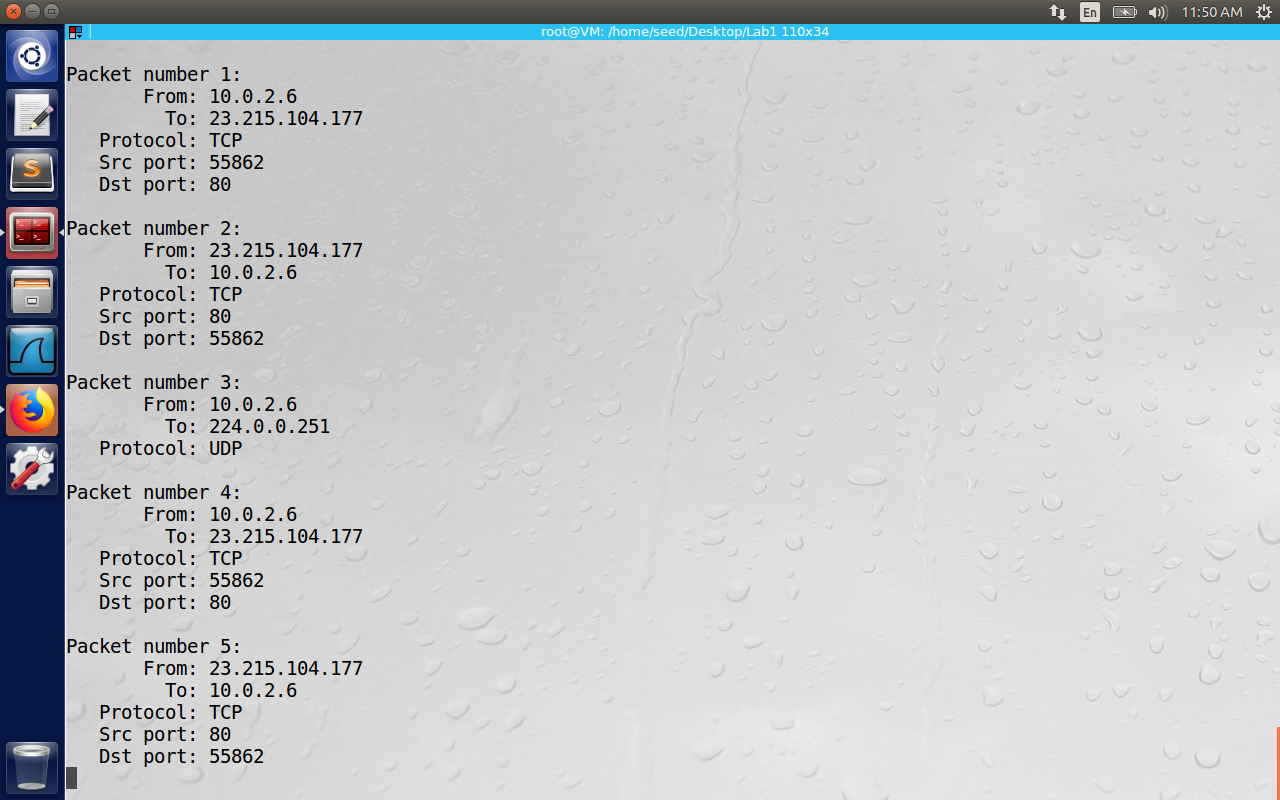
Calculations like these can be used to find the ethernet , ip and tcp headers. But Ip and Tcp headers lengths are not fixed and the information about these lengths should be fetched from the packet itself.

**Problem2:** Why do we need root privilege to run sniffex? Where does the program fail if executed without the root privilege?

We need to sniff on the network devices, set filter at the kernel level. All these read or write operations to the kernel /network devices needs special privileges. Hence the program fails if it run as a normal user.

**Problem3:** Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this?

In standard, non-promiscuous sniffing, a host is sniffing only traffic related to it. Promiscuous mode on the other hand sniffs all the traffic on the wire. In a network with switches and hubs, the host can detect that network is sniffed, because of some port mirroring done on switches to send the traffic to middle ear dropper. In our case with VM, we can connect to the network either using bridge adapter or NAT adapter. If we use NAT adapter, VM will be in the private network and reaches the internet through NAT gateway, but for VM to even receive the packets of the host, we change the change the network to bridged adapter where the VM is also connected to one of the network cards on the host, hence the packets can be sniffed easily in the bridged mode.



Task 1.b: Writing Filters:

Capture ICMP packets between two specific hosts.  
We can use primitives like negation, and , or to achieve the logical isolation of traffic. In case of ICMP packets between host1 and host2, the filer expression can be “icmp and (src host <host1> and dst host <host2>) or (src host <host2> and dst host <host1>)”.

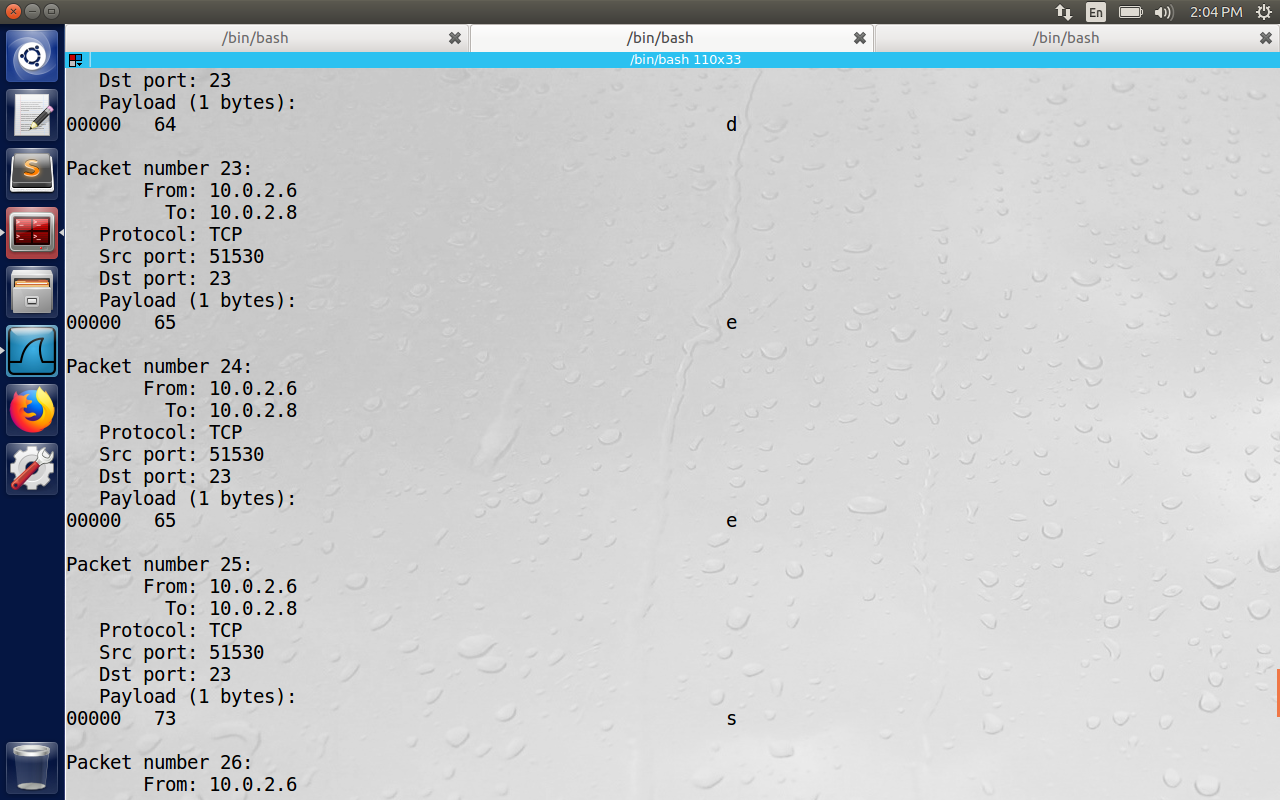
Capture the TCP packets that have a destination port range from to port 10 - 100.

The filter expression looks like: “tcp dst portrange 10-100”. This expressions needs to be filled in pcap\_compile() to compile and then install this filter to capture packets.

Task 1.c: Sniffing Passwords

Here we start telnet daemon on VM2 using the command “sudo service openbsd-inet start”

Also we modify the sniffer program to filter the traffic on port 23 , as telnet is started on port 23. Char filter\_expression[] = “dst port 23”.



We can observe the sniffer got the alphabets d,e,e,s which is the password for the user name - seed.

**Task2: Spoofing:**

The OS allows the users to set only specific fields like destination address, destination port etc. The rest of the fields are added by the OS, however with root privileges the user has full control over the packet fields, using raw sockets to send the packets with custom headers and payloads.

Task 2.a: Write a spoofing program

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/ip.h>

#include "myheader.h"

#define SRC\_PORT 80

#define DEST\_PORT 9090

#define SRC\_IP "[10.0.2.6](https://l.facebook.com/l.php?u=http%3A%2F%2F10.0.2.6%2F&h=ATMXfROJm8Xxegl7BdtU2dya0phOoqFQVPIJXNJ1k5hUeyYZVCeSqfpVvErCmfNWZyyQPizo8OvrZeWeiMnJV1shZ-I0FrpL4Ob7MQ5NKikzat7xQDPOZ2CnlgRduci0HMpZ6L0rk0eSCXr1d9pwow)"

#define DEST\_IP "[10.0.2.8](https://l.facebook.com/l.php?u=http%3A%2F%2F10.0.2.8%2F&h=ATMXfROJm8Xxegl7BdtU2dya0phOoqFQVPIJXNJ1k5hUeyYZVCeSqfpVvErCmfNWZyyQPizo8OvrZeWeiMnJV1shZ-I0FrpL4Ob7MQ5NKikzat7xQDPOZ2CnlgRduci0HMpZ6L0rk0eSCXr1d9pwow)"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given a buffer of data, calculate the checksum

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

unsigned short in\_cksum(unsigned short \*buf, int length)

{

unsigned short \*w = buf;

int nleft = length;

int sum = 0;

unsigned short temp=0;

/\*

\* The algorithm uses a 32 bit accumulator (sum), adds

\* sequential 16 bit words to it, and at the end, folds back all the

\* carry bits from the top 16 bits into the lower 16 bits.

\*/

while (nleft > 1) {

sum += \*w++;

nleft -= 2;

}

/\* treat the odd byte at the end, if any \*/

if (nleft == 1) {

\*(u\_char \*)(&temp) = \*(u\_char \*)w;

sum += temp;

}

/\* add back carry outs from top 16 bits to low 16 bits \*/

sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16

sum += (sum >> 16); // add carry

return (unsigned short)(~sum);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Given an IP packet, send it out using a raw socket.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void send\_raw\_ip\_packet(struct ipheader\* ip)

{

struct sockaddr\_in dest\_info;

int enable = 1;

// Step 1: Create a raw network socket.

int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

// Step 2: Set socket option.

setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL, &enable, sizeof(enable));

// Step 3: Provide needed information about destination.

dest\_info.sin\_family = AF\_INET;

dest\_info.sin\_addr = ip->iph\_destip;

// Step 4: Send the packet out.

sendto(sock, ip, ntohs(ip->iph\_len), 0,

(struct sockaddr \*)&dest\_info, sizeof(dest\_info));

close(sock);

}

int main()

{

char buffer[1500];

int sd;

struct sockaddr\_in sin;

/\* Create a raw socket with IP protocol. The IPPROTO\_RAW parameter

\* tells the system that the IP header is already included;

\* this prevents the OS from adding another IP header.

\*/

sd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

if (sd < 0) {

perror("socket() error");

exit(-1);

}

/\* This data structure is needed when sending the packets

\* using sockets. Normally, we need to fill out several

\* fields, but for raw sockets, we only need to fill out

\* this one field \*/

sin.sin\_family = AF\_INET;

// Here you can construct the IP packet using buffer[]

// - construct the IP header ...

// - construct the TCP/UDP/ICMP header ...

// - fill in the data part if needed ...

// Note: you should pay attention to the network/host byte order.

memset(buffer, 0, sizeof(buffer));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 1: Fill in the ICMP Header

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct icmpheader \*icmp = (struct icmpheader \*)(buffer + sizeof(struct ipheader));

icmp->icmp\_type = 8; // ICMP: Type 8 is request, 0 is reply

// Calculate the checksum for integrity

icmp->icmp\_chksum = 0;

icmp->icmp\_chksum = in\_cksum((unsigned short \*)icmp, sizeof(struct icmpheader));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 2: Fill in the IP Header

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct ipheader \*ip = (struct ipheader \*)buffer;

ip->iph\_ver = 4;

ip->iph\_ihl = 5;

ip->iph\_ttl = 20;

ip->iph\_sourceip.s\_addr = inet\_addr(SRC\_IP);

ip->iph\_destip.s\_addr = inet\_addr(DEST\_IP);

ip->iph\_protocol = 1; // IPPROTO\_ICMP is 1, representing ICMP

ip->iph\_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 3: Finally, send the spoofed packet

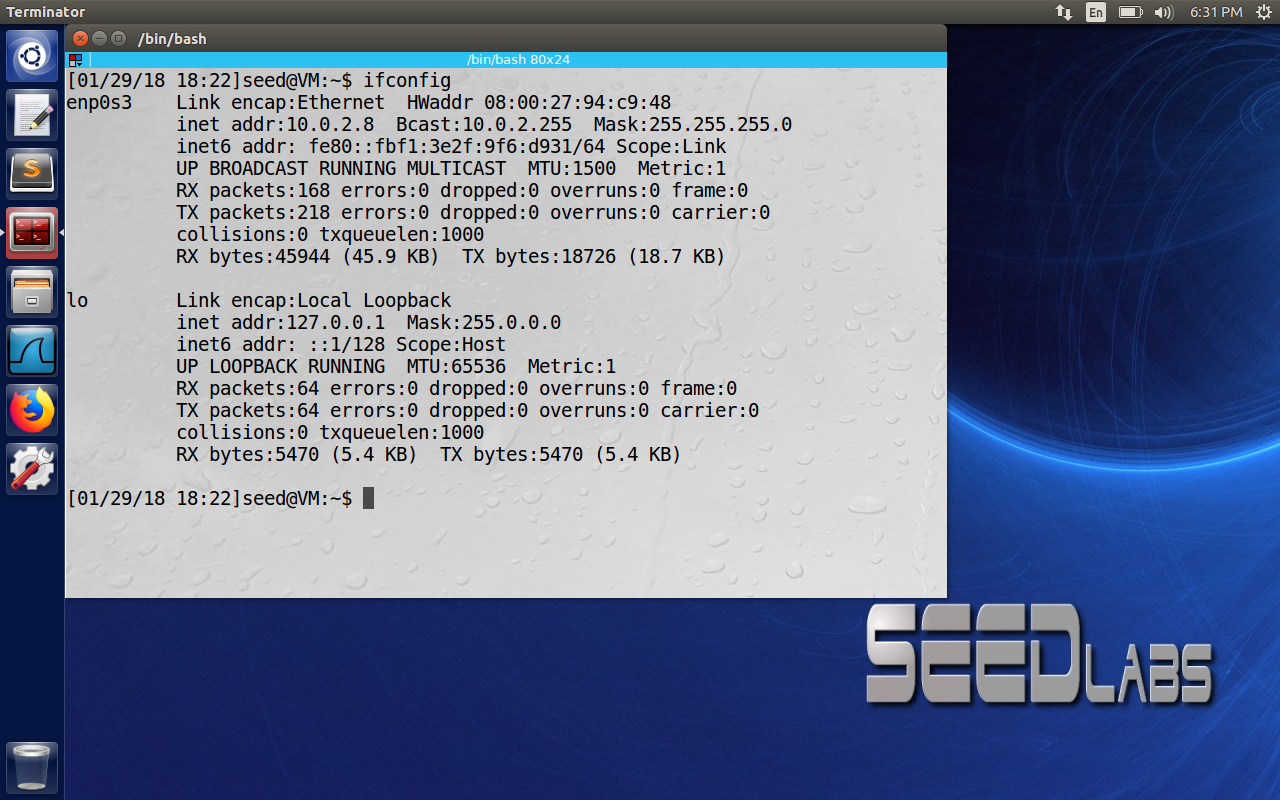
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

send\_raw\_ip\_packet(ip);

return 0;

}

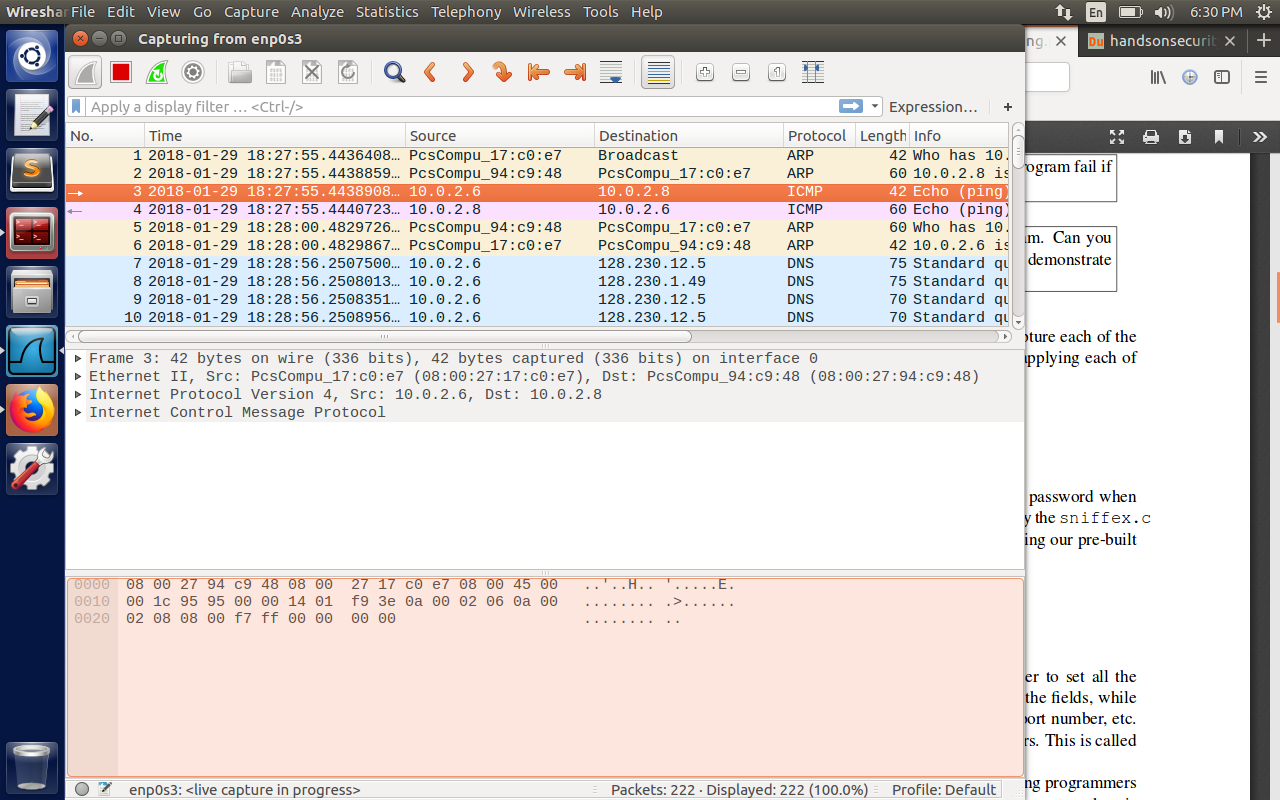
For the above program, I have created 2 VM with IP addresses 10.0.2.6 and 10.0.2.8. To get a different IP for VM2(we have cloned it from VM1) we need to refresh MAC address under advanced settings of Network to get a different IP.



**Task 2.b:**

The above program constructs a ICMP echo request by filling all the necessary headers. The IP

header is constructed first and then ICMP header is added to that. All the fields should be converted from host to network order as per convention. The checksum for icmp header is calculated using the in\_cksum() function provided in the lab code. If the IP Header checksum is zero, kernel calculates it for you. The figure below shows the ICMP request in red color captured using wireshark, and ICMP echo reply in pink color from the destination. Hence the packet spoofing was successful.



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

The socket call to send the program is sendto(). If we set the packet length more than 1500 bytes, it will throw an error message as “Message too long”.

(Reference: <https://stackoverflow.com/questions/36821280/can-i-set-an-arbitrary-value-for-a-ip-packets-total-length> )

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

We don’t have to calculate the checksum for IP header. If the kernel finds out that IP header checksum is zero, it automatically does it for us.

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?

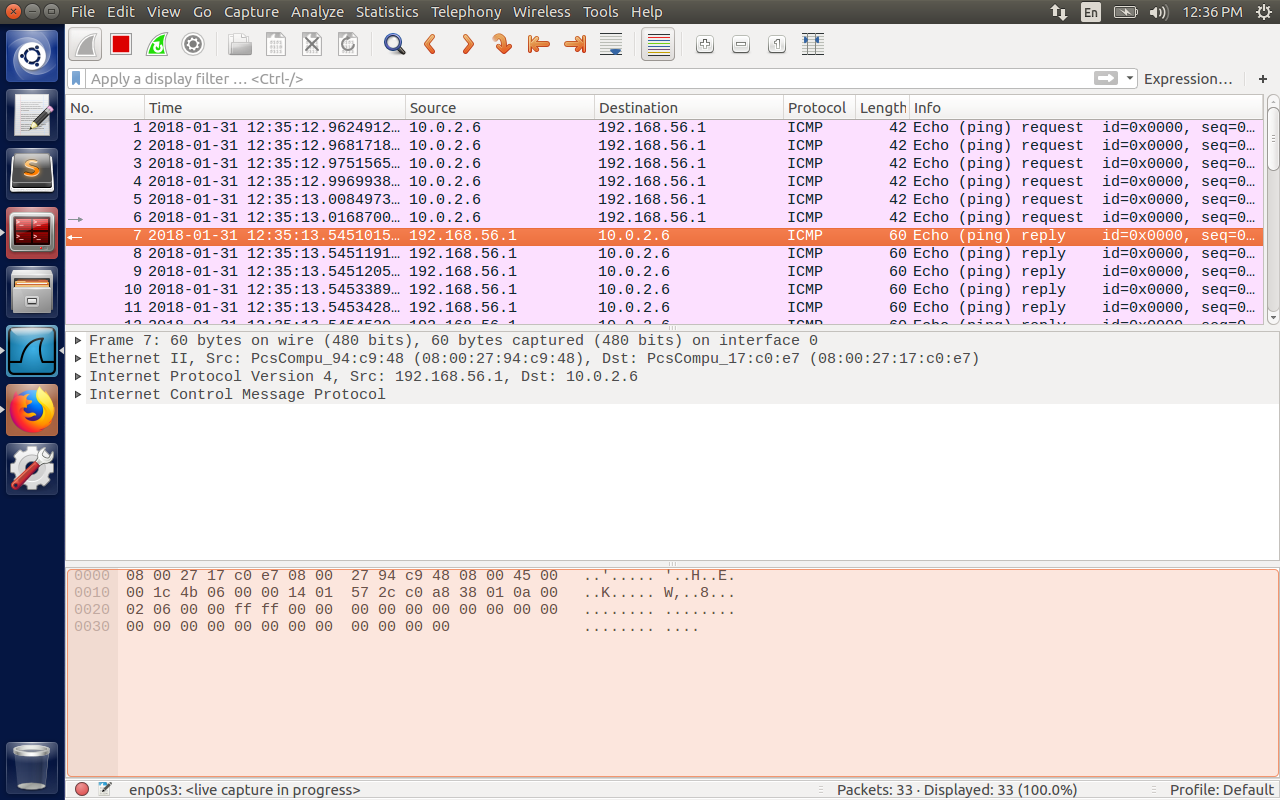
Opening the device is a system call that needs root privilege, pcap\_open\_live() call returns a handle to the opened network device. All the rest of calls use this handle to set the filters. Hence we need root privileges to open the device for capture.

**Task 3: Sniff and then Spoof**

Here I have created a static arp entry on VM with IP address 192.168.56.1 using the command

sudo arp –I enp0s3 –s 192.168.56.1 AA:AA:AA:AA:AA:AA. This is done inorder to prevent the arp request from being generated (assuming this IP lies on the LAN network 10.0.2.0/24.)

Now the sniffex program on 10.0.2.8 receives the packet, checks the protocol as ICMP request, creates a ICMP reply through ICMP reply, and sends the IP packet using raw sockets. So, the VM2 here acts as a man in the middle spoofing the ICMP reply as if the non-existing IP is on the same LAN.



Here a script is written to send 6 ICMP echo requests from VM1 with IP 10.0.2.6 as follows:

#!/bin/bash

var = 0

while [$var –lt 6 ]

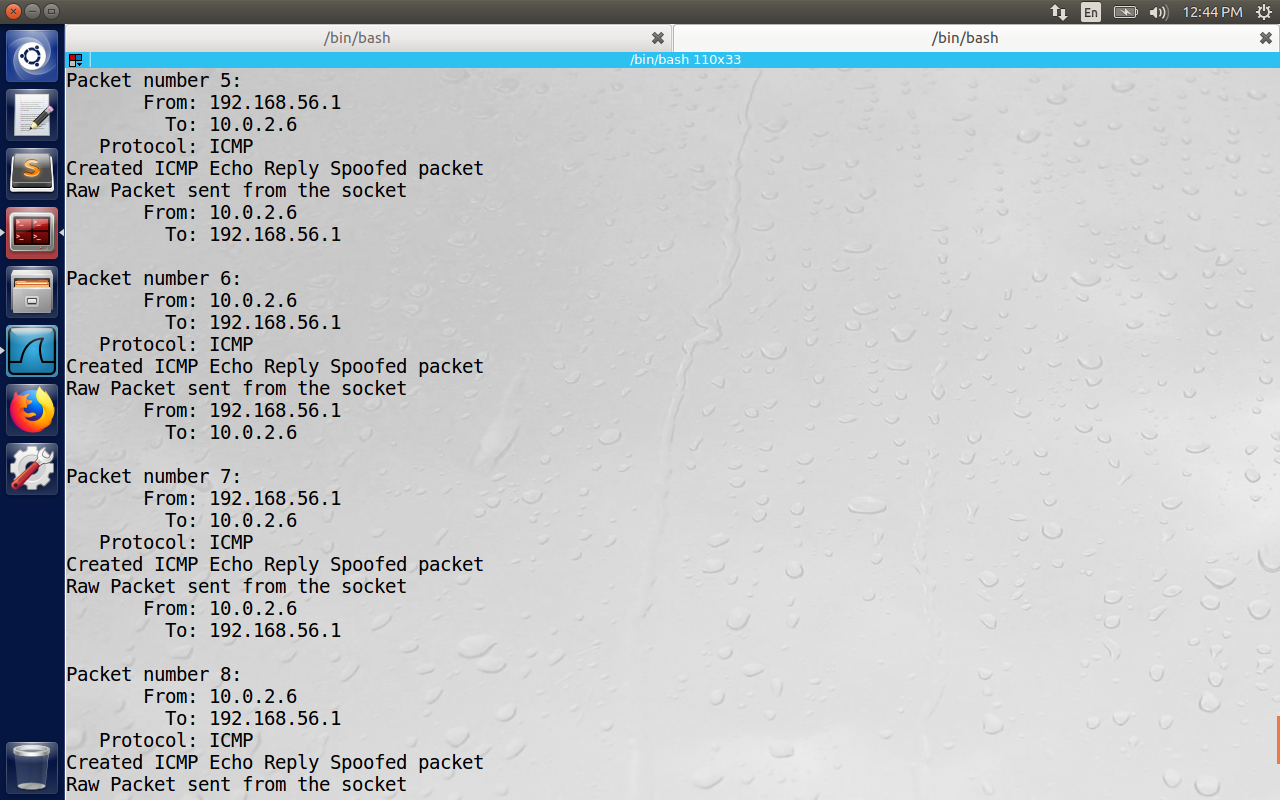
do

sudo ./myICMP

var=$[var+1]]

done

The Wireshark receives 6 echo replies as shown in the screenshot from VM2 which has spoofed the replies as in the below screenshot.



Here 6 packets are received as request and from the seventh packet echo replies are generated by the sniffex program. The programs on myICMP.c and sniffex.c will be attached in addition.