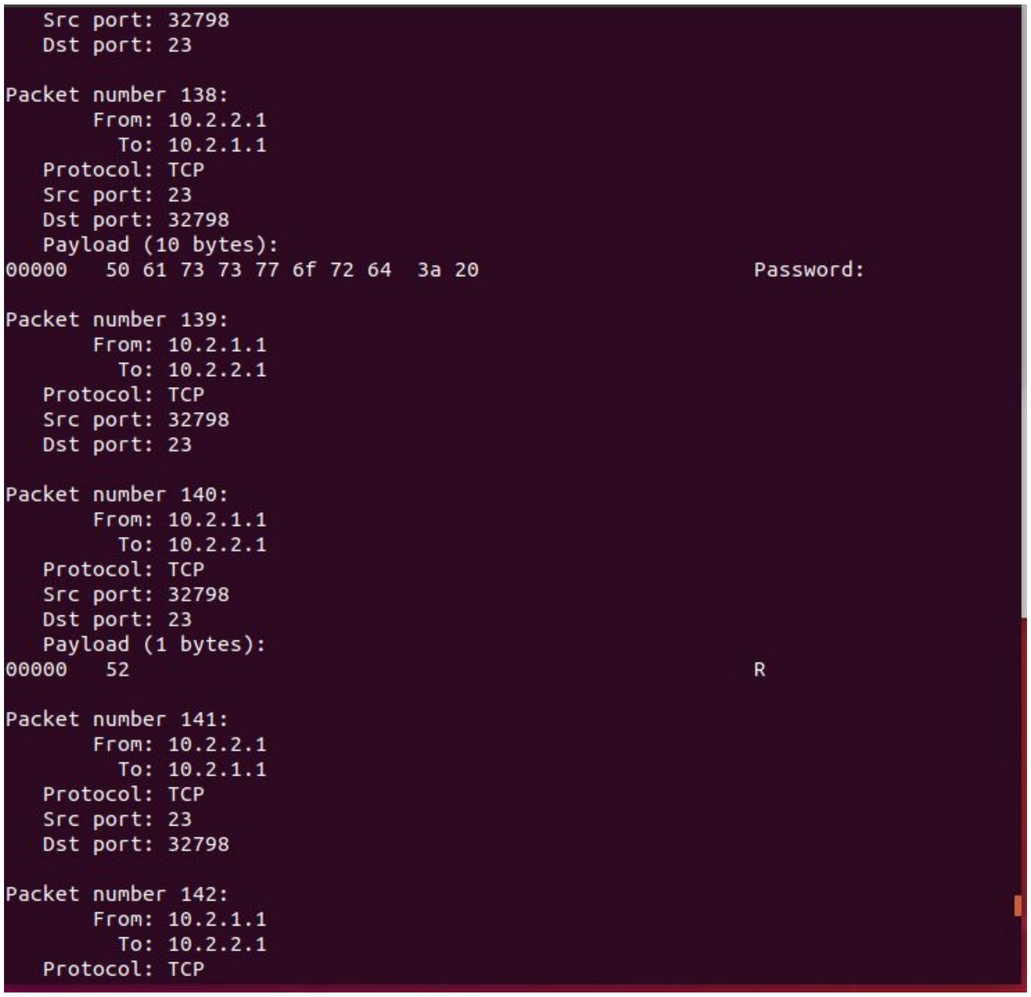
Donald Elrod
CSCE 465-501
1/29/2019

Homework 1

1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not a detailed explanation like the one in the tutorial
    a. First, the program must initialize a listening device, which can be done by manually inputting the device name on the filesystem (in Linux) or having the pcap library find one for you, and then the device is opened like a file descriptor. Then, the program can apply filters for when/what to sniff for, which is 'compiled' into a language pcap can understand. Finally, pcap starts capturing packets in accordance with what the program requires, such as getting individual packets, capturing x number of packets and stopping, or constantly catching packets and dispatching callbacks based on the results.
2. Why do you need the root privilege to run sniffex? Where does the program fail if executed without the root privilege?
    a. Root privilege is needed to open the raw socket, and when not run in sudo mode will not work. It crashes at the point of executing the open_pcap* command
3. Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? (Note: even if the promiscuous mode does not work in TCR, you can still demonstrate your trial with the mode on and off in the sniffer program) Please describe the use of promiscuous mode (explained in the class) and explain your observations.
    a. Promiscuous mode allows the program to receive all packets that are sent over the network the computer has access to, whereas without it the program would only receive traffic routed to the computer and the port it is running on. I was not able to move the sniffex file over to my VMs on AWS, so I created my own local linux VM and ran/compiled it on that on my home network. When I ran in promiscuous mode, it did indeed capture packets from other hosts on the network, while it only captured its own traffic in normal mode. I had to run both modes with sudo, which I was not expecting to have to do but of course it is due to raw sockets.
4. Please write filter expressions to capture each of the following. In your lab reports, you need to include screen dumps to show the results of applying each of these filters.
    a. Capture the ICMP packets between two specific hosts.
        i. "icmp"
    b. Capture the TCP packets that have a destination port range from to port 10 - 100.
        i. "portrange 10-100"
5. Please show how you can use sniffex to capture the password(s) when somebody is using telnet on the network that you are monitoring. You can start from modifying sniffex.c to implement the function. You also need to start the telnetd server on your VM. If you are using our pre-built VM, the telnetd server is already installed; just type the

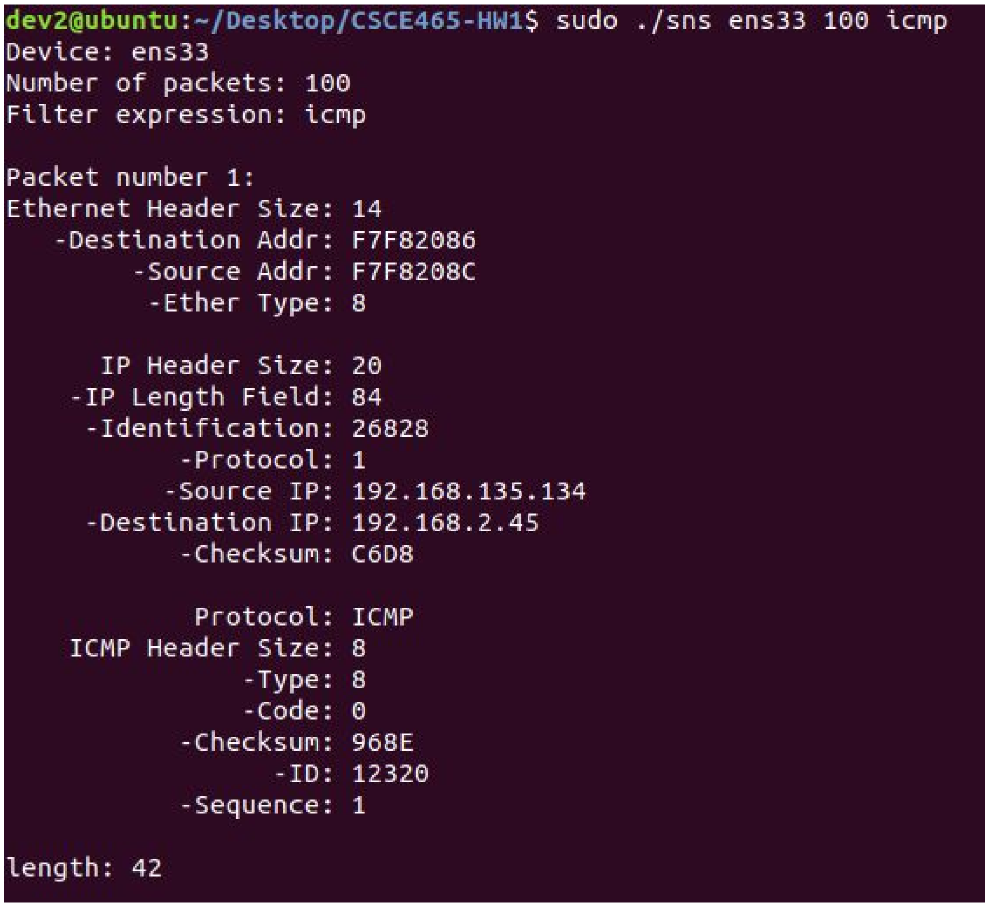
following command to start it.
% sudo service openbsd-inetd start
In addition to live traffic, your program should also support reading from an offline trace file. For your convenience, we also provide a network trace that contains some test Telnet traffic. It is available at http: //faculty.cse.tamu.edu/guofei/download/tfsession.pcap.

a. From the file above, the username and password from the telnet session is:
b. Username: cs6262
c. Password: Re=mi3vE4
d. 

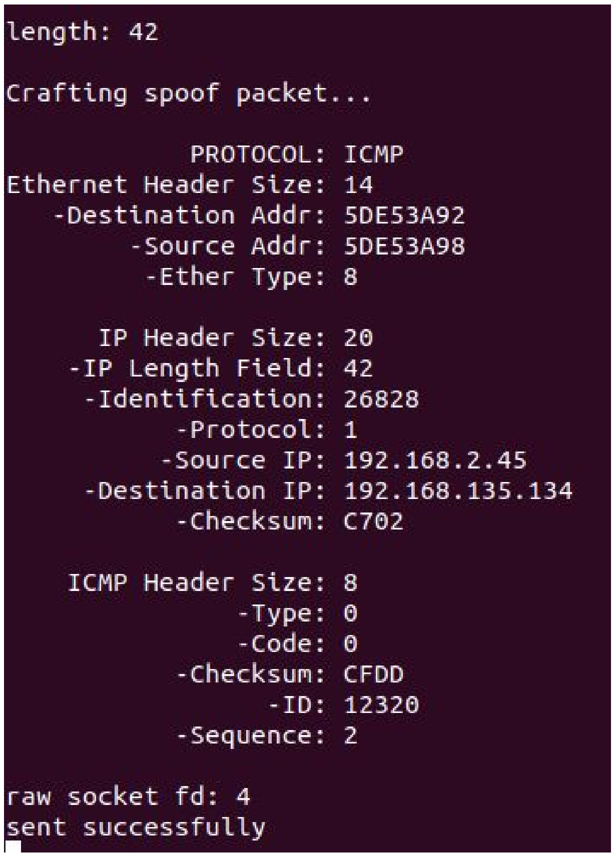
```
   Src port: 32798
   Dst port: 23

Packet number 138:
       From: 10.2.2.1
         To: 10.2.1.1
   Protocol: TCP
   Src port: 23
   Dst port: 32798
   Payload (10 bytes):
00000   50 61 73 73 77 6f 72 64  3a 20                      Password:

Packet number 139:
       From: 10.2.1.1
         To: 10.2.2.1
   Protocol: TCP
   Src port: 32798
   Dst port: 23

Packet number 140:
       From: 10.2.1.1
         To: 10.2.2.1
   Protocol: TCP
   Src port: 32798
   Dst port: 23
   Payload (1 bytes):
00000   52                                                  R

Packet number 141:
       From: 10.2.2.1
         To: 10.2.1.1
   Protocol: TCP
   Src port: 23
   Dst port: 32798

Packet number 142:
       From: 10.2.1.1
         To: 10.2.2.1
   Protocol: TCP
```

e. I filtered by port 23 and scrolled to see the username and password

6. Please use your own words to describe the sequence of the library calls that are essential for packet spoofing. This is meant to be a succinct summary.
   a. First and foremost, a buffer the size of the packet must be created to write into. After that, it helps to make structs that match the structure of the packet, but is not necessary. Then a raw socket is opened and the packet is written into the buffer. Finally, the spoofed packet is sent to the destination through the raw socket

7. 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?
    a. They are needed to use raw sockets as a raw socket object receives all packets on the network, unlike normal sockets which declare a port and have other traffic filtered out. Since this could be potentially dangerous for the other applications running on the machine and the network, sudo is needed to allow the program to run.
8. Please combine your sniffing and the spoofing programs to implement a sniff-and-then spoof program. This program monitors its local network; whenever it sees an ICMP echo request packet, it spoofs an ICMP echo reply packet with the information of another machine (other than the sender and receiver) in the network (in your AWS enclave). Please include screendump in your report to show that your program works (you might capture ICMP echo request and reply from the sender machine). Please also attach the code in your report.
    a. 

```
dev2@ubuntu:~/Desktop/CSCE465-HW1$ sudo ./sns ens33 100 icmp
Device: ens33
Number of packets: 100
Filter expression: icmp

Packet number 1:
Ethernet Header Size: 14
   -Destination Addr: F7F82086
        -Source Addr: F7F8208C
         -Ether Type: 8

      IP Header Size: 20
    -IP Length Field: 84
     -Identification: 26828
           -Protocol: 1
          -Source IP: 192.168.135.134
     -Destination IP: 192.168.2.45
           -Checksum: C6D8

            Protocol: ICMP
    ICMP Header Size: 8
               -Type: 8
               -Code: 0
           -Checksum: 968E
                 -ID: 12320
           -Sequence: 1

length: 42
```

b. 

```
length: 42

Crafting spoof packet...

            PROTOCOL: ICMP
Ethernet Header Size: 14
   -Destination Addr: 5DE53A92
        -Source Addr: 5DE53A98
         -Ether Type: 8

      IP Header Size: 20
    -IP Length Field: 42
     -Identification: 26828
           -Protocol: 1
          -Source IP: 192.168.2.45
     -Destination IP: 192.168.135.134
           -Checksum: C702

    ICMP Header Size: 8
               -Type: 0
               -Code: 0
           -Checksum: CFDD
                 -ID: 12320
           -Sequence: 2

raw socket fd: 4
sent successfully
```

c. AWS is refusing to work with me, as I cannot copy and paste or drag-and-drop files into the RDP session, so I did this on a local VM. It correctly receives the packet, and then reconstructs it as you see above, and throws no errors in sending the packet, however it never actually responds. I am not sure if this is due to both OS's being on the same machine and I would try it on a RasPi but I didn't have time to set it up, please accept this as enough. Below is the code for my program:

```
#include <pcap.h>
#include <unistd.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 <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/ip_icmp.h>

#define SNAP_LEN 1518
```

```
/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN   6

//using namespace std;
#pragma pack(push, 1)
/* Ethernet header */
struct ethernet_header {
       u_char  dest_addr[ETHER_ADDR_LEN];      /* destination host address */
       u_char  source_addr[ETHER_ADDR_LEN];    /* source host address */
       u_short type;                           /* IP? ARP? RARP? etc */
};

/* IP header */
struct ip_header {
       u_char  ip_vhl;                         /* version << 4 | header length >> 2
*/
       u_char  type_of_service;                /* type of service */
       u_short length;                         /* total length */
       u_short id;                             /* identification */
       u_short fragment_offset;                /* 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  time_to_live;                   /* time to live */
       u_char  proto;                          /* protocol */
       u_short checksum;                       /* 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 tcp_header {
       u_short source_port;               /* source port */
       u_short dest_port;               /* destination port */
       tcp_seq seq_num;                 /* sequence number */
       tcp_seq ack_num;                 /* acknowledgement number */
       u_char  th_offx2;               /* data offset, rsvd */

          #define TH_OFF(th)      (((th)->th_offx2 & 0xf0) >> 4)
```

```
        u_char  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 window;                 /* window */
        u_short checksum;                 /* checksum */
        u_short urgent_pointer;                 /* urgent pointer */
};

struct icmp_header {
    u_char type;
    u_char code;
    u_short checksum;
    u_short id;
   u_short seq;
};
#pragma pack(pop)


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

void print_payload(const u_char *payload, int len);

void print_hex_ascii_line(const u_char *payload, int len, int offset);

/**
* Calculates the checksum of length bytes, starting at *addr
*/
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) {
```

```c
        *(unsigned char *) (&answer) = *(unsigned char *) w;
        sum += answer;
    }

    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    answer = ~sum;
    return (answer);
}


void print_hex_ascii_line(const u_char *payload, int len, int offset) {

    int i;
    int gap;
    const u_char *ch;

    /* offset */
    printf("%05d   ", offset);

    /* hex */
    ch = payload;
    for (i = 0; i < len; i++) {
        printf("%02x ", *ch);
        ch++;
        /* print extra space after 8th byte for visual aid */
        if (i == 7)
            printf(" ");
    }
    /* print space to handle line less than 8 bytes */
    if (len < 8)
        printf(" ");

    /* fill hex gap with spaces if not full line */
    if (len < 16) {
        gap = 16 - len;
        for (i = 0; i < gap; i++) {
            printf("   ");
        }
    }
    printf("   ");

    /* ascii (if printable) */
    ch = payload;
    for (i = 0; i < len; i++) {
        if (isprint(*ch))
            printf("%c", *ch);
        else
            printf(".");
        ch++;
    }

    printf("\n");

return;
}


/*
* print packet payload data (avoid printing binary data)
*/
void print_payload(const u_char *payload, int len) {

    int len_rem = len;
    int line_width = 16;            /* number of bytes per line */
    int line_len;
    int offset = 0;                 /* zero-based offset counter */
    const u_char *ch = payload;

    if (len <= 0)
        return;

    /* data fits on one line */
    if (len <= line_width) {
        print_hex_ascii_line(ch, len, offset);
        return;
    }

    /* data spans multiple lines */
    for ( ;; ) {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width) {
            /* print last line and get out */
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }

return;
}

void echo_imcp(u_char* packet, int ethernet_length, int ip_length) {
   struct ethernet_header  ethernet = *(struct ethernet_header*)packet;
   struct ip_header        ip = *(struct ip_header*)((char*)(packet +
ethernet_length));
   struct icmp_header      icmp = *(struct icmp_header*)((char*)(packet +
ethernet_length + ip_length));       /* The ICMP header */

   if (icmp.type != 8)
       return;

   char buffer[1024];

   //copy ethernet header into buffer
   memcpy(buffer, (char*)&ethernet, ethernet_length);
   //cast buffer to ethernet_header
   struct ethernet_header bufeth = *(struct ethernet_header*)buffer;

   in_addr ip_src = ip.ip_src;
   in_addr ip_dst = ip.ip_dst;

   //copy ip header into buffer
   memcpy((char*)(buffer + ethernet_length),(char*)&ip, ip_length);
   //cast buffer to ip_header
   struct ip_header bufip = *(struct ip_header*)((char*)(buffer +
ethernet_length));

   bufip.ip_src = ip_dst;
   bufip.ip_dst = ip_src;

   bufip.length = htons(ethernet_length + ip_length + sizeof(icmp_header));
   bufip.id;
   bufip.checksum = 0;
   bufip.checksum = in_cksum((u_short*)&bufip, ip_length);

   //cast buffer to icmp_header
   struct icmp_header buficmp = *(struct icmp_header*)((char*)(buffer +
ethernet_length + ip_length));

   buficmp.type = 0;
   buficmp.code = 0;
```

```
    buficmp.checksum = 0;
    buficmp.id = icmp.id;
    buficmp.seq = htons(ntohs(icmp.seq)+1);
    buficmp.checksum = in_cksum((u_short*)&buficmp, sizeof(icmp_header));




    printf("length: %d\n", ethernet_length + ip_length + sizeof(struct
icmp_header));
    printf("\nCrafting spoof packet...\n\n");
    printf("            PROTOCOL: ICMP\n");
    // printf("From: %s\n", inet_ntoa(bufip.ip_src));
     // printf("To: %s\n", inet_ntoa(bufip.ip_dst));
    printf("Ethernet Header Size: 14\n");
    printf("   -Destination Addr: %X\n", bufeth.dest_addr);
    printf("        -Source Addr: %X\n", bufeth.source_addr);
    printf("         -Ether Type: %X\n\n", bufeth.type);
     printf("      IP Header Size: %d\n", ip_length);
    printf("    -IP Length Field: %d\n", ntohs(bufip.length));
    printf("     -Identification: %d\n", ntohs(bufip.id));
    printf("           -Protocol: %X\n", bufip.proto);
    printf("          -Source IP: %s\n", inet_ntoa(bufip.ip_src));
    printf("     -Destination IP: %s\n", inet_ntoa(bufip.ip_dst));
    printf("           -Checksum: %X\n\n", ntohs(bufip.checksum));
    printf("    ICMP Header Size: %d\n", sizeof(buficmp));
    printf("               -Type: %d\n", buficmp.type);
    printf("               -Code: %d\n", buficmp.code);
    printf("           -Checksum: %X\n", ntohs(buficmp.checksum));
    printf("                 -ID: %d\n", ntohs(buficmp.id));
    printf("           -Sequence: %d\n\n", ntohs(buficmp.seq));
    //printf("               -Data: %X\n\n", ntohl(buficmp.data));


     int raw_sock;

     if ( (raw_sock = socket(AF_INET, SOCK_RAW, IPPROTO_UDP)) < 0) {
         perror("raw socket");
         exit(1);
     }

     const int on = 1;

     if (setsockopt(raw_sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
         perror("setsockopt");
         exit(1);
     }
```

```
   printf("raw socket fd: %d\n", raw_sock);

    struct sockaddr_in sin;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr = ip_src;

    if (sendto(raw_sock, buffer, ethernet_length + ip_length + sizeof(buficmp), 0,
(struct sockaddr*)&sin, sizeof(struct sockaddr)) < 0) {
        perror("sendto");
        exit(1);
    }
    printf("sent successfully\n");
    close(raw_sock);
    return;
}


/*
* 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 ethernet_header *ethernet;  /* The ethernet header [1] */
    const struct ip_header *ip;              /* The IP header */
    const struct tcp_header *tcp;            /* The TCP header */
   const struct icmp_header *icmp;

    const char *payload;                     /* Packet payload */

    int size_ip;
    int size_icmp;
    int size_payload;



    printf("\nPacket number %d:\n", count);
    count++;

    /* define ethernet header */
    ethernet = (struct ethernet_header*)(packet);
```

```
 /* define/compute ip header offset */
 ip = (struct ip_header*)(packet + SIZE_ETHERNET);
 size_ip = IP_HL(ip)*4;
 if (size_ip < 20) {
     printf("   * Invalid IP header length: %u bytes\n", size_ip);
     return;
 }


 /* print source and destination IP addresses */
 // printf("From: %s\n", inet_ntoa(ip->ip_src));
 // printf("  To: %s\n", inet_ntoa(ip->ip_dst));
printf("Ethernet Header Size: 14\n");
printf("   -Destination Addr: %X\n", ethernet->dest_addr);
printf("        -Source Addr: %X\n", ethernet->source_addr);
printf("         -Ether Type: %X\n\n", ethernet->type);
 printf("      IP Header Size: %d\n", size_ip);
printf("    -IP Length Field: %d\n", ntohs(ip->length));
printf("     -Identification: %d\n", ntohs(ip->id));
printf("           -Protocol: %X\n", ip->proto);
printf("          -Source IP: %s\n", inet_ntoa(ip->ip_src));
printf("     -Destination IP: %s\n", inet_ntoa(ip->ip_dst));
printf("           -Checksum: %X\n\n", ntohs(ip->checksum));

// printf("        -Source Port: %d\n", ntohs(ip_.source_port));
// printf("   -Destination Port: %d\n", ntohs(ip_.dest_port));

 u_char* packet_copy;

 /* determine protocol */
 switch(ip->proto) {
     case IPPROTO_TCP:
         printf("            Protocol: TCP\n");
         break;
     case IPPROTO_UDP:
         printf("            Protocol: UDP\n");
         break;
     case IPPROTO_ICMP:
         printf("            Protocol: ICMP\n");
        icmp = (struct icmp_header*)(packet + SIZE_ETHERNET + size_ip);
        printf("    ICMP Header Size: %d\n", sizeof(*icmp));
        printf("               -Type: %d\n", icmp->type);
        printf("               -Code: %d\n", icmp->code);
        printf("           -Checksum: %X\n", ntohs(icmp->checksum));
        printf("                 -ID: %d\n", ntohs(icmp->id));
        printf("           -Sequence: %d\n\n", ntohs(icmp->seq));
```

```
           packet_copy = new u_char[SIZE_ETHERNET + size_ip + sizeof(struct
icmp_header)];
           memcpy(packet_copy, packet, SIZE_ETHERNET + size_ip + sizeof(struct
icmp_header));
           echo_imcp(packet_copy, SIZE_ETHERNET, size_ip);
            break;
        case IPPROTO_IP:
            printf("            Protocol: IP\n");
            break;
        default:
            printf("            Protocol: unknown\n");
            break;
    }

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* fileLocation;
    bool useFile = false;

    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 = 100;                    /* number of packets to capture */

    /* check for capture device name on command-line */
    if (argc == 2) {
        dev = argv[1];
        useFile = false;
       filter_exp = "icmp";
    } else if (argc == 3) {
        fileLocation = argv[1];
        useFile = true;
        num_packets = atoi(argv[2]);
       filter_exp = "icmp";
    } else if (argc == 4) {
       dev = argv[1];
       useFile = false;
       num_packets = atoi(argv[2]);
    filter_exp = argv[3];
} else if (argc > 4) {
     fprintf(stderr, "error: unrecognized command-line options\n\n");
     exit(EXIT_FAILURE);
 } else {
     /* find a capture device if not specified on command-line */
     dev = pcap_lookupdev(errbuf);
    filter_exp = "icmp";
     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);

 if (useFile)
     handle = pcap_open_offline(fileLocation, errbuf);
 else
     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) {
        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;
}
```