

# Project 3 Sniffing and Spoofing

Yang Guo [yguo3@clemson.edu](mailto:yguo3@clemson.edu)

## Task 1 Writing Packet Sniffing Program

### Task1.a: Understanding sniffex

```
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.
```

```
Device: eth0
Number of packets: 10
Filter expression: ip
```

```
Packet number 1:
  From: 10.0.2.15
  To: 224.0.0.251
  Protocol: UDP
```

```
Packet number 2:
  From: 10.0.2.15
  To: 4.34.16.35
  Protocol: TCP
  Src port: 45795
  Dst port: 80
```

```
Packet number 3:
  From: 4.34.16.35
  To: 10.0.2.15
  Protocol: TCP
  Src port: 80
  Dst port: 45795
```

```
Packet number 4:
  From: 4.34.16.35
  To: 10.0.2.15
  Protocol: TCP
  Src port: 80
  Dst port: 45795
```

```
Packet number 5:
  From: 10.0.2.15
  To: 4.34.16.35
  Protocol: TCP
  Src port: 45795
  Dst port: 80
```

```
Packet number 6:
  From: 10.0.2.15
  To: 130.127.255.250
  Protocol: UDP
```



The screenshot shows that sniffex runs successfully and capture all the packets.

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.

(1) Setting the device.

```
Dev = pcap_lookupdev(errbuf);
```

pcap just sets the device on its own. In the event that the pcap\_lookupdev(errbuf) fails, it will store an error message in errbuf.

(2) Open the device for sniffing

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)
```

Tell pcap which device we are sniffing now.

(3) Filtering traffic. Before we sniff on a specific device, we must know its ip address and net mask.

```
pcap_lookupnet()
```

Then apply the filter.

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

(4) The actual sniffing.

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

It will capture a single packet at one time.

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

It will call back function every time a packet is sniffed and pass our filter test.

(5) Close the sniffer session.

```
pcap_close()
```

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

Without the root privilege we can't find the default device. So that we can't run sniffex successfully.

```
[10/19/2016 07:53] seed@ubuntu:~/project3$  
[10/19/2016 07:53] seed@ubuntu:~/project3$ ./sniffex  
sniffex - Sniffer example using libpcap  
Copyright (c) 2005 The Tcpdump Group  
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.  
  
Couldn't find default device: no suitable device found  
[10/19/2016 07:53] seed@ubuntu:~/project3$ █
```

The problem fails due to the code here:

```
dev = pcap_lookupdev(errbuf);
```

```
if (dev == NULL) {
```

```

        fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
        return(2);
    }

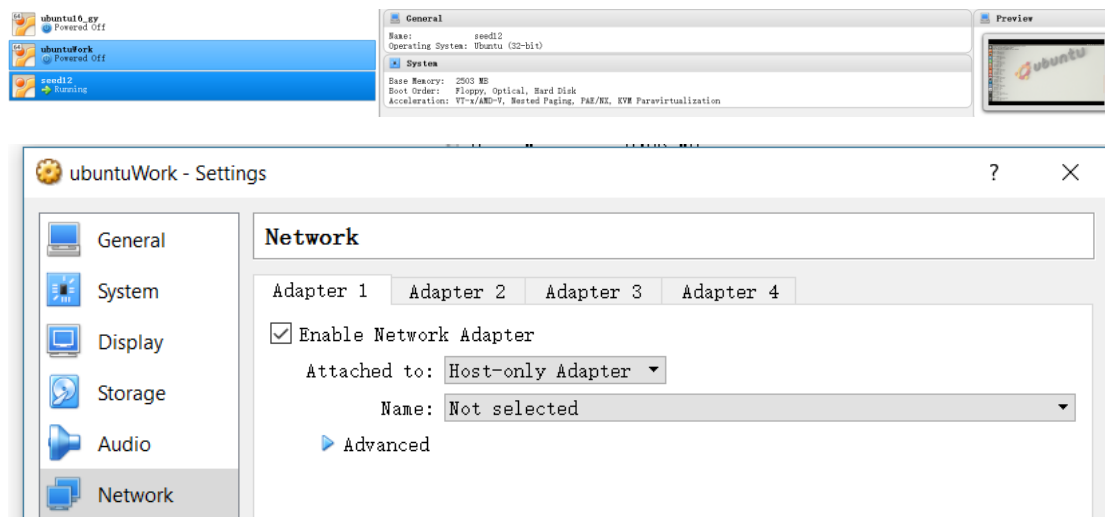
```

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.

These two modes are very different. In non-promiscuous mode, the host machine only capture traffic and packets related to it. In promiscuous mode, the host can capture all the traffic.

To do demonstration test, first I set up three Virtual Machine. And set all machines' network setting as host-only.



Turn on promiscuous mode. We need to set the parameters as below.

```

/* 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] */

```

Use two machines to ping each other. The IP of them are 192.168.52.132 and 192.168.52.129.

```
root@ubuntu:/home# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:71:62:37
          inet addr:192.168.52.132  Bcast:192.168.52.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe71:6237/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:919 errors:0 dropped:0 overruns:0 frame:0
          TX packets:133 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:61939 (61.9 KB)  TX bytes:16528 (16.5 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:164 errors:0 dropped:0 overruns:0 frame:0
          TX packets:164 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:12051 (12.0 KB)  TX bytes:12051 (12.0 KB)
```

```
root@ubuntu:/home# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:71:62:37
          inet addr:192.168.52.132  Bcast:192.168.52.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe71:6237/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:983 errors:0 dropped:0 overruns:0 frame:0
          TX packets:110 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:71344 (71.3 KB)  TX bytes:14495 (14.4 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:159 errors:0 dropped:0 overruns:0 frame:0
          TX packets:159 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11547 (11.5 KB)  TX bytes:11547 (11.5 KB)

root@ubuntu:/home# ping 192.168.52.132
PING 192.168.52.132 (192.168.52.132) 56(84) bytes of data.
64 bytes from 192.168.52.132: icmp_seq=1 ttl=64 time=2.29 ms
64 bytes from 192.168.52.132: icmp_seq=2 ttl=64 time=0.638 ms
64 bytes from 192.168.52.132: icmp_seq=3 ttl=64 time=0.483 ms
64 bytes from 192.168.52.132: icmp_seq=4 ttl=64 time=1.76 ms
^Z
[1]+  Stopped                  ping 192.168.52.132
root@ubuntu:/home#
```

The host machine can capture packets between two communicating machines.

```
[10/18/2016 21:20] root@ubuntu:/home/seed/Downloads# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: ip

Packet number 1:
  From: 192.168.52.129
  To: 192.168.52.132
  Protocol: ICMP

Packet number 2:
  From: 192.168.52.132
  To: 192.168.52.129
  Protocol: ICMP

Packet number 3:
  From: 192.168.52.129
  To: 192.168.52.132
```

Turn off promiscuous mode, set parameters as below.

```
/* open capture device */
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
if (handle == NULL) {
    fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
    exit(EXIT_FAILURE);
}
```

While machine 192.168.52.129 ping 192.168.52.132. Host machine can't capture any packets.

```
root@ubuntu:/home# ping 192.168.52.132
PING 192.168.52.132 (192.168.52.132) 56(84) bytes of data.
64 bytes from 192.168.52.132: icmp_seq=1 ttl=64 time=0.455 ms
64 bytes from 192.168.52.132: icmp_seq=2 ttl=64 time=0.634 ms
64 bytes from 192.168.52.132: icmp_seq=3 ttl=64 time=0.779 ms
64 bytes from 192.168.52.132: icmp_seq=4 ttl=64 time=0.558 ms
64 bytes from 192.168.52.132: icmp_seq=5 ttl=64 time=0.534 ms
64 bytes from 192.168.52.132: icmp_seq=6 ttl=64 time=0.532 ms
64 bytes from 192.168.52.132: icmp_seq=7 ttl=64 time=0.591 ms
^Z
[2]+  Stopped                  ping 192.168.52.132

[10/18/2016 21:24] root@ubuntu:/home/seed/Downloads# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: ip
```

Only when one machine ping host machine, program sniffex can capture packets.

```
root@ubuntu:/home# ping 192.168.52.131
PING 192.168.52.131 (192.168.52.131) 56(84) bytes of data.
64 bytes from 192.168.52.131: icmp_seq=1 ttl=64 time=0.599 ms
64 bytes from 192.168.52.131: icmp_seq=2 ttl=64 time=0.591 ms
64 bytes from 192.168.52.131: icmp_seq=3 ttl=64 time=0.425 ms
^Z
[3]+  Stopped                  ping 192.168.52.131
root@ubuntu:/home#

[10/18/2016 21:26] root@ubuntu:/home/seed/Downloads# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: ip

Packet number 1:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: ICMP

Packet number 2:
  From: 192.168.52.131
  To: 192.168.52.129
  Protocol: ICMP

Packet number 3:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: ICMP
```



Now we have demonstrated differences between promiscuous and non-promiscuous mode, only in promiscuous mode we can all the packets in the same local network.

### Task1.b: Writing Filters.

- **Capture ICMP packets**

First we should change the filter expression as below.

```
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();
```

Use one machine to ping another.

```
root@ubuntu:/home# ping 192.168.52.131
PING 192.168.52.131 (192.168.52.131) 56(84) bytes of data.
64 bytes from 192.168.52.131: icmp_seq=1 ttl=64 time=0.432 ms
64 bytes from 192.168.52.131: icmp_seq=2 ttl=64 time=0.678 ms
64 bytes from 192.168.52.131: icmp_seq=3 ttl=64 time=0.718 ms
^Z
[4]+  Stopped                  ping 192.168.52.131
root@ubuntu:/home#
```

Then run sniffex in host machine and we can see the result.

```
[10/18/2016 21:29] root@ubuntu:/home/seed/Downloads# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: icmp

Packet number 1:
    From: 192.168.52.129
    To: 192.168.52.131
    Protocol: ICMP

Packet number 2:
    From: 192.168.52.131
    To: 192.168.52.129
    Protocol: ICMP

Packet number 3:
```

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

Change the filter expression as below.

```

char filter_exp[] = "tcp and 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 = 10;                                  /* number of packets to capture */

```

Since telnet use TCP protocol, telnet another machine from one machine.

```

root@ubuntu:/home# telnet 192.168.52.131
Trying 192.168.52.131...
Connected to 192.168.52.131.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
ubuntu login: ^Z

```

Run sniffex in sniffing host machine, we can see the result.

```

[10/18/2016 21:32] root@ubuntu:/home/seed/Downloads# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: tcp and portrange 10-100

Packet number 1:
    From: 192.168.52.129
    To: 192.168.52.131
    Protocol: TCP
    Src port: 51628
    Dst port: 23

Packet number 2:
    From: 192.168.52.131
    To: 192.168.52.129
    Protocol: TCP
    Src port: 23
    Dst port: 51628

Packet number 3:

```

### Task1.c: Sniffing Passwords.

First we need to change the filter expression as below.

```

char filter_exp[] = "tcp and port 23";                  /* 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 */

```

Then start to telnet to the sniffing host machine from one machine with username and password.

```
root@ubuntu:/home# telnet 192.168.52.131
Trying 192.168.52.131...
Connected to 192.168.52.131.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
ubuntu login: seed
Password:
Last login: Tue Oct 18 21:35:13 PDT 2016 from ubuntu-2.local on pts/1
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.
```

We can see the result from the sniffing host machine.

```
    Payload (12 bytes):
000000  0d 0a 50 61 73 73 77 6f  72 64 3a 20                ..Password:

Packet number 33:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: TCP
  Src port: 51632
  Dst port: 23

Packet number 34:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: TCP
  Src port: 51632
  Dst port: 23
  Payload (1 bytes):
000000  64                          d

Packet number 35:
  From: 192.168.52.131
  To: 192.168.52.129
  Protocol: TCP
  Src port: 23
  Dst port: 51632

Packet number 36:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: TCP
  Src port: 51632
  Dst port: 23
  Payload (1 bytes):
000000  65                          e

Packet number 37:
  From: 192.168.52.131
  To: 192.168.52.129
  Protocol: TCP
  Src port: 23
  Dst port: 51632

Packet number 38:
  From: 192.168.52.129
  To: 192.168.52.131
  Protocol: TCP
```

As we can see from above, using sniffex we can capture the password when somebody is using telnet on the network we are monitoring.



## Task 2 Spoofing

### Task2.a Write a spoofing program

I search and download the source code from website.

URL: [https://github.com/DaNotKnow/Raw\\_Socket](https://github.com/DaNotKnow/Raw_Socket)

Here is the main part of the code:

```
struct ipheader{
    unsigned int ip_version:4, ip_ihl:4;
    unsigned char ip_typeOfService;
    unsigned short int ip_totalLength;
    unsigned short int ip_identification;
    unsigned char ip_flags;
    unsigned short int ip_fragmentOffset;
    unsigned char ip_timeToLive;
    unsigned char ip_protocol;
    unsigned short int ip_headerChecksum;
    unsigned long int ip_sourceIpAddress;
    unsigned long int ip_destinationIpAddress;
};

struct udpheader{
    unsigned short int udp_sourcePortNumber;
    unsigned short int udp_destinationPortNumber;
    unsigned short int udp_length;
    unsigned short int udp_checksum;
};

unsigned short csum(unsigned short *ptr,int nbytes)
{
    register long sum;
    unsigned short oddbyte;
    register short answer;

    sum=0;
    while(nbytes>1) {
        sum+=*ptr++;
        nbytes-=2;
    }
    if(nbytes==1) {
        oddbyte=0;
        *((u_char*)&oddbyte)=*(u_char*)ptr;
        sum+=oddbyte;
    }

    sum = (sum>>16)+(sum & 0xffff);
    sum = sum + (sum>>16);
    answer=(short)~sum;
    return(answer);
}
```

```

//zera a stack relacionada ao packet
memset( packet, 0, sizeof(packet) );

//Monta os cabecinhos no packet
// [ [ IP HEADER] [UDP HEADER] [PAYLOAD] ]
ip = (struct iphdr *)packet;
udp = (struct udpheader *) (packet + sizeof(struct iphdr) );
data = (char *) (packet + sizeof(struct iphdr) + sizeof(struct udpheader) );
strcpy(data, PAYLOAD);

//Preenche o cabecalho IP
ip->version = 4; //IPv4
ip->ihl = 5; //Tamanho do cabecalho, minimo eh 5
ip->tos = 0; //Tipo de servico, default eh 0 que representa opcoes normais
ip->tot_len = sizeof(packet); //tamanho do datagrama
ip->id = htonl(1234); // Numero de identificacao do pacote
ip->frag_off = 0; // Flags, zero representa reservado (Preciso pes
// ip->ip_fragmentOffset = 0; // Caso haja fragmentacao de datagramas, este nu
ip->ttl = 255; // Maximo 255
ip->protocol = IPPROTO_UDP; // Apresenta UDP como protocolo a ser usado na p
ip->check = 0; // Calculado depois
ip->saddr = inet_addr( argv[1] ); //Ip de origem ( spoofing ocorre aqui)
ip->daddr = inet_addr( argv[3] ); //IP de destino

//Preenche o cabecalho UDP
udp->udp_sourcePortNumber = htonl( atoi(argv[2]) ); //Porta de origem
udp->udp_destinationPortNumber = htonl( atoi(argv[4]) ); //Porta de destino
udp->udp_length = htonl( sizeof(struct udpheader) ); //Tamanho do cabecalho udp
udp->udp_checksum = 0; //Calculado depois

//Calcula os checksums
ip->check = csum((unsigned short *)packet, ip->tot_len); //Calcula o checksum do cab
udp->udp_checksum = csum((unsigned short *)udp, udp->udp_length); //Calcul

//Armazena os dados do destino
dest.sin_family = AF_INET; // Address Family Ipv4
dest.sin_port = htons( atoi( argv[4] ) ); // Porta de destino, redundant
dest.sin_addr.s_addr = inet_addr( argv[3] ); // Endereço para onde se quer

//Loop principal
int count = 0;
while( count < 100){

    // Envia os pacotes
    status = sendto(sockfd, packet, ip->tot_len, 0, (struct sockaddr *)&dest, sizeof(dest) );
    if(status < 0){
        printf("Erro ao enviar os pacotes\n");
        exit(1);
    }
    count++;
}
printf("Enviados %d pacotes de tamanho: %d, olhe o Wireshark!!!\n", count, ip->tot_len);
exit(0);

```

After compiling the code, I run this program as root user. The result is as below:

```

[10/18/2016 19:35] seed@ubuntu:~/project3/raw$ ./simpleRaw 1.2.3.4 22 8.8.8.8 53
Erro ao iniciar o socket
[10/18/2016 19:37] seed@ubuntu:~/project3/raw$ ./simpleRaw 1.2.3.4 22 8.8.8.8 53
^C
[10/18/2016 19:37] seed@ubuntu:~/project3/raw$ ifconfig
eth0
    Link encap:Ethernet  HWaddr 08:00:27:82:df:82
    inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
    inet6 addr: fe80::a00:27ff:fe82:df82/64  Scope:Link
    UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
    RX packets:7322 errors:0 dropped:0 overruns:0 frame:0
    TX packets:3602 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:8227633 (8.2 MB)  TX bytes:404967 (404.9 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:70 errors:0 dropped:0 overruns:0 frame:0
    TX packets:70 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:4895 (4.8 KB)  TX bytes:4895 (4.8 KB)

[10/18/2016 19:39] seed@ubuntu:~/project3/raw$ ./simpleRaw 10.0.3.25 23 8.8.8.8
53
[10/18/2016 19:40] seed@ubuntu:~/project3/raw$ su
Password:
su: Authentication failure
[10/18/2016 19:40] seed@ubuntu:~/project3/raw$ su
Password:
[10/18/2016 19:41] root@ubuntu:/home/seed/project3/raw# ./simpleRaw 10.0.2.25 22
8.8.8.8 53
Enviados 100 pacotes de tamanho: 57, olhe o Wireshark!!!
[10/18/2016 19:42] root@ubuntu:/home/seed/project3/raw# su seed
[10/18/2016 19:42] seed@ubuntu:~/project3/raw$

```

1	2016-10-18 19:41:08.06	fe80::a00:27ff:fe82:df82::fb	MDNS	101 Standard query PTR sane-port. tcp.local, "QM" question
2	2016-10-18 19:41:08.06	10.0.2.15	MDNS	81 Standard query PTR sane-port. tcp.local, "QM" question
3	2016-10-18 19:41:09.01	10.0.2.15	DNS	76 Standard query A daisy.ubuntu.com
4	2016-10-18 19:41:09.02	8.8.8.8	DNS	108 Standard query response A 162.213.33.133 A 162.213.33.164
5	2016-10-18 19:41:14.01	CadmusCo_82:df:82	ARP	42 Who has 10.0.2.2? Tell 10.0.2.15
6	2016-10-18 19:41:14.01	RealtekU_12:35:02	ARP	60 10.0.2.2 is at 52:54:00:12:35:02
7	2016-10-18 19:41:38.04	10.0.2.15	DNS	76 Standard query A daisy.ubuntu.com
8	2016-10-18 19:41:38.05	8.8.8.8	DNS	108 Standard query response A 162.213.33.133 A 162.213.33.164
9	2016-10-18 19:41:39.04	10.0.2.15	DNS	76 Standard query A daisy.ubuntu.com
10	2016-10-18 19:41:39.05	8.8.8.8	DNS	108 Standard query response A 162.213.33.164 A 162.213.33.133
11	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
12	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
13	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
14	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
15	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
16	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
17	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
18	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
19	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
20	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
21	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
22	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
23	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]
24	2016-10-18 19:42:24.93	10.0.2.25	UDP	71 Source port: 0 Destination port: 0 [BAD UDP LENGTH 0 < 8]

As you can see from screenshots above, the IP of my computer is 10.0.2.15. In the Wireshark it has successfully faked as 10.0.2.25.

Task2.b: Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

I download the code from web.

URL <http://www.pdbuchan.com/rawsock/icmp4.c>

First, set an arbitrary IP address as the IP address of my machine and set the target address in the code.

```
// Source IPv4 address: you need to fill this out
strcpy (src_ip, "192.168.1.166");

// Destination URL or IPv4 address: you need to fill this out
strcpy (target, "www.clemson.edu");
```

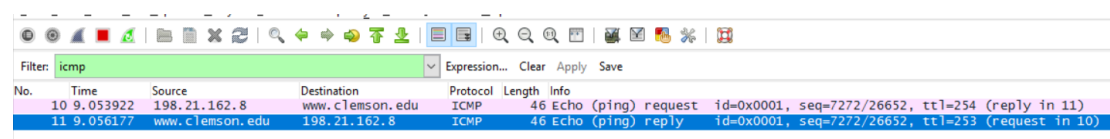
After this, compile the code and run it as root user.

```
[10/19/2016 12:24] seed@ubuntu:~/project3$ cd icmp
[10/19/2016 12:24] seed@ubuntu:~/project3/icmp$ ls
icmp4.c icmp4.c~
[10/19/2016 12:25] seed@ubuntu:~/project3/icmp$ gcc -o icmp icmp4.c
[10/19/2016 12:25] seed@ubuntu:~/project3/icmp$ ls
icmp icmp4.c icmp4.c~
[10/19/2016 12:25] seed@ubuntu:~/project3/icmp$ su
Password:
[10/19/2016 12:25] root@ubuntu:/home/seed/project3/icmp# ./icmp
Index for interface eth0 is 2
[10/19/2016 12:25] root@ubuntu:/home/seed/project3/icmp#
```

Capture packets using Wireshark. Here is the result.

1	2016-10-19 12:25:58.6110.0.2.15	130.127.255.250	DNS	75 Standard query A www.clemson.edu
2	2016-10-19 12:25:58.61130.127.255.250	10.0.2.15	DNS	258 Standard query response A 130.127.204.30
3	2016-10-19 12:25:58.6110.0.2.15	130.127.255.250	DNS	87 Standard query PTR 30.204.127.130.in-addr.arpa
4	2016-10-19 12:25:58.61130.127.255.250	10.0.2.15	DNS	283 Standard query response PTR www.clemson.edu
5	2016-10-19 12:25:58.61192.168.1.166	130.127.204.30	ICMP	46 Echo (ping) request id=0x03e8, seq=0/0, ttl=255
6	2016-10-19 12:25:58.62RealtekU_12:35:02	Broadcast	ARP	60 Who has 192.168.1.166? Tell 10.0.2.2

From above we can see that the actual IP address is 10.0.2.15. And machine send an ICMP packet on behalf of 192.168.1.166. After sending it, there is an ARP packet because gateway has received the responding message and need to redirect the message. To demonstrate this, I use Wireshark to capture packets in Windows. And it shows that when I run program using the faked IP address in seed Ubuntu system, it can send ICMP and get response from target.



No.	Time	Source	Destination	Protocol	Length	Info
10	9.053922	198.21.162.8	www.clemson.edu	ICMP	46	Echo (ping) request id=0x0001, seq=7272/26652, ttl=254 (reply in 11)
11	9.056177	www.clemson.edu	198.21.162.8	ICMP	46	Echo (ping) reply id=0x0001, seq=7272/26652, ttl=253 (request in 10)

It demonstrates spoofing is successful.

## Task2.c: Spoof an Ethernet Frame.

I download the code from web.

URL [http://www.pdbuchan.com/rawsock/icmp4\\_frag.c](http://www.pdbuchan.com/rawsock/icmp4_frag.c)

First, as the same as before, set an arbitrary MAC address and IP address as the faked address and set the target address in the code.



```

// Copy source MAC address.
//memcpy (src_mac, ifr.ifr_hwaddr.sa_data, 6 * sizeof (uint8_t));
src_mac[0] = 0x01;
src_mac[1] = 0x02;
src_mac[2] = 0x03;
src_mac[3] = 0x04;
src_mac[4] = 0x05;
src_mac[5] = 0x06;

// Source IPv4 address: you need to fill this out
strcpy (src_ip, "192.168.1.166");

// Destination URL or IPv4 address: you need to fill this out
strcpy (target, "www.clemson.edu");

// Submit request for a raw socket descriptor.
if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
    perror ("socket() failed ");
    exit (EXIT_FAILURE);
}

```

We can check the actual MAC and IP address.

```

[10/19/2016 12:39] seed@ubuntu:~/project3/ethernetFrame$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:82:df:82
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe82:df82/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:31983 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13260 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:34438461 (34.4 MB)  TX bytes:1404667 (1.4 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:16436  Metric:1
          RX packets:94 errors:0 dropped:0 overruns:0 frame:0
          TX packets:94 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6981 (6.9 KB)  TX bytes:6981 (6.9 KB)

```

Compile the code and run it as root user. We can see from screenshot below MAC address is successfully faked as 01:02:03:04:05:06

```

[10/19/2016 14:05] seed@ubuntu:~/project3/ethernetFrame$ su
Password:
[10/19/2016 14:05] root@ubuntu:/home/seed/project3/ethernetFrame# ./icmp_f
Current MTU of interface eth0 is: 1500
MAC address for interface eth0 is 01:02:03:04:05:06
Index for interface eth0 is 2
Upper layer protocol header length (bytes): 8
Payload length (bytes): 12
Total fragmentable data (bytes): 20
Frag: 0, Data (bytes): 20, Data Offset (8-byte blocks): 0
Total number of frames to send: 1
Sending fragment: 0
[10/19/2016 14:05] root@ubuntu:/home/seed/project3/ethernetFrame#

```



Using Wireshark to capture packets and check the result with the output information from command line, we can see the result and specific information of packet as below.

No.	Time	Source	Destination	Protocol	Length	Info
1	2016-10-19 12:59:48.81	10.0.2.15	130.127.255.250	DNS	75	Standard query A www.clemson.edu
2	2016-10-19 12:59:48.86	130.127.255.250	10.0.2.15	DNS	258	Standard query response A 130.127.204.30
3	2016-10-19 12:59:48.86	10.0.2.15	130.127.255.250	DNS	87	Standard query PTR 30.204.127.130.in-addr.arpa
4	2016-10-19 12:59:48.91	130.127.255.250	10.0.2.15	DNS	283	Standard query response PTR www.clemson.edu
5	2016-10-19 12:59:48.91	192.168.1.166	130.127.204.30	ICMP	54	Echo (ping) request id=0x03e8, seq=0/0, ttl=255
6	2016-10-19 12:59:48.91	RealtekU 12:35:02	Broadcast	ARP	60	Who has 192.168.1.166? Tell 10.0.2.2

▶ Frame 5: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
▼ Ethernet II, Src: Woonsang 04:05:06 (01:02:03:04:05:06), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▶ Destination: Broadcast (ff:ff:ff:ff:ff:ff)
▶ Source: Woonsang 04:05:06 (01:02:03:04:05:06)
Type: IP (0x0800)
▶ Internet Protocol Version 4, Src: 192.168.1.166 (192.168.1.166), Dst: 130.127.204.30 (130.127.204.30)
▶ Internet Control Message Protocol

As we can see from above. The ICMP packet is sent on before of the faked IP 192.168.1.166. And it will get a response. (This has been demonstrated in task2.b) From the specific information of ICMP packet we can see that the source MAC address has been faked as 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?

You can set it to an arbitrary value, but it will cause some problems. When router receives a packet, it will check the length field and determine the size of the packet and how many bytes of data is going to be transmitted.

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

Yes, you must calculate the checksum for the IP header. In the IP header there is a checksum field used for error checking. When the packets arrive at the router, it will calculate checksum and compare it with the checksum field of the header. Using this method, it can ensure integrity.

In the raw socket code, there is a part to calculate the checksum. We can set the checksum as we want, but the packet can't be sent out. I modify the code of task2.c to illustrate this.

```
// IPv4 header checksum (16 bits): set to 0 when calculating checksum
iphdr.ip_sum = 0;
//iphdr.ip_sum = checksum ((uint16_t *) &iphdr, IP4_HDRLEN);
```

1	2016-10-19 16:43:38.92	10.0.2.15	8.8.8.8	DNS	75	Standard query A www.clemson.edu
2	2016-10-19 16:43:38.93	8.8.8.8	10.0.2.15	DNS	91	Standard query response A 130.127.204.30
3	2016-10-19 16:43:38.93	10.0.2.15	8.8.8.8	DNS	87	Standard query PTR 30.204.127.130.in-addr.arpa
4	2016-10-19 16:43:38.94	8.8.8.8	10.0.2.15	DNS	116	Standard query response PTR www.clemson.edu
5	2016-10-19 16:43:38.94	10.0.2.15	130.127.204.30	ICMP	54	Echo (ping) request id=0x03e8, seq=0/0, ttl=255
6	2016-10-19 16:43:43.94	CadmusCo 82:df:82	RealtekU 12:35:02	ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
7	2016-10-19 16:43:43.94	RealtekU 12:35:02	CadmusCo 82:df:82	ARP	60	10.0.2.2 is at 52:54:00:12:35:02

```

▶ Frame 5: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
▶ Ethernet II, Src: CadmusCo_82:df:82 (08:00:27:82:df:82), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▼ Internet Protocol Version 4, Src: 10.0.2.15 (10.0.2.15), Dst: 130.127.204.30 (130.127.204.30)
  Version: 4
  Header length: 20 bytes
  ▶ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
  Total Length: 40
  Identification: 0x7ab7 (31415)
  ▶ Flags: 0x00
  Fragment offset: 0
  Time to live: 255
  Protocol: ICMP (1)
  ▼ Header checksum: 0x0000 [incorrect, should be 0xe670 (maybe caused by "IP checksum offload"?)]
    [Good: False]
    ▼ [Bad: True]
      ▼ [Expert Info (Error/Checksum): Bad checksum]
        [Message: Bad checksum]
        [Severity level: Error]
        [Group: Checksum]
        Source: 10.0.2.15 (10.0.2.15)
        Destination: 130.127.204.30 (130.127.204.30)
  ▶ Internet Control Message Protocol

```

As we can see from above, when we don't calculate the checksum, the packet can't be sent successfully.

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?

Only root user has the privilege to listen to ports below 1024 (also called privileged ports), normal user can't open raw socket. And if normal users can run raw socket program, it's not secure because raw socket program can be used to spoof or do other dangerous things.

Using code of task2.b to illustrate this problem. The program fails without root privilege is due to the code shown below:

```

[10/19/2016 16:23] seed@ubuntu:~/project3/icmp$ gcc -o icmp icmp4.c
[10/19/2016 16:24] seed@ubuntu:~/project3/icmp$ ls
icmp icmp4.c icmp4.c~
[10/19/2016 16:24] seed@ubuntu:~/project3/icmp$ ./icmp
socket() failed to get socket descriptor for using ioctl() : Operation not permitted
[10/19/2016 16:24] seed@ubuntu:~/project3/icmp$ █

```

```

// Submit request for a socket descriptor to look up interface.
if ((sd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    perror ("socket() failed to get socket descriptor for using ioctl() ");
    exit (EXIT_FAILURE);
}

```