COMPUTER NETWORK SECURITY - UE18CS335

ASSIGNMENT 2: SNIFFING AND SPOOFING USING PCAP LIBRARY

Name: Ankitha P

Class: 6 'D'

Date : 07/02/2021

OBJECTIVE

1. Writing programs to sniff and spoof packets using pcap library
2. Understanding how a sniffer works, writing filters, sniffing passwords
3. Understand how spoofing works

Task 1: Writing Programs to Sniff and Spoof Packets using pcap library

Sniffer programs can be easily written using the pcap library. With pcap, the task of sniffers becomes invoking a simple sequence of procedures in the pcap library. At the end of the sequence, packets will be put in buffer for further processing as soon as they are captured. All the details of packet capturing are handled by the pcap library

Task 1.1: Writing Packet Sniffing Program

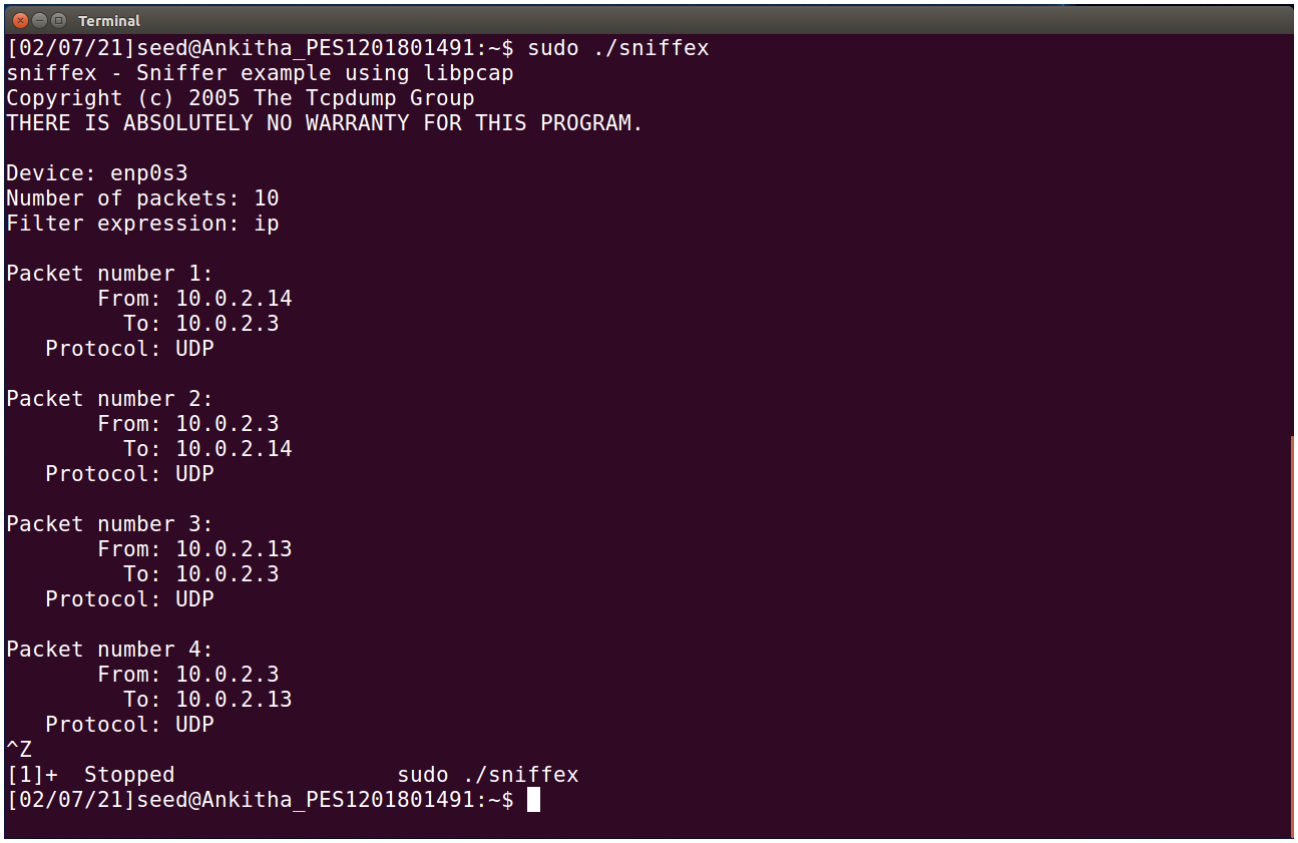
Attacker machine: 10.0.2.14

Victim machine: 10.0.2.13

sniffex.c program was downloaded from the tutorial mentioned in the manualcompiles and run using:

```
gcc -Wall -o sniffex sniffex.c -lpcap
sudo ./sniffex
```

screen dump evidence to show that the program runs successfully and produces expected results:



The filter used was:

```
        char filter_exp[] = "ip";/* filter expression */
        struct bpf_program fp;
        bpf_u_int32 mask;
        bpf_u_int32 net;
        int num_packets = 10;
```

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.

The following calls are within **sniffex.c:**

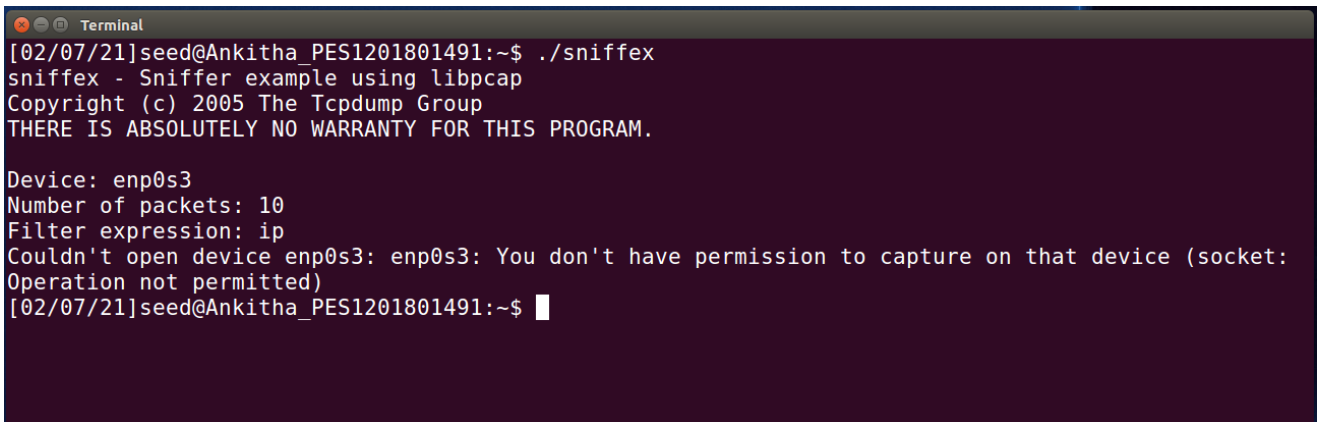
1. **pcap_lookupdev**: Finds a capture device to sniff on
2. **pcap_lookupnet**: Returns the network number and mask for the capture device
3. **pcap_open_live**: Starts sniffing on the capture device
4. **pcap_datalink**: Returns the kind of device we're capturing on

5. **pcap_compile**: Compiles the filter expression stored in a regular stringin order to set the filter
6. **pcap_setfilter**: Sets the compiled filter
7. At this point, we can either sniff one packet at a time (pcap_next) or continuously sniff (pcap_loop). Since sniffex.c uses we'll continue with pcap_loop
8. **pcap_loop**: Sets callback function for new (filtered) packets
9. **pcap_freecode:** Frees up allocated memory generated by pcap_compile
10. **pcap_close:** Closes the sniffing session

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 in order for sniffex to run because sniffex will need to access a network device which a non-root user cannot do. Because of the hierarchy of network, the OS hides the details below the application layer. For users, they do not need to know what the packet's content is, but for some attackers they could get useful things for their evil. Another reason is that the program calls the system API. So the root privilege is requested.

The error obtained without root privilege:

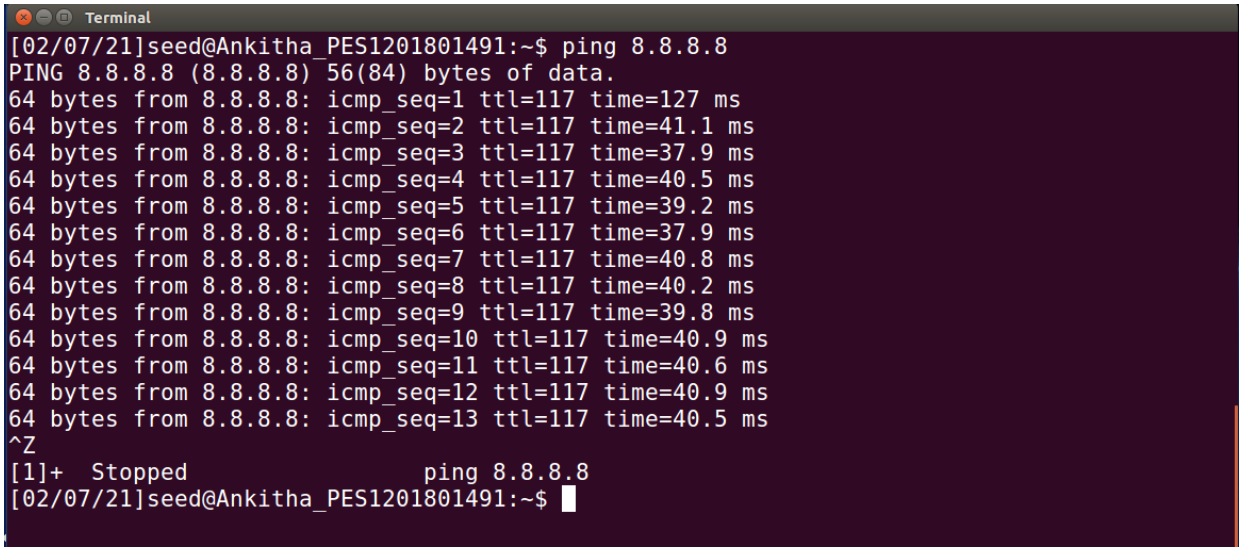


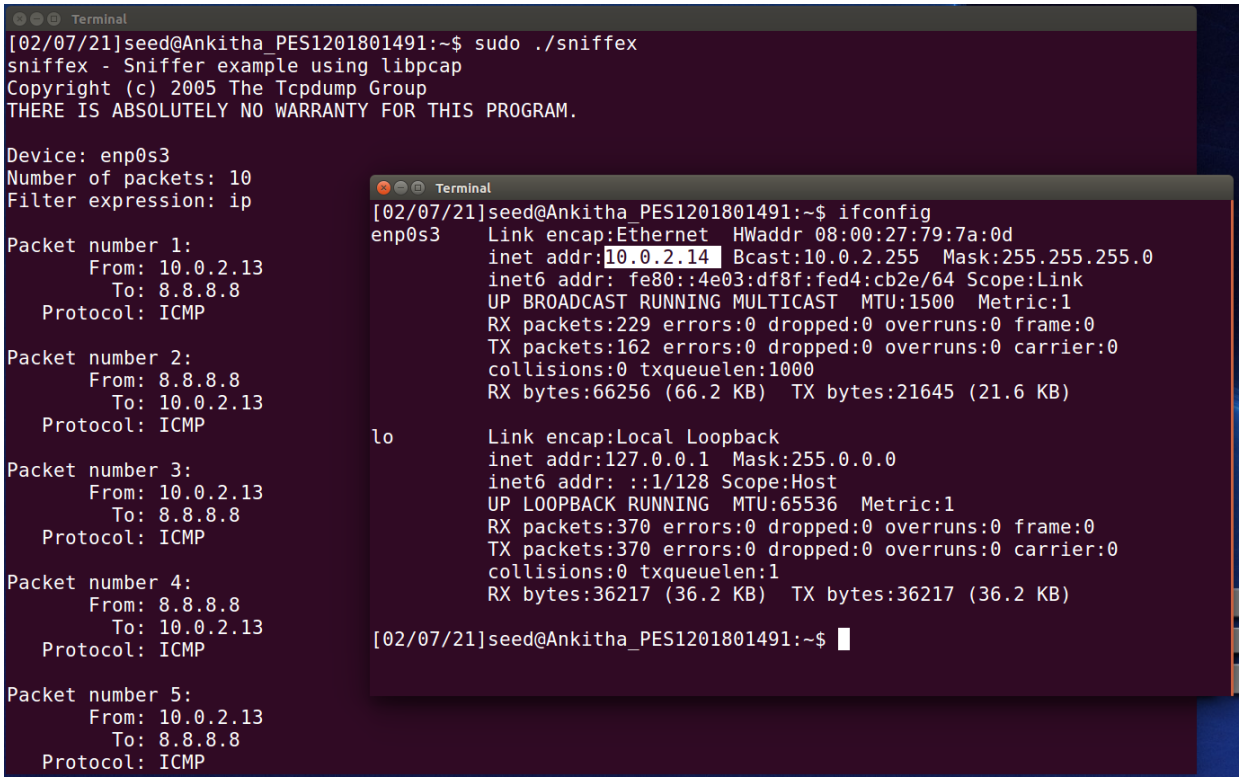
The code snippet that is responsible for the program to fail without the root privilege.

```
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);
        }
}
```

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 **Promiscuous mode** is turned **on**:





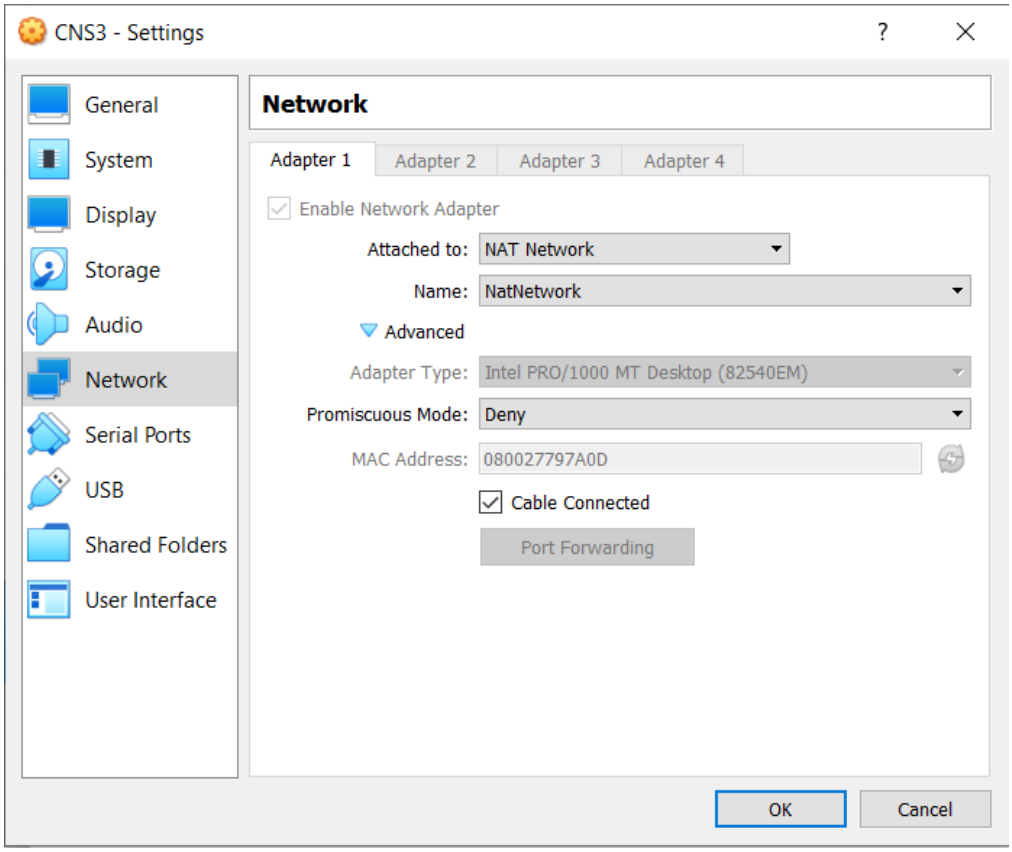
Promiscuous mode allows for a network sniffer to pass *all* traffic from a network controller and not just the traffic that the network controller was intended to receive.

Whether or not the capture device is in promiscuous mode determines on the third parameter (a 'boolean' int) in pcap_open_live . The code below highlights it:

```
/* 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);
}
```

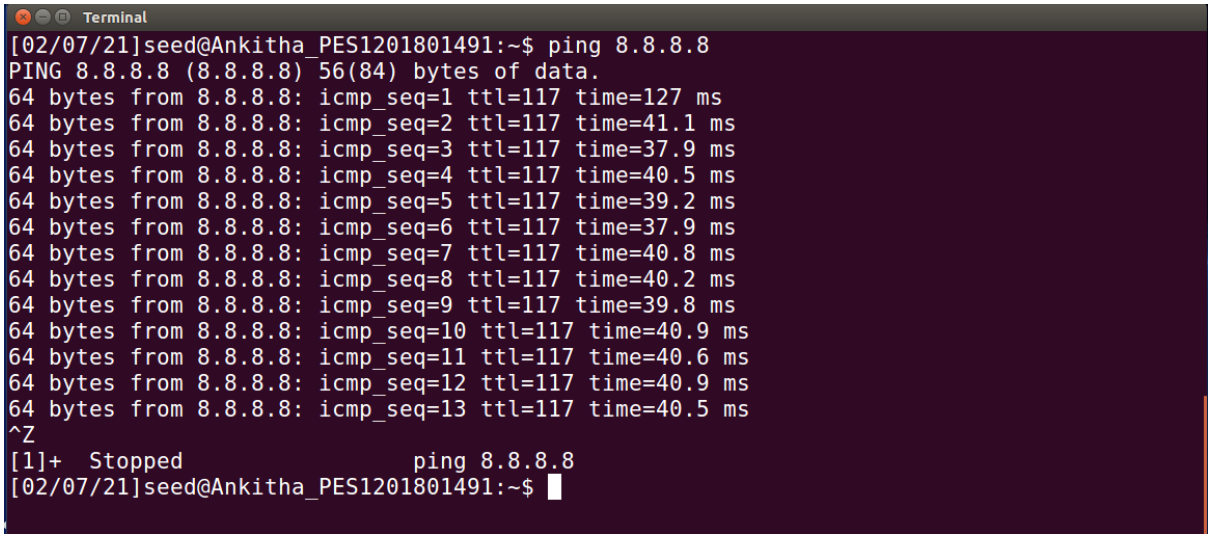
When **Promiscuous mode** is turned **OFF** :

The promiscuous mode is turned off in the networks setting:

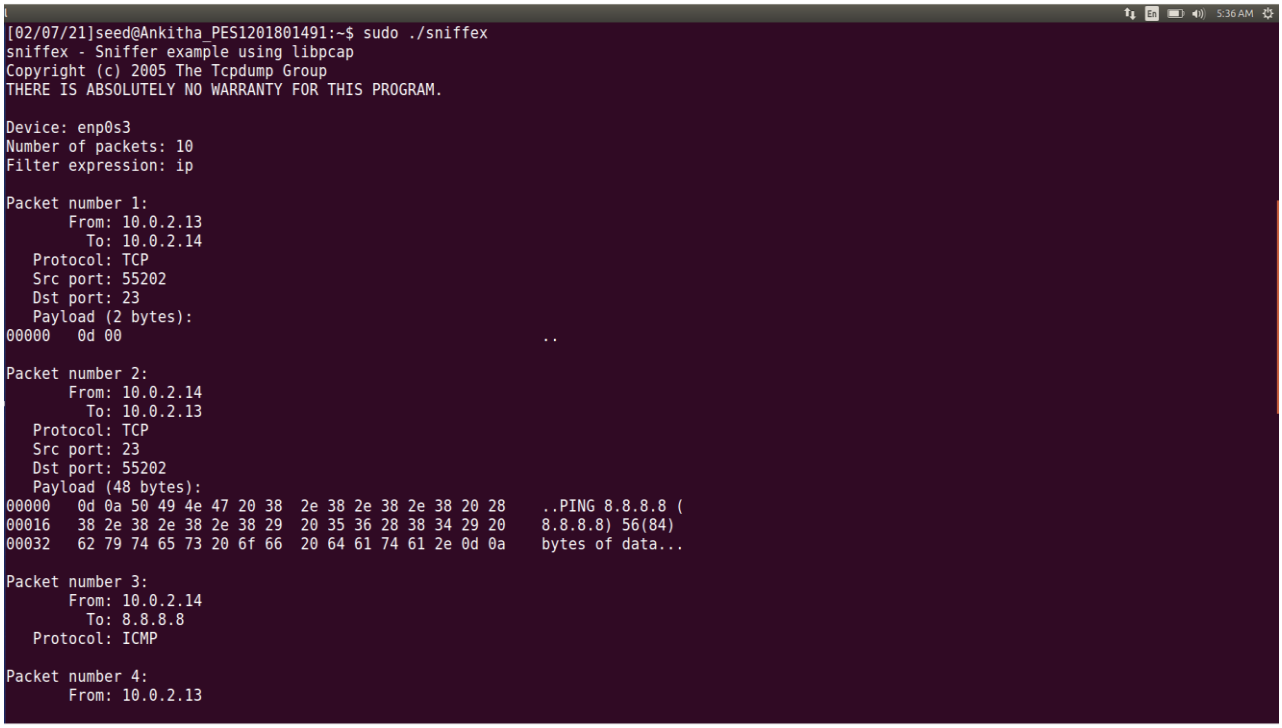



```
/* open capture device */
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
if (handle == NULL) {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        exit(EXIT_FAILURE);
}
```

Victim machine pings to 8.8.8.8



Attacker machine cannot capture all the packets when the promiscuous mode is turned off:



When promiscuous mode is switched off the sniffer program is not able to sniff through the packets which destination address being that of the machine running the sniffer program. When we send packets to a random address 8.8.8.8 from victim machine (10.0.2.10), the attacker using the sniffer program cannot capture the

packets as the promiscuous mode is off since the NIC (hardware device) discards the packets that are not being sent to the sniffing machine. But if we send packets to the attacker machine (10.0.2.9) the sniffer program captures this packet since the destination is the 10.0.2.9.

Task 1.2B: Writing Filters

The objective of this task to capture certain traffic on the network based on filters. We can provide filters to the sniffer program. In pcap sniffer, when we have a sniffing session opened using “pcap_open_live”, we can create a rule set to filter the traffic which needs to be compiled. The rule set which is in the form of a string is compiled to a form which can be read by pcap. The rule set provided here sniffs the ICMP requests and responses between two given hosts. After compiling, the filter needs to be applied using pcap_setfilter () which preps the sniffer to sniff all the traffic based on the filter. Now, actual packets can be captured using pcap_loop().

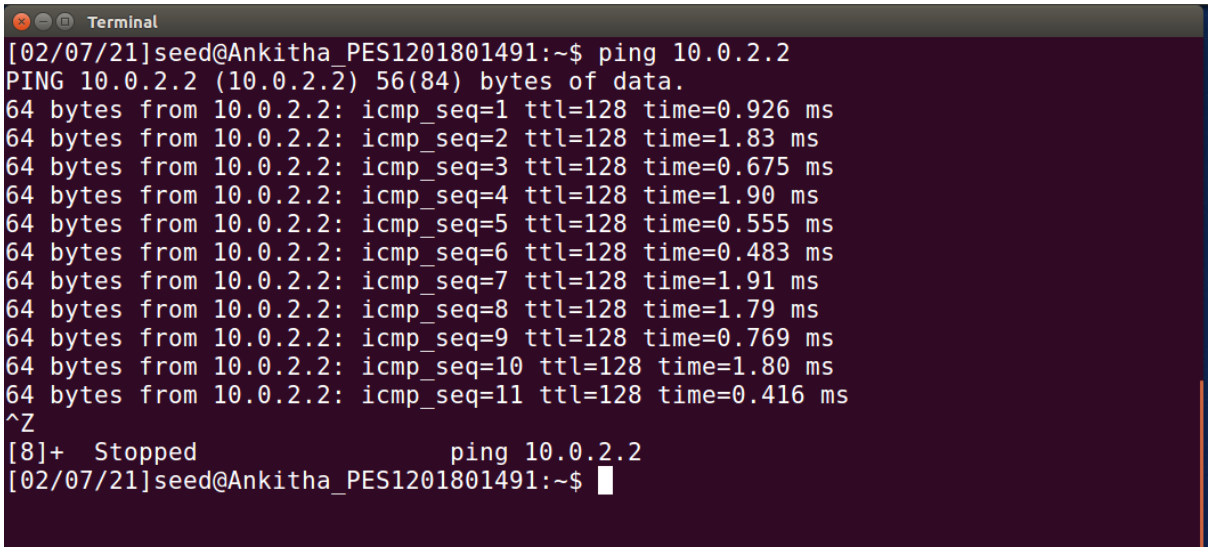
Capture the ICMP packets between two specific hosts

we can filter only the ICMP packets by modifying the filter_exp[] string.

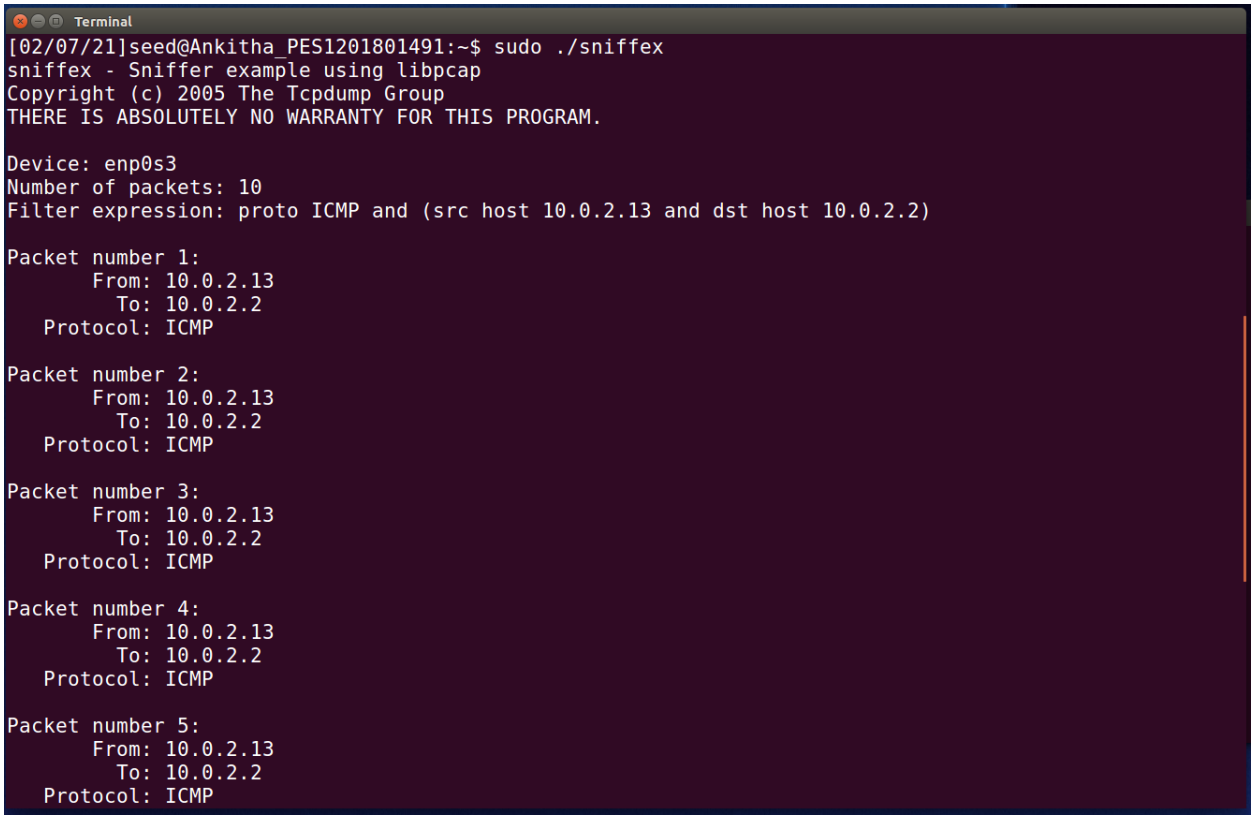
```
char filter_exp[] = "proto ICMP and (src host 10.0.2.13 and dst host 10.0.2.2)";/* filter expression */
struct bpf_program fp;
bpf_u_int32 mask;
bpf_u_int32 net;
int num_packets = 10;
```

The above filter code captures all the icmp packets sent from src host 10.0.2.13 to the machine 10.0.2.2

Victim pings to machine 10.0.2.2:



Attacker machine sniffs the packet sent from 10.0.2.13 to machine 10.0.2.2



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

Changing the `filter_exp[]` variable again we can sniff only TCP packets with a destination port from 10 to 100.

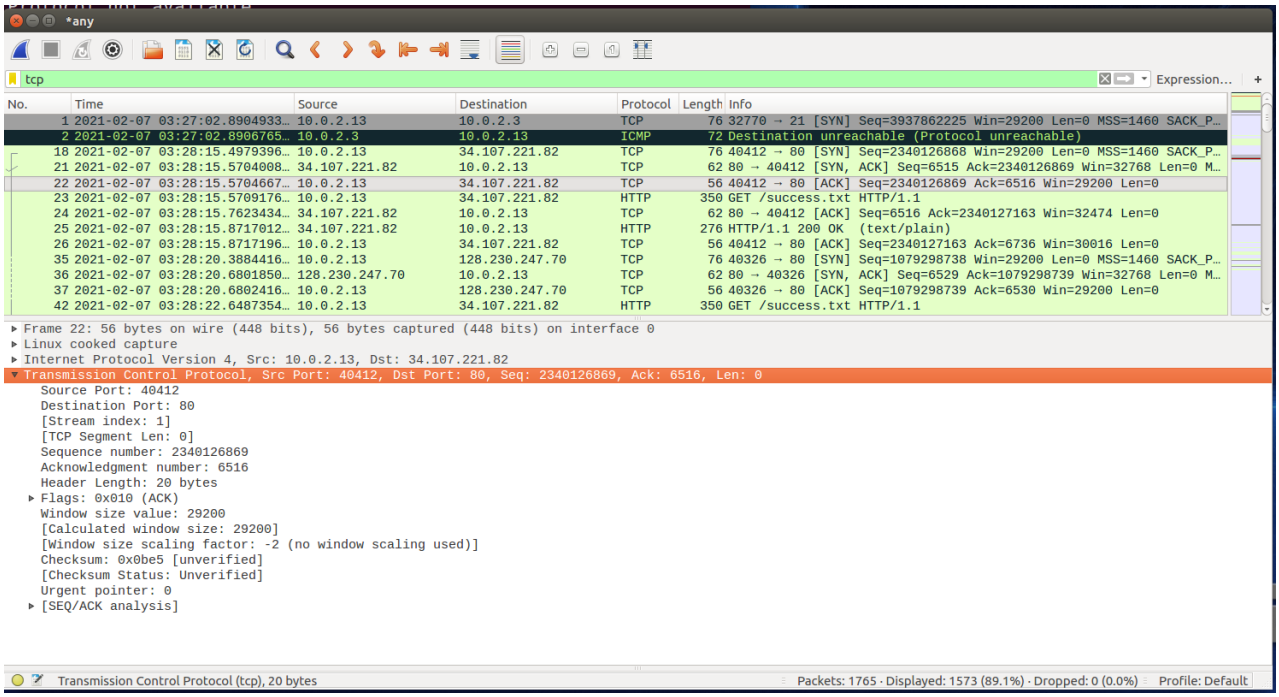
```
char filter_exp[] = "tcp dst portrange 10-100";/* filter expression */
struct bpf_program fp;
bpf_u_int32 mask;
bpf_u_int32 net;
int num_packets = 10;
```

We know HTTP transfer happens over port 80.

Victim Machine:

Navigating to a webpage within Firefox (over HTTP – port 80) gives the following:

TCP packet captured in wireshark on victim machine when opening a webpage in Firefox (**HTTP request over port 80**)



Attacker machine:

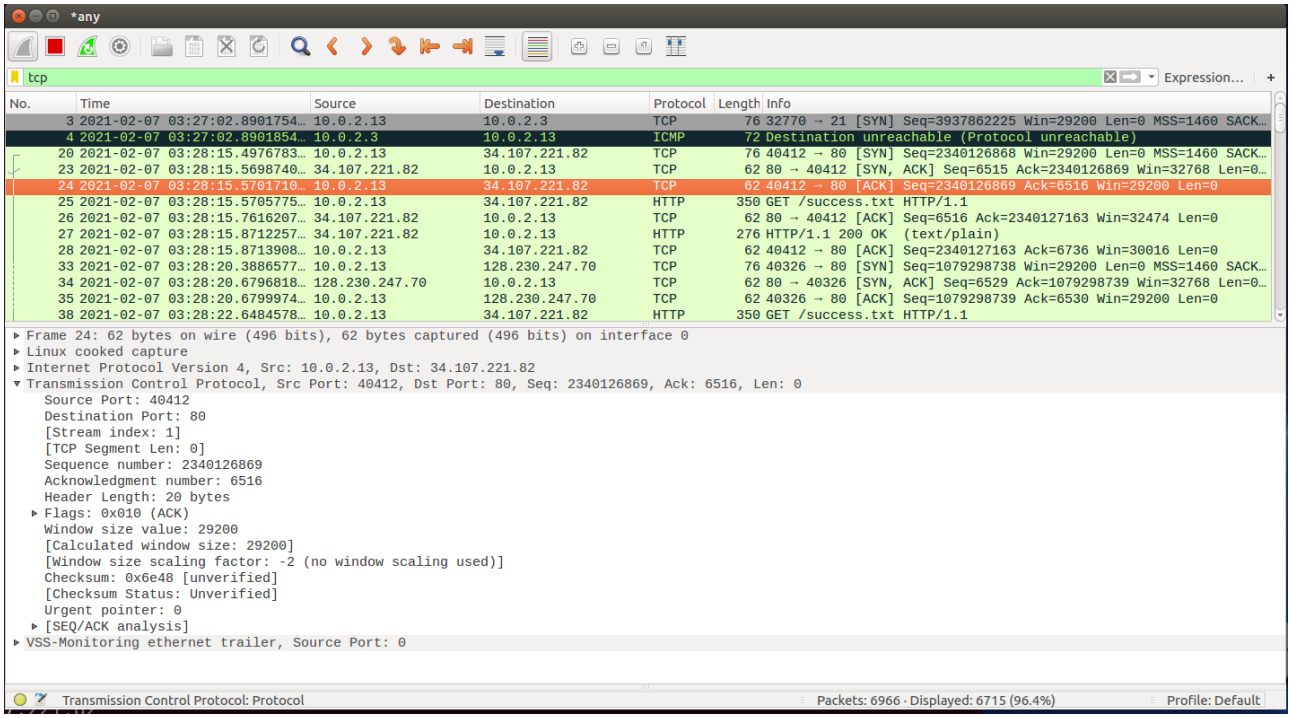
The attacker machine sniffs all the packets sent by the victim machine to port 80

```
Terminal
[02/07/21]seed@Ankitha_PES1201801491:~$ sudo ./sniffex
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 dst portrange 10-100

Packet number 1:
       From: 10.0.2.13
         To: 10.0.2.3
   Protocol: TCP
   Src port: 32770
   Dst port: 21

Packet number 2:
       From: 10.0.2.13
         To: 34.107.221.82
   Protocol: TCP
   Src port: 40412
   Dst port: 80

Packet number 3:
       From: 10.0.2.13
         To: 34.107.221.82
   Protocol: TCP
   Src port: 40412
   Dst port: 80

Packet number 4:
       From: 10.0.2.13
         To: 34.107.221.82
```

The TCP packet containing payload:

```
Terminal
         To: 34.107.221.82
   Protocol: TCP
   Src port: 40412
   Dst port: 80
   Payload (294 bytes):
00000   47 45 54 20 2f 73 75 63  63 65 73 73 2e 74 78 74    GET /success.txt
00016   20 48 54 54 50 2f 31 2e  31 0d 0a 48 6f 73 74 3a     HTTP/1.1..Host:
00032   20 64 65 74 65 63 74 70  6f 72 74 61 6c 2e 66 69     detectportal.fi
00048   72 65 66 6f 78 2e 63 6f  6d 0d 0a 55 73 65 72 2d    refox.com..User-
00064   41 67 65 6e 74 3a 20 4d  6f 7a 69 6c 6c 61 2f 35    Agent: Mozilla/5
00080   2e 30 20 28 58 31 31 3b  20 55 62 75 6e 74 75 3b    .0 (X11; Ubuntu;
00096   20 4c 69 6e 75 78 20 69  36 38 36 3b 20 72 76 3a     Linux i686; rv:
00112   36 30 2e 30 29 20 47 65  63 6b 6f 2f 32 30 31 30    60.0) Gecko/2010
00128   30 31 30 31 20 46 69 72  65 66 6f 78 2f 36 30 2e    0101 Firefox/60.
00144   30 0d 0a 41 63 63 65 70  74 3a 20 2a 2f 2a 0d 0a    0..Accept: */*..
00160   41 63 63 65 70 74 2d 4c  61 6e 67 75 61 67 65 3a    Accept-Language:
00176   20 65 6e 2d 55 53 2c 65  6e 3b 71 3d 30 2e 35 0d     en-US,en;q=0.5.
00192   0a 41 63 63 65 70 74 2d  45 6e 63 6f 64 69 6e 67    .Accept-Encoding
00208   3a 20 67 7a 69 70 2c 20  64 65 66 6c 61 74 65 0d    : gzip, deflate.
00224   0a 43 61 63 68 65 2d 43  6f 6e 74 72 6f 6c 3a 20    .Cache-Control:
00240   6e 6f 2d 63 61 63 68 65  0d 0a 50 72 61 67 6d 61    no-cache..Pragma
00256   3a 20 6e 6f 2d 63 61 63  68 65 0d 0a 43 6f 6e 6e    : no-cache..Conn
00272   65 63 74 69 6f 6e 3a 20  6b 65 65 70 2d 61 6c 69    ection: keep-ali
00288   76 65 0d 0a 0d 0a                                  ve....

Packet number 9:
       From: 10.0.2.13
         To: 128.230.247.70
   Protocol: TCP
   Src port: 40326
   Dst port: 80
   Payload (417 bytes):
00000   47 45 54 20 2f 7e 77 65  64 75 2f 73 65 65 64 2f    GET /~wedu/seed/
```

Wireshark capture on the Attacker machine:



Task 1.3C: Sniffing Passwords

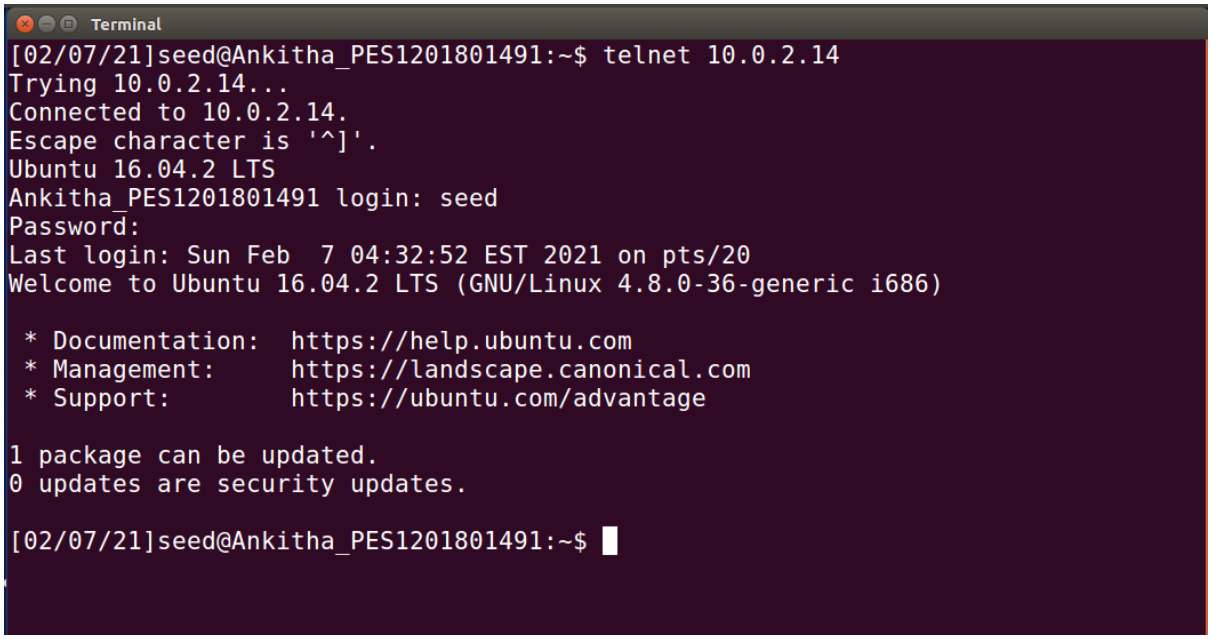
The objective of this task is to sniff passwords using the sniffer program. We will connect to a telnet server (running in our VM) and get password of the user. Our telnet server is running in machine with IP address 10.0.2.

Since we're sniffing telnet passwords, we can just look for tcp packets on port 23:

```
char filter_exp[] = "tcp port 23";/* filter expression */
struct bpf_program fp;
bpf_u_int32 mask;
bpf_u_int32 net;
int num_packets = 10;
```

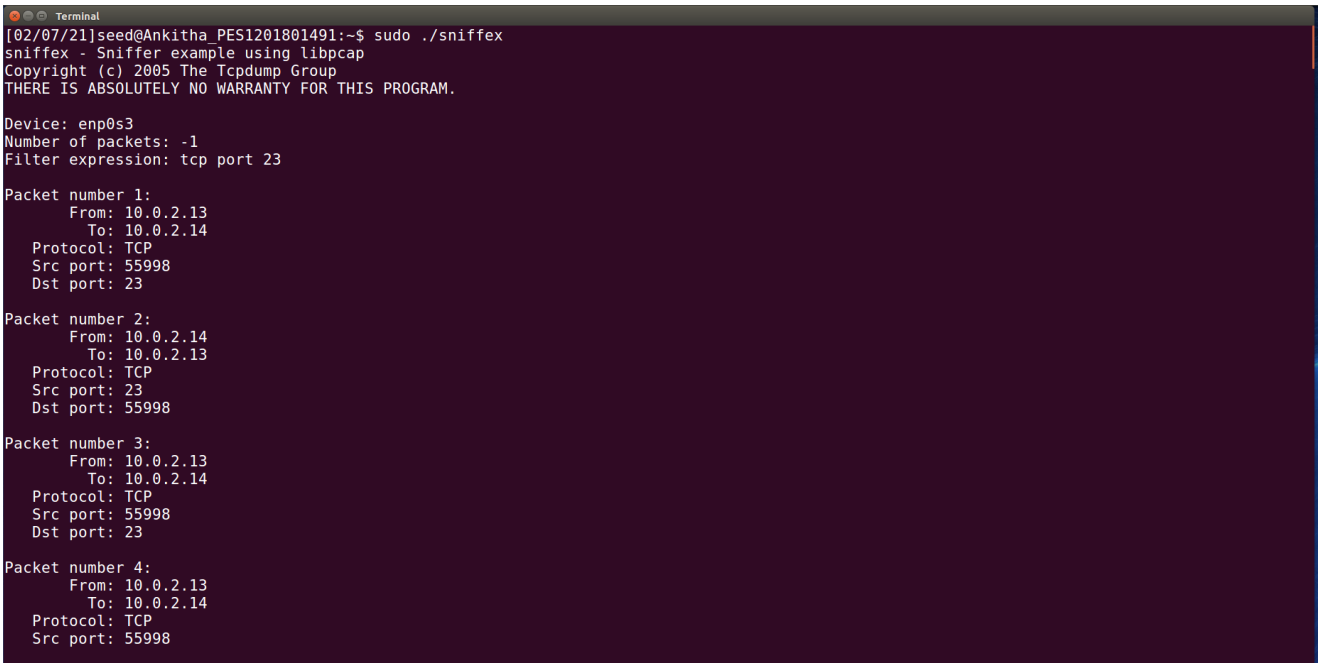
Victim machine:

Using the Victim machine, I telnet into the attacker machine:



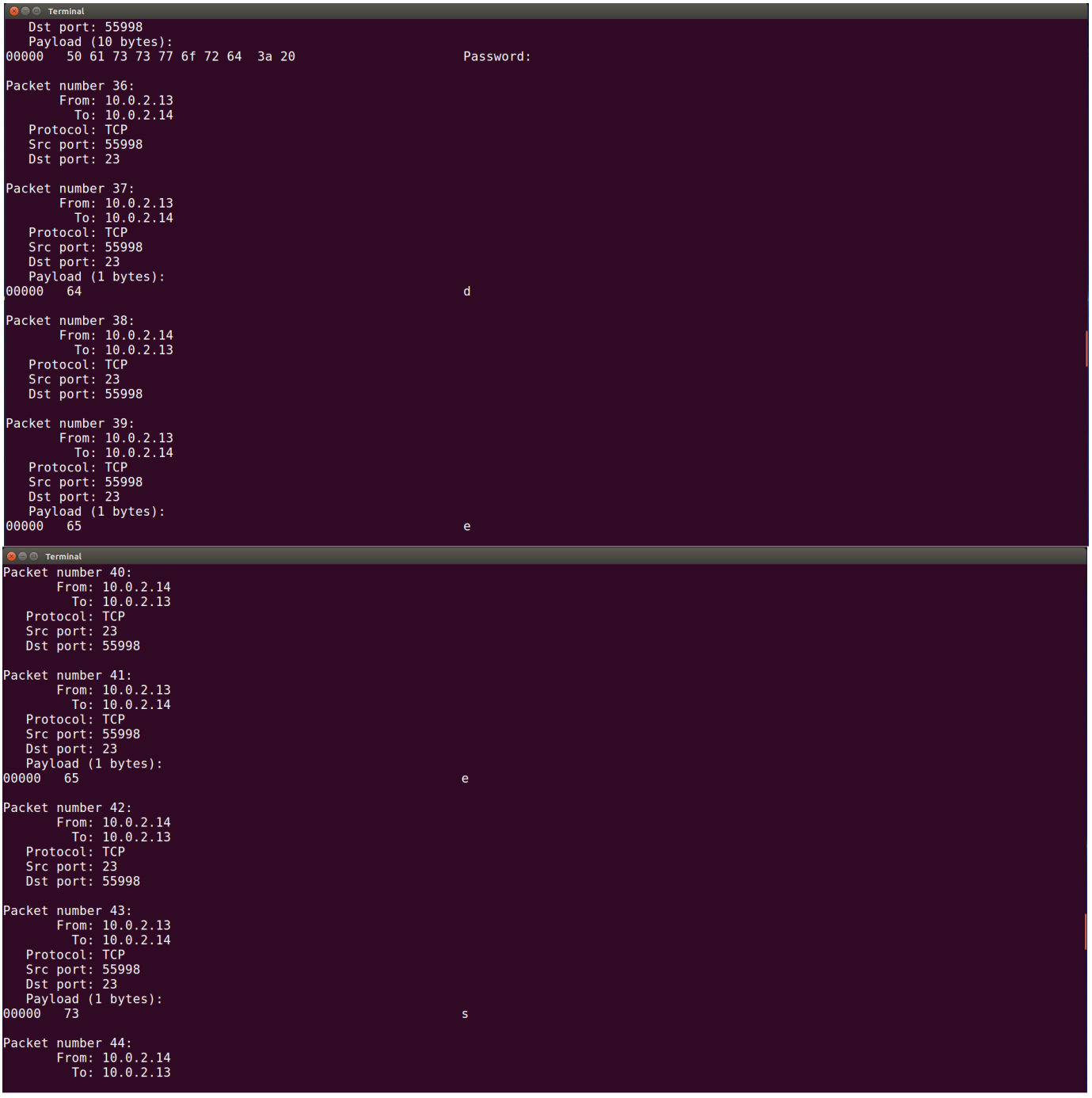
Attacker machine:

Obviously sniffex would need some modification in order for us to clearly sniff the passwords, but, for short passwords like the password to the SEED VM, we don't need to output just the payload!



The password that has been extracted by the attacker:

Username: seed Password: dees



When we run the telnet the attacker machine, i.e. 10.0.2.14 at port 23, it requests for username and password. As these details are being given, the attacker has sniffed the details including Password using the sniffing program.

Task 2.2: Spoofing

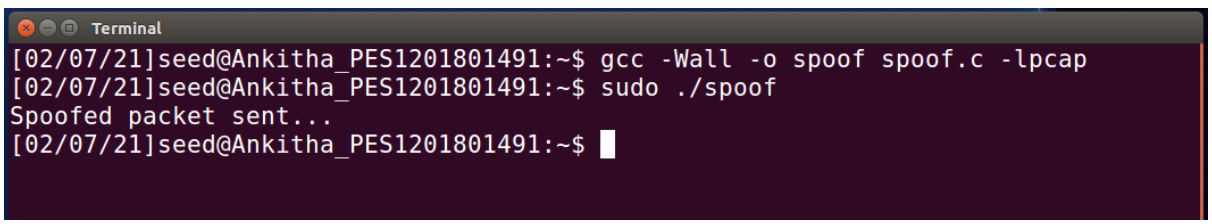
The objectives of this task is to create raw sockets and send spoof packets to the user/victim machine raw sockets give programmers the absolute control over the packet construction

The following code has been written to spoof the packets to an unknown destination

Open Save

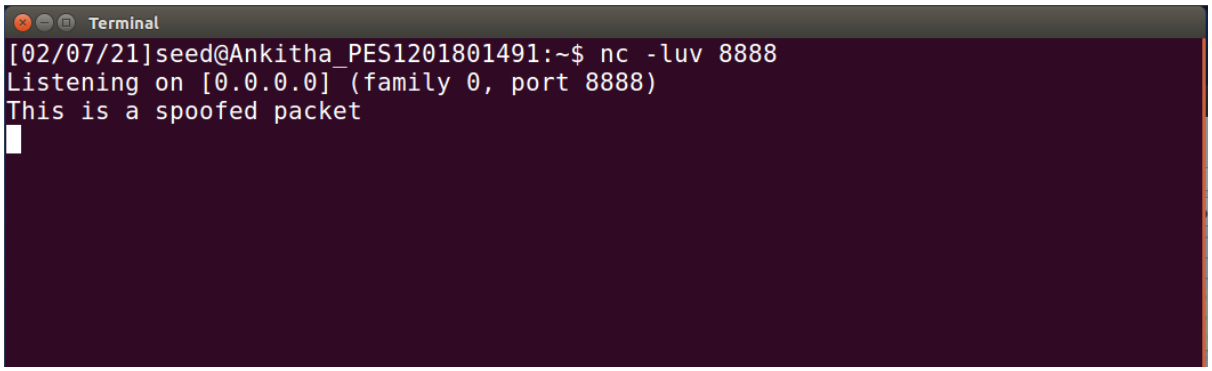
```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <stdlib.h>

struct udpheader
{
        u_int16_t udp_sport;
        u_int16_t udp_dport;
        u_int16_t udp_ulen;
        u_int16_t udp_sum;
};
struct ipheader {
        unsigned char ip_ihl:4, ip_ver:4;
        unsigned char ip_tos;
        unsigned short int ip_len;
        unsigned short int ip_ident;
        unsigned short int ip_flag:3, iph_offset:13;
        unsigned char ip_ttl;
        unsigned char ip_protocol;
        unsigned short int ip_chksum;
        struct in_addr ip_sourceip;
        struct in_addr ip_destip;
};

void send_raw_ip_packet (struct ipheader *ip)
{
        int sd;
        int enable = 1;
        struct sockaddr_in sin;
        sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
        if(sd < 0) {
                perror("CANNOT CREATE RAW SOCKET"); exit(-1);
        }
        setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
        sin.sin_family = AF_INET;
        sin.sin_addr = ip->ip_destip;
        if(sendto(sd, ip, ntohs(ip->ip_len), 0, (struct sockaddr *)&sin,sizeof(sin)) < 0) {
                perror("CANNOT SEND OUT PACKET"); exit(-1);
        }
        printf("Spoofed packet sent...\n");

}

int main() {
        char buffer[1024];
        memset(buffer, 0, 1024);
        struct ipheader *ip = (struct ipheader *) buffer;
        struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
        // Filling in UDP Data field
        char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
        const char *msg="This is a spoofed packet\n";
        int data_len = strlen(msg);
        strncpy(data, msg, data_len);
        // Fill in the UDP header
        udp->udp_sport = htons(9999);
        udp->udp_dport = htons(8888);
        udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
        udp->udp_sum = 0;
        // Fill in the IP header
        ip->ip_ver = 4;
        ip->ip_ihl = 5;
        ip->ip_ttl = 20;
        ip->ip_sourceip.s_addr = inet_addr("1.2.3.4");
        ip->ip_destip.s_addr = inet_addr("10.0.2.13");
        ip->ip_protocol = IPPROTO_UDP;
        ip->ip_len=htons(sizeof(struct ipheader)+sizeof(struct udpheader) + data_len);
        // Send the spoofed packet
        send_raw_ip_packet(ip);
        return 0;
}
```

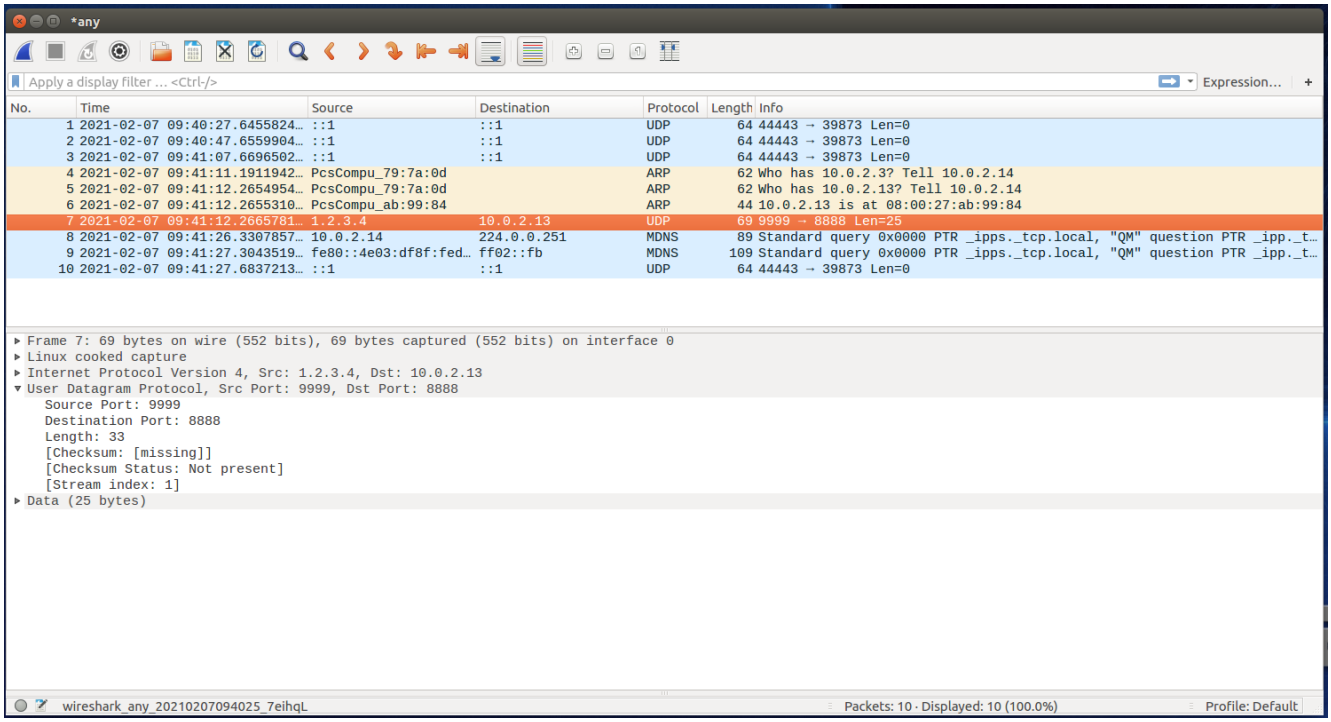
Attacker Machine: The attacker machine sends a spoofed packet to 10.0.2.13 from a non existing ip 1.2.3.4



Victim Machine:

nc listening on port 8888.





The spoof program created raw socket and sent spoof packet to the victim machine using the raw sockets from a non-existing ip. The spoofed packet was accepted by the victim machine and has been captured in the wireshark with following details :

Source ip: 1.2.3.4

Destination ip: 10.0.2.13

Source port: 9999

Destination port: 888