Sniffing and Spoofing

CSE644

Internet Security

Chirag Sachdev

680231131

Homework 1

Lab task 1

Task 1.1A

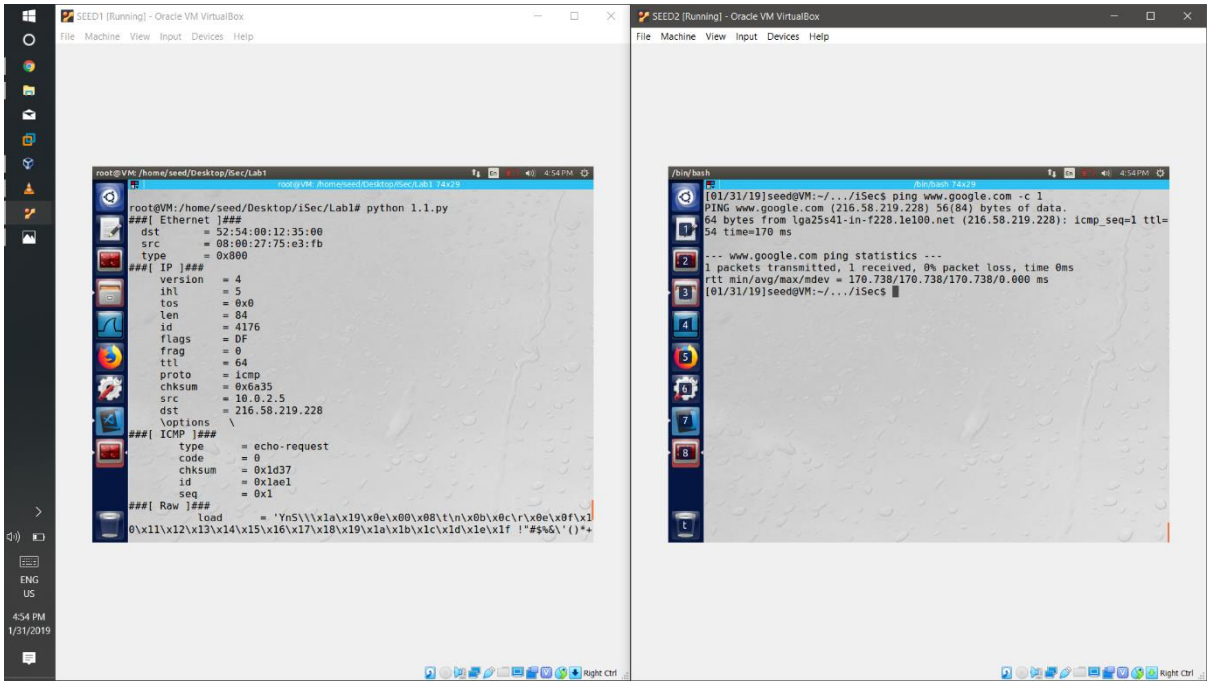
Using tools for sniffing and spoofing.

Using the following code and using the scapy library, we can create the script to perform sniffing operations on the network.

Code:

```
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

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

Output:



Observation:

By using the scapy library, we can sniff on the network using the sniff() function.

For this example, we pass the arguments of the filter for the traffic and the function call prn which has the argument of the packet.

The function show() prints the contents of the packet as shown in the terminal on the left of the output.

Task 1.1B

Using filters for network traffic

In the first argument of the function show() of the scapy library, we can see that we can pass filters to sniff on selected packets of the traffic. The syntax of the filter expressions followed is of the Berkeley Packet Filter(BPF) syntax.

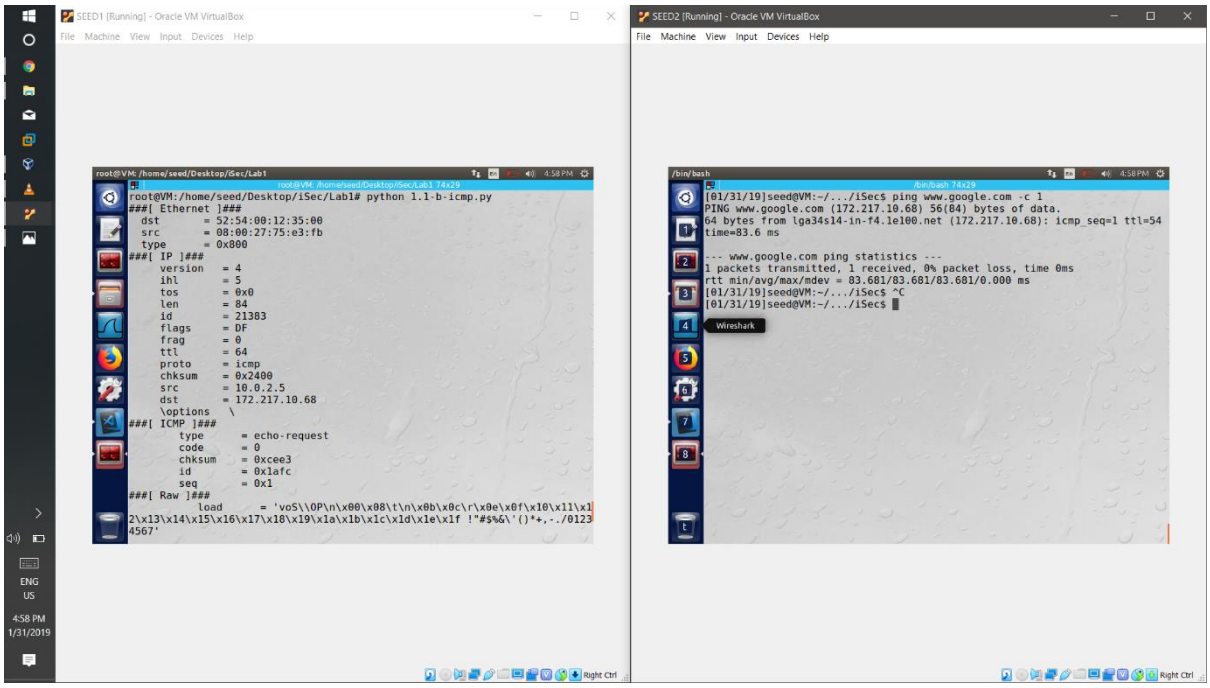
I used the following as reference for the same:

http://alumni.cs.ucr.edu/~marios/ethereal-tcpdump.pdf

Code:

```
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter="icmp && host 10.0.2.5",prn=print_pkt)
```

Output:



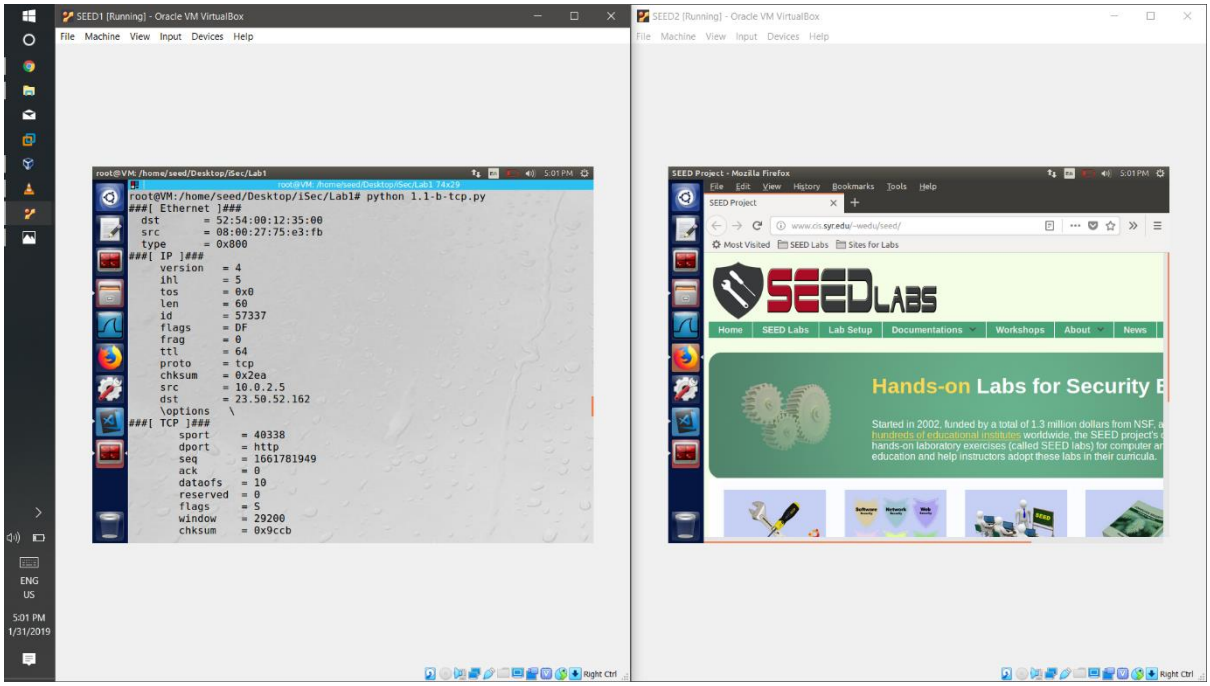
Observation:

The filter expression only filters out ICMP packets from the traffic.

Code:

```
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter="tcp && host 10.0.2.5",prn=print_pkt)
```

Output:

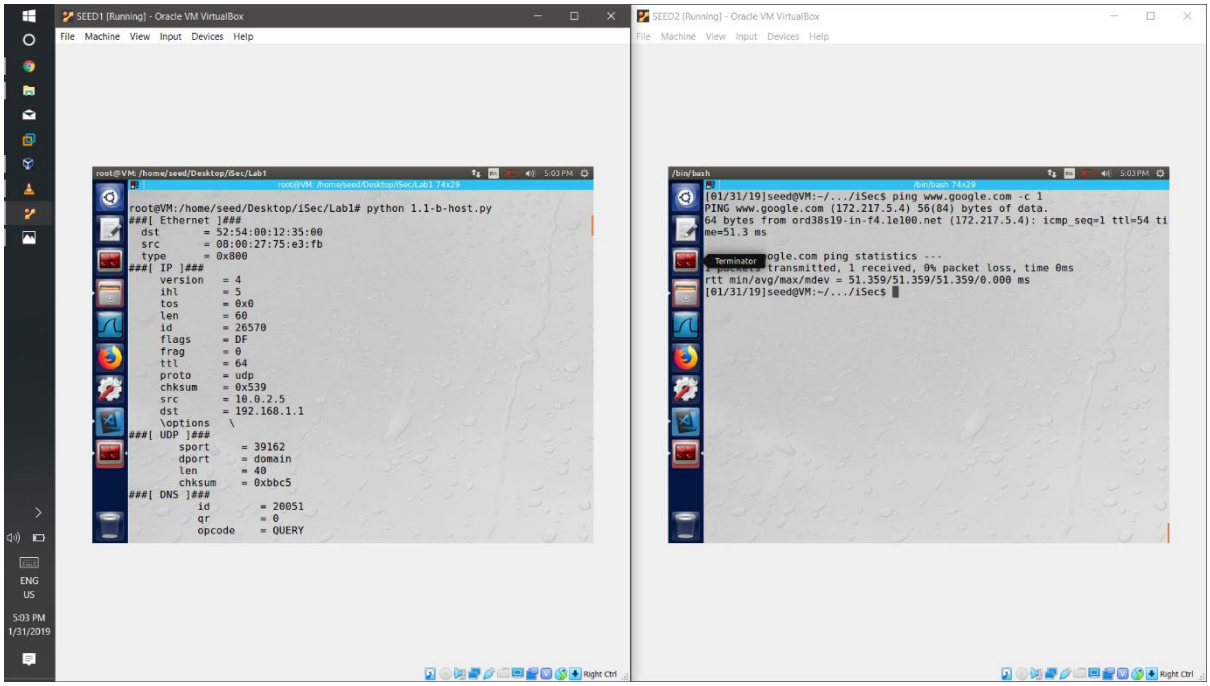


Observation: The above filter only filters out the TCP traffic from the host “10.0.2.5” from the VM on the right when it tries to load the course website.

Code:

```
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter="host 10.0.2.5",prn=print_pkt)
```

Output:



Observation:

The filter only sniffs packets from the host ’10.0.2.5” which is the VM on the right. Here the second VM is sending 1 PING to “www.google.com” and the host machine using the script above captures it.

Task 1.2

Spoofing packets

Here we use scapy to send out spoofed packets.

We create an object “a” of the IP header type and set the destination as the IP address of our target machine.

We create an object “b” of the ICMP header type.

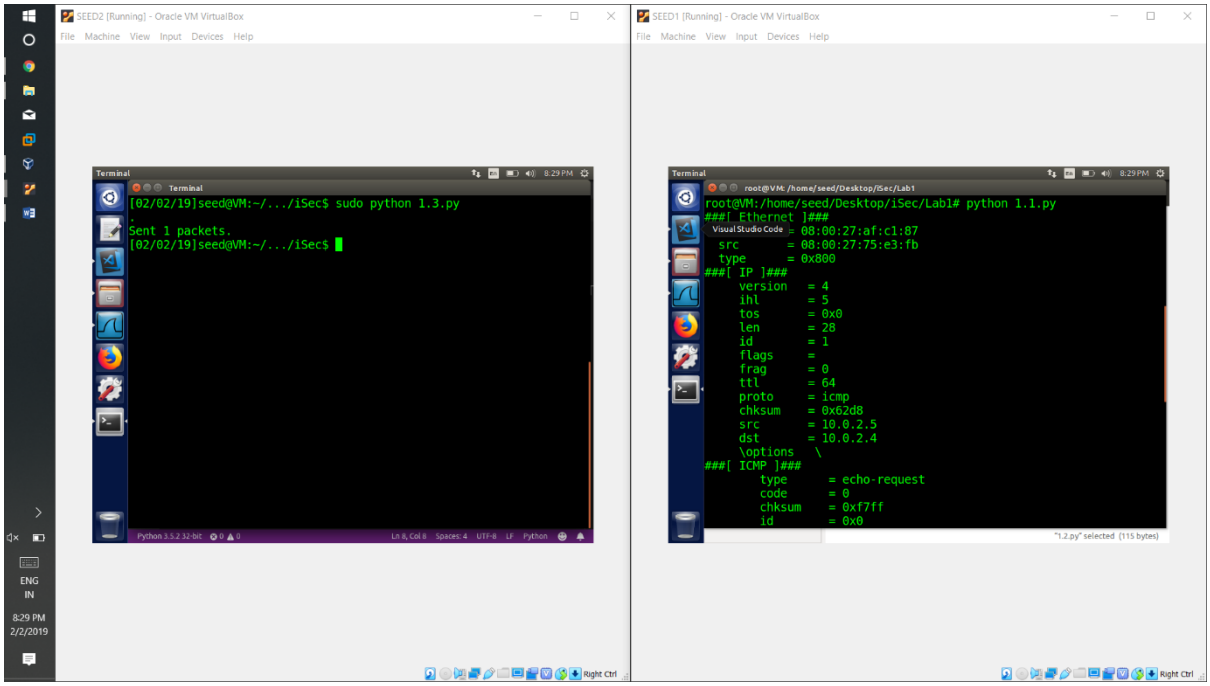
Using the stack operator “/”, we stack the layers in a packet p and it is concatenated in one buffer now.

We then use the function send() to send out the packet on the network.

Code:

```
from scapy.all import *
a=IP()
a.dst='10.0.2.4'
b=ICMP()
p=a/b
send(p)
```

Output:



Observation:

Sending out the packet from the second machine and using the script from task 1.1 to sniff on the target machine, we can see that the code above we can see that a packet is sent with the destination entered by us.

Task 1.3

Performing a traceroute operation by altering the time to live of a packet.

The time to live quantity of the packet is the number of hops the packet is allowed to make.

For e.g., the packet jumps from the host to the router, this it is the first jump it makes.

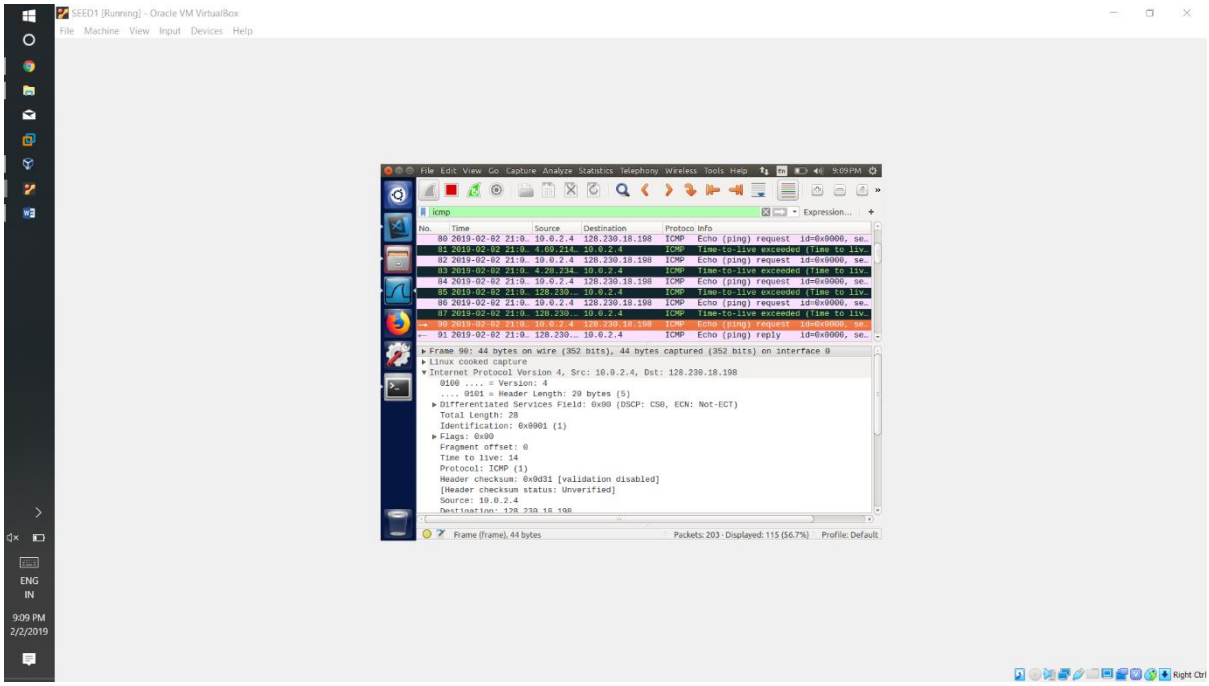
A packet has to make multiple jumps till it reaches the destination. The TTL of a PING request is 64, this means the packet can make 64 jumps before it dies.

To make a traceroute script, we keep incrementing the time to live for a packet before sending it till we receive a reply.

Code:

```
from scapy.all import *
for i in range(1,65):
    a=IP()
    a.dst="128.230.18.198"
    a.ttl=i
    b=ICMP()
    p=a/b
    send(p)
    for j in range(10000000):
#       time delay loop
```

Output:



Observation:

The packet is sent in a loop with every increment in the loop, the TTL is incremented. It was only after the value of TTL was set to 16, we received a reply. Before that the packets were dropped before reaching the destination.

Task 1.4

Sniffing and then spoofing.

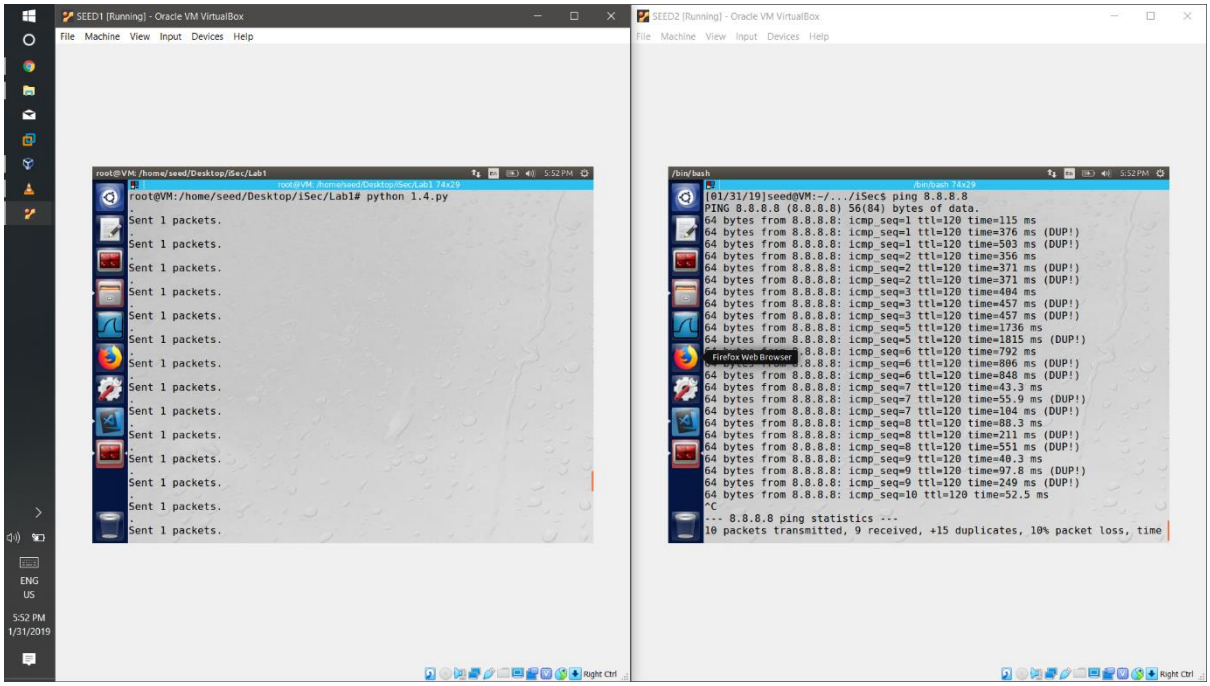
Here we first sniff on the network for ICMP requests and then spoof a reply to the host.

Here we sniff on the network till we capture an ICMP request packet and then spoof a response to the same.

Code:

```
from scapy.all import *
def spoof_pkt(pkt):
    a=pkt[IP]
    dest=str(pkt[IP].dst)
    src=str(pkt[IP].src)
    a.dst=src
    a.src=dest
    b=pkt[ICMP]
    p=a/b
    send(p)

pkt = sniff(filter="icmp[icmptype] == icmp-echo",prn=spoof_pkt)
```

Output:



Observation:

When a ping request is sent from a machine to a destination, the program captures that request and spoofs a reply to the machine as the destination.

We can see that for 1 request sent out by the host, it receives multiple replies from the destination even though the destination sends out only 1 reply.

Lab Task 2

Writing programs to perform sniffing and spoofing

Lab task 2.1

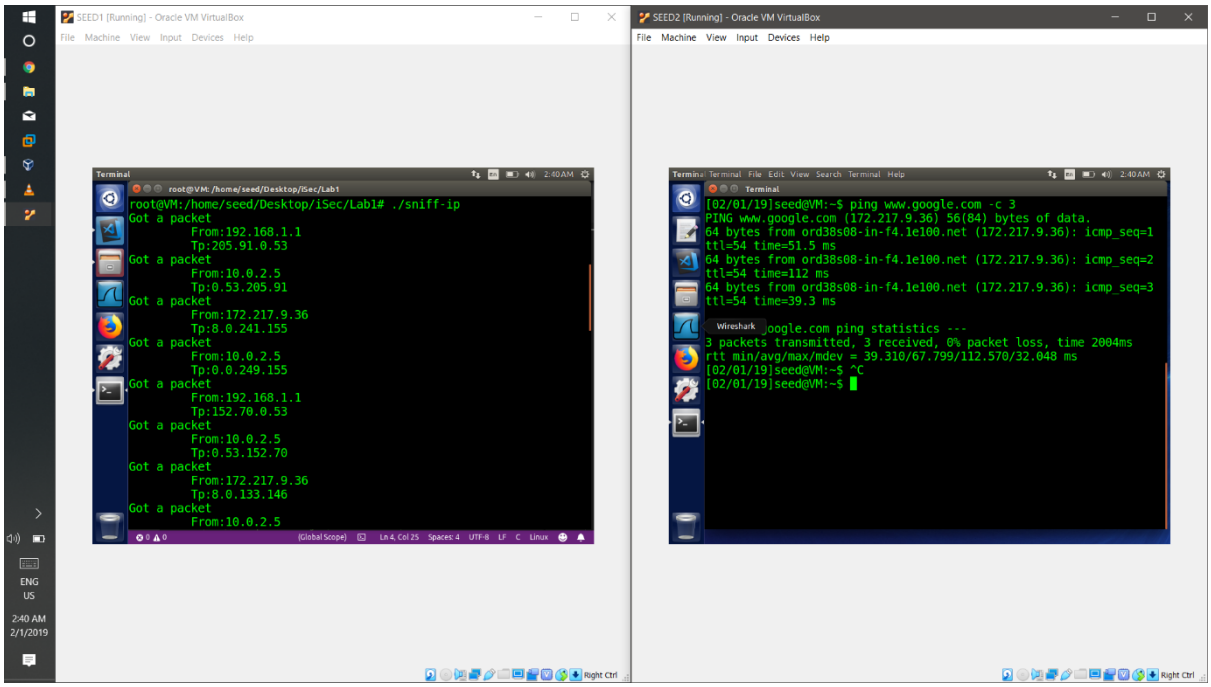
We write a sniffing program in C to perform a sniffing operation.

Code:

```
#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>
#define ETHER_ADDR_LEN 6

// ethernet header
struct ethheader
{
    // host destination IP
    unsigned char ether_dhost[ETHER_ADDR_LEN];
    // host source IP
    unsigned char ether_shost[ETHER_ADDR_LEN];
    // Protocol
    unsigned short ether_type;
};

// IP header
struct ipheader
{
    // IP header length
    unsigned char       iph_length:4;
    // IP version
    unsigned char       iph_version:4;
    // Type of service
    unsigned short int  iph_tos;
    // IP packet length(header + data)
    unsigned short int  ip_len;
    // Identification
    unsigned short int  iph_ident;
    // Fragmentation flags
    unsigned short int  iph_flag:3;
    // Flags offset
    unsigned short int  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;
};

// validating if a packet has been received
void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{
    printf("Got a packet\n");
```

```
    struct ethheader *eth = (struct ethheader *)packet;

    // 0x0800 is the IP header type
    if(ntohs(eth->ether_type) == 0x0800)
    {
        struct ipheader * ip = (struct ipheader *)(packet+sizeof(struct
ethheader));

        // printing source IP
    printf("\t From:%s\n",inet_ntoa(ip->iph_sourceip));
    //   printing destination IP
    printf("\t Tp:%s\n",inet_ntoa(ip->iph_destip));
    }
}


void main()
{
    // pcap structure handle
    pcap_t *handle;
    char errbuff[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    //filtering only icmp protocol
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

    // opening socket
    handle = pcap_open_live("enp0s3", BUFSIZ,1,1000,errbuff);

    // filter_exp -> BPF pseudocode
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle,&fp);

    // capture
    pcap_loop(handle, -1, got_packet, NULL);

    // closing socket
    pcap_close(handle);
}
```

Output:



Observation: The program prints a message Got a packet every time it sniffs a packet along with the source and destination of each packet.

Q1. For a simple sniffer, we invoke 2 libraries:

1. Stdio.h: it's the basic library for input and output of files
2. Pacap.h: pcap is the packet capture library.

The pcap library is responsible for handling the socket required for network communication. Sniffing is made very simple using the pcap library as we do not have to code the packets. It opens up a raw socket in promiscuous mode which allows the socket to interact with all traffic on the same network. We open a raw socket by invoking the pcap_open_live() function.

We can also set up filters for filtering out selected part of the traffic using the pcap_compile() and pcap_setfilter() functions.

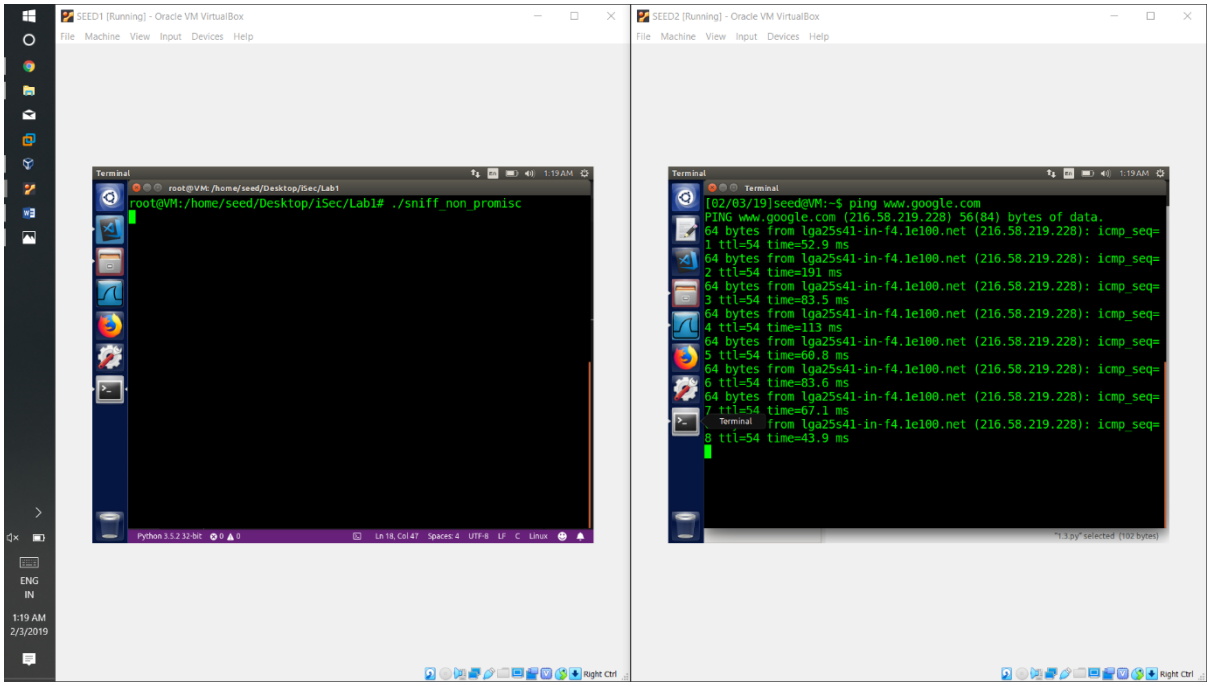
We can capture the packets on the traffic using the pcap_loop() function.

Q2. Raw sockets can read all the traffic on a network. To open a raw socket we require root privileges as the rules set by the OS by default do not allow any program to read all network traffic for security reasons. TCPdump is a program that has root privileges and hence it can read all traffic. Wireshark by default does not capture the packets, since it does not have root access, it simple reads the packets which are captured by TCPdump.

We require root privilege to read all traffic on the network not only the one which is directed to the host.

Q3. We open a raw socket using pcap_open_live() function which has a flag for promiscuous mode. When set to 1, the promiscuous mode is enabled. To turn it off, we pass the flag as 0.

Without promiscuous mode, the program simply does not receive any packet.:

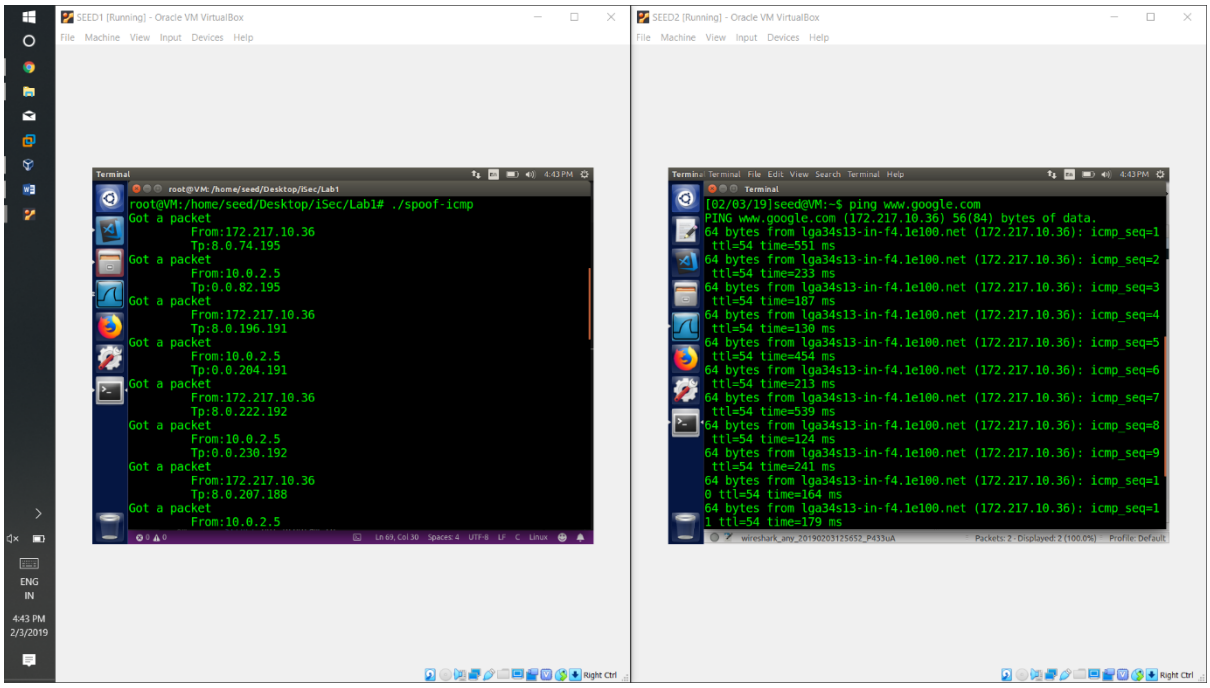


Explanation. Here the VM: SEED2 is pinging www.google.com, although on the same network, SEED1 does not see any traffic as the promiscuous mode is turned off.

Task 2.1B

- We can capture the ICMP packets by setting the filter as “ip proto icmp”

Output:

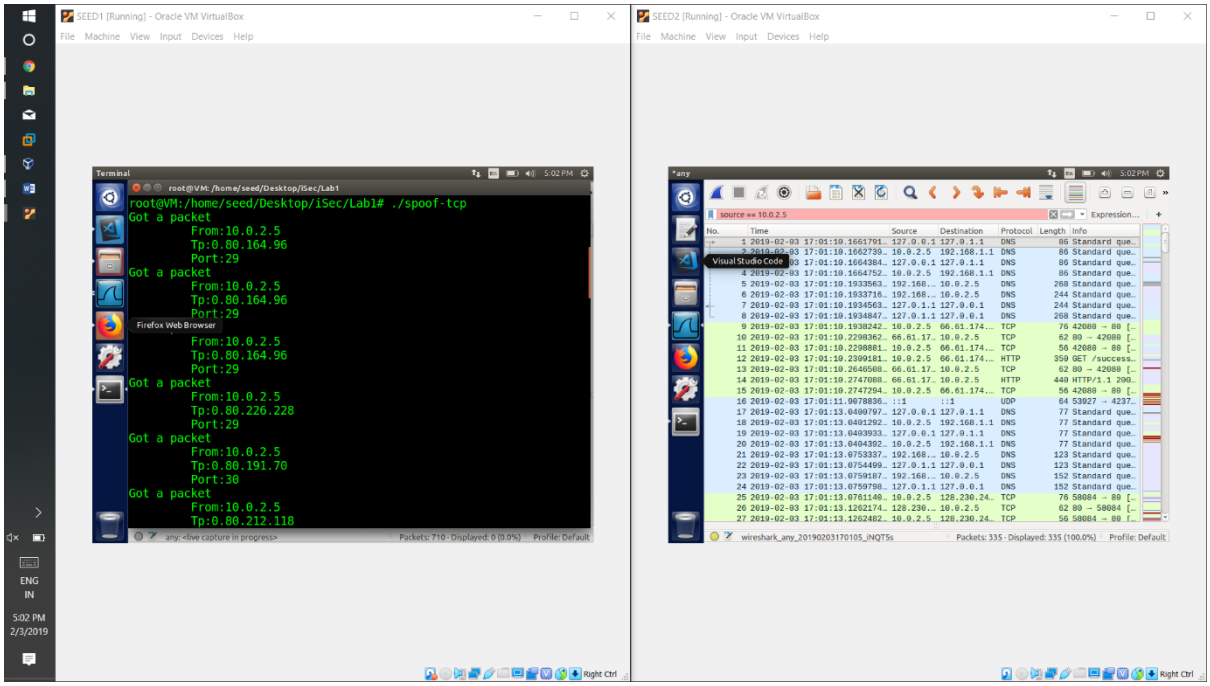


Observation:

Only ICMP packets are captured through this filter

- We can capture TCP packets within the port range by the filter “tcp src portrange 10-100”

Output:



Observation:

We can see the sniffer only captures TCP packets within the port range whereas the Host machine captures all packets which are going through its traffic.

Task 2.1C

Sniffing passwords

For this task we need to dump the payload of a TCP packet.

We set the filter as TCP and the pointer to the payload is calculated as

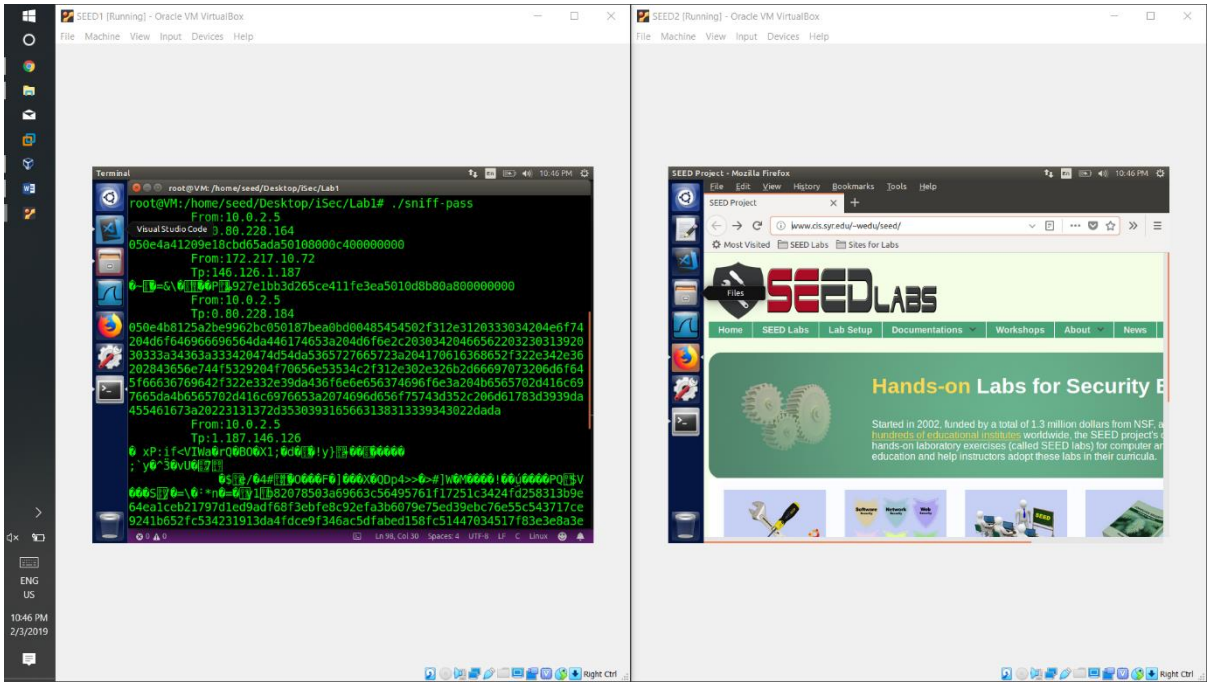
```
payload=(u_char *)(packet+sizeof(struct ipheader)+sizeof(struct
                                ethheader)+sizeof(struct tcpheader));
```

We print the payload using the following code:

```
const char *payload;
        int iph_length=ip->iph_length*4;
        int eth_length=14;
        int tcp_length=ntohs(tcp->th_offx2);
        int payload_length=header->caplen-
                           (iph_length+eth_length+tcp_length);
        if (payload_length<1)
            return;
        printf("\t From:%s\n",inet_ntoa(ip->iph_sourceip));
        //   printing destination IP
        printf("\t Tp:%s\n",inet_ntoa(ip->iph_destip));
        // printing source port
```

```
        payload=(u_char *)(packet+(iph_length+eth_length+tcp_length));
        const u_char *temp=payload;
        // printf("%s",temp);
        for(int i=0;i<payload_length;i++)
            printf("%x",temp[i]);
        printf("\n");
```

Output:



Observation:

I could figure out hot to use a telnet login, However I was able to print the payload of the packet.

Task 2.2A

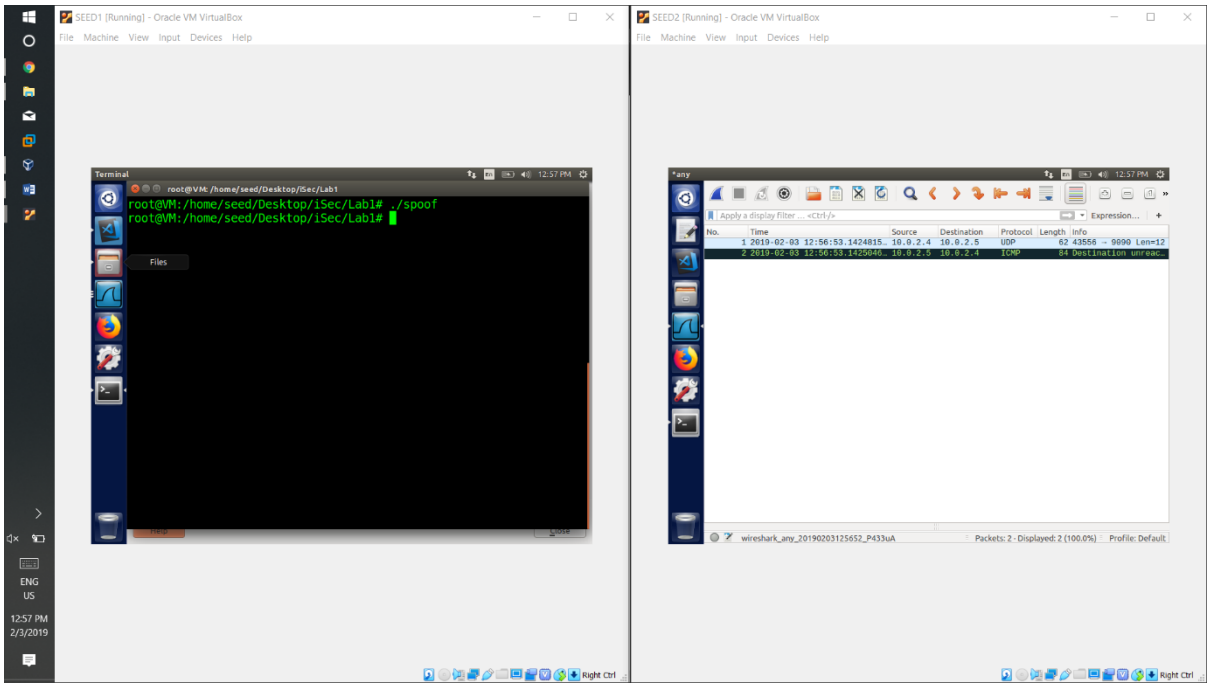
Spoofing

A simple spoofing code using raw sockets.

Code:

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

void main()
{
    struct sockaddr_in dest_info;
    char *data = "UDP message\n";

    // creating network socket
    int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    // destination info
    memset((char *) &dest_info, 0, sizeof(dest_info));
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr.s_addr = inet_addr("10.0.2.5");
    dest_info.sin_port = htons(9090);

    // sending packet
    sendto(sock, data, strlen(data), 0,(struct sockaddr *)&dest_info,
sizeof(dest_info));
    // closing socket
    close(sock);
}
```

Output:



Observation:

The packet was successfully sent using raw sockets.

Task 2.2B

Spoofing ICMP request

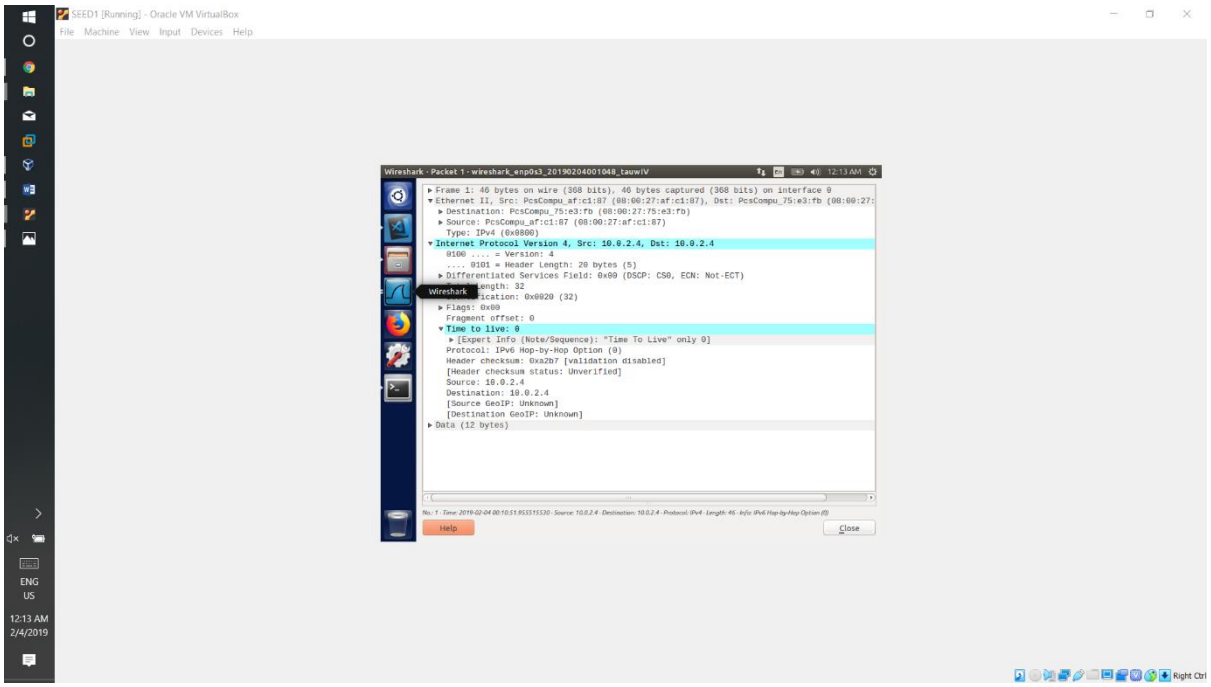
Code:

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

// icmp header
struct icmpheader
{
    // icmp message type
    unsigned char       icmp_type;
    // error code
    unsigned char       icmp_code;
    // checksum for icmp header and data
    unsigned short int  icmp_chksum;
    // used for identifying request
    unsigned short int  icmp_id;
    // sequence number
    unsigned short int  icmp_seq;
};

// ip header
struct ipheader
{
    // IP header length
    unsigned char       iph_length:4;
    // IP version
    unsigned char       iph_ver:4;
    // Type of service
    unsigned short int  iph_tos;
    // IP packet length(header + data)
    unsigned short int  iph_len;
    // Identification
    unsigned short int  iph_ident;
    // Fragmentation flags
    unsigned short int  iph_flag:3;
    // Flags offset
    unsigned short int  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;
};

// calculating checksum
```

```
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)
    {
        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);
}
// sending raw IP
void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_info;
    int enable=1;

    // creating raw network packet
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // setting up socket option
    setsockopt(sock,IPPROTO_IP, IP_HDRINCL, &enable,sizeof(enable));

    // destination info
    dest_info.sin_family=AF_INET;
    dest_info.sin_addr=ip->iph_destip;
    // dest_info.sin_port = htons(9090);

    // sending raw packet
    sendto(sock,ip,htons(ip->iph_len),0,(struct sockaddr *)&dest_info,
sizeof(dest_info));
    // printf("%u\t\t%u",ip->iph_sourceip, ip->iph_destip);

    // closing socket
    close(sock);
}

// main
int main()
{
    // buffer of packet
    char buffer[1500];
    memset(buffer,0,1500);
    struct ipheader *ip=(struct ipheader *)buffer;
    /*********************************************/
    /* filling icmp header                       /
    /*********************************************/
    struct icmpheader *icmp = (struct icmpheader *)(buffer+sizeof(struct
ipheader));
    // type 8= request, 0=reply
    icmp->icmp_type=8;
```

```
    // calculating checksum
    icmp->icmp_chksum=0;
    icmp->icmp_chksum=in_chksum((unsigned short *)icmp, sizeof(struct
icmpheader));

    /*******************************************/
    /* filling ip header                       /
    /*******************************************/
    ip->iph_ver=4;
    ip->iph_length=5;
    ip->iph_ttl=64;
    ip->iph_sourceip.s_addr=inet_addr("10.0.2.4");
    ip->iph_destip.s_addr=inet_addr("10.0.2.5");
    ip->iph_protocol=IPPROTO_ICMP;
    ip->iph_len=ntohs(sizeof(struct ipheader)+sizeof(struct icmpheader));

    /*******************************************/
    /* sending raw packet                      /
    /*******************************************/
    // for(int i=0;i<10;i++)
    send_raw_ip_packet(ip);
    // show(ip);
    return 0;
}
```

Output:



Observation:

I could not send out the spoofed ICMP packet successfully. The packet shows the TTL as 0 and bounces back to the same machine.

Q4. We cannot set the value to an arbitrary value as these fields have to be set according to the IP protocol in the IP packet.

The total length of an ethernet packet is 1500, if the size of packet out exceeds this value, then we have to fragment it.

Q5. No we do not need to calculate checksum for the IP packet as it is calculated by the OS.

Q6. Opening a raw socket allows to read anything that is received in each interface, so, basically, you can read any packet that is directed to any application - even if that application is owned by another user. That basically means that the user with this capability, can read any and all communications of all users. We require root permissions as raw sockets break the rules of networking by reading all traffic on a network.

Task 2.3

Snoofing:

Here we sniff ICMP packets on the network and spoof back packets using the spoofing program.

Code:

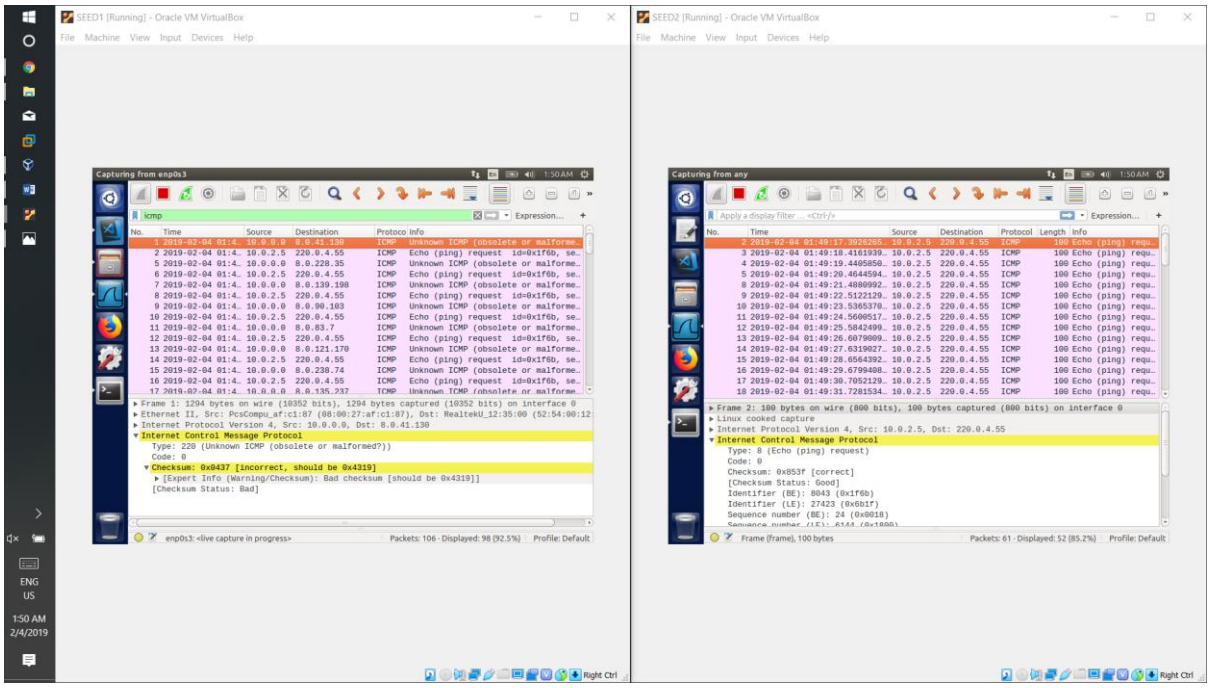
```
#include <stdio.h>
#include <pcap.h>
#include <arpa/inet.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
// #include <netinet/ip_icmp.h>

#define ETHER_ADDR_LEN 6

// ethernet header
struct ethheader
{
    // host destination IP
    unsigned char ether_dhost[ETHER_ADDR_LEN];
    // host source IP
    unsigned char ether_shost[ETHER_ADDR_LEN];
    // Protocol
    unsigned short ether_type;
};

// IP header
struct ipheader
{
    // IP header length
    unsigned char       iph_length:4;
    // IP version
    unsigned char       iph_version:4;
    // Type of service
    unsigned short int  iph_tos;
    // IP packet length(header + data)
    unsigned short int  ip_len;
    // Identification
    unsigned short int  iph_ident;
    // Fragmentation flags
    unsigned short int  iph_flag:3;
    // Flags offset
    unsigned short int  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;
};

// icmp header
struct icmpheader
{
```

```
    // icmp message type
    unsigned char       icmp_type;
    // error code
    unsigned char       icmp_code;
    // checksum for icmp header and data
    unsigned short int  icmp_chksum;
    // used for identifying request
    unsigned short int  icmp_id;
    // sequence number
    unsigned short int  icmp_seq;
};

// calculating checksum
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)
    {
        sum+=*w++;
        nleft-=2;
    }
    if(nleft==1)
    {
        *(unsigned char *)(&temp)=*(unsigned char *)w;
        sum+=temp;
    }
    sum=(sum >> 16)+(sum & 0xFFFF);
    sum+=(sum >> 16);
    sum=~sum;
    sum=(unsigned short)(sum);
    return sum;
}
// sending raw IP
void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_info;
    int enable=1;

    // creating raw network packet
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // setting up socket option
    setsockopt(sock,IPPROTO_IP, IP_HDRINCL, &enable,sizeof(enable));

    // destination info
    dest_info.sin_family=AF_INET;
    dest_info.sin_addr=ip->iph_destip;
    // dest_info.sin_port = htons(9090);

    // sending raw packet
    sendto(sock,ip,htons(ip->iph_length),0,(struct sockaddr *)&dest_info,
sizeof(dest_info));
    printf("Packet sent\n");

    // closing socket
    close(sock);
}
```

```
// validating if a packet has been received
void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{
    char buffer[BUFSIZ];
    memset(buffer,0,BUFSIZ);
    printf("Got a packet\n");
    struct ethheader *eth = (struct ethheader *)packet;
    struct ethheader *new_eth=(struct ethheader *)buffer;
    struct ipheader *ip = (struct ipheader *)(packet+sizeof(struct
ethheader));
    struct ipheader *new_ip=(struct ipheader *)(buffer+sizeof(struct
ethheader));
    struct icmpheader *icmp=(struct icmpheader *)(packet+sizeof(struct
ethheader)+sizeof(struct ipheader));
    struct icmpheader *new_icmp=(struct icmpheader *)(buffer+sizeof(struct
ethheader)+sizeof(struct ipheader));
    // filling new eth packet
    // strcpy(eth->ether_shost,new_eth->ether_dhost);
    // strcpy(eth->ether_dhost,new_eth->ether_shost);
    // new_eth->ether_type=eth->ether_type;
    // filling ip packet
    new_ip->ip_len=ip->ip_len;
    new_ip->iph_chksum=ip->iph_chksum;
    new_ip->iph_destip=ip->iph_sourceip;
    new_ip->iph_sourceip=ip->iph_destip;
    new_ip->iph_flag=ip->iph_flag;
    new_ip->iph_ident=ip->iph_ident;
    new_ip->iph_length=ip->iph_length;
    new_ip->iph_offset=ip->iph_offset;
    new_ip->iph_protocol=ip->iph_protocol;
    new_ip->iph_tos=ip->iph_tos;
    new_ip->iph_ttl=ip->iph_ttl;
    new_ip->iph_version=ip->iph_version;
    // filling icmp packet
    new_icmp->icmp_id=icmp->icmp_id;
    new_icmp->icmp_code=icmp->icmp_code;
    new_icmp->icmp_seq=icmp->icmp_seq;
    new_icmp->icmp_type=0;
    new_icmp->icmp_chksum=0;
    new_icmp->icmp_chksum=in_chksum((unsigned short *)icmp, sizeof(struct
icmpheader));
    send_raw_ip_packet(new_ip);
    // }
}


void main()
{
    // pcap structure handle
    pcap_t *handle;
    char errbuff[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    //filtering only icmp protocol
    char filter_exp[] = "icmp && icmp[icmptype] == icmp-echo";
    bpf_u_int32 net;

    // opening socket
    handle = pcap_open_live("enp0s3", BUFSIZ,1,1000,errbuff);

    // filter_exp -> BPF pseudocode
```

```
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle,&fp);

    // capture
    pcap_loop(handle, -1, got_packet, NULL);

    // closing socket
    pcap_close(handle);
}
```

Output:



Observation:

The code was able to capture the packet from the network and sends out an echo reply. The code does not work properly as the protocol is automatically set to 220 even though I set it as 0(echo-reply). The checksum calculation is off and hence the spoofed packet does not reach the host machine.