Assigment 6

By:

Yehonatan Amosi 209542349

Liav Levi 206603193

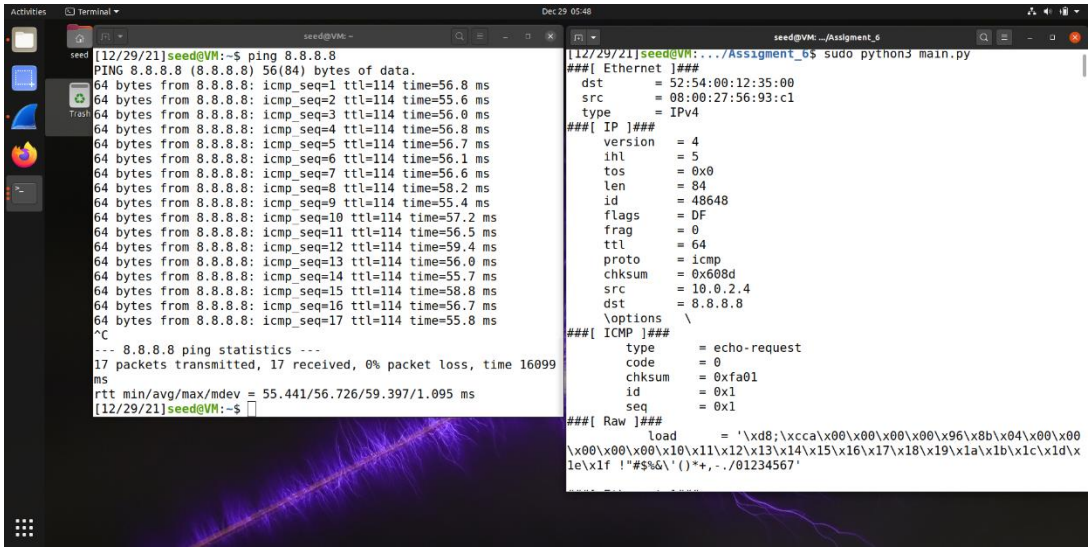
Task 1

A:

in this task we wrote a short program using scapy library and used the method show() so all the data of the packet we capture will be print.

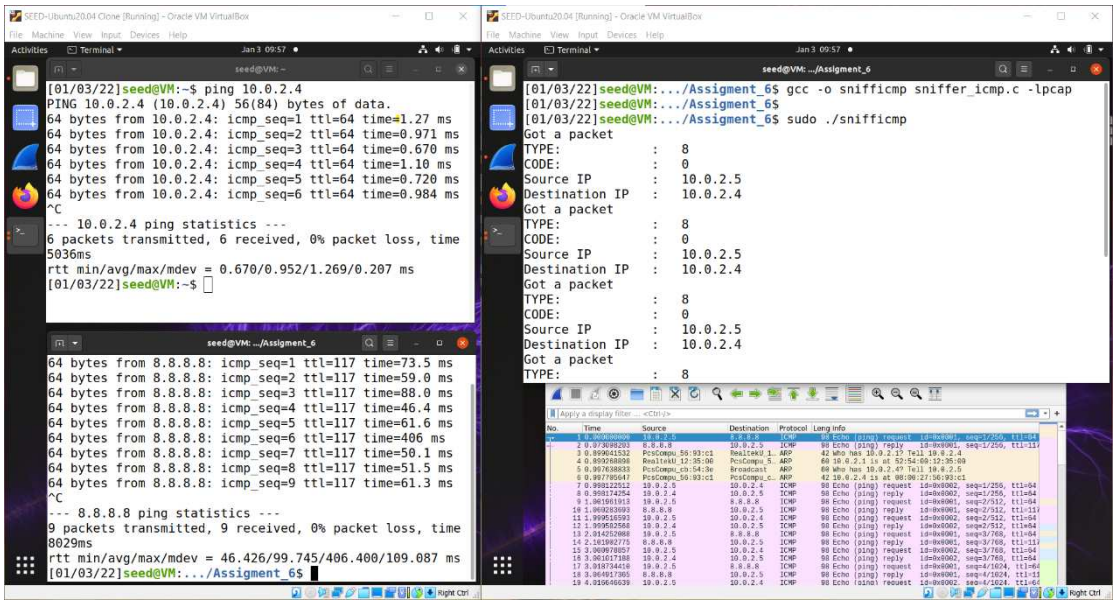
Note: the code is add In the folder of the task.

Result:



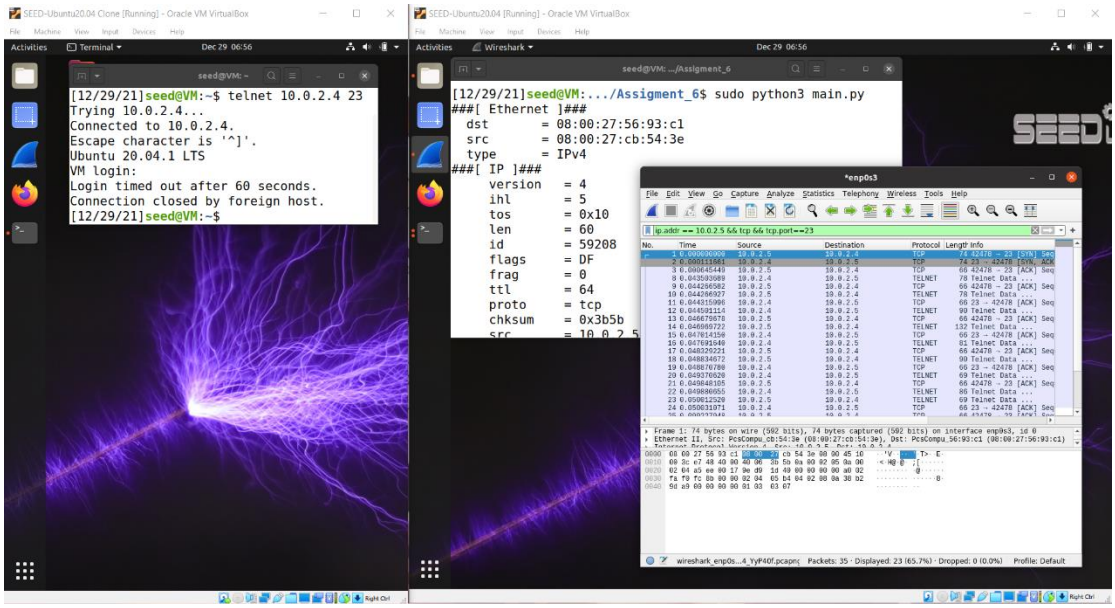
B:

a) Capture only ICMP packet



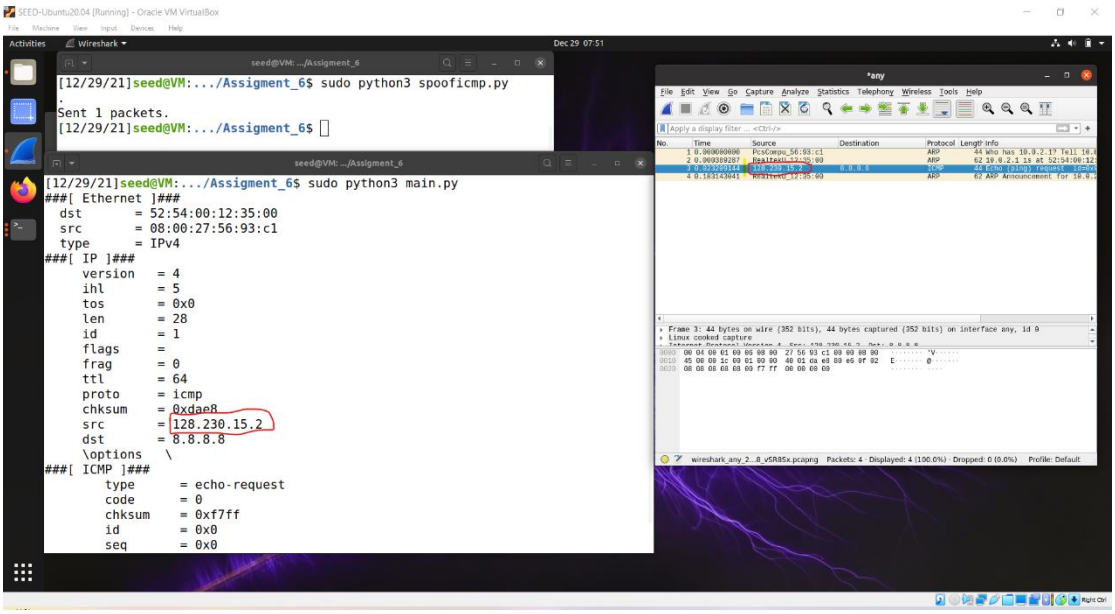
We send ping to my computer and we can see that we capture the packets by looking in the wireshark and for the fact that our sniffer print the data.

b) Capture any TCP packet that comes from a particular IP and with a destination port number 23:



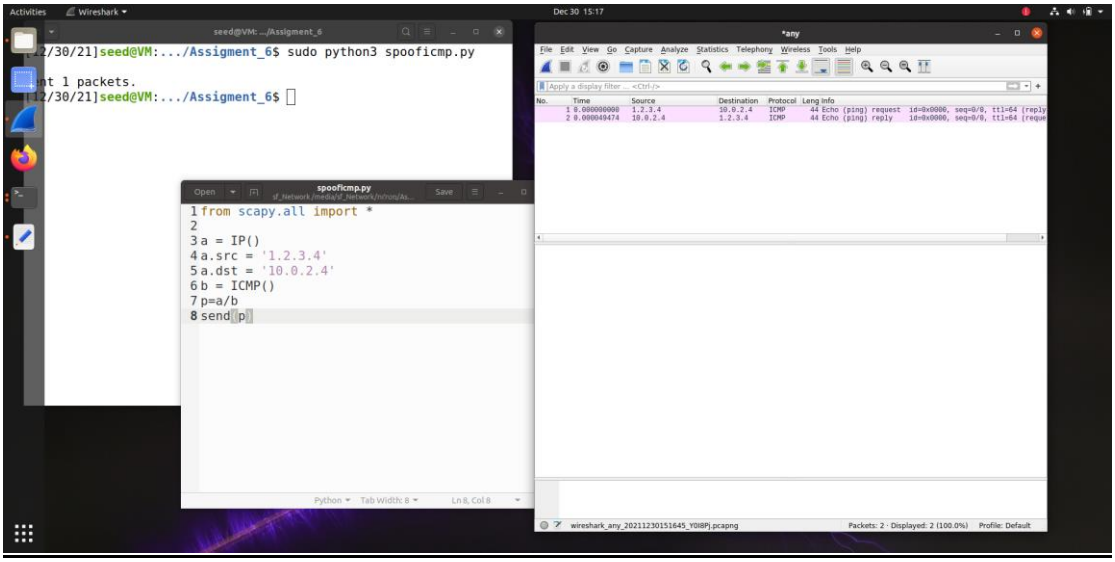
As we can see our sniffer capture only the packets that came from the left computer and print them.

c) 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:



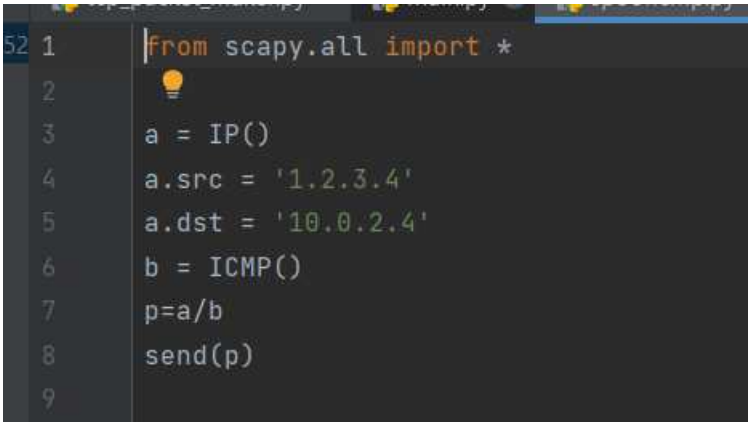
As we can see we capture the requested packets from the requested subnet.

Task 1_2: Spoofing ICMP Packets



as we can see we send packet from IP -1.2.3.4 to the IP – 10.0.2.4

the code:



In the code above, Line 3 creates an IP object from the IP class. Line 3 shows how to set the destination IP address field. If a field is not set, a default value will be used.

Line 6 creates an ICMP object. The default type is echo request. In Line 7, we stack a and b together to form a new object. We can now send out this packet using send() in Line 8.

Task 1_3: Traceroute

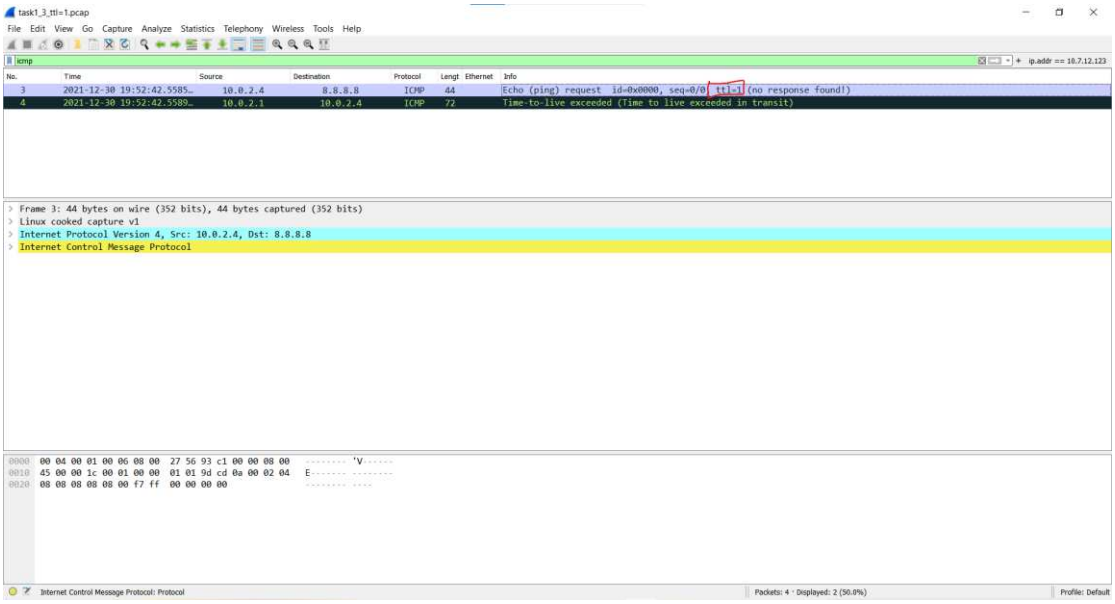
Code:

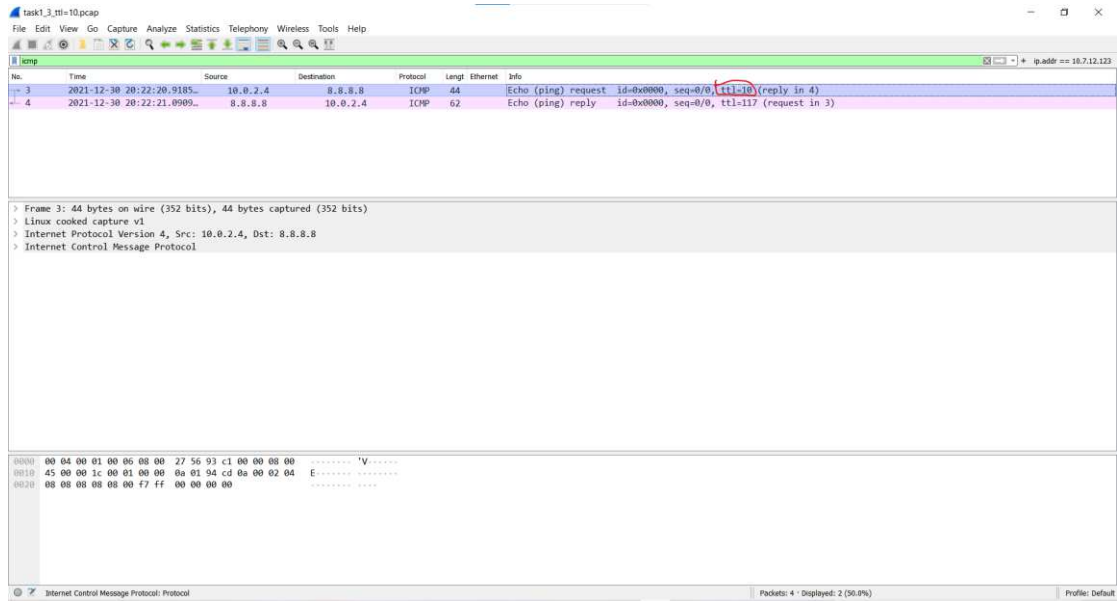
```
from scapy.all import *
a = IP()
a.dst = '8.8.8.8'
a.ttl = 10  # change the number from 1 to 10
b = ICMP()
send(a/b)
```

We were change the ttl value from 1 to 10 and the we got the back echo respond.

To see the full result we add the pcap record and you can see there the full results.

We add the first record and the last one:





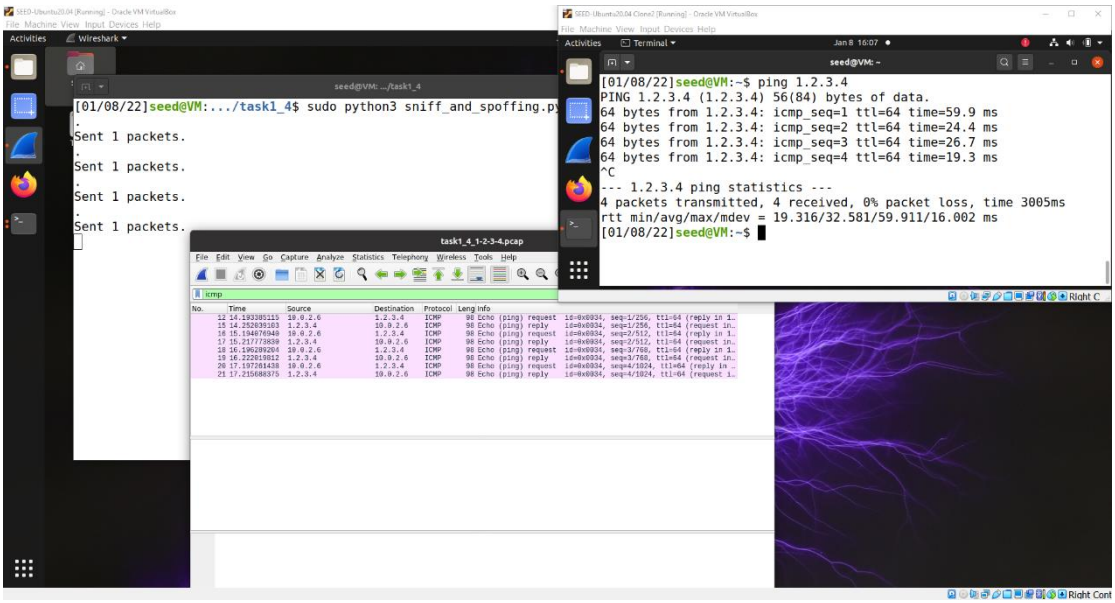
Task 1_4: Sniffing and-then Spoofing

Code:

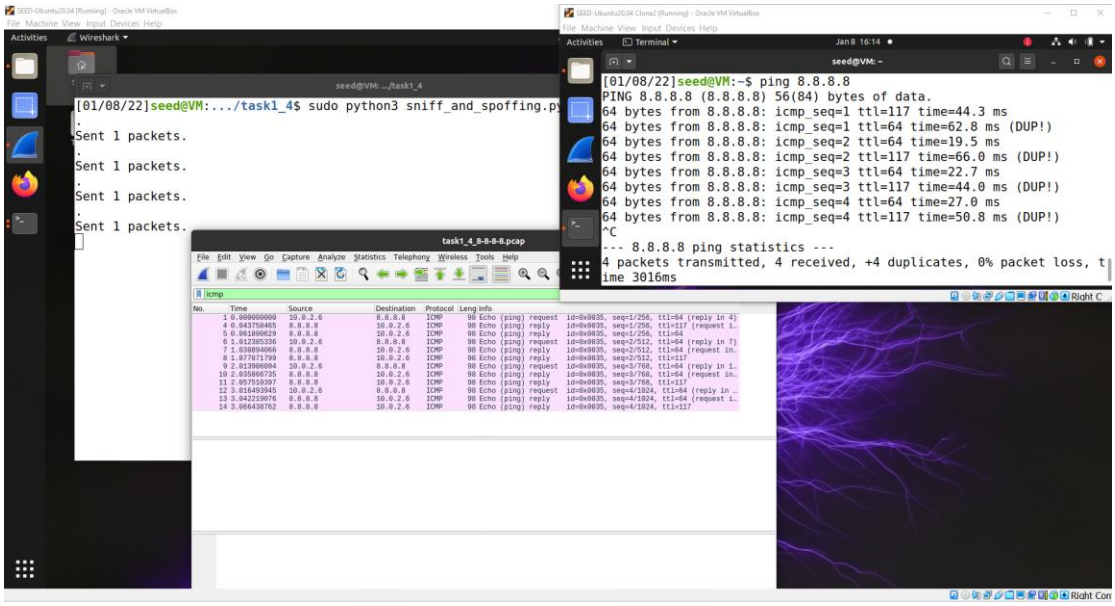
```python
#!/usr/bin/env python3
from scapy.all import *


def sniff_pkt(pkt):
    if pkt[ICMP].type != 8:
        return

    ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
    icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
    data = pkt[Raw].load
    new_pkt = ip / icmp / data
    send(new_pkt)


while 1:
    pkt = sniff(filter='icmp', prn=sniff_pkt)
```

In the method sniff_pkt() we make a fake packets and just switching between source and destination IP.

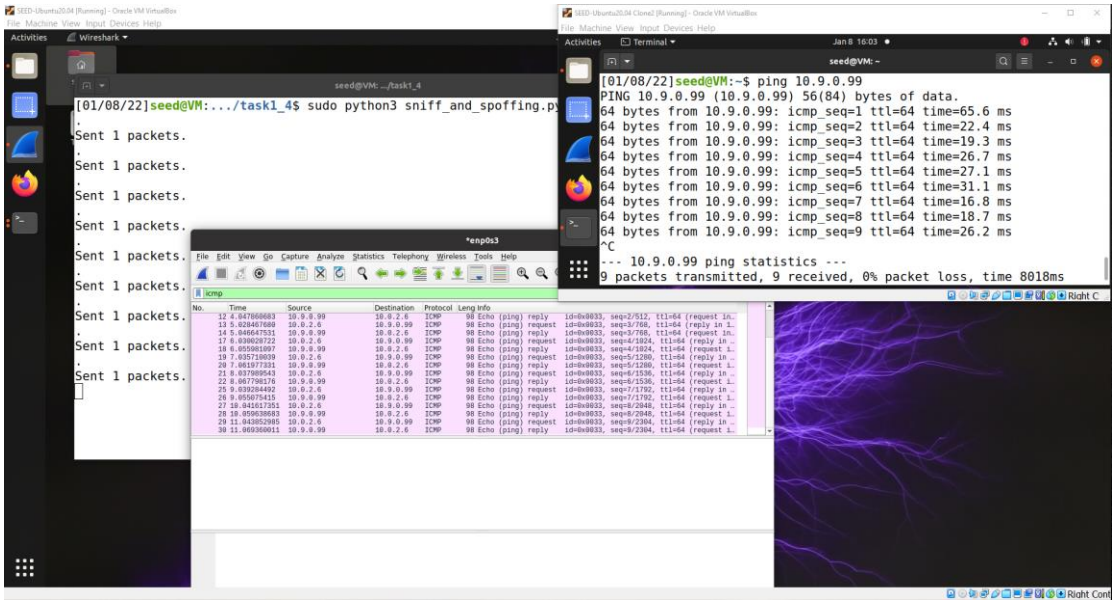
Ping to 1.2.3.4 :



Ping to 8.8.8.8:



Ping to 10.9.0.99:



Task 2.1A: Understanding How a Sniffer Works- In this task, we wrote a sniffer program that use pcap library and print out the source and destination IP addresses of each captured packet.

The code:

sniffer.c 2

Assigment_6 > task2_1 > sniffer.c > main()

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

struct ethheader
{
    u_char ether_dhost[6]; /* destination host address */
    u_char ether_shost[6]; /* source host address */
    u_short ether_type;    /* IP? ARP? RARP? etc */
};
struct ipheader
{
    unsigned char iph_ihl : 4,       //IP header length in byte
        iph_ver : 4;                 //IP version
    unsigned char iph_tos;           //Type of service
    unsigned short int iph_len;      //IP Packet length (data + header)
    unsigned short int iph_ident;    //Identification
    unsigned short int iph_flag : 3, //Fragmentation flags
        iph_offset : 13;             //Flags offset
    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
};
#define IP_HL(ip) (((ip)->iph_ihl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

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) // check if its type IP
    {
        // copy the data to an ip header, the buffer includes all the data we need
        struct ipheader *ipHeader = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("Got a packet\n");
        printf("Source IP        :   %s\n", inet_ntoa(ipHeader->iph_sourceip));
        printf("Destination IP   :   %s\n", inet_ntoa(ipHeader->iph_destip));
    }
}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name eth3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke

    // 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;
}
```

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:

pcap_open_live - this function used to open a listener socket to the network

pcap_compile and Pcap_setfilter - these functions used for making a filter so when we capture packets, we are not capturing all of them, we capture only the packets that we define on the filter like ICMP packets and more.

pcap_loop – this function used for capturing the packets the goes on the network after we define the filter we want.

pcap_close – this function closes the listener socket we opened.

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:

because we are using raw socket and also use promiscuous mode the OS is blocking the access to the raw socket we are using to build the packets. The program fails at access the packets that goes on the network.

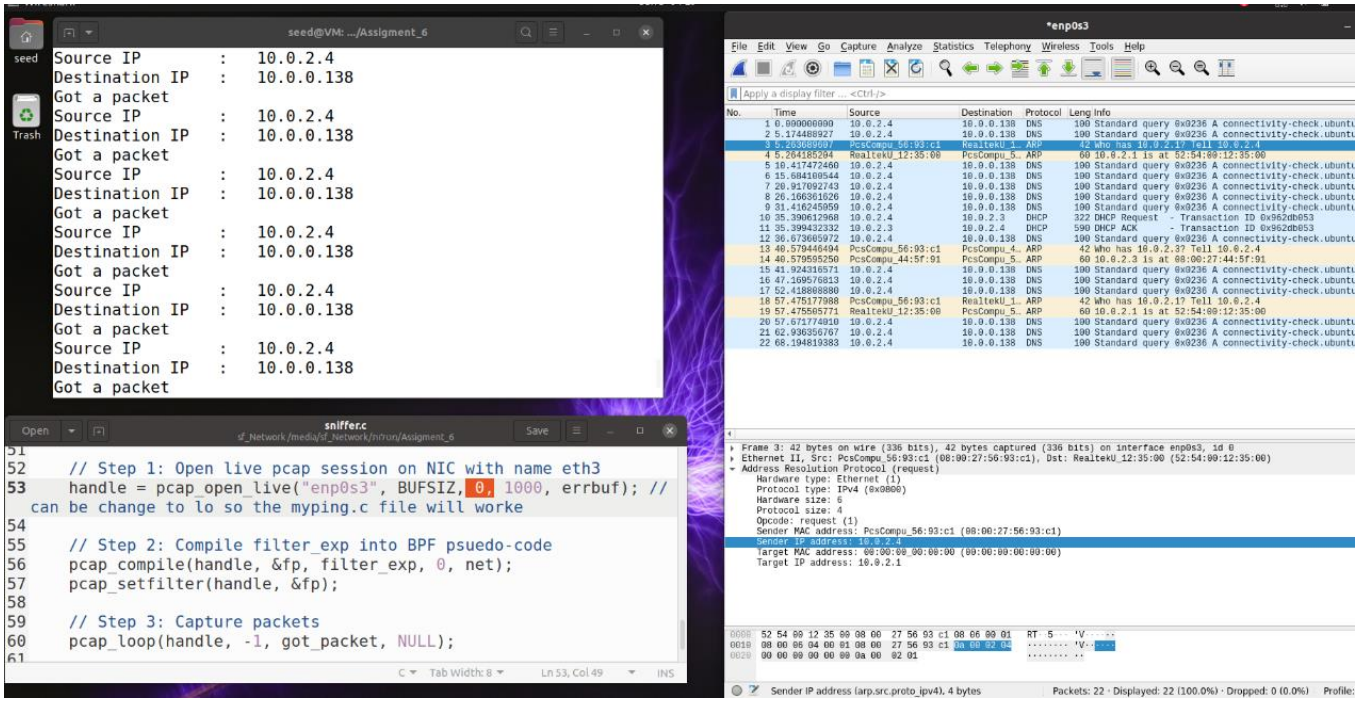
Question 3:

Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

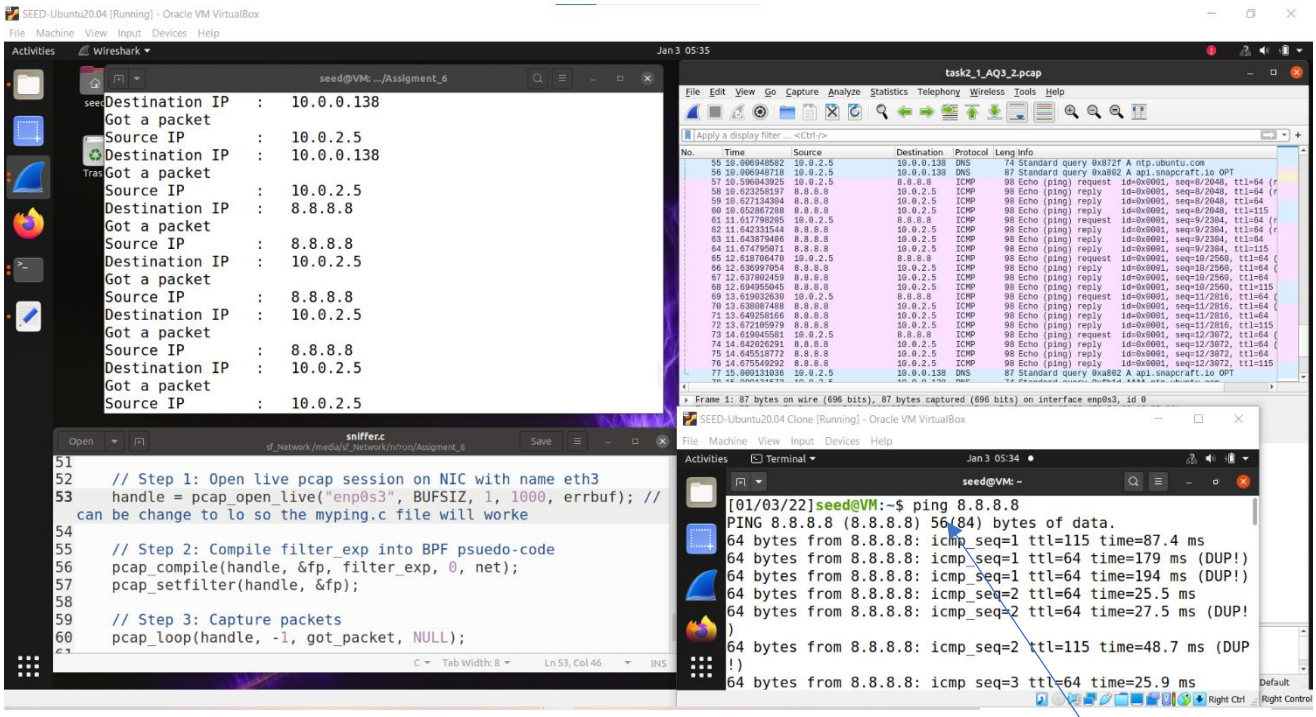
Answer:

when promiscuous mode is off we can see that the program sniffing packets that my IP was the source (including when someone is answering back to me and then my IP isn't the source IP) at the other hand when promiscuous is on we can see that we capturing packet that going from different computers. Here some screen shots to show that:

Promiscuous mode off:



Promiscuous mode on:

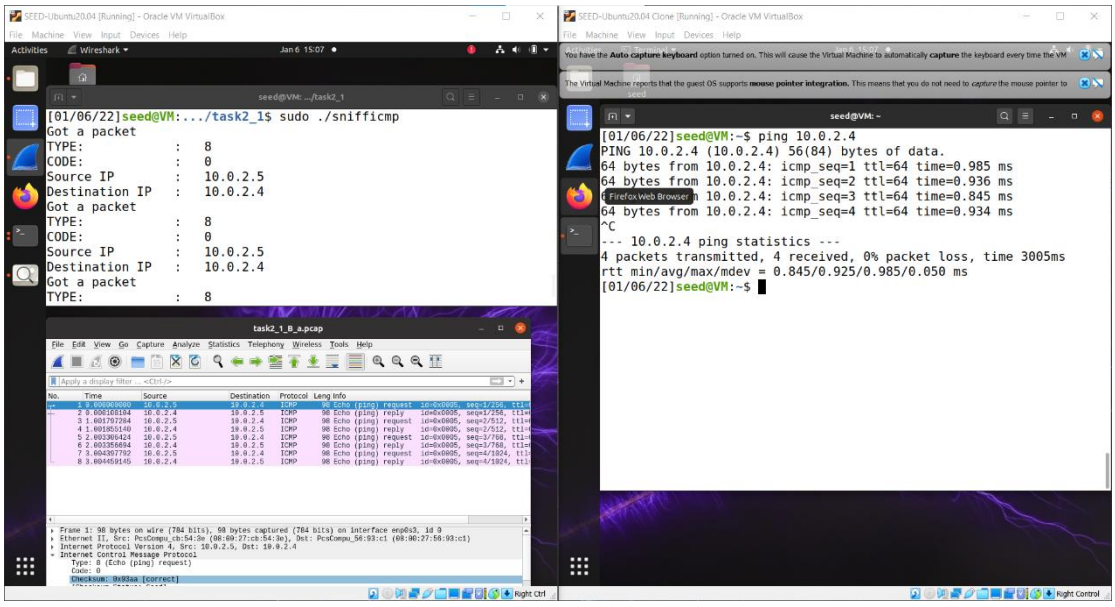


Here we send a ping message to 8.8.8.8 from another VM and her IP is 10.0.2.5

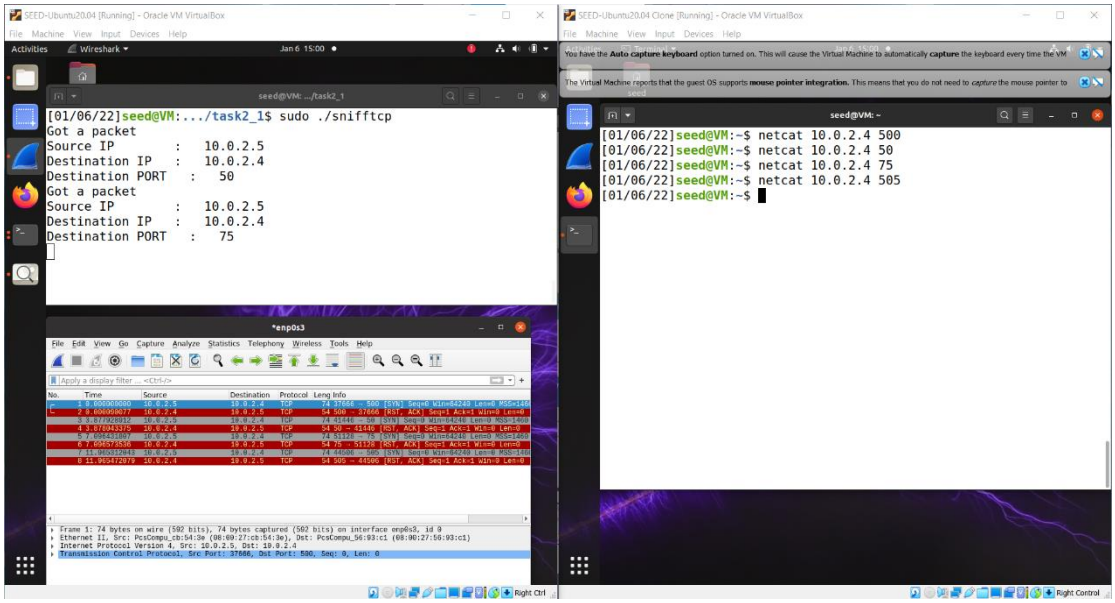
Task 2_1 B

a) Capture the ICMP packets between two specific hosts

We send icmp packet from 10.0.2.5 to 10.0.2.4:



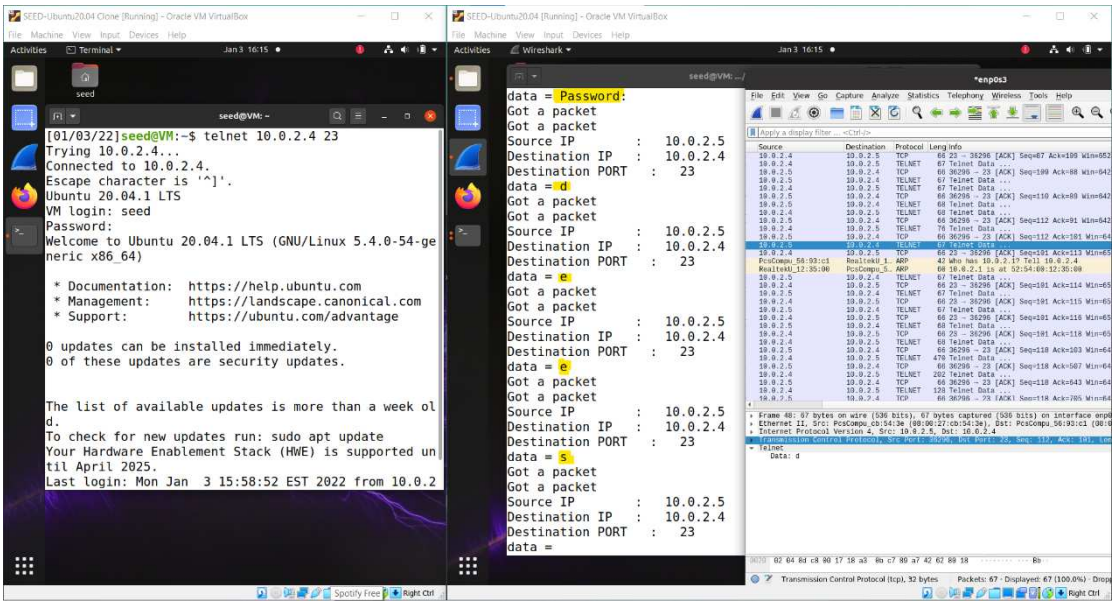
b) Capture the TCP packets with a destination port number in the range from 10 to 100



As we can see only the packets that goes between the asked range ports are capture in our sniffer and printed.

Task 2_1_C: Sniffing Passwords

We use our sniffer program to capture the password when somebody is using telnet on the network that we are monitoring:



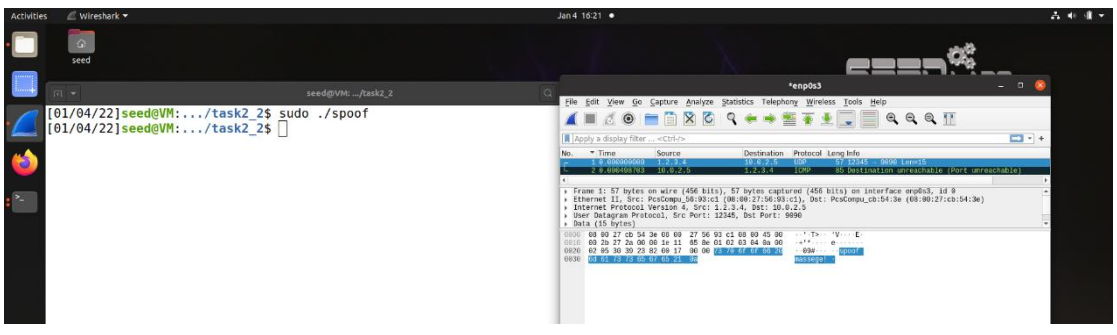
As we can the password is "dees".

Task 2_2:

A: Write a spoofing program. we write a packet spoofing program in C.

code:

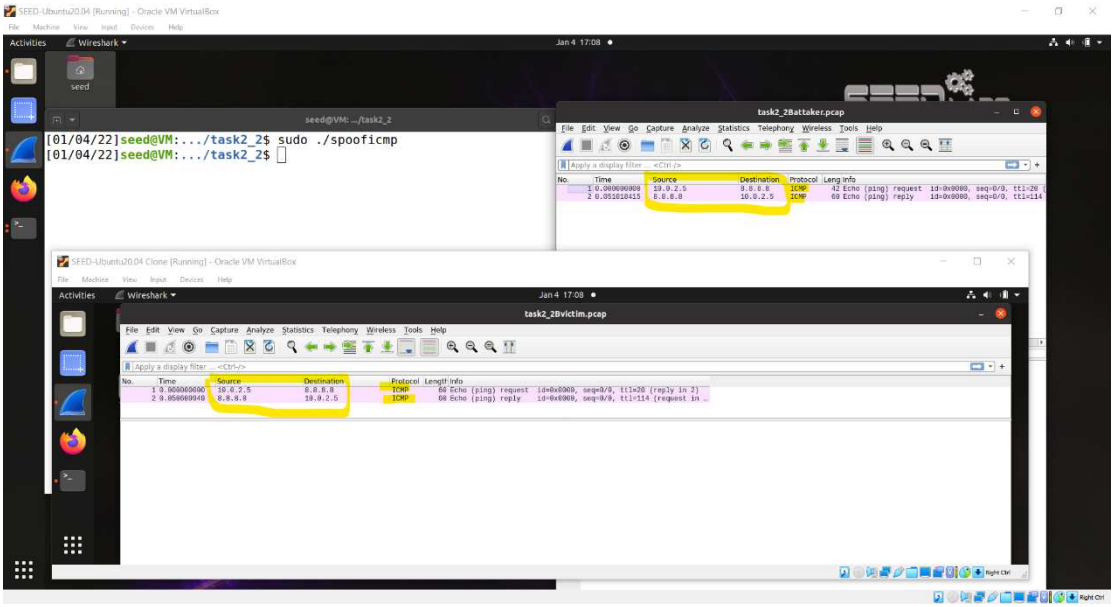
```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <ctype.h>

struct ethheader
{
    u_char ether_dhost[6]; /* destination host address */
    u_char ether_shost[6]; /* source host address */
    u_short ether_type;    /* IP? ARP? RARP? etc */
};
struct ipheader
{
    unsigned char iph_ihl : 4,       //IP header length in byte
        iph_ver : 4;                 //IP version
    unsigned char iph_tos;           //Type of service
    unsigned short int iph_len;      //IP Packet length (data + header)
    unsigned short int iph_ident;    //Identification
    unsigned short int iph_flag : 3, //Fragmentation flags
        iph_offset : 13;             //Flags offset
    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
};
#define IP_HL(ip) (((ip)->iph_ihl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
#define IP_V(ip) (((ip)->ip_vhl) >> 4)

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) // check if its type IP
    {
        // copy the data to an ip header, the buffer includes all the data we need
        struct ipheader *ipHeader = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("Got a packet\n");
        printf("Source IP        :   %s\n", inet_ntoa(ipHeader->iph_sourceip));
        printf("Destination IP   :   %s\n", inet_ntoa(ipHeader->iph_destip));
    }
}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name eth3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke

    // 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;
}
```



B: 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).

The code is add at the folder of Task2_2.

We send a spoofed packet from 10.0.2.5 to 8.8.8.8 but our reals IP was 10.0.2.4:



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

Answers:

yes, the IP packet length can be any arbitrary value, although the packet's total length is overwritten to its original size when its sent.

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

Answers:

we don't have, the reason is that we can tell the karnel to calculate the checksum for the IP header. In IP header fields we have the parameter "ip_check", if we do "ip_check = 0" that will let the karnel to do the checksum by default unless we change it to different value but then we will have to use a checksum method.

• **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?

Answers:

we must 2have the root privileges and they are necessary to run the program that make use of raw socket. If you use non-privileges user you will not have the permissions to change all the fields in the protocol headers. The root privileges will give the ability to change and set any fields in the protocol headers and access to the socket and put the interface card in promiscuous mode. If we run the program without the root privileges, it will fail at socket setup.

Task 2_3:

Code:

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

struct icmpheader
{
    unsigned char icmp_type; // ICMP message type
    unsigned char icmp_code; // Error code
    unsigned short int icmp_chksum;
    unsigned short int icmp_id;
    unsigned short int icmp_seq;
};
struct ethheader
{
    u_char ether_dhost[6]; /* destination host address */
    u_char ether_shost[6]; /* source host address */
    u_short ether_type;    /* IP? ARP? RARP? etc */
};
struct ipheader
{
    unsigned char iph_ihl : 4,       //IP header length in byte
        iph_ver : 4;                 //IP version
    unsigned char iph_tos;           //Type of service
    unsigned short int iph_len;      //IP Packet length (data + header)
    unsigned short int iph_ident;    //Identification
    unsigned short int iph_flag : 3, //Fragmentation flags
        iph_offset : 13;             //Flags offset
    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
};
```

```c
#define IP_HL(ip) (((ip)->iph_ihl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
#define IP_V(ip) (((ip)->iph_ver) >> 4)
#define packt_size 512

unsigned short in_cksum(unsigned short *buf, int length)
{
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;

    /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
    while (nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* treat the odd byte at the end, if any */
    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 hi 16 to low 16
    sum += (sum >> 16);                 // add carry
    return (unsigned short)(~sum);
}
```

```c
void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_addr;
    const int on = 1;
    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    if (sock < 0)
    {
        perror("socket() eroor");
    }
    // Step 2: Set socket option.
    int se = setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
    if (se < 0)
    {
        perror("didnt allowed spoof");
    }
    // Step 3: Provide needed information about destination.
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_addr, sizeof(dest_addr));
    printf("spoofed packet sent :\n");
    printf("    from :  %s\n", inet_ntoa(ip->iph_sourceip));
    printf("    to : %s\n", inet_ntoa(ip->iph_destip));
    close(sock);
    return;
}
```

```c
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) // check if its type IP
    {
        // copy the data to an ip header, the buffer includes all the data we need
        struct ipheader *ipHeader = (struct ipheader *)(packet + sizeof(struct ethheader));
        // check the protocol
        if (ipHeader->iph_protocol == IPPROTO_ICMP)
        {
            int size_ip = IP_HL(ipHeader) * 4;
            struct icmpheader *newicmp = (struct icmpheader *)(packet + size_ip + sizeof(struct ethheader));
            printf("%d \n", newicmp->icmp_type);
            if (newicmp->icmp_type == 8)
            {
                // Construct IP: swap src and dest in faked ICMP packet
                struct in_addr source = ipHeader->iph_sourceip;
                ipHeader->iph_sourceip = ipHeader->iph_destip;
                ipHeader->iph_destip = source;
                ipHeader->iph_ttl = 64;

                // Fill in all the needed ICMP header information.
                // ICMP Type: 8 is request, 0 is reply.
                newicmp->icmp_type = 0;
                newicmp->icmp_chksum = 0;

                int data_len = ntohs(ipHeader->iph_len) - sizeof(struct ipheader)- sizeof(struct icmpheader);
                printf("%d",data_len);
                newicmp->icmp_chksum = in_cksum((unsigned short *)newicmp, sizeof(struct icmpheader) + data_len);
                send_raw_ip_packet(ipHeader);
                return;
            }
        }
    }
    return;
}
```

```c

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp[icmptype] = icmp-echo";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name eth3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke

    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    if (pcap_setfilter(handle, &fp) != 0)
    {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
    }

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);

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

Explanation:

With the attacker machine that was in promiscuous we executed our sniff-spoof program.

The NIC captured all the packets that were relevant (decided by our filter),we modified the packets by swap between source and destination IP and send it buck to the original source as an echo replay.