Computer Networking

Final Project

Packet Sniffing and Spoofing Lab

Dor Azaria

Year 2021

Ariel University

Wireshark recordings (PDF) and code files are included in the folder.

Scapy is a packet manipulation tool. Craft and send packets.

It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies.

Code:

init.py

```
#!/bin/bin/python
from scapy.all import *
a = IP()
a.show()
```

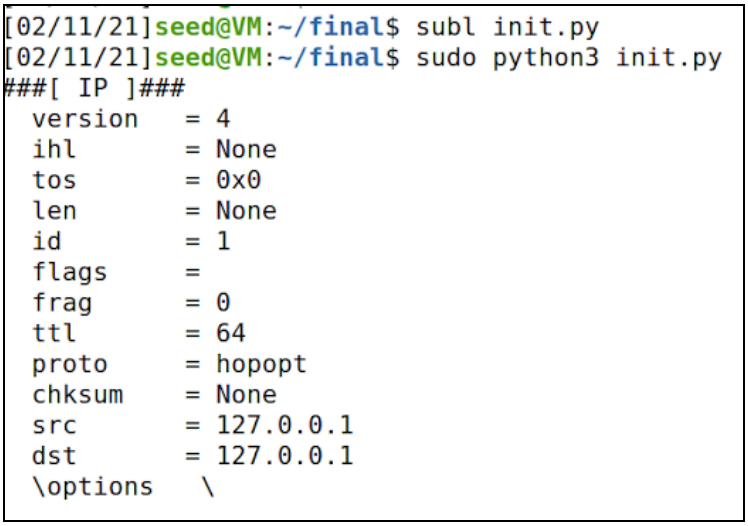
Explanation:

In this code (*init.py*), the `IP()` method creates and returns a new IP default and empty packet.

If we execute the `show()` command it will display the contents of the packet.

I used '*sudo*' the root privilege because it's required for packet manipulations.

Screenshot:



Task 1.1A.
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. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

Code:

sniffer.py

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-5b2e5b5961ac','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

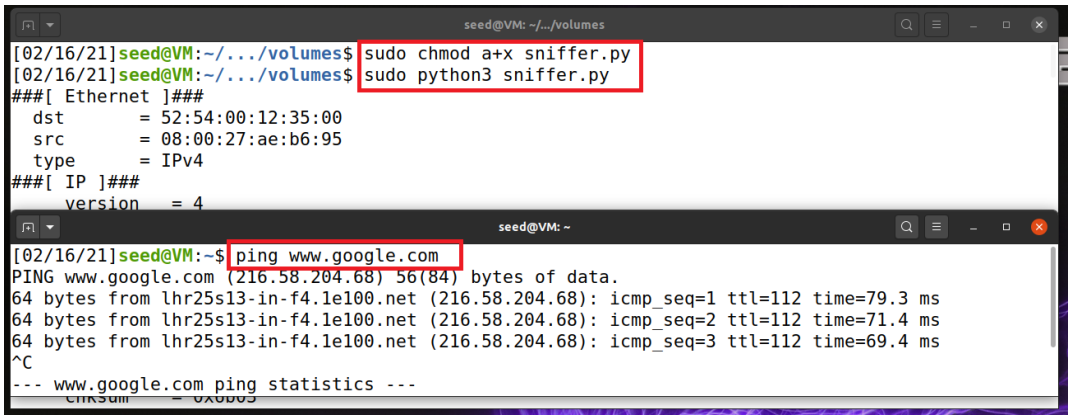
1. **About the code**: I picked these interfaces using the command ‘ifconfig’ in terminal and mentioned them in a list of the interfaces I wish to sniff the packets from. Using the filter, Scapy will show us only ICMP packets.
2. **About the question**: In order to run the program with the root privilege, I made this possible with the help of the following command: `‘sudo chmod a+x sniffer.py’`, the ‘a+x’ means: execute + all.
   I used the ‘ping’ command with ‘google.com’ for ICMP echo request and reply packets. (We can see this case at ‘Screenshot 1’ below).
   I ran the program again, but this time, without using the root privilege.
   As we know, the ‘sudo’ method (superuser do) requires that the user be set up with the power to run "as root".
   So this time, we will run the program without ‘sudo’. (‘Screenshot 2’ below).

Why did this happen?

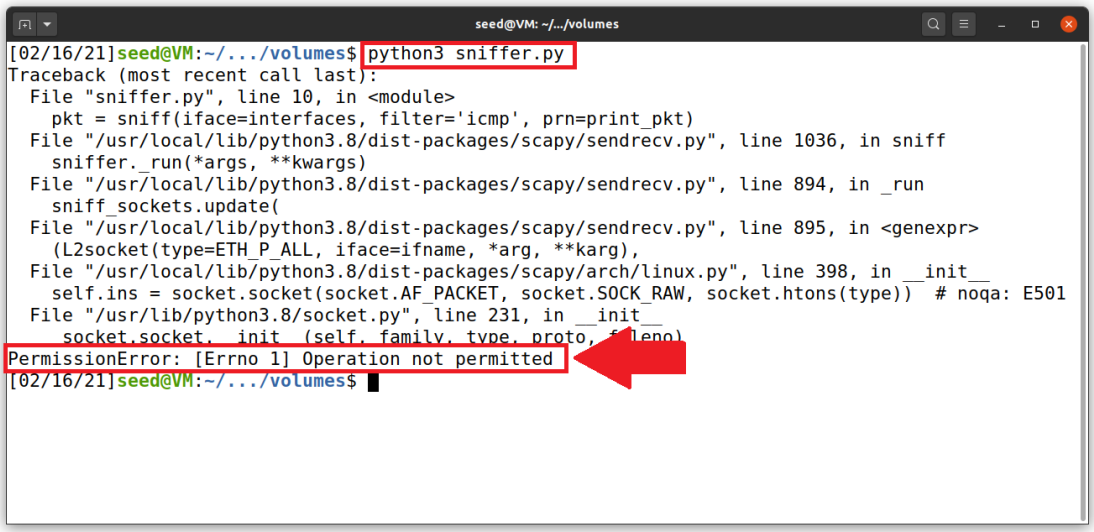
Running the program using sudo allows us to see the whole network traffic in our given interfaces. As we can see in 'Screenshot 2', we got a **PermissionError - "operation not permitted"**.

So, if we want to sniff packets, we need root privileges to see the traffic and to capture the relevant packets.

Screenshot 1:



Screenshot 2:



Task 1.1B.
Usually, when we sniff packets, we are only interested in certain types of packets.
We can do that by setting filters in sniffing.
Scapy's filter use the BPF (Berkeley Packet Filter) syntax;
you can find the BPF manual from the Internet.

Task 1.1B. - Capture only the ICMP packet.

Code:

sniff_only_icmp.py

```
#!/usr/bin/python

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(f"\tSource: {pkt[IP].src}")
                  print(f"\tDestination: {pkt[IP].dst}")

                  if pkt[ICMP].type == 0:
                        print(f"\tICMP type: echo-reply")

                  if pkt[ICMP].type == 8:
                        print(f"\tICMP type: echo-request")

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

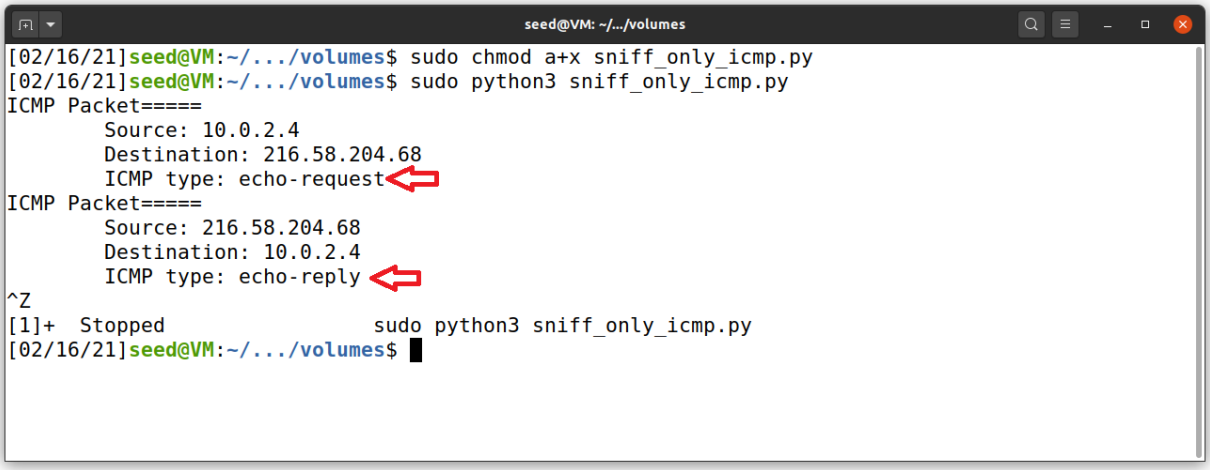
1. **About the code**: I used the same filter like the last code 'sniffer.py' which accepts the syntax of Berkeley Packet Filter. I could have used the method 'show()' but I wanted to print only the relevant details.
   So I printed only the source and destination IP and the ICMP type.

2. **About the question:** I sent a ping an IP of X='www.google.com'.
   This will generate an ICMP echo request packet. If X is alive, the program will receive an echo reply, and print out the response.

Wireshark:

A recording capture file named '*sniff_only_icmp.pdf*' is attached.

Screenshot:



Task 1.1B. - Capture any TCP packet that comes from a particular IP and with a destination port number 23.

Code:

tcp_sniffer.py

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
       if pkt[TCP] is not None:
              print("TCP Packet=====")
              print(f"\tSource: {pkt[IP].src}")
              print(f"\tDestination: {pkt[IP].dst}")
              print(f"\tTCP Source port: {pkt[TCP].sport}")
              print(f"\tTCP Destination port: {pkt[TCP].dport}")

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.0.2.4',
prn=print_pkt)
```

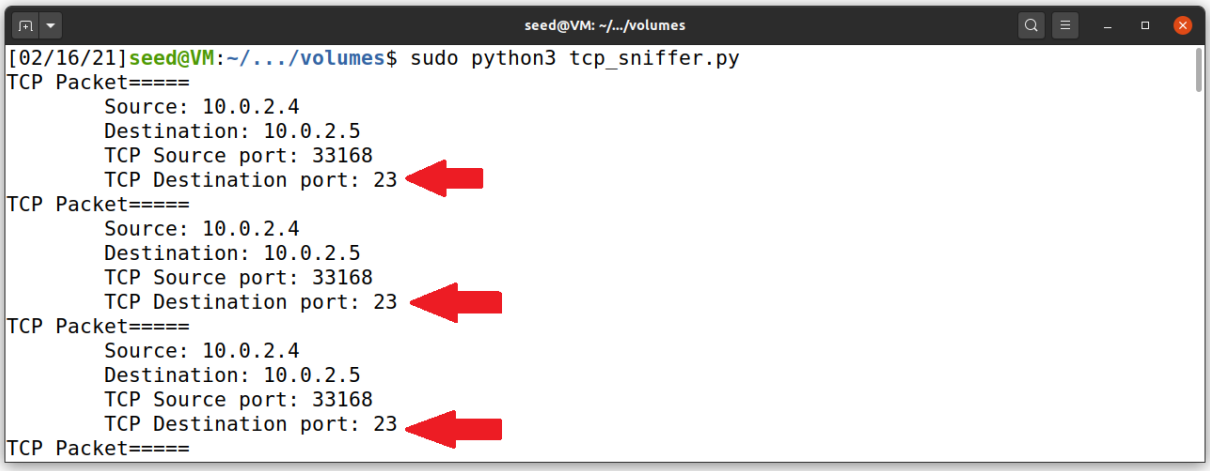
Explanation:

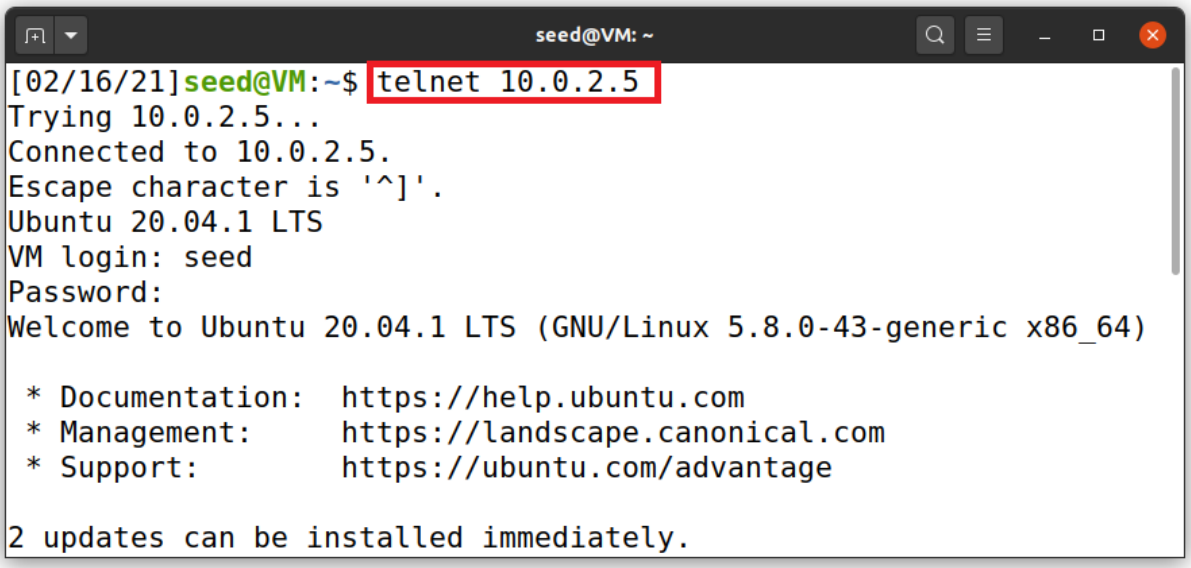
1. **About the code**: I filtered this time with `'tcp port 23 and src host 10.0.2.4'`. The syntax I used for this filter is from BPF syntax website.
   And again like the previous code, I printed only the relevant data for this question. That's why I didn't use 'show()'.
2. **About the question**:
   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.

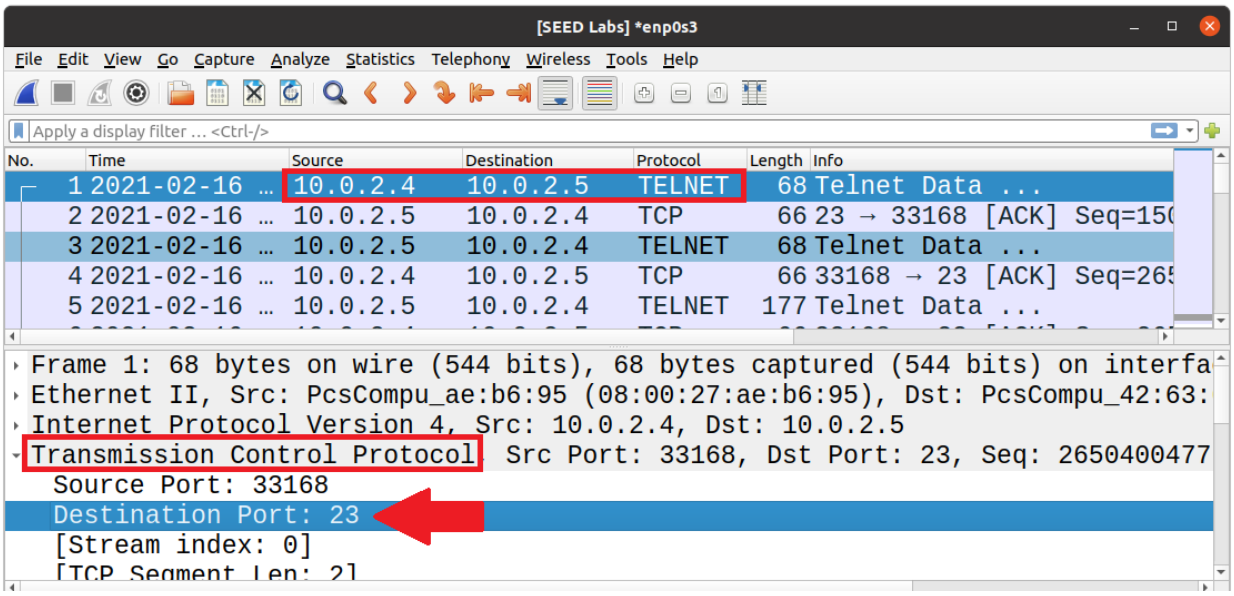
Wireshark:

A recording capture file named '*sniffer_tcp.pdf*' is attached.

Screenshots:







Task 1.1B. - 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.

Code:

subnet_sniffer.py

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16', prn=print_pkt)
```

send_subnet_packet.py

```
from scapy.all import *
ip=IP()
ip.dst='128.230.0.0/16'
send(ip,4)
```

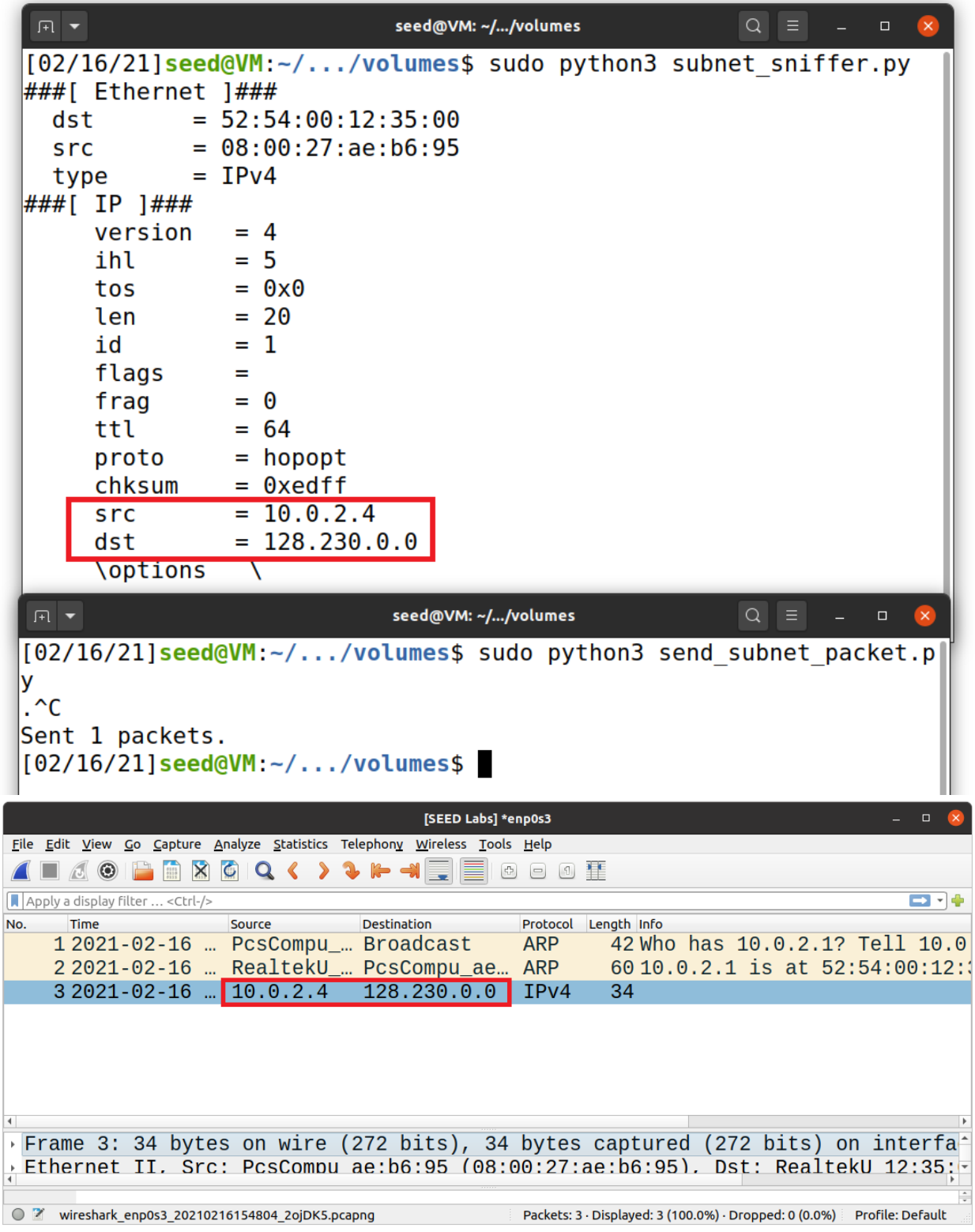
Explanation:

1. **About the code**: in 'subnet_sniffer.py' I picked the filter using Berkeley Packet Filter syntax: `'dst net 128.230.0.0/16'` .
   'dst' means a possible direction.
   'net' returns true if there is a possible type of net, in this case, a subnet which represented in the task details. I wrote the program 'send_subnet_packet.py' the same like the example shown at page 5 of the task file.
2. **About the question**:
   A packet was sent to a particular subnet and the program sniffed only packets that were sent from source '10.0.2.4' to destination IP from the other subnet.

Wireshark:

A recording capture file named '*subnet_sniffer.pdf*' is attached.

Screenshots:



Task 1.2: Spoofing ICMP Packets
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.

What is **packet spoofing**? - When a normal user sends a packet, the OS usually does not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). The OS will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, the destination port number, etc. However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called **packet spoofing.**

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)
```

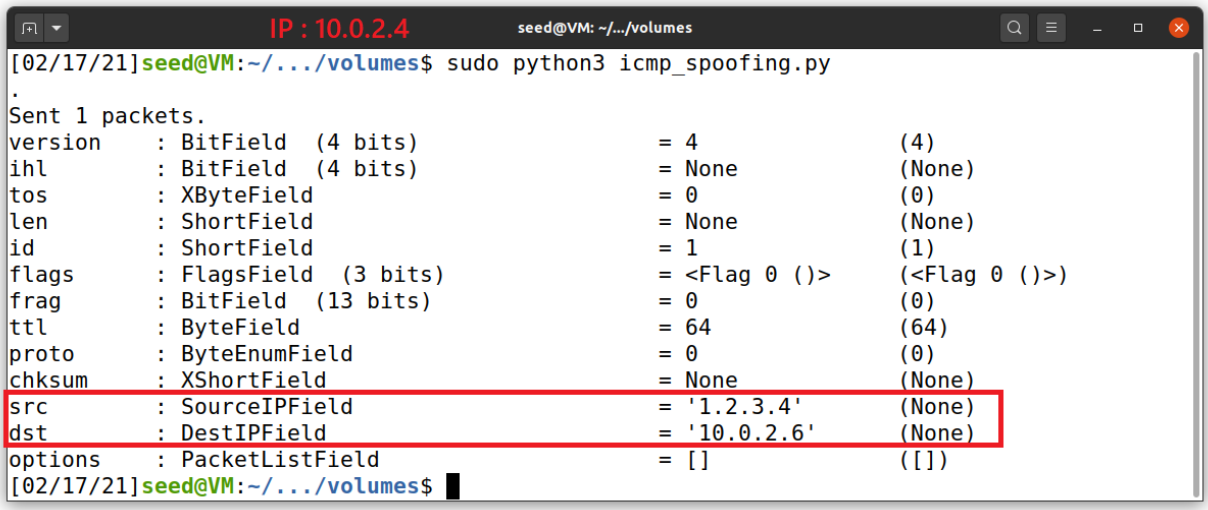
Explanation:

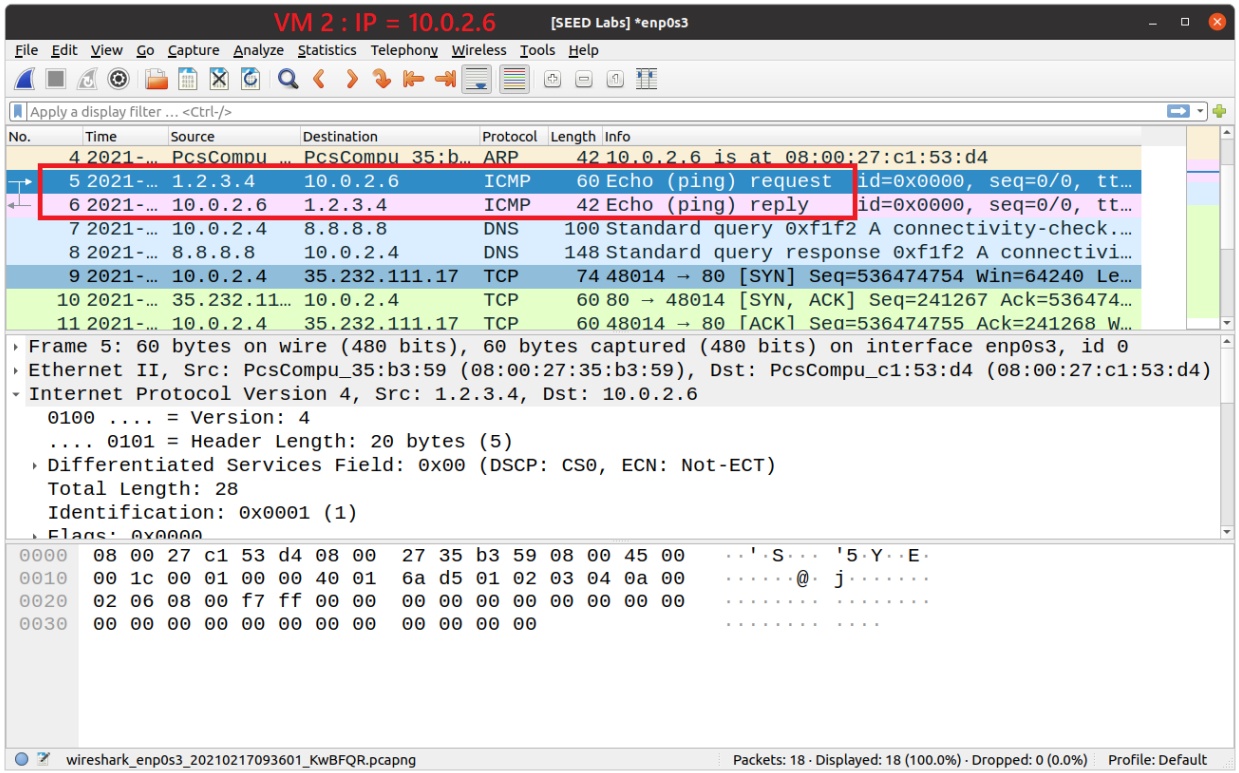
1. **About the code**: I used another VM (IP destination '10.0.2.6') and sent an ICMP packet using a random IP source '1.2.3.4'.
2. **About the question**: I spoofed ICMP echo request packet, and sent it to another VM on the same subnet.
   I used Wireshark to observe whether our request will be accepted by the receiver.
   Using scapy library, the source ip was overwritten with our own ip: 1.2.3.4 and sent the packet to destination 10.0.2.6; the packet was received by 10.0.2.6 and sent an echo reply back to 1.2.3.4.

Wireshark:

A recording capture file named '*icmp_spoofing.pdf*' is attached.

Screenshot:





Task 1.3: Traceroute
write your tool to perform the entire procedure automatically.

What is **traceroute**? - traceroute is a computer network diagnostic command for displaying possible routes and measuring transit delays of packets across an Internet Protocol network.

Code:

my_traceroute.py

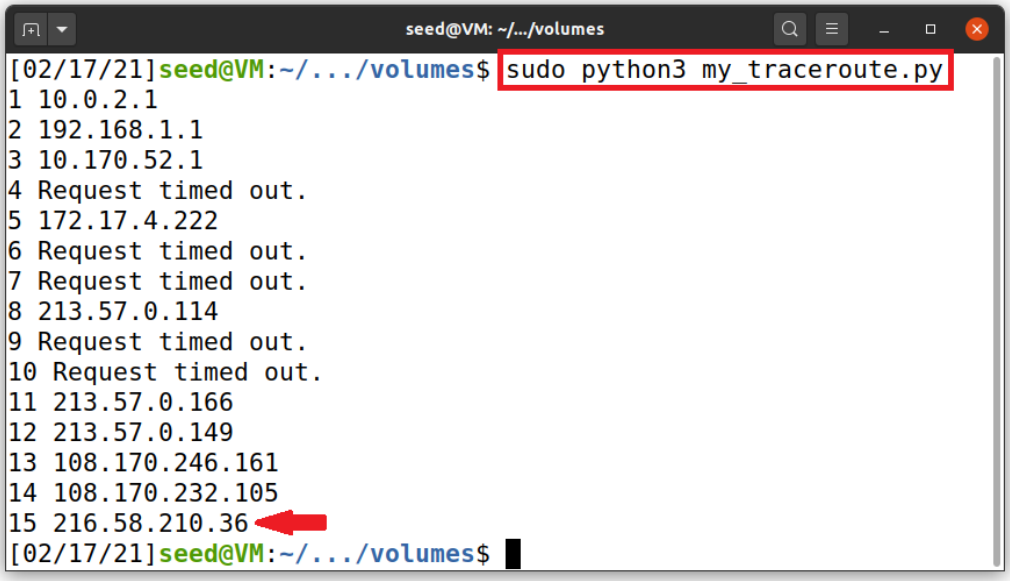
```
from scapy.all import *

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(f"{i} Request timed out.")
	elif response.type == 0:
		print(f"{i} {response.src}")
		inRoute = False
	else:
		print(f"{i} {response.src}")

	i = i + 1
```

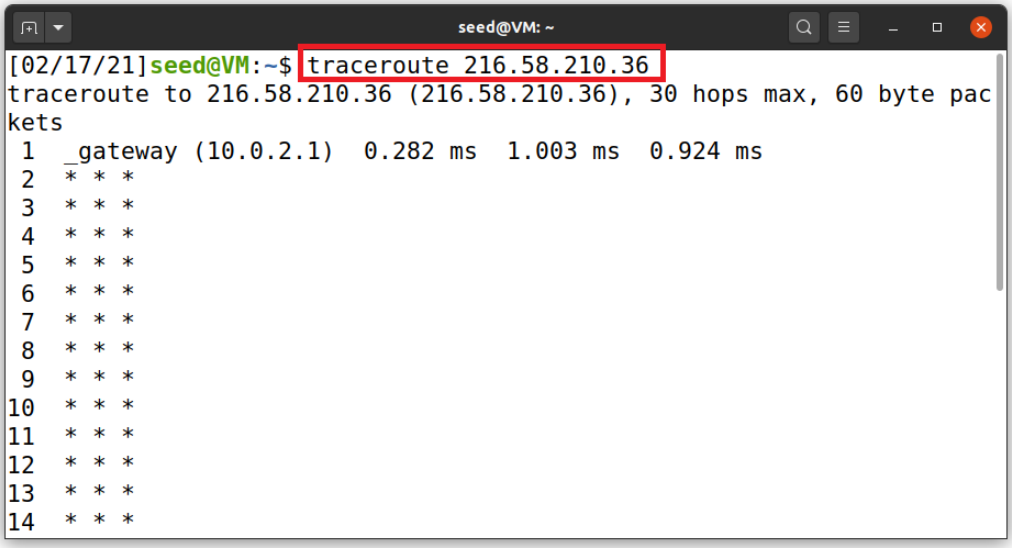
Explanation:

1. **About the code**: I programmed my own traceroute using Scapy library.
   I asked for the destination IP '216.58.210.36' which is Google LLC.
   The flag 'ttl' of the packet is increasing by one in each given packet.
   I used a while-loop which will keep iterating as long as it's routing.
   The sr1() method of Scapy is a method which will listen and wait for a packet response. (timeout = time limit for response, verbose = ignore printing unnecessary details).

2. **About the question**: This program figures out how many routers (hops) it takes to send out that packet all the way to the IP address destination.
   Each line at the display is a different router. The time- to-live is used to return an error of each hop till the destination, this way we can print each IP router till it stops. In this case we reached 15 different routers, 5 of them are timed out.
   A "Request timed out" message at the beginning/middle of a traceroute is very common and can be ignored. This is typically a device that doesn't respond to ICMP or traceroute requests.

Screenshots:





Task 1.4: Sniffing and-then Spoofing.
you will combine the sniffing and spoofing techniques to implement the following sniff-and then- spoof program.

What is an **ARP** protocol? - a communication protocol used for discovering the link layer address, such as a MAC address, associated with a given internet layer address, typically an IPv4 address. Such a request consists of special packets commonly known as 'who-has' packets. 'Who-has' packets are packets that the IP system broadcasts to all devices on a network (or VLAN), to determine the owner of a specific IP address.

Code:

sniffing_and_spoofing.py

```
#!/usr/bin/python
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(f"Flip: src {src} dst {dst} type 8 REQUEST")
            print(f"Flop: src {dst} dst {src} type 0 REPLY\n")
            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)
```

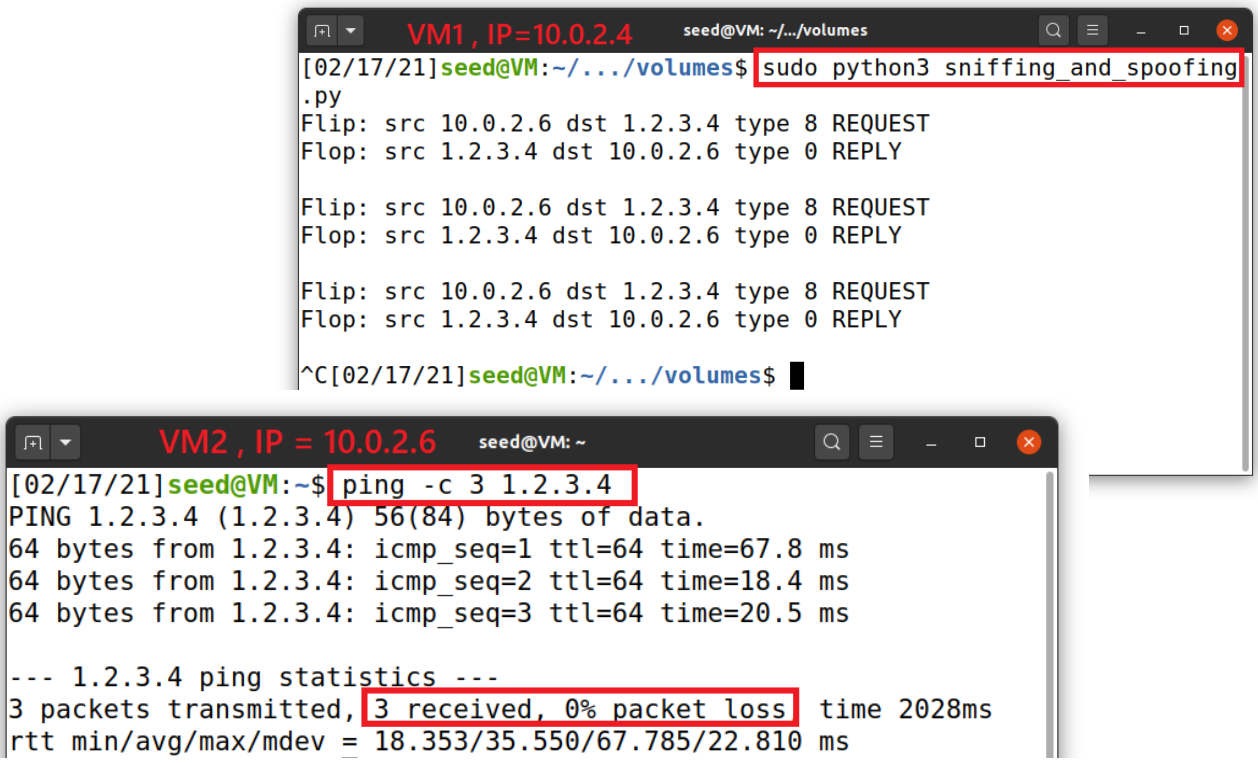
Explanation:

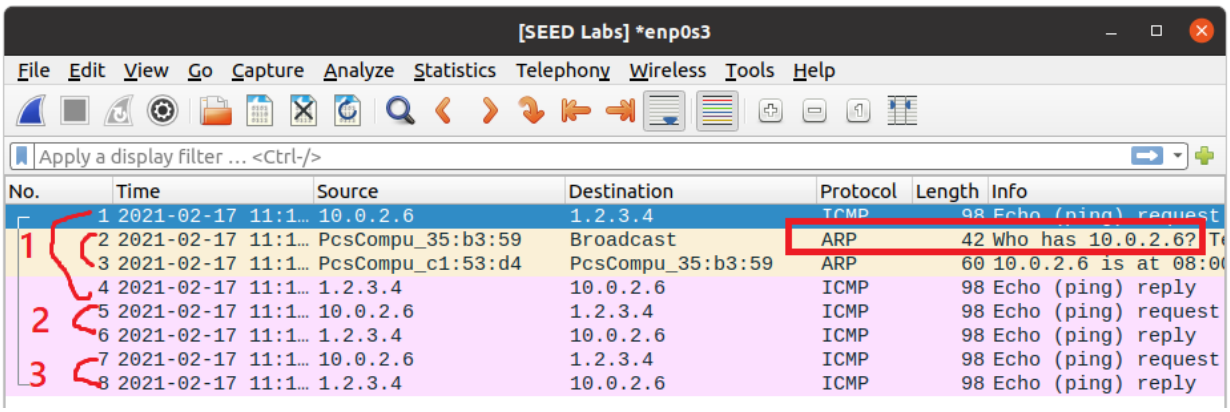
1. **About the code**: the 'if' block is checking if an ICMP is a request.
   If it's true, the reply packet will be based on details derived from the original packet, but it will flip dst and src so whenever it sees an ICMP echo request, regardless of what the target IP address is, the program should immediately

send out an echo reply using this packet spoofing technique.
The pkt[Raw].load is used to store the original packet data payload this way it Will return properly to the sender.

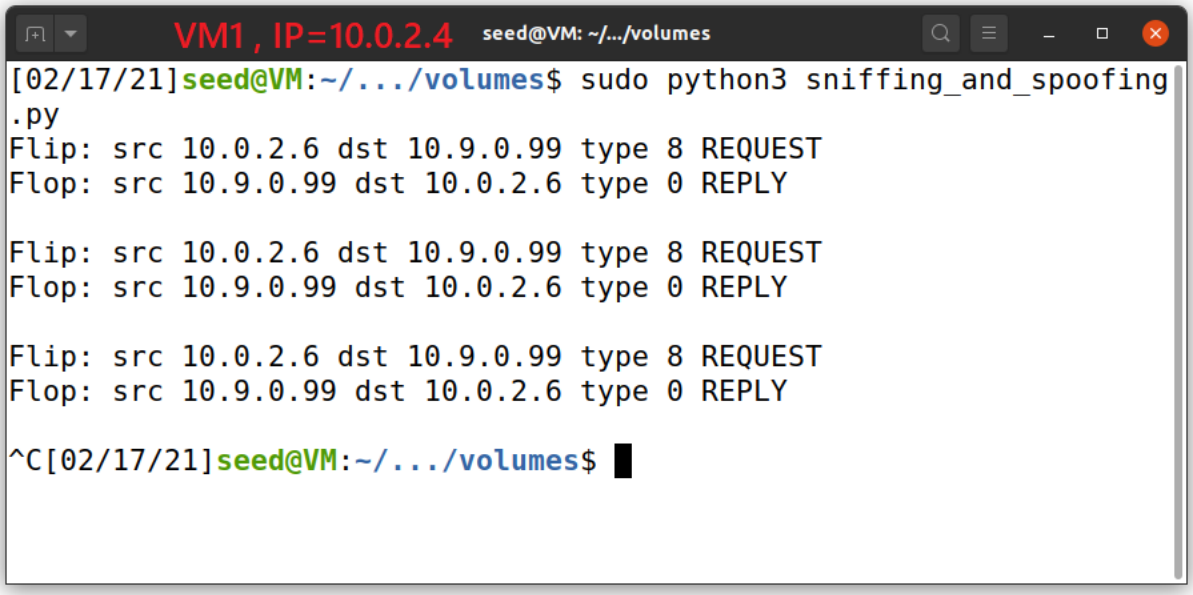
2. **About the question (explanation with screenshots):**
   In this task I tested 3 scenarios of 3 different IPs to ping.
   The program 'sniffing_and_spoofing.py' will sniff for any ICMP packets in the subnet, and when it catches one, the program will return to the sender an ICMP reply packet back. So even if the IP echo request isn't available at all, the program will always return to the sender a reply packet.
   I used two VMs for this task.
   My original VM will be known as 'VM1' with IP address of '10.0.2.4' and the other one will be 'VM2' with IP address of '10.0.2.5'.

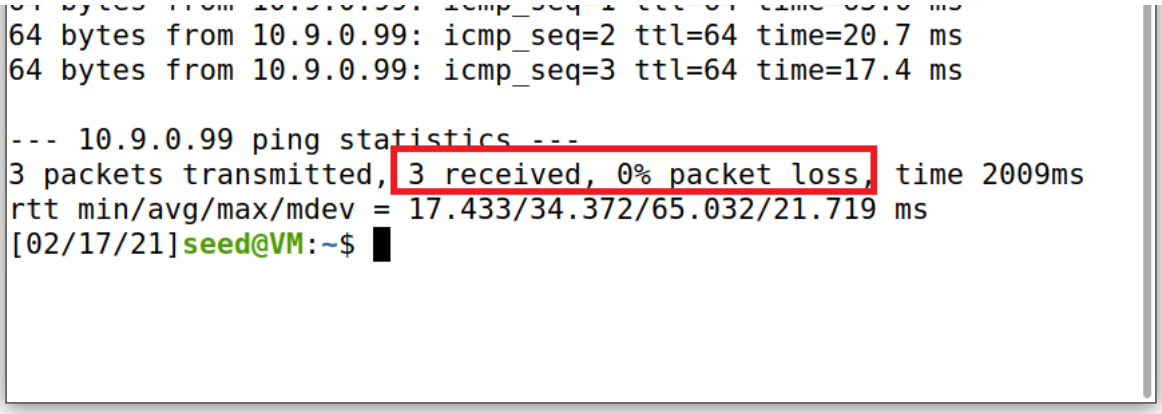
At the first scenario VM2 sends a ping to '1.2.3.4' which is a non-existing host on the Internet. Without the program we will get a 100% packet loss because it will never return to the source. We can see in the screenshot of the wireshark that the ARP protocol is asking for 1.2.3.4 and asking on the network who has that IP destination? So the attacker (my program on VM1) returns with an answer to it and the ICMP packet reply is coming back to VM2.(A wireshark capture is attached: 'snifsnof1.pdf').





At the second scenario, VM2 sends a ping to '10.9.0.99' which is a non-existing host on the LAN. In this scenario we're getting the same concept with the same idea of the ARP protocol. Even though the host doesn't even exist. The program from VM1 will send back an ICMP response packet. (A wireshark capture is attached: 'snifsnof2.pdf').





At the third scenario, VM2 sends a ping to '8.8.8.8' which is an existing host on the Internet. This scenario is different from the others because it really exists on the net. So in this case we're getting duplicate responses, that's because the real destination is responding to the source, but my program is also responding to the source.
We can see it very clear in the screenshots and in the wireshark recording.
(A wireshark capture is attached: 'snifsnof3.pdf').

VM1 , IP=10.0.2.4 — seed@VM: ~/.../volumes

```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY
```

VM2 , IP = 10.0.2.6 — seed@VM: ~

```
[02/17/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=26.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=96.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=31.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=82.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=22.2 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss,
 time 2045ms
rtt min/avg/max/mdev = 22.174/51.877/96.870/31.460 ms
[02/17/21]seed@VM:~$
```

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
2	2021-02…	PcsCompu_35:…	Broadcast	ARP	60	Who has 10.0.2.6? T
3	2021-02…	PcsCompu_c1:…	PcsCompu_35:b3:59	ARP	42	10.0.2.6 is at 08:0
4	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
5	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
6	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
7	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
8	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
9	2021-02…	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
10	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
11	2021-02…	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
12	2021-02…	PcsCompu_c1:	RealtekU_12:35:00	ARP	42	Who has 10.0.2.12 T

1
2
3

Task 2.1: Writing Packet Sniffing Program

What is **pcap**? - **pcap** is an application programming interface (API) for capturing network traffic.

Code:

sniffer.c

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
  }
}

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

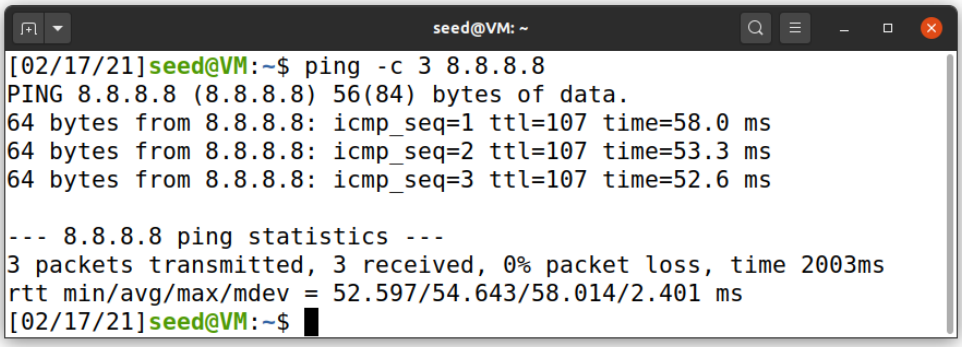
Explanation:

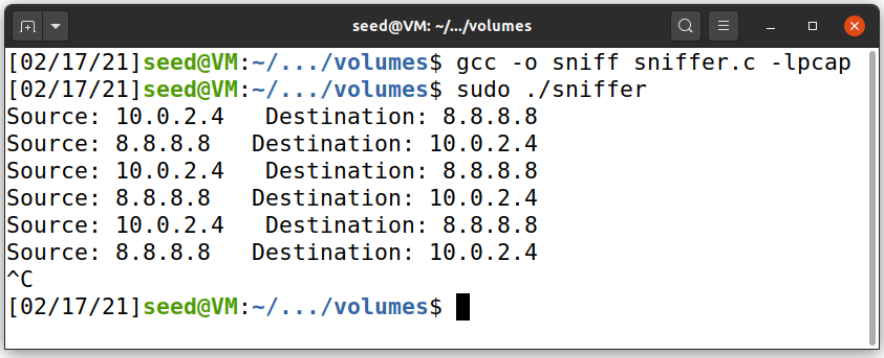
1. **About the code:** I created a sniffer program using pcap library for capturing network traffic and displays the source and the destination IP addresses.
   I used the same filter syntax of BPF for filtering only ICMP packets.
   When the program captures a packet, It checks if the header is IPv4 type and if it's true, it will print the source and destination of that IP header packet.
2. **About the question:** I sent a ping with IP and the program sniffed it and printed the correct source and destination.

Wireshark:

A recording capture file named '*snifferc.pdf*' is attached.

Screenshots:





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.

First step, we open a live pcap session on NIC with name enp0s3, this operation is done by the 'pcap_open_live' (a function from the pcap library). This function lets us see the whole network traffic in the interface and binds the socket.

Second step, we are setting the filter by using the following methods:

pcap_compile() is used to compile the string str into a filter program

pcap_setfilter() is used to specify a filter program.

Third step, we capture the packets in a loop and process the captured packets using the 'pcap_loop' function, the -1 means an infinity loop.

Question 2.

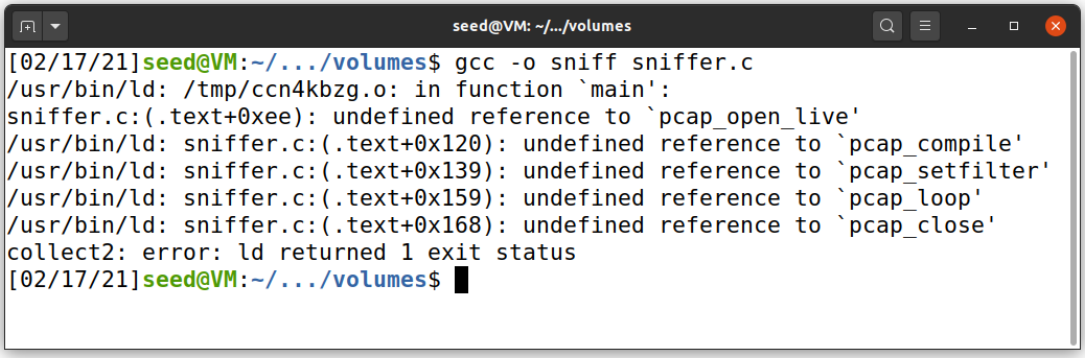
Why do you need the root privilege to run a sniffer program?

Where does the program fail if it is executed without the root privilege?

Solution Q2.

A root privilege is required to set up the card in promiscuous mode and raw socket, this way we can see the whole network traffic in the interface.

If we run the program without a root user, where the pcap_open_live function fails to access the device and so it will cause an error to the whole program.



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.

The promiscuous mode is a part of the chip in my NIC card, which is within the computer, activated using the 'pcap_open_live' function.
If you change the third param of the 'pcap_open_live' function to 0 = OFF and anything other than 0 will be ON.

If I'll turn the promiscuous mode OFF, a host is sniffing only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer.

On the other hand, if I turn the promiscuous mode ON, it sniffs all traffic on the wire and you will get all packets your device sees, whether they are intended for you or not.

Task 2.1B: Writing Filters - Capture the ICMP packets between two specific hosts.

Code:

sniffer_icmp.c

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));

        /* determine protocol */
    switch(ip->iph_protocol) {
        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 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:

1. **About the code:** I took the previous code and added some features to it.

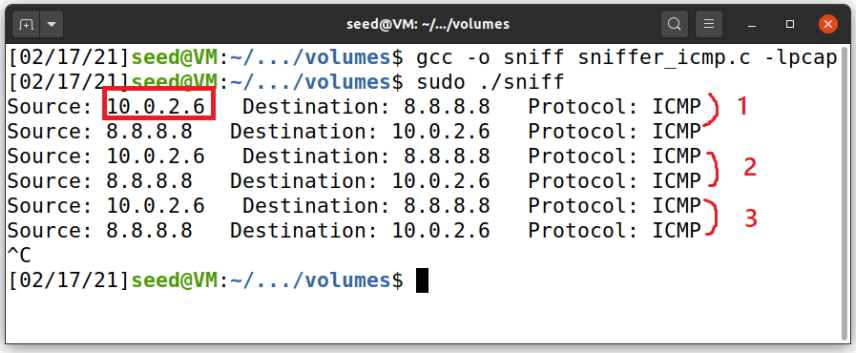
   The pcap filter which is based on the BPF syntax is now: `"ip proto icmp"`.

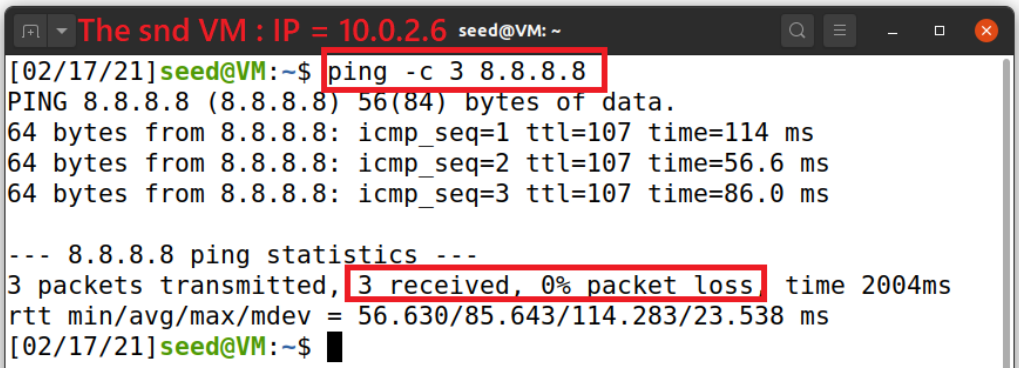
   My current IP address is 10.0.2.4. The program checks if it's IPv4 type and if it's true, it's also checks if the protocol is ICMP - In this case we'll also print that It's an ICMP protocol type.
2. **About the question**: I used another VM (10.0.2.6 - the victim) which sent a ping to '8.8.8.8'. The attacker (10.0.2.4) sniffed the packet and displayed it.

Wireshark:

A recording capture file named '*sniffer_icmpc.pdf*' is attached.

Screenshots:





Task 2.1B: Writing Filters - Capture the TCP packets with a destination port number in the range from 10 to 100.

Code:

sniffer_tcp.c

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

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("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s", inet_ntoa(ip->iph_destip));
        /* determine protocol */
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\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[] = "proto TCP and dst portrange 10-100";
  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;
}
```

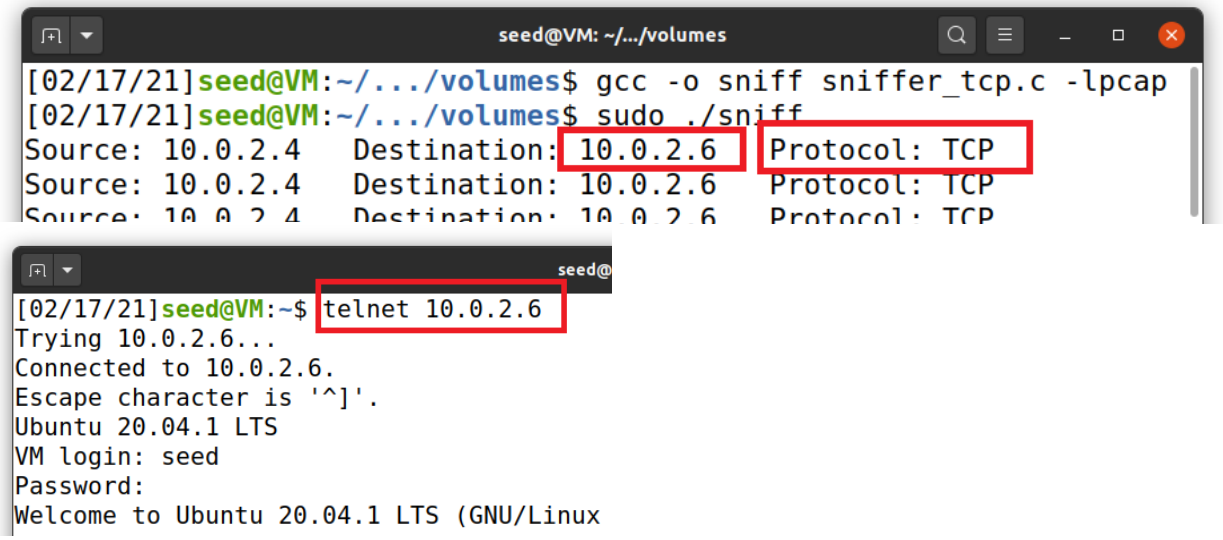
Explanation:

1. **About the code:** My pcap filter is: "proto TCP and dst portrange 10-100".
   The program captures the packet and checks if the header is IPv4 type.
   If it's true, it also checks if the protocol type is TCP, and if it's also true it will print it.
2. **About the question:** I used telnet to capture the TCP packet and sent it to another alive VM . The program captures the packet and displays it.
   The syntax used for this filter is from BPF syntax website.

Wireshark:

A recording capture file named '*sniffer_tcpc.pdf*' is attached.

Screenshot:



Task 2.1C: Sniffing Passwords

Code:

pwd_sniffer.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 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;
}
```

Explanation:

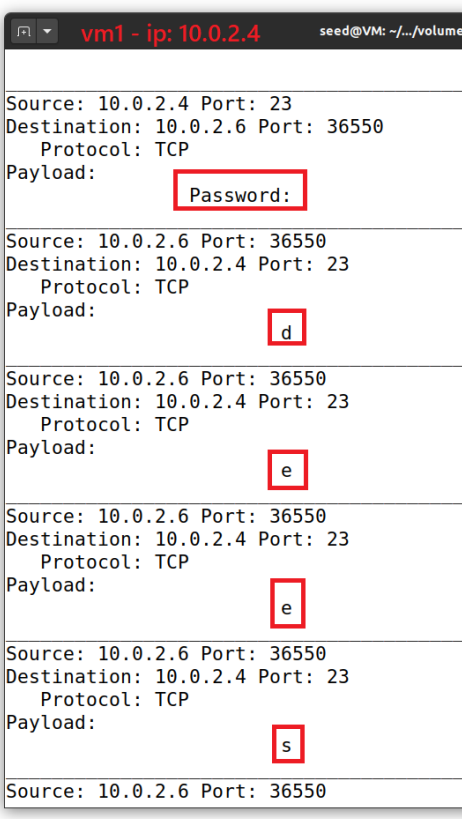
1. **About the code**: My pcap filter is: "tcp port telnet".
   The syntax used for this filter is from BPF syntax website.
   The program was set to sniff the tcp packets of telnet and when executed and performed a telnet from machine 10.0.2.4 to 10.0.2.6; the data was captured which includes password.

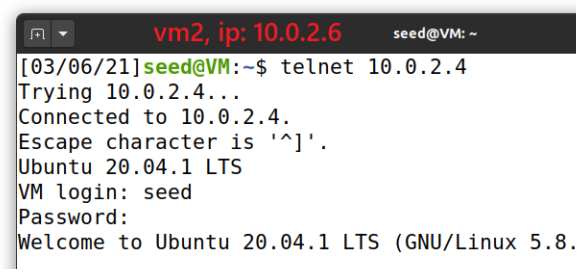
2. **About the question**:The 'pwd_sniffer.c' program is running and listening to the tcp packets.
   As telnet is a tcp program, the packets are captured, and the payload was displayed and in a clear text. I marked the password in red as we can see in the screenshots down below.

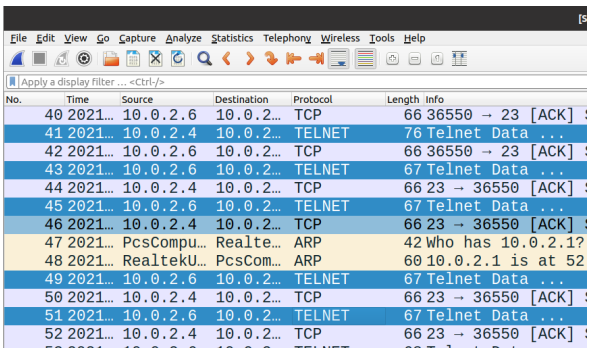
Wireshark:

A recording capture file named '*password.pdf*' is attached.

Screenshot:







Task 2.2A: Write a spoofing 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"

void send_raw_ip_packet(struct ipheader* ip) {
	struct sockaddr_in dest_info;
	int enable = 1;
	//Step1: Create a raw network socket
	int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

	//Step2: Set Socket option
	setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

	//Step3: Provide destination information
	dest_info.sin_family = AF_INET;
	dest_info.sin_addr = ip->iph_destip;

	//Step4: Send the packet out
	sendto(sock, ip, ntohs(ip->iph_len),0, (struct sockaddr *)&dest_info,
sizeof(dest_info));
	close(sock);
}

void main() {
	int mtu = 1500;
	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);
	memcpy(data, msg, data_len);

	udp->udp_sport=htons(9190);
```

```
	udp->udp_dport=htons(9090);
	udp->udp_ulen=htons(sizeof(struct udpheader) + data_len);
	udp->udp_sum=0;

	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_raw_ip_packet(ip);

}
```

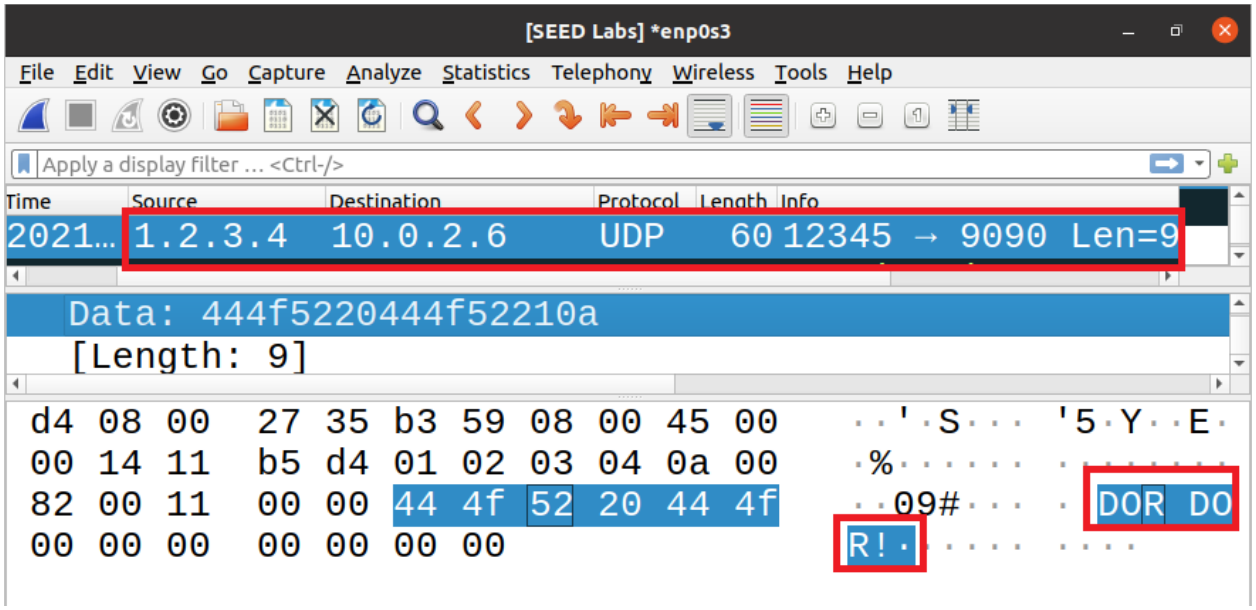
Explanation:

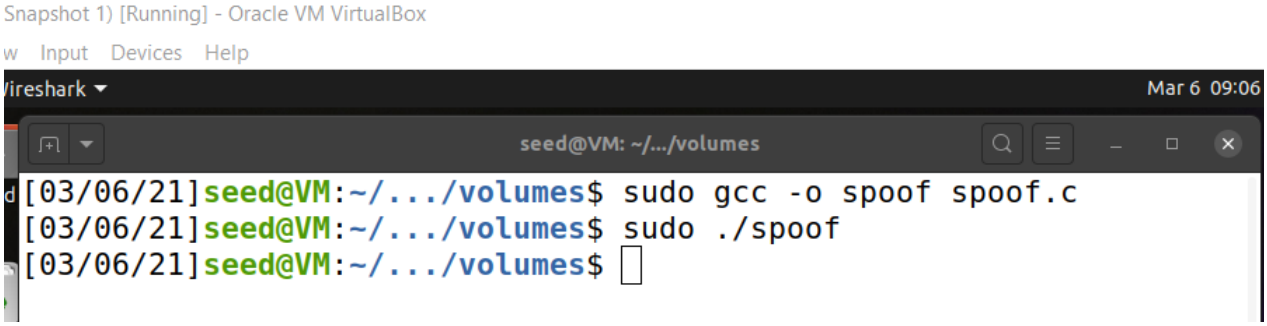
1. **About the code**: The program spoofing between another active VM (10.0.2.6) to 1.2.3.4 .
   I took the example from the task info and added to it a header that contains a UDP protocol and sent it to destination 10.0.2.6 (from - 1.2.3.4) but faking it.
   The program was created with a pcap library and modified the IP headers to use the source IP as 1.2.3.4 and destination as victim IP (10.0.2.6).
   When executed the packet was created with 1.2.3.4 and sent to the victim.
2. **About the question**: I worked with 2 VMs (my main VM and another one just to be alive for the task). I Created the spoof program using pcap library and when executed the spoofing machine (10.0.2.4) sent a packet to the victim machine (10.0.2.6) with a fake IP address (1.2.3.4).

Wireshark:

A recording capture file named '*spoofingc.pdf*' is attached.

Screenshot:


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>
Time Source Destination Protocol Length Info
2021... 1.2.3.4 10.0.2.6 UDP 60 12345 → 9090 Len=9
Data: 444f5220444f52210a
[Length: 9]
d4 08 00 27 35 b3 59 08 00 45 00
00 14 11 b5 d4 01 02 03 04 0a 00
82 00 11 00 00 44 4f 52 20 44 4f
00 00 00 00 00 00 00
DOR DO
R!.


Snapshot 1) [Running] - Oracle VM VirtualBox
w Input Devices Help
Wireshark
Mar 6 09:06
seed@VM: ~/.../volumes
[03/06/21]seed@VM:~/.../volumes$ sudo gcc -o spoof spoof.c
[03/06/21]seed@VM:~/.../volumes$ sudo ./spoof
[03/06/21]seed@VM:~/.../volumes$

Task 2.2B: Spoof an ICMP Echo Request.

Code:

spoof_icmp.c

```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:

1. **About the code**:

   Though the ICMP request originated from 10.0.2.4, the attacker created the packet with a spoofed IP (victim's).

   So, the remote server once received the ICMP packet, it responded back to the source IP that is present in the packet instead of sending to the attacker.

   Thus, the attacker spoofed an ICMP Echo request.

2. **About the question**:

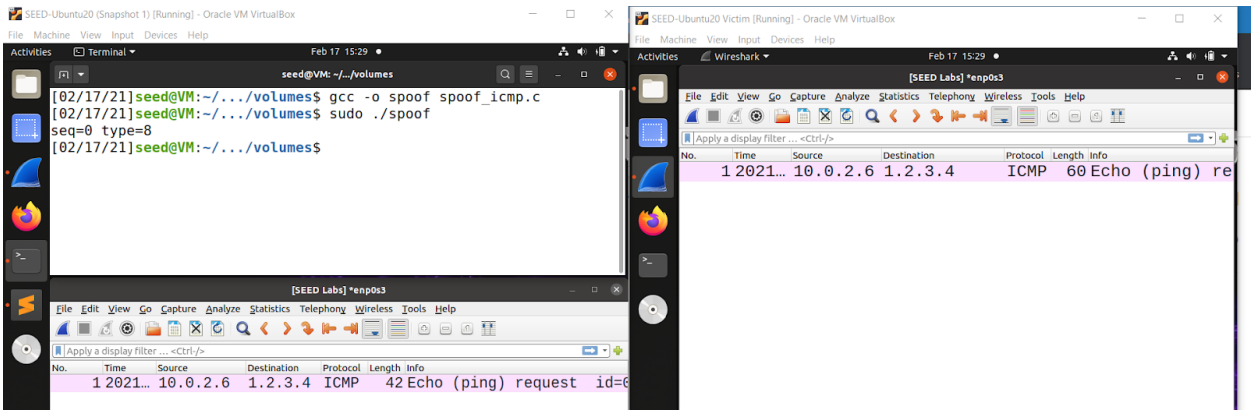
   Created a spoof ICMP request from attacker machine with source IP as victim (10.0.2.6) and sent to remote server (1.2.3.4);

   the remote server responded to ICMP request and sent it to the victim (10.0.2.6).

Wireshark:

A recording capture file named '*spoofing_icmpc.pdf*' is attached.

Screenshot:



Questions.

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. non-privileges 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:

sniffspoff.c

```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 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 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 live pcap session on NIC with 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;
}
```

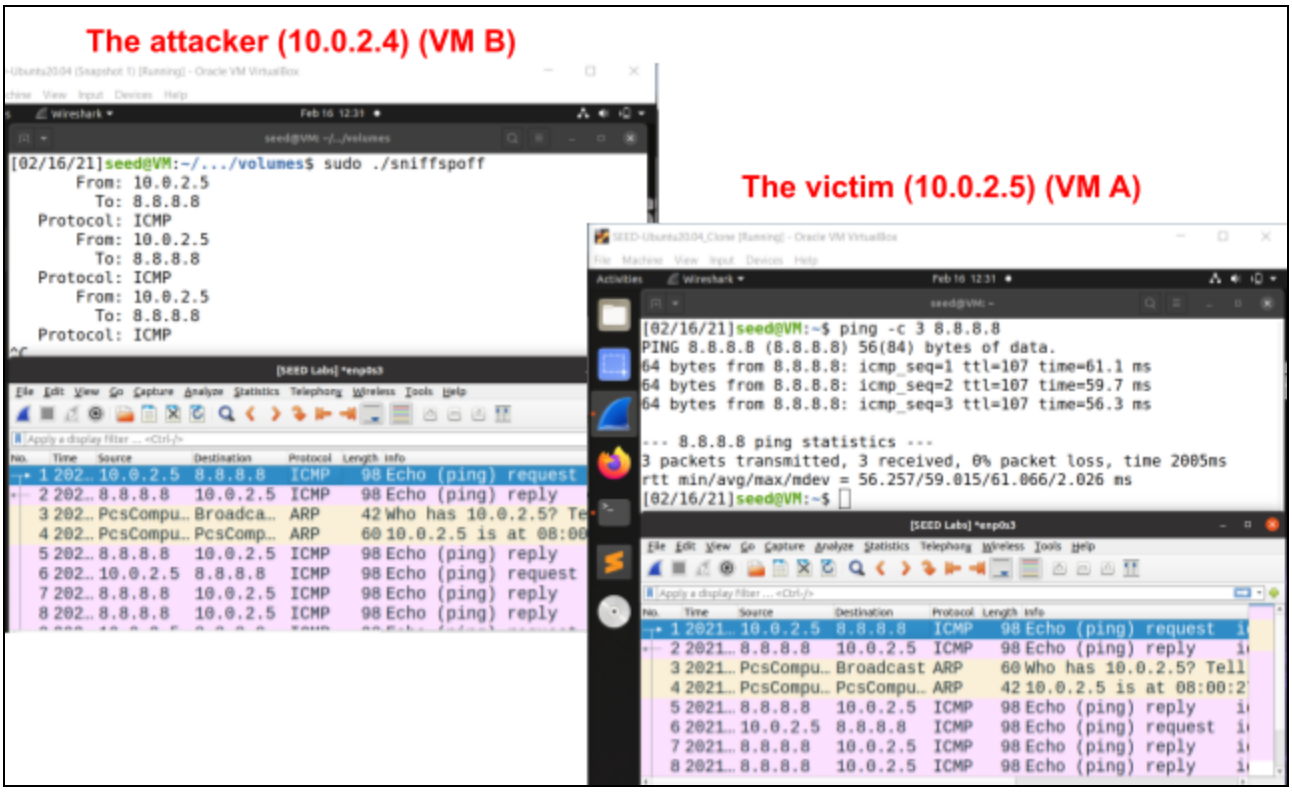
Explanation:

The attacker machine was in promiscuous mode and then when we executed our spoofing program, the NIC captured all the packets that reached and the program then processed in such a way, it modified the destination as source and source as destination. Once the packet is created it sends the packet out and the victim has received it. Thus, we spoofed the ICMP echo request.

Wireshark:

A recording capture file named ‘sniffspoff.pdf’ is attached.

Screenshot:


The attacker (10.0.2.4) (VM B)
[02/16/21]seed@VM:~/.../volumes$ sudo ./sniffspoff
From: 10.0.2.5
To: 8.8.8.8
Protocol: ICMP
From: 10.0.2.5
To: 8.8.8.8
Protocol: ICMP
From: 10.0.2.5
To: 8.8.8.8
Protocol: ICMP
The victim (10.0.2.5) (VM A)
[02/16/21]seed@VM:~$ ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=61.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=59.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=107 time=56.3 ms
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 56.257/59.015/61.066/2.026 ms
[02/16/21]seed@VM:~$