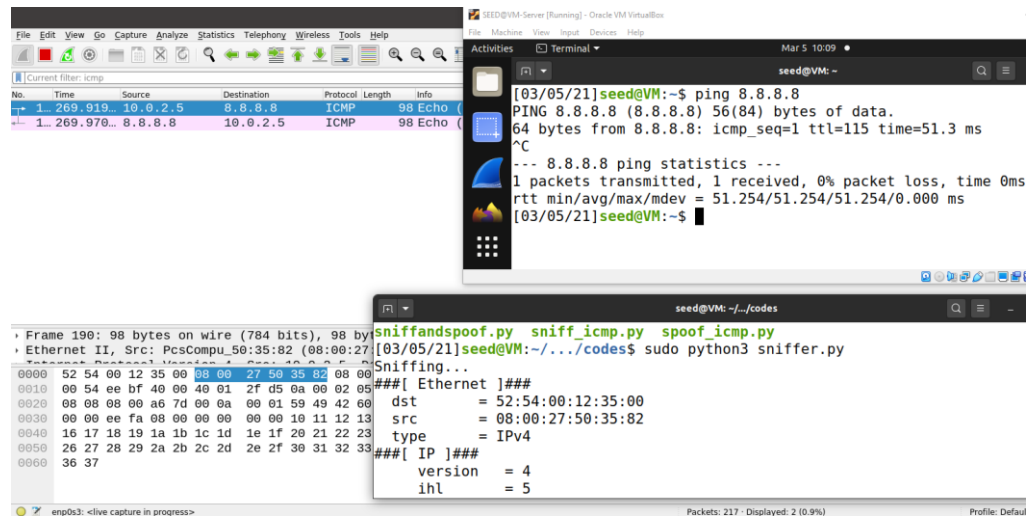


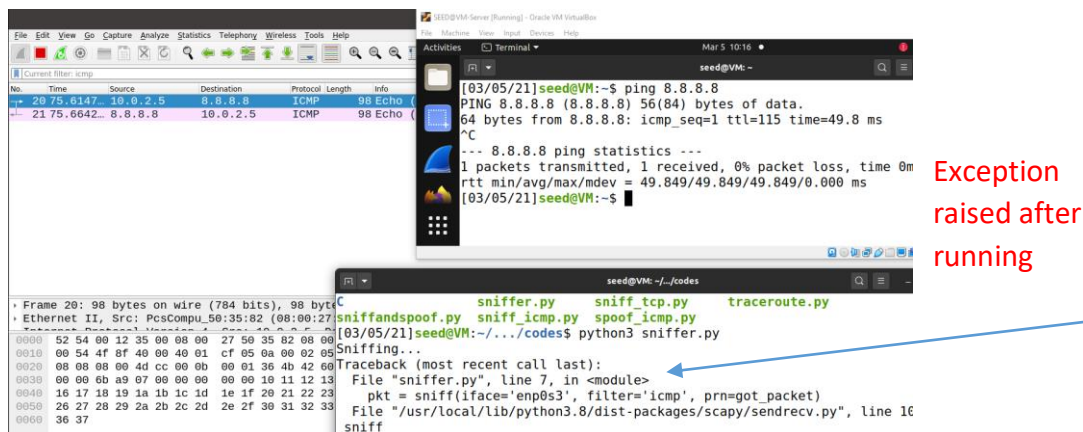
Sniffing and Spoofing

Task 1:

1A:



The image above describes running the python code 'sniffer.py', **with root privileges**, which sniffs packets on the machine's default network interface.



The image above describes running the same code (sniffer.py),

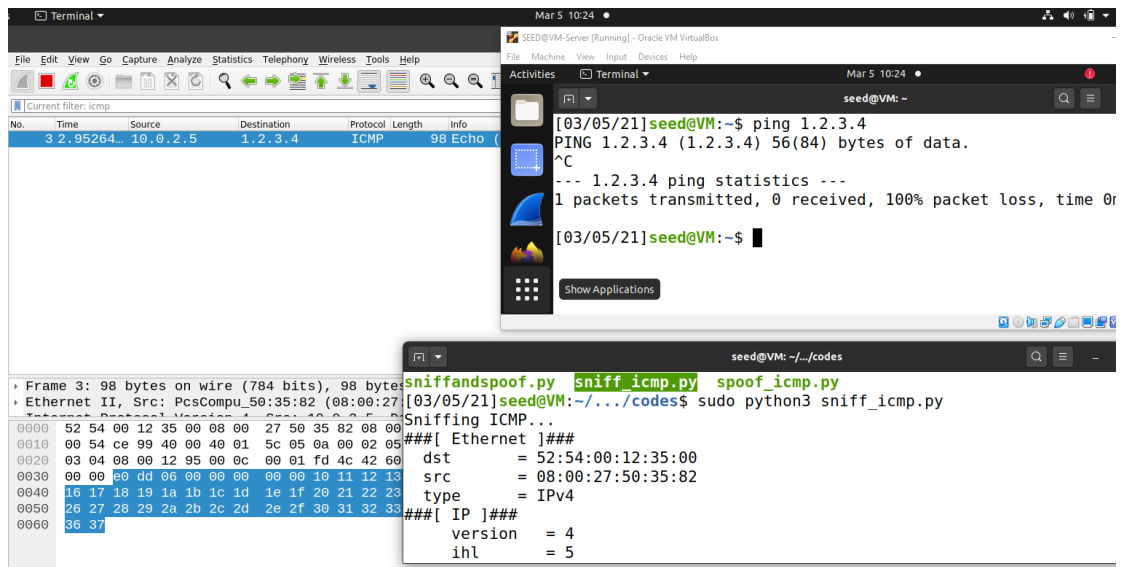
Without root privileges. Which will not succeed to run the sniffing program.

Sniffing/Spoofing programs require root privileges in order to receive packets from our network interface device.

1B: ICMP:

As you can see, our ICMP sniffing program successfully sniffed an ICMP-Echo request to 1.2.3.4, which got no response.

The message was sent from our server (10.0.2.5), and the code which was running on our attacker's VM terminal (10.0.2.4) printed out the packet.



The screenshot shows a network capture window with a table of ICMP packets. The first packet is an Echo request from 10.0.2.5 to 1.2.3.4. Below the table, the packet details for Frame 3 are shown, including the Ethernet II header and the IP header. The terminal window shows the command `ping 1.2.3.4` being executed, resulting in a 100% packet loss. The attacker's VM terminal shows the command `sudo python3 sniff_icmp.py` being executed, which outputs the details of the sniffed packet.

```
Current filter: icmp
No. Time Source Destination Protocol Length Info
3 2.95264... 10.0.2.5 1.2.3.4 ICMP 98 Echo (ping)

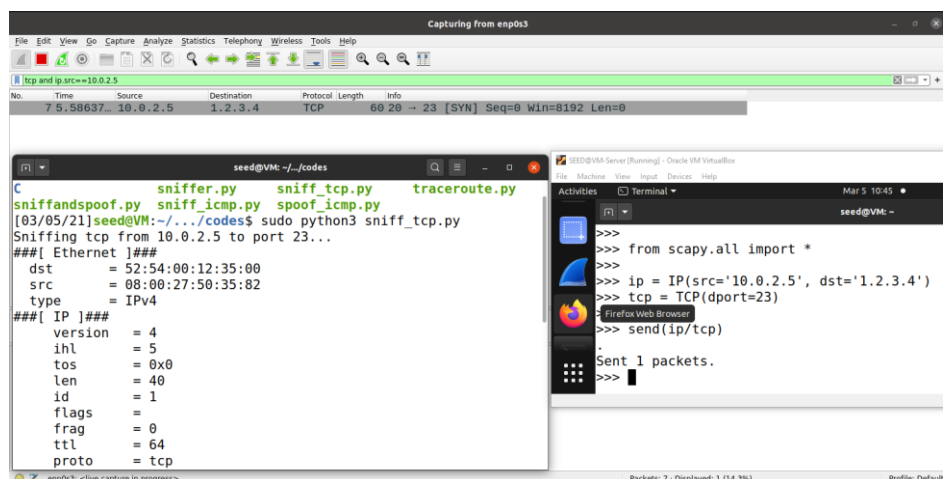
Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface veth-pair0
Ethernet II, Src: PcsCompu_50:35:82 (08:00:27:50:35:82), Dst: 02:00:00:00:00:00
Internet Protocol Version 4, Src: 10.0.2.5, Destination: 1.2.3.4
ICMP Echo (ping) request, id: 0, sequence: 0
0000 52 54 00 12 35 00 08 00 27 50 35 82 08 00
0010 00 54 ce 99 40 00 40 01 5c 05 0a 00 02 05
0020 03 04 08 00 12 95 00 0c 00 01 fd 4c 42 60
0030 00 00 00 dd 06 00 00 00 00 00 10 11 12 13
0040 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23
0050 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33
0060 36 37

sniffandspoofer.py sniff_icmp.py spoof_icmp.py
[03/05/21]seed@VM:~/..../codes$ sudo python3 sniff_icmp.py
Sniffing ICMP...
###[ Ethernet ]###
dst = 52:54:00:12:35:00
src = 08:00:27:50:35:82
type = IPv4
###[ IP ]###
version = 4
ihl = 5
```

TCP:

Our TCP program successfully sniffed 2 TCP packets that arrived at our Attacker's VM (10.0.2.4) at port 23.

The IP/TCP packet was forged using scapy and sent through our Server VM (Could have been sent to any other destination as well out-side of 10.0.2.0/24. **Port 23 is reserved for the Telnet protocol**, which is used to open connections unencrypted between machines.



The screenshot shows a network capture window with a table of TCP packets. The first packet is a SYN packet from 10.0.2.5 to 1.2.3.4 on port 23. Below the table, the packet details for Frame 7 are shown, including the Ethernet II header and the IP header. The terminal window shows the command `sudo python3 sniff_tcp.py` being executed, which outputs the details of the sniffed packet. The attacker's VM terminal shows the command `sudo python3 sniff_tcp.py` being executed, which outputs the details of the sniffed packet.

```
Capturing from enp0s3
No. Time Source Destination Protocol Length Info
7 5.58637... 10.0.2.5 1.2.3.4 TCP 60 20 -> 23 [SYN] Seq=0 Win=8192 Len=0

[03/05/21]seed@VM:~/..../codes$ sudo python3 sniff_tcp.py
Sniffing tcp from 10.0.2.5 to port 23...
###[ Ethernet ]###
dst = 52:54:00:12:35:00
src = 08:00:27:50:35:82
type = IPv4
###[ IP ]###
version = 4
ihl = 5
tos = 0x0
len = 40
id = 1
flags =
frag = 0
ttl = 64
proto = tcp
```

Specific subnets:

Our specific subnet sniffer successfully sniffs packets with IP destination: 10.11.12.0/24 (Any client in the subnet 10.11.12.X).

The screenshot displays a network traffic capture window (top) and a terminal window (bottom) running a subnet sniffer.

Network Traffic Capture (Top):

No.	Time	Source	Destination	Protocol	Length	Info
11	13.9734...	10.0.2.5	10.11.12.13	ICMP	98	Echo (ping) request id=0x0010, seq=1/256, ttl=64 ...
12	19.0094...	PcsCompu_50...	RealtekU_12...	ARP	60	Who has 10.0.2.1? Tell 10.0.2.5
13	19.0094...	RealtekU_12...	PcsCompu_50...	ARP	60	10.0.2.1 is at 52:54:00:12:35:00

Terminal Output (Bottom Left):

```
seed@VM: ~/../codes
[03/05/21]seed@VM:~/../codes$ sudo python3 sniff_subnet.py
Sniffing on subnet 10.11.12.0/24
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:50:35:82
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 19313
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x0d1h
```

Terminal Output (Bottom Right):

```
seed@VM: ~
[03/05/21]seed@VM:~$ ping 10.11.12.13
PING 10.11.12.13 (10.11.12.13) 56(84) bytes
^C
--- 10.11.12.13 ping statistics ---
1 packets transmitted, 0 received, 100% pac
[03/05/21]seed@VM:~$
```

1.2:

Spoofing ICMP:

The screenshot displays a network traffic capture window (top) and a terminal window (bottom) running an ICMP spoofing script.

Network Traffic Capture (Top):

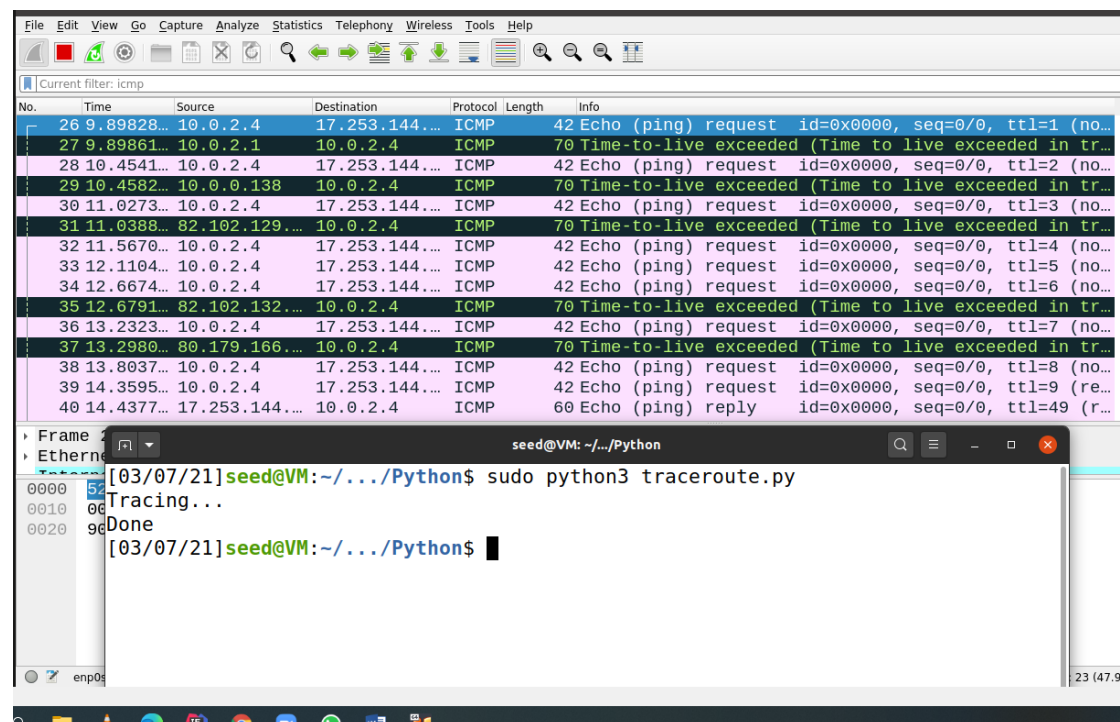
No.	Time	Source	Destination	Protocol	Length	Info
7	25.2342...	10.0.2.1	10.0.2.5	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (r...
8	25.2347...	10.0.2.5	10.0.2.1	ICMP	60	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (r...

Terminal Output (Bottom):

```
seed@VM: ~/../Python
[03/07/21]seed@VM:~/../Python$ ls
arp.py          sniff_arp.py    sniff_icmp.py    sniff_tcp.py    test.py
sniffandspooof.py sniffer.py      sniff_subnet.py  spoof_icmp.py   tracerou
[03/07/21]seed@VM:~/../Python$ sudo python3 spoof_icmp.py
Spoofed an icmp ping-req from 10.0.2.1 to 10.0.2.5!
[03/07/21]seed@VM:~/../Python$
```

1.3:

Traceroute:



The image above displays how we use tracerouting to establish the connection with 'apple.com'.

In our code we generate ICMP packets with TTL in range of 1-13, sent to the 'apple.com' website holding IP address '17.253.144.10'.

Each router that the packet pass through decreases the packets TTL by 1, the first router which returns the TTL exceeded response is our gateway (i.e. 10.0.2.1), which connects us with the outer network.

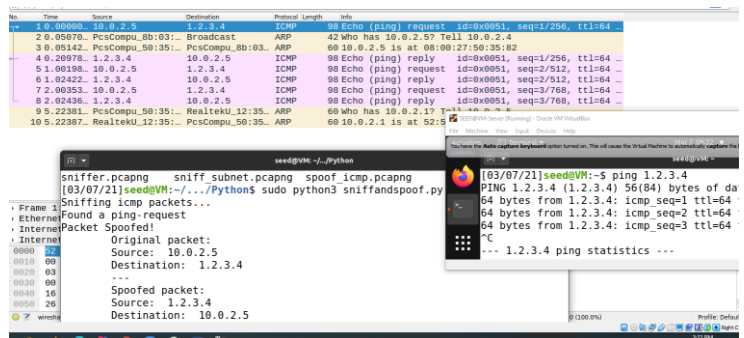
Eventually, we send out a packet with a sufficient amount of TTL to reach its final destination. That happens with a TTL sum of 5 for this case (a total amount of 5 routers), and setting TTL to 8 or higher, allows us to communicate and receive replies from the destination server.

1.4:

Sniff and Spoof:

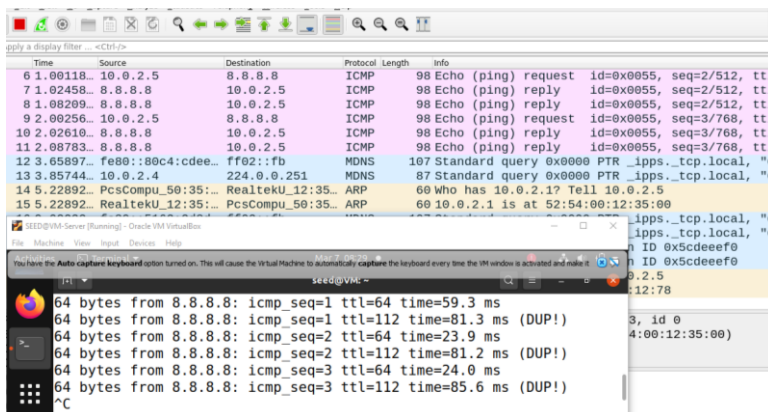
The images below demonstrate our sniffing and spoofing tool, which monitors our network interface (enp0s3) and awaits ICMP Echo-Request packets.

The first image below shows that our server (10.0.2.5) pings a Non-Existing host on the Internet, and successfully spoofs an ICMP reply packet, which keeps the PING program running and receiving responses, although the host 1.2.3.4 does not exist.



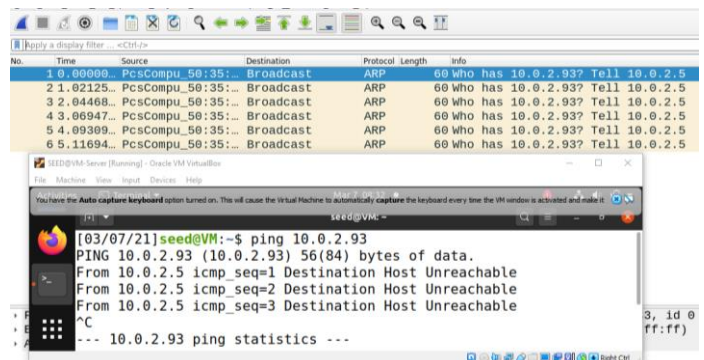
The second image here shows our server pinging an existing host on the internet (outside of our LAN), and successfully spoofing replies, although the remote server is also sending replies.

This causes a Duplicate ICMP reply packet for the pinging server.



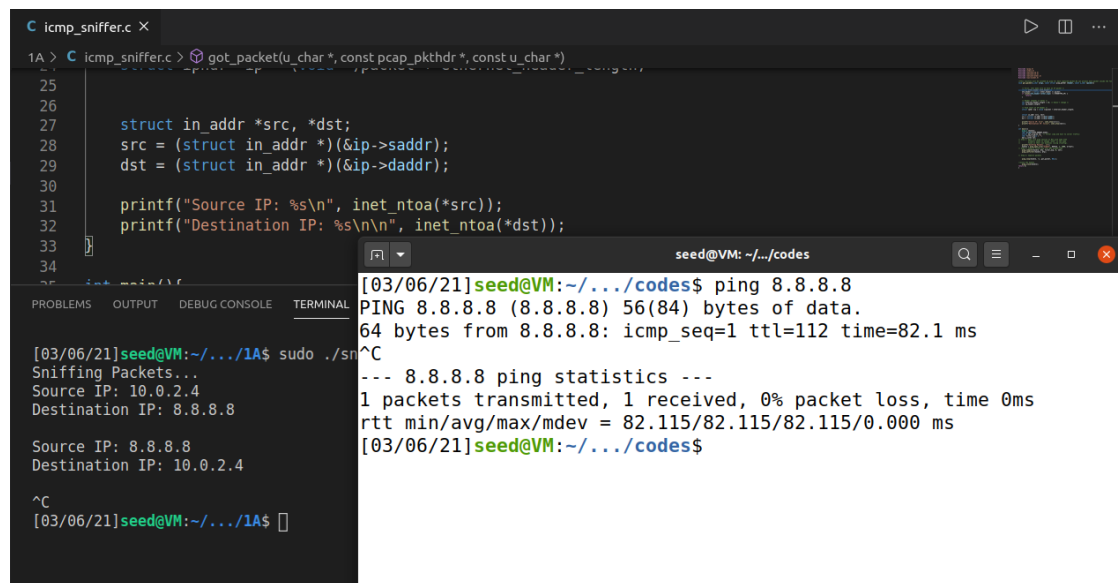
(Non-Existing host
on our LAN)

The third image is an example of pinging a non-existing host in our LAN resulting an ARP spam that asks: **Who has the non-existing host?**, which no one replies to. Thus no packet is sent and cannot be sniffed and then spoofed.



C:

2.1A:



The image shows a VS Code editor window with a file named `icmp_sniffer.c`. The code is a C program that uses libpcap to sniff network packets. It defines a `struct in_addr` and prints the source and destination IP addresses of each captured packet. Below the editor, there are two terminal windows. The left terminal shows the output of the program after running `sudo ./sniffer`, displaying two captured packets with their source and destination IP addresses. The right terminal shows the output of a `ping 8.8.8.8` command, including the ping statistics and the command prompt.

```
1A > C icmp_sniffer.c > got_packet(u_char *, const pcap_pkthdr *, const u_char *)
25
26
27 struct in_addr *src, *dst;
28 src = (struct in_addr *)(&ip->saddr);
29 dst = (struct in_addr *)(&ip->daddr);
30
31 printf("Source IP: %s\n", inet_ntoa(*src));
32 printf("Destination IP: %s\n", inet_ntoa(*dst));
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```
[03/06/21]seed@VM: ~/.../codes$ ping 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=112 time=82.1 ms
^C
--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 82.115/82.115/82.115/0.000 ms
[03/06/21]seed@VM: ~/.../codes$
```

```
[03/06/21]seed@VM: ~/.../1A$ sudo ./sniffer
Sniffing Packets...
Source IP: 10.0.2.4
Destination IP: 8.8.8.8

Source IP: 8.8.8.8
Destination IP: 10.0.2.4

^C
[03/06/21]seed@VM: ~/.../1A$
```

The image above shows how running 'sudo ./sniffer' starts sniffing all kinds of packets, as long as they have an IP header, with a source and destination.

In the terminal on the right we have pinged google (8.8.8.8) and got a reply. Both of the packets are displayed on the left terminal, with their source and destination printed.

Follow-up Questions:

1. A. The initial call for the function `pcap_open_live()` determines to which interface we are currently listening to, and capture mode (promiscuous, etc...).

- B. Later calls to:
`pcap_set_promisc();`
`pcap_setfilter();`
`pcap_compile();`
`etc...`

are used to compile the string to filter the packets we're interested in, and insert more sniffing options.

C. the function `pcap_loop()` allows up to continue live listening in a loop on the specified network interface.

D. the function `pcap_close()` releases all resources and close the program.

2. While running the program, we have to use the ROOT privileges, in-order to take the data from our network interface, which receives and sends network data (packets).

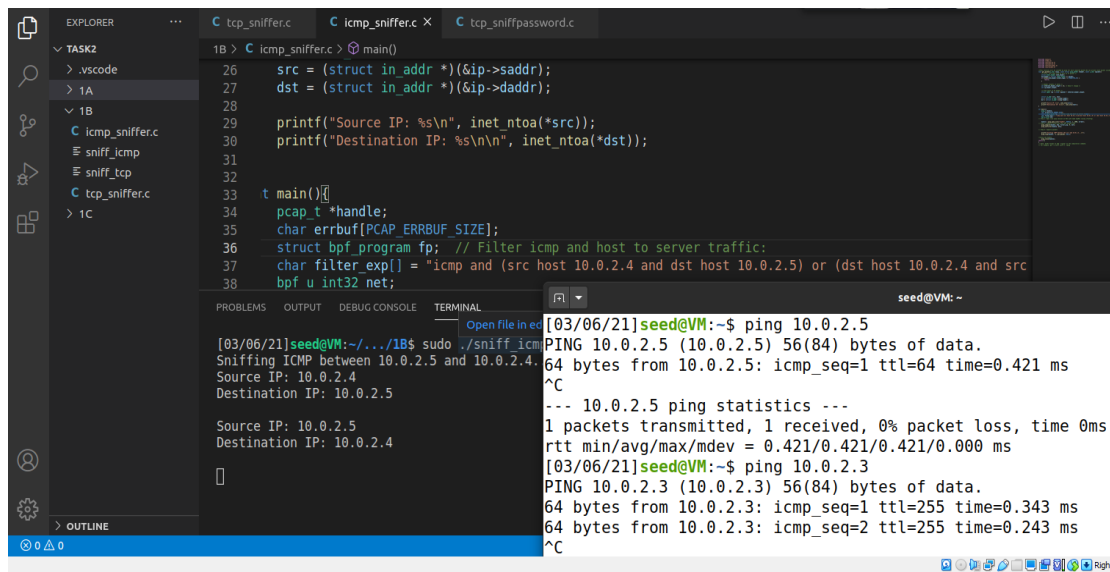
Running without the ROOT privileges will deny access to the network interface and will not allow creating a Raw Socket.

The program will fail executing the **pcap_compile()** function (Because it is called before creating a socket), because the network interface passed to `pcap_open_live()` needs to be compiled before running, and notes that root privileges were not given.

3. We have tried **turning off** the promiscuous mode in-order to receive only packets that are destined to our Attacker's VM (10.0.2.4), and **turning on** the promiscuous mode in-order to **receive all packets** that are passing through our network interface.

This could be described by spoofing an ICMP packet to another machine from a spoofed src address(Ex: 10.0.2.3 to 10.0.2.5), and if the promisc mode is **OFF**, then the packets should not show up. If promisc mode is **ON**, then the packets should show up.

2.1B:



```
1B > C icmp_sniffer.c > main()
26   src = (struct in_addr *)(&ip->saddr);
27   dst = (struct in_addr *)(&ip->daddr);
28
29   printf("Source IP: %s\n", inet_ntoa(*src));
30   printf("Destination IP: %s\n\n", inet_ntoa(*dst));
31
32
33   t main()
34   pcap_t *handle;
35   char errbuf[PCAP_ERRBUF_SIZE];
36   struct bpf_program fp; // Filter icmp and host to server traffic:
37   char filter_exp[] = "icmp and (src host 10.0.2.4 and dst host 10.0.2.5) or (dst host 10.0.2.4 and src
38   bpf_u_int32 net;

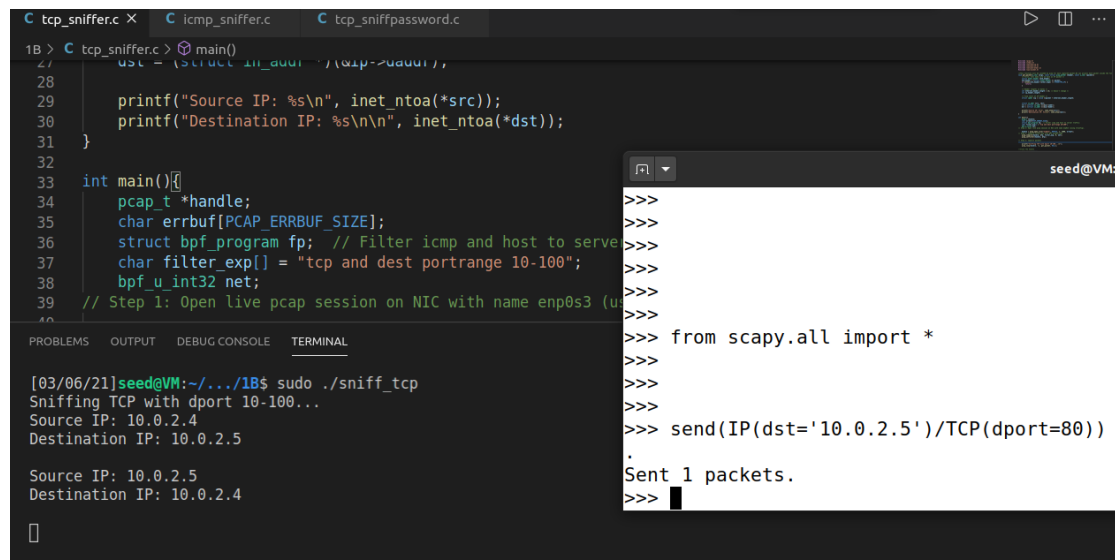
[03/06/21]seed@VM:~/.../1B$ sudo ./sniff icmp
Sniffing ICMP between 10.0.2.5 and 10.0.2.4
Source IP: 10.0.2.4
Destination IP: 10.0.2.5

Source IP: 10.0.2.5
Destination IP: 10.0.2.4

[03/06/21]seed@VM:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data.
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=0.421 ms
^C
--- 10.0.2.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.421/0.421/0.421/0.000 ms
[03/06/21]seed@VM:~$ ping 10.0.2.3
PING 10.0.2.3 (10.0.2.3) 56(84) bytes of data.
64 bytes from 10.0.2.3: icmp_seq=1 ttl=255 time=0.343 ms
64 bytes from 10.0.2.3: icmp_seq=2 ttl=255 time=0.243 ms
^C
```

In the above image you can see our sniffer program 'icmp_sniffer.c' captures and prints packets that are from 10.0.2.4 to 10.0.2.5 and the other way around, and are of type ICMP.

Note: that the second ping is from 10.0.2.4 to 10.0.2.3, and our sniffing program did not capture these packets!!! (which means the filtering works!)



The image shows a C code editor with three files: `tcp_sniffer.c`, `icmp_sniffer.c`, and `tcp_sniffpassword.c`. The `tcp_sniffer.c` file contains the following code:

```
1B > C tcp_sniffer.c > main()
27 struct in_addr src, dst;
28
29 printf("Source IP: %s\n", inet_ntoa(*src));
30 printf("Destination IP: %s\n\n", inet_ntoa(*dst));
31 }
32
33 int main(){
34     pcap_t *handle;
35     char errbuf[PCAP_ERRBUF_SIZE];
36     struct bpf_program fp; // Filter icmp and host to server
37     char filter_exp[] = "tcp and dest portrange 10-100";
38     bpf_u_int32 net;
39     // Step 1: Open live pcap session on NIC with name enp0s3 (u
40
```

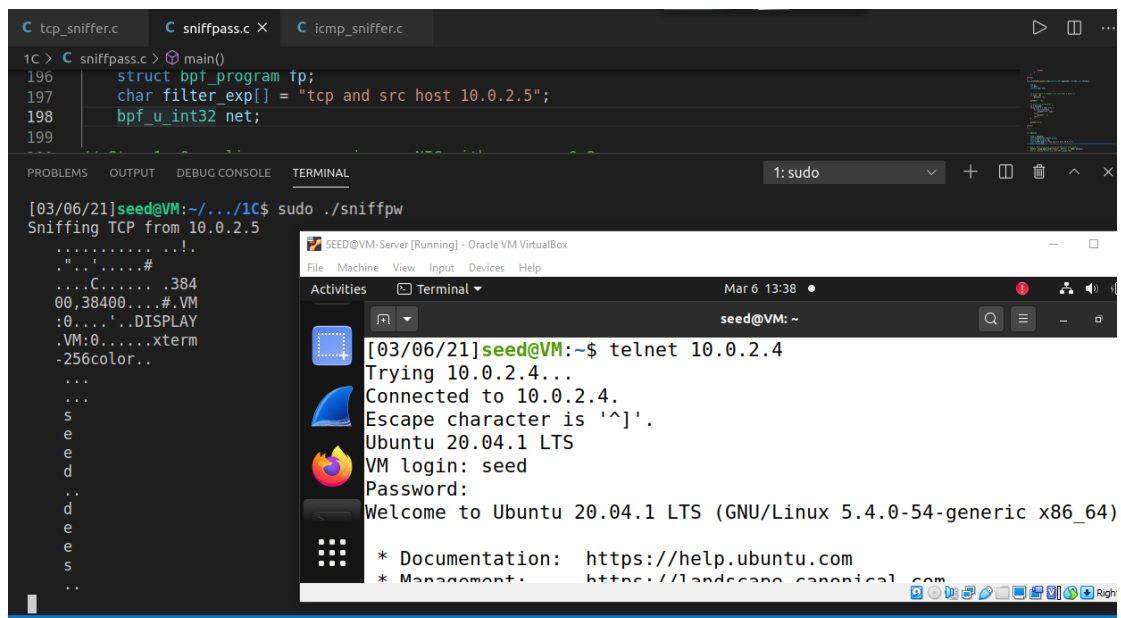
The terminal output shows the execution of the program:

```
[03/06/21]seed@VM:~/.../1B$ sudo ./sniff_tcp
Sniffing TCP with dport 10-100...
Source IP: 10.0.2.4
Destination IP: 10.0.2.5

Source IP: 10.0.2.5
Destination IP: 10.0.2.4
```

In the above image you can see our sniffer program 'tcp_sniffer.c' successfully captures and prints packets that have a destination port in range 10-100.

2.1C:



The image shows a terminal window with the following output:

```
[03/06/21]seed@VM:~/.../1C$ sudo ./sniffpw
Sniffing TCP from 10.0.2.5
.....!..
..C.....#
00,38400...#.VM
:0....'.DISPLAY
.VM:0.....xterm
-256color..
...
s
e
d
..
d
e
e
s
..
```

The terminal also shows a telnet connection to 10.0.2.4:

```
[03/06/21]seed@VM:~$ telnet 10.0.2.4
Trying 10.0.2.4...
Connected to 10.0.2.4.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

The image above shows how we have successfully sniffed out all the TCP traffic from our server VM (10.0.2.5) to our Attacker's VM (10.0.2.4) through a telnet (port 23) connection (**Unencrypted**).

On the left terminal, you can see the stream of characters that match the VM Login field('seed'), and after 1 line, the stream that matches the password field('dees').

40	14.1661...	10.0.2.4	10.0.2.5	TEL...	76 Telnet Data ...
41	14.1665...	10.0.2.5	10.0.2.4	TCP	66 45342 → 23 [ACK] Seq=109 Ack=88 Win=64256 Len=0 TS...
Frame 40: 76 bytes on wire (608 bits), 76 bytes captured (608 bits) on interface enp0s3, id 0					
Ethernet II, Src: PcsCompu 8b:03:4a (08:00:27:8b:03:4a), Dst: PcsCompu 50:35:82 (08:00:27:50:35:82)					
00	08 00 27 50 35 82 08 00 27 8b 03 4a 08 00 45 10	..P5... 'J..E.			
10	00 3e 3a 1a 40 00 40 06 e8 87 0a 00 02 04 0a 00	->: @. @.			
20	02 05 00 17 b1 1e 96 0b 66 96 3b 1c 9f 5e 80 18 f.; ^..			
30	01 fd 18 39 00 00 01 01 08 0a 90 03 6e 89 1e 05	...9... ..n...			
40	47 41 56 4d 20 6c 6f 67 69 6e 3a 20	GA VM log in:			

In the above image you can see host VM (10.0.2.4) requires VM Login info from the other connection (10.0.2.5).

```

.....!. "!. '.....#.....#..!.....!. ".....#.....!.....C.
.....38400,38400.....#..VM:0.....'..DISPLAY.VM:0.....xterm-256color.....Ubuntu 20.04.1 LTS
VM login: sseeedd
Password: dees
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

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

64 updates can be installed immediately.
64 of these updates are security updates.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update
Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Sat Mar  6 13:37:30 EST 2021 from 10.0.2.5 on pts/5
.]0;seed@VM: ~. [03/06/21]. [01;32mseed@VM. [00m:.. [01;34m~. [00m$

```

In the above image you can see a Wireshark tool, called TCP -> Stream follower, which Follows the unencrypted TCP data stream, and extracts the Login and password details.

2.2A:

The screenshot shows a terminal window with the execution of the `spoof.c` program. The program defines a source IP of `8.8.8.8` and a destination IP of `10.0.2.5`. The terminal output shows the compilation and execution of the program. To the right, Wireshark is capturing traffic on `enp0s3`. It shows two ICMP Echo (ping) requests: one from `8.8.8.8` to `10.0.2.5` (No. 32) and a reply from `10.0.2.5` to `8.8.8.8` (No. 33). Below the packet list, the raw data of the second packet is shown in hexadecimal and ASCII.

```

2 > C spoof.c > main()
11 #include <stdio.h>
12 #include <sys/time.h>
13
14 unsigned short calculate_checksum(unsigned short *addr, int len)
15 {
16     #define IP4_HDRLEN 20
17     #define ICMP_HDRLEN 8
18     #define SOURCE_IP "8.8.8.8"
19     #define DESTINATION_IP "10.0.2.5"
20
21     int main ()
22     {
23         struct ip iphdr; // IPv4 header
24
25         [03/06/21]seed@VM:~/.../2$ gcc -o spoof spoof.c
26         [03/06/21]seed@VM:~/.../2$ sudo ./spoof
27         [03/06/21]seed@VM:~/.../2$

```

Wireshark Packet List:

No.	Time	Source	Destination	Protocol	Length	Info
32	11.0206...	8.8.8.8	10.0.2.5	ICMP	61	Echo (ping) request
33	11.0209...	10.0.2.5	8.8.8.8	ICMP	61	Echo (ping) reply

Frame 33: 61 bytes on wire (488 bits) - 61 bytes captured (488 bits) on enp0s3

```

0000  08 00 27 50 35 82 08 00 27 8b 03 4a 08 00 45 00  ...P5...
0010  00 2f 50 fe 00 00 32 01 1b bc 08 08 08 08 0a 00  .../P...2...
0020  02 05 08 00 ae 36 12 00 00 00 54 68 69 73 20 69  ....6...

```

In the above image, you can see how running the `spoof.c` program successfully spoofs and ICMP packet.

Successful ICMP spoofing causes the destination host to reply (if it is alive) to the spoofed source (Result achieved).

2.2B:

The screenshot shows a terminal window with the execution of the `spoof_icmp.c` program. The program defines a source IP of `10.0.2.1` and a destination IP of `10.0.2.5`. The terminal output shows the compilation and execution of the program. To the right, Wireshark is capturing traffic on `enp0s3`. It shows two ICMP Echo (ping) requests: one from `10.0.2.1` to `10.0.2.5` (No. 1) and a reply from `10.0.2.5` to `10.0.2.1` (No. 2).

```

2 > C spoof_icmp.c > main()
11 #include <stdio.h>
12 #include <sys/time.h>
13
14 unsigned short calculate_checksum(unsigned short *addr, int len)
15 {
16     #define IP4_HDRLEN 20
17     #define ICMP_HDRLEN 8
18     #define SOURCE_IP "10.0.2.1"
19     #define DESTINATION_IP "10.0.2.5"
20
21     int main ()
22     {
23         struct ip iphdr; // IPv4 header
24         struct icmp icmphdr; // ICMP-header
25
26         [03/06/21]seed@VM:~/.../2$ sudo ./spoof_icmp
27         [03/06/21]seed@VM:~/.../2$

```

Wireshark Packet List:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.00000...	10.0.2.1	10.0.2.5	ICMP	61	Echo (ping) request
2	0.00077...	10.0.2.5	10.0.2.1	ICMP	61	Echo (ping) reply

In the above image, you can see how running the `spoof_icmp.c` program successfully spoofs an ICMP packet.

Successful ICMP spoofing causes the destination host to reply (if it is alive) to the spoofed source (Result achieved).

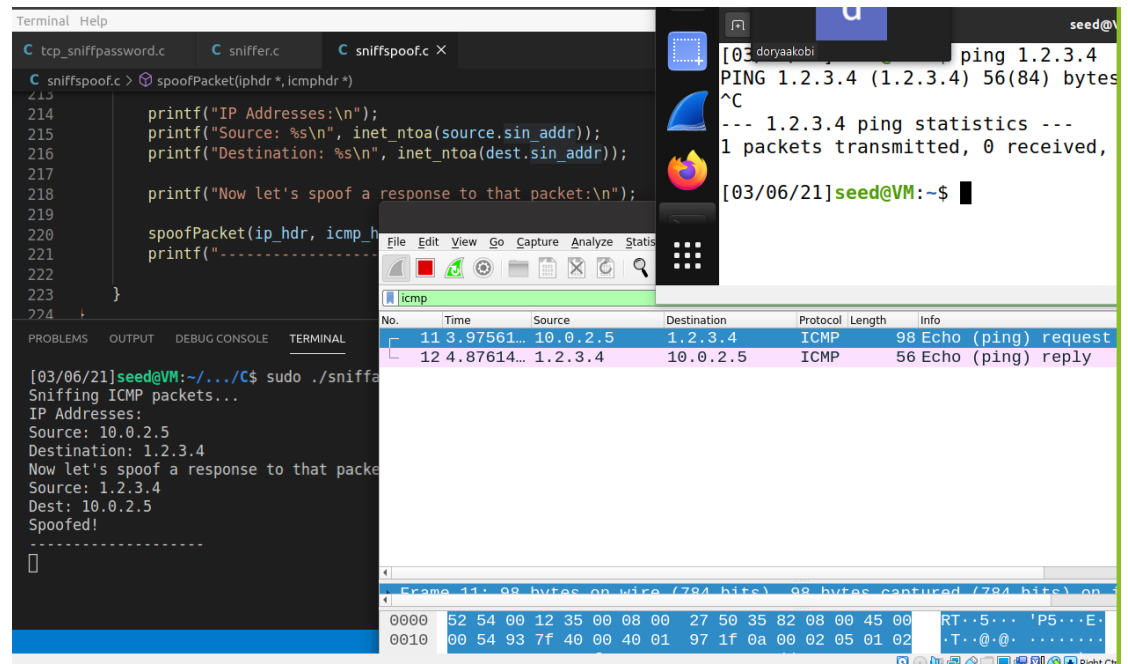
Follow-up Questions:

4. Yes, we can set the IP packet length field to an arbitrary value, causing the payload to become a chunk of zeroes (0's) because the memory location of the payload is lost.
5. No, we do not have to calculate the checksum field for **IP** headers, because the raw socket functionality does that for us.
6. You have to provide root privileges in order to use Raw Sockets, because raw sockets enable us to set all the fields of a packet as we would like, and disable the OS's Kernel from intervening with adding the headers it-self. By doing so, it is possible to construct our own SPOOFED packet and send it. If no root privileges are required -> then it was very easy to forge packets and send them around the world and create kind of a chaos.

Our program still fails in pcap_compile() function, because we are compiling and running pcap before creating a socket.

If we were to create a **Raw Socket** before compiling pcap, the program will fail in the socket creation (sock = socket(...)).

2.3 Sniff and Spoof:



In this sniff and spoof program, we have accomplished spoofing sniffed ICMP Echo-request packets, with all the needed packet fields, except for the payload (Which probably causes the ping program to not receive a reply).

In this task we were able to ping all required addresses (i.e. Existing on the LAN, Non-Existing on the Internet and Existing on the Internet Addresses), only in the Wireshark packet trace.

If we were to copy the packets payload successfully into our new **Spoofed** packet, then the PING program should work the same as 1.4(python sniff and spoof).

In the images above and below you can see in our Wireshark trace, that we have spoofed 1 Echo-Reply for 1 Echo-Request packet.

8	10.2193...	10.0.2.5	8.8.8.8	ICMP	98 Echo (ping) request	id=0x0058, seq=1/256, ttl=64 ...
9	10.3098...	8.8.8.8	10.0.2.5	ICMP	98 Echo (ping) reply	id=0x0058, seq=1/256, ttl=112...
10	10.9490...	8.8.8.8	10.0.2.5	ICMP	56 Echo (ping) reply	id=0x0058, seq=1/256, ttl=100...
11	11.2214...	10.0.2.5	8.8.8.8	ICMP	98 Echo (ping) request	id=0x0058, seq=2/512, ttl=64 ...
12	11.3018...	8.8.8.8	10.0.2.5	ICMP	98 Echo (ping) reply	id=0x0058, seq=2/512, ttl=112...
13	11.9724...	8.8.8.8	10.0.2.5	ICMP	56 Echo (ping) reply	id=0x0058, seq=2/512, ttl=100...
14	12.2234...	10.0.2.5	8.8.8.8	ICMP	98 Echo (ping) request	id=0x0058, seq=3/768, ttl=64 ...
15	12.3080...	8.8.8.8	10.0.2.5	ICMP	98 Echo (ping) reply	id=0x0058, seq=3/768, ttl=112...
16	12.9964...	8.8.8.8	10.0.2.5	ICMP	56 Echo (ping) reply	id=0x0058, seq=3/768, ttl=100...
17	13.2254...	10.0.2.5	8.8.8.8	ICMP	98 Echo (ping) request	id=0x0058, seq=4/1024, ttl=64...
18	13.3067...	8.8.8.8	10.0.2.5	ICMP	98 Echo (ping) reply	id=0x0058, seq=4/1024, ttl=11...
19	14.0213...	8.8.8.8	10.0.2.5	ICMP	56 Echo (ping) reply	id=0x0058, seq=4/1024, ttl=100...
20	15.2316...	PcsCompu 50:35...	RealtekU 12:35...	ARP	60 Who has 10.0.2.1? Tell 10.0.2.5	