Eric Wong: 501174088
Rohan Manoharan: 501189408
Thylane Rossi: 501340990

Lab 6

Task 1.1A

> The command

ifconfig

is used to get the interface name, which would be used in program *sniffer.py*. Searching for the IP of the VM, 10.9.0.1, the corresponding interface name would be *br-15511391c179*

```
[11/19/24]seed@VM:~/.../Labsetup$ ifconfig
br-15511391c179: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:5c:2d:88:cb  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

> *sniffer.py* is written with the following code (similar to the sample code given, with a substituted interface name)

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3 def print_pkt(pkt):
4         pkt.show()
5
6 pkt = sniff(iface='br-15511391c179', filter='icmp', prn=print_pkt)
```

> The following commands are used to make the program executable, then having the program run with root privileges.

chmod a+x sniffer.py

sudo ./sniffer.py

As per the screenshot below, the program does not print anything, nor does it ever complete (eventually interrupted whilst waiting several minutes for a result). This is assumedly the program communicating, waiting for a packet to sniff.

```
[11/19/24]seed@VM:~/.../Labsetup$ chmod a+x sniffer.py
[11/19/24]seed@VM:~/.../Labsetup$ sudo ./sniffer.py
```

> Conversely, when switching to the “seed” account and executing the program without root permissions, it results in a permission error. The commands used were the following:

su seed (inputted password being “dees”)

./sniffer.py

With the following being the outputted result

```
^C[11/19/24]seed@VM:~/.../Labsetup$ su seed
Password:
[11/19/24]seed@VM:~/.../Labsetup$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 6, in <module>
    pkt = sniff(iface='br-15511391c179', filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in
 sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in
_run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e))  # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

Task 1.1B

The following filters were applied to *sniffer.py*

> Capturing only the ICMP packet: *icmp*

```
6 pkt = sniff(iface='br-15511391c179', filter='icmp', prn=print_pkt)
```

> Capturing any TCP packet from a particular IP and destination port number 23: *tcp dst port 23 and src host 10.9.0.1*

```
6 pkt = sniff(iface='br-15511391c179', filter='tcp dst port 23 and src host
  10.9.0.1', prn=print_pkt)
```

> Capture packets comes from subnet 128.230.0.0/16: *net 128.230.0.0/16*

```
6 pkt = sniff(iface='br-15511391c179', filter='net 128.230.0.0/16', prn=print_pkt)
```

> As with the results of attempting to execute the program in part 1.1A, the desired result seemed to not have been outputted, resulting in an empty process that needs to be interrupted manually.

Task 1.2

> The following code was used for the spoofing program, *spoofer.py*

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4 a = IP(src='10.9.0.1')
5 b = ICMP()
6 p = a/b
7 send(p)
```

> Upon executing the program, it outputs the following, matching the desired output from the instructions

```
[11/20/24]seed@VM:~/.../Labsetup$ sudo ./spoofer.py
.
Sent 1 packets.
```

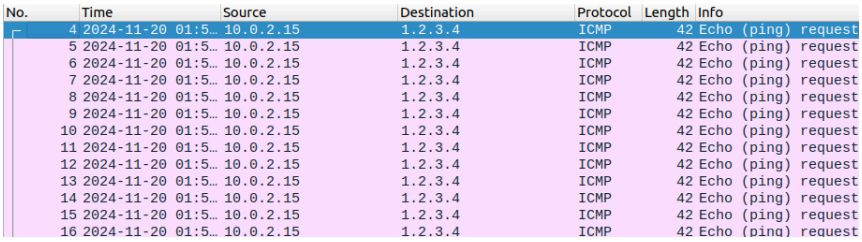
Task 1.3

> The following code was used in the program, *traceroute.py*. It uses the sample code given and adds a loop, incrementing the TTL field with each loop.

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4 a = IP()
5 a.dst = '1.2.3.4'
6 b = ICMP()
7
8 for i in range(1,65):
9         a.ttl = i
10        send(a/b)
```

> Unfortunately, no ICMP error message seemed to have been caught, outputting the following message multiple times, similar to task 1.2. Upon referencing wireshark as well, there seemed to have been no caught error messages there either.

```
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```



No.	Time	Source	Destination	Protocol	Length	Info
4	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
5	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
6	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
7	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
8	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
9	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
10	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
11	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
12	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
13	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
14	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
15	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request
16	2024-11-20 01:5…	10.0.2.15	1.2.3.4	ICMP	42	Echo (ping) request

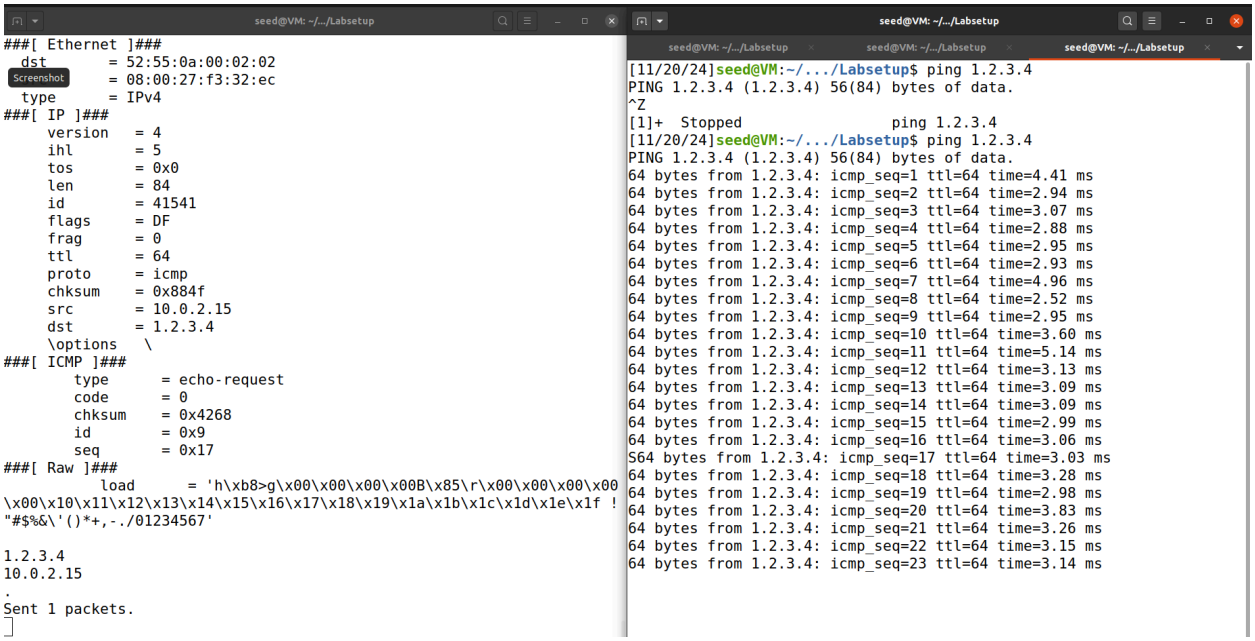
Task 1.4

> In order to complete this task we need to create a new file that will, as the title suggests, sniff and spoof.

>Sniff_Snoof.py

```python
from scapy.all import *

def spoof_pkt(pkt):
    print(pkt[IP].src)
    print(pkt[IP].dst)
    pkt.show()

    a = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
    b = ICMP(type=0, seq=pkt[ICMP].seq, id=pkt[ICMP].id)
    new_pkt = a/b/pkt[Raw]

    print(  new_pkt[IP].src    )
    print(  new_pkt[IP].dst    )

    send(new_pkt)

pkt = sniff(filter="icmp", prn=spoof_pkt)
~
```

> Ping 1.2.3.4



> Despite 1.2.3.4 being a “non-existing host on the internet”, we are still able to get replies from the IP, because of there being a mapping for the address in the routing table. We can f ind this information when we run

> *ip route get 1.2.3.4*

> ping 8.8.4.4

```
###[ Ethernet ]###
  dst       = 52:55:0a:00:02:02
  src       = 08:00:27:f3:32:ec
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x628e
     src       = 10.0.2.15
     dst       = 8.8.4.4
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x2a0b
        id        = 0xa
        seq       = 0x12
###[ Raw ]###
           load      = 'j\xc8>g\x00\x00\x00\x00\x00l\xd6\x01\x00\x00\x00\x
00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'

8.8.4.4
10.0.2.15
.
Sent 1 packets.
s
```

```
64 bytes from 8.8.4.4: icmp_seq=9 ttl=64 time=3.16 ms
64 bytes from 8.8.4.4: icmp_seq=9 ttl=255 time=21.9 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=9 ttl=64 time=33.4 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=10 ttl=64 time=3.47 ms
64 bytes from 8.8.4.4: icmp_seq=10 ttl=255 time=31.6 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=10 ttl=64 time=41.6 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=11 ttl=64 time=2.85 ms
64 bytes from 8.8.4.4: icmp_seq=11 ttl=255 time=29.7 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=11 ttl=64 time=39.9 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=12 ttl=64 time=3.28 ms
64 bytes from 8.8.4.4: icmp_seq=12 ttl=255 time=28.8 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=12 ttl=64 time=38.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=13 ttl=64 time=3.04 ms
64 bytes from 8.8.4.4: icmp_seq=13 ttl=255 time=20.5 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=13 ttl=64 time=31.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=14 ttl=64 time=3.01 ms
64 bytes from 8.8.4.4: icmp_seq=14 ttl=255 time=27.6 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=14 ttl=64 time=38.6 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=15 ttl=64 time=4.22 ms
64 bytes from 8.8.4.4: icmp_seq=15 ttl=255 time=18.5 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=15 ttl=64 time=25.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=16 ttl=64 time=3.12 ms
64 bytes from 8.8.4.4: icmp_seq=16 ttl=255 time=22.1 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=16 ttl=64 time=36.0 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=17 ttl=64 time=2.16 ms
64 bytes from 8.8.4.4: icmp_seq=17 ttl=255 time=27.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=17 ttl=64 time=34.5 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=18 ttl=64 time=2.29 ms
64 bytes from 8.8.4.4: icmp_seq=18 ttl=255 time=30.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=18 ttl=64 time=39.6 ms (DUP!)
^Z
[3]+  Stopped                 ping 8.8.4.4
[11/21/24]seed@VM:~/.../Labsetup$
```

> Here we are getting duplicates, denoted with (DUP!), since we are getting replies from the real machine, and from sniff_spoof.py as well.

>ping 10.9.0.99

```
###[ Ethernet ]###
  dst       = 52:55:0a:00:02:02
  src       = 08:00:27:f3:32:ec
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x628e
     src       = 10.0.2.15
     dst       = 8.8.4.4
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x2a0b
        id        = 0xa
        seq       = 0x12
###[ Raw ]###
           load      = 'j\xc8>g\x00\x00\x00\x00\x00l\xd6\x01\x00\x00\x00\x
00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
 !"#$%&\'()*+,-./01234567'

8.8.4.4
10.0.2.15
.
Sent 1 packets.
```

```
64 bytes from 8.8.4.4: icmp_seq=17 ttl=64 time=34.5 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=18 ttl=64 time=2.29 ms
64 bytes from 8.8.4.4: icmp_seq=18 ttl=255 time=30.3 ms (DUP!)
64 bytes from 8.8.4.4: icmp_seq=18 ttl=64 time=39.6 ms (DUP!)
^Z
[3]+  Stopped                 ping 8.8.4.4
[11/21/24]seed@VM:~/.../Labsetup$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable
From 10.9.0.1 icmp_seq=4 Destination Host Unreachable
From 10.9.0.1 icmp_seq=5 Destination Host Unreachable
From 10.9.0.1 icmp_seq=6 Destination Host Unreachable
From 10.9.0.1 icmp_seq=7 Destination Host Unreachable
From 10.9.0.1 icmp_seq=8 Destination Host Unreachable
From 10.9.0.1 icmp_seq=9 Destination Host Unreachable
From 10.9.0.1 icmp_seq=10 Destination Host Unreachable
From 10.9.0.1 icmp_seq=11 Destination Host Unreachable
From 10.9.0.1 icmp_seq=12 Destination Host Unreachable
From 10.9.0.1 icmp_seq=13 Destination Host Unreachable
From 10.9.0.1 icmp_seq=14 Destination Host Unreachable
From 10.9.0.1 icmp_seq=15 Destination Host Unreachable
From 10.9.0.1 icmp_seq=16 Destination Host Unreachable
From 10.9.0.1 icmp_seq=17 Destination Host Unreachable
From 10.9.0.1 icmp_seq=18 Destination Host Unreachable
From 10.9.0.1 icmp_seq=19 Destination Host Unreachable
From 10.9.0.1 icmp_seq=20 Destination Host Unreachable
From 10.9.0.1 icmp_seq=21 Destination Host Unreachable
From 10.9.0.1 icmp_seq=22 Destination Host Unreachable
From 10.9.0.1 icmp_seq=23 Destination Host Unreachable
From 10.9.0.1 icmp_seq=24 Destination Host Unreachable
```

> Here we get the error that states “Destination Host Unreachable”. This is due to the fact that the IP address when the program attempts to send an ARP request to the destination. Therefore no one is sending an ARP back for a result.

Task 2

> In this task, we create a sniffer program that runs in the host machine using pcap

> Code “sniff.c”

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>
/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function.
*/
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                const u_char *packet)
{
    printf("Got a packet\n");
}

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

    // Step 1: Open live pcap session on NIC with name eth3.
    // Students need to change "eth3" to the name found on their own
    //         machines (using ifconfig). The interface to the 10.9.0.0/2
    // network has a prefix "br-" (if the container setup is used).
    handle = pcap_open_live("br-625b70bf2ec5", BUFSIZ, 1, 1000, errbuf);

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

     // Step 3: Capture packets
     pcap_loop(handle, -1, got_packet, NULL);
     pcap_close(handle); // Close the handle
     return 0;
}
// Note: don’t forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```

> Result when executing *ping -c 3 8.8.8.8* in the victim VM while sniff.c is running in the attacker VM

```
root@VM:/tmp# ./a.out
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
Got a packet
```

```
[11/21/24]seed@VM:~/.../Labsetup$ 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=255 time=51.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=255 time=54.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=255 time=28.7 ms
```

Task 2.1A

Question 1

- Step 1: Open a live pcap session on NIC using 'pcap_open_live'. This allows the user to see the network traffic.
- Step 2: Set a filter by using 'pcap_compile()' and 'pcap_setfilter()'. These functions compile the string str into a filter program, and specify a filter program respectively.
- Step 3: Capture the packets caught in a loop, and then process captured packets using 'pcap_loop()' by setting it with -1, which puts it in an infinite loop.

Question 2:

- We need root privilege so we can see the whole network traffic in the interface.
- If we don't use run sniffer.c without a root user, pcap_open_live will fail, and in turn the program will return an error

Question 3:

- If you turn promiscuous mode off via setting the 3rd argument of pcap_open_live to 0, it will only intercept/sniff network traffic that is related to it.
- However, if the parameter is set to 1, which means that promiscuous mode is on, it will intercept/sniff all traffic.

Task 2.1B

> Unfortunately, I was unable to produce a result when running both of these programs as I would receive invalid BPF program errors. Hypothetically for this to work, you would need to alter filter_exp[] into "ip proto icmp" and ""proto TCP and dst portrange 10-100" in order to solve this task's problems respectively

Task 2.1C

> Once again, I was unable to get a result from the program, however in a hypothetical situation where this program works, the program would output the password, char by char. This would occur when you try to connect to telnet via the ip: 10.2.2.15

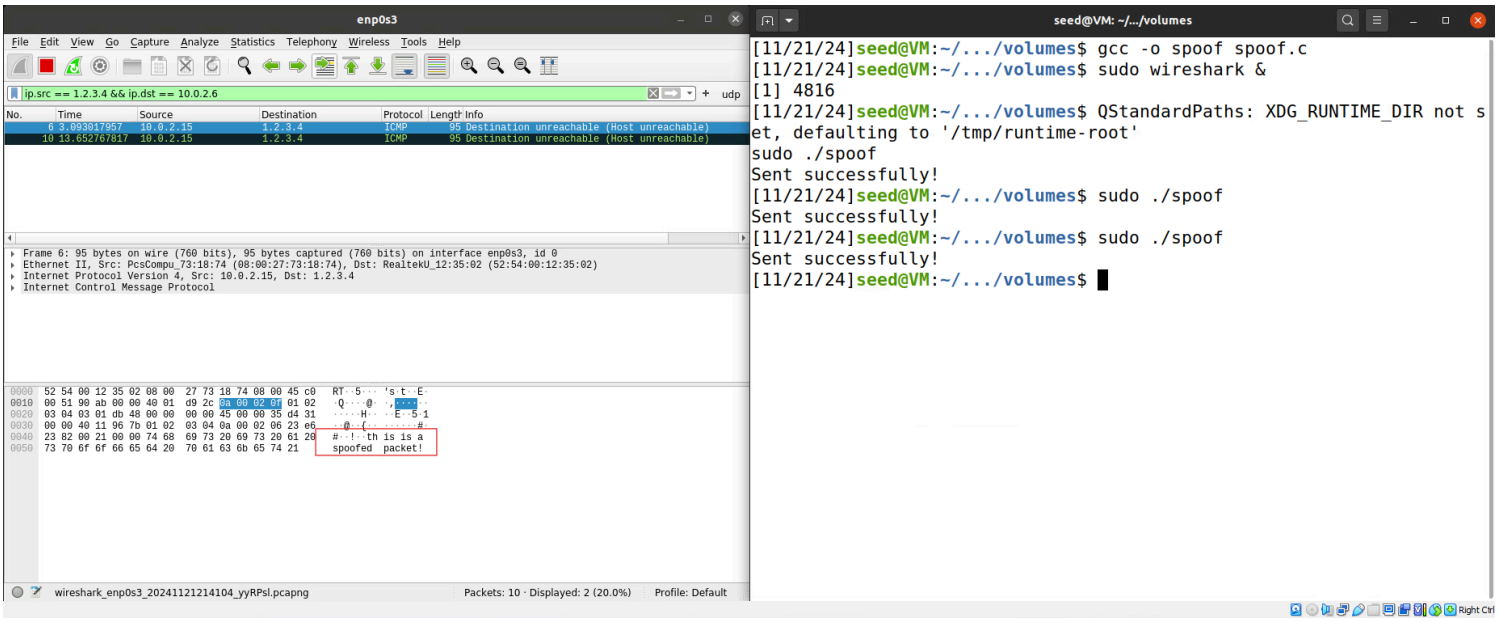
```
[11/21/24]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
Connected to 10.0.2.15.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
```

> After inputting the password, the program would sniff the tcp packets of telnet, via the filter in the program “tcp port telnet”. See pwd_sniff.c in submission for the code used.

Task 2.2A: Write a spoofing program

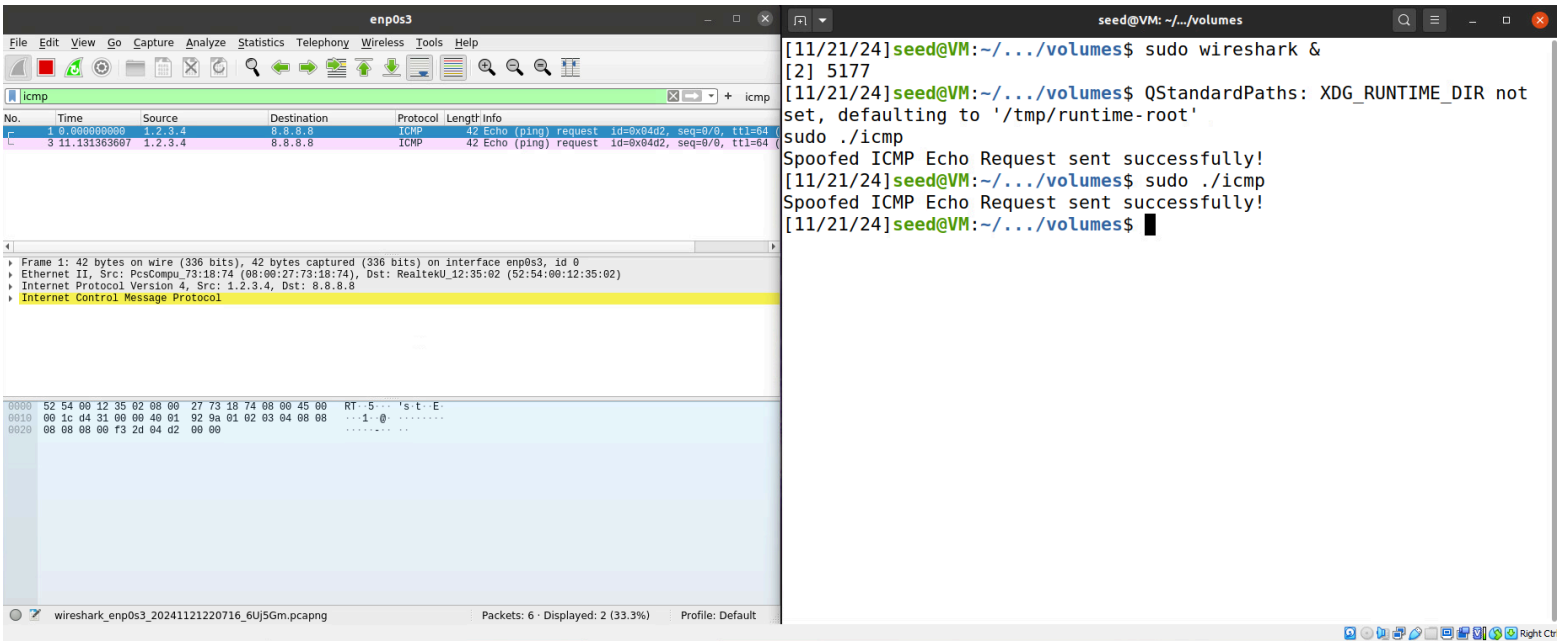
The program spoof.c (present in the submitted lab file) sends a fake IP packet using raw sockets. It manually creates the IP and UDP headers, setting the source and destination IPs, ports, and protocol. A custom message is added as the UDP payload, and the IP checksum is calculated to make the packet valid. The packet is then sent using the *sendto()* function. By using raw sockets, the program gives full control over the packet, making it useful for learning or testing network concepts.

Wireshark shows that the packets have been sent to the correct destination address (red box on the screen).



Task 2.2B: Spoof an ICMP Echo Request

Using Wireshark, we can see that the packets have been sent to the Google DNS IP address 8.8.8.8:



Question 4:

No, the iph_len must match the real size of the packet (IP header + payload). If the length is wrong, the packet will be rejected by the receiver or routers.

Question 5:

Yes, with raw sockets, you need to calculate the IP header checksum yourself. The operating system does not do it for you, so the packet might get dropped if the checksum is incorrect.

Question 6:

You need root privileges because raw sockets let you bypass normal network protections. Without root, the program will fail when trying to create the socket and show an error like "Operation not permitted."

Task 2.3: Sniff and then Spoof

In this task, we need to capture ICMP Echo Request packets (type 8) sent to a machine and respond with spoofed ICMP Echo Reply packets (type 0). The issue is that if we ping our own external IP address (e.g., 10.0.2.15), the operating system optimizes the process by using the loopback interface (lo) instead of going through the actual network interface (enp0s3). This prevents the program from seeing the packets on enp0s3. To make it work, we need a second machine or another VM on the same network. This machine will send ICMP requests that will actually pass through enp0s3. The program can then capture these packets, generate spoofed

ICMP replies, and send them back to the other machine. This demonstrates that both sniffing and spoofing are working correctly.
The sniff_and_spoof.c file in the rendering is the program that was intended to run for this task. On the attacker's VM we should have run sudo ./sniff_and_spoof and on the victim's VM ping 10.0.2.15. This would show how an attacker can manipulate the victim's network by intercepting and forging the packets he receives.