Task 1.1: Sniffing Packets

```
[09/25/25]seed@1006859:~/.../Labsetup$ ifconfig
br-ee3f4b5252f1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:aff:fe22:28bb  prefixlen 64  scopeid 0x20<link>
        ether 02:42:0a:22:28:bb  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 47  bytes 5585 (5.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The network interface is: br-ee3f4b5252fl

Task 1.1A

sniffer.py:

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-ee3f4b5252fl', filter='icmp', prn=print_pkt)
```

```
[09/25/25]seed@1006859:~/.../volumes$ dockps
2fcf3ce16ffd  hostB-10.9.0.6
2ea4542bf776  hostA-10.9.0.5
d99acb2d7ab7  seed-attacker
```

After running sniffer.py, I started a shell on host A and pinged host B. This should send an ICMP request from host A (10.9.0.5) to host B (10.9.0.6).

```
[09/25/25]seed@1006859:~/.../volumes$ docksh 2ea
root@2ea4542bf776:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.513 ms
^C
--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 1
ms
rtt min/avg/max/mdev = 0.513/0.513/0.513/0.000 ms
root@2ea4542bf776:/#
```

Below shows the sniffed packet on seed-attacker. It contains an ICMP echo-request from host A to host B and an ICMP echo-reply from host B to host A.

```
[09/25/25]seed@1006859:~/.../volumes$ docksh d99
root@1006859:/# cd volumes/
root@1006859:/volumes# chmod a+x sniffer.py
root@1006859:/volumes# sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 36013
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x99df
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x9146
        id        = 0x28
        seq       = 0x1
###[ Raw ]###
           load      = '\x8e\xf9\xd4h\x00\x00\x00\x00=[\x07\x00\x00\x00\x00\x00\x10
\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./012345
67'

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 46276
     flags     =
```

Now, we try to run sniffer.py without root privileges on seed-attacker.

```
root@1006859:/volumes# su seed
seed@1006859:/volumes$ sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface="br-ee3f4b5252f1", filter="icmp", prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sn
iff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _ru
n
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in _
_init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  # 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
```

This shows that there is a permission error. This is because root privileges are needed to create raw sockets, hence a normal user like “seed” will not have the necessary privileges to create a raw socket.

Task 1.1B

I need to set the following filters separately:

- Capture only ICMP packet
- Capture any TCP packet that comes from a particular IP with destination port number 23
- Capture packets coming from or going to a particular subnet, like 128.230.0.0/16

Capture only ICMP packet:

sniffer_icmp.py icmp.py udp.py

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

def print_pkt(pkt):
    print("Source IP:", pkt[IP].src)
    print("Destination IP:", pkt[IP].dst)
    print("Protocol:", pkt[IP].proto)
    print("\n")

pkt = sniff(iface="br-ee3f4b5252f1", filter="icmp", prn=print_pkt)
```

sniffer_icmp.py is the filter implemented to only capture ICMP packets.

On Host A, I pinged Host B to simulate ICMP packets between both hosts.
“nc 10.9.0.5 23” simulates sending TCP packets between both hosts as port 23 is the default port for Telnet, which also uses TCP.

```
root@2ea4542bf776:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.227 ms
^C
--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.227/0.227/0.227/0.000 ms
root@2ea4542bf776:/# nc 10.9.0.5 23
```

On the attacker machine, I ran sniffer_icmp.py to capture only ICMP packets, as shown by Protocol: 1.

```
root@1006859:/volumes/sniffer# sniffer_icmp.py
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Protocol: 1


Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Protocol: 1
```

The picture below shows the protocol field number specified by IETF RFC 790.

- ICMP – 1
- UDP – 17
- TCP – 6

RFC 790 Historic

Assigned Internet Protocol Numbers

Decimal	Octal	Protocol Numbers	References
0	0	Reserved	[JBP]
1	1	ICMP	[53,JBP]
2	2	Unassigned	[JBP]
3	3	Gateway-to-Gateway	[48,49,VMS]
4	4	CMCC Gateway Monitoring Message	[18,19,DFP]
5	5	ST	[20,JWF]
6	6	TCP	[34,JBP]
7	7	UCL	[PK]
8	10	Unassigned	[JBP]
9	11	Secure	[VGC]
10	12	BBN RCC Monitoring	[VMS]
11	13	NVP	[12,DC]
12	14	PUP	[4,EAT3]
13	15	Pluribus	[RDB2]
14	16	Telenet	[RDB2]
15	17	XNET	[25,JFH2]
16	20	Chaos	[MOON]
17	21	User Datagram	[42,JBP]
18	22	Multiplexing	[13,JBP]
19	23	DCN	[DLM1]
20	24	TAC Monitoring	[55,RH6]
21-62	25-76	Unassigned	[JBP]
63	77	any local network	[JBP]
64	100	SATNET and Backroom EXPAK	[DM11]
65	101	MIT Subnet Support	[NC3]
66-68	102-104	Unassigned	[JBP]
69	105	SATNET Monitoring	[DM11]
70	106	Unassigned	[JBP]
71	107	Internet Packet Core Utility	[DM11]
72-75	110-113	Unassigned	[JBP]
76	114	Backroom SATNET Monitoring	[DM11]
77	115	Unassigned	[JBP]
78	116	WIDEBAND Monitoring	[DM11]

Capture any TCP packet that comes from a particular IP with destination port number 23

sniffer_destport23.py is designed to capture any TCP packet that comes from a particular IP, 10.9.0.5, in this scenario, with destination port 23.

Similar to the previous example, I used ping to simulate ICMP request from 10.9.0.5 to 10.9.0.6 and ICMP response from 10.9.0.6 to 10.9.0.5.
“nc 10.9.0.6 23” will simulate TCP request from 10.9.0.5 to 10.9.0.6 and TCP response from 10.9.0.6 to 10.9.0.5. This is because port 23 is also the default port for Telnet, which uses TCP too.

```
root@2ea4542bf776:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.171 ms
^C
--- 10.9.0.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.171/0.171/0.171/0.000 ms
root@2ea4542bf776:/# nc 10.9.0.6 23
[illegible]
```

As expected, only the TCP requests from 10.9.0.5, headed to destination port 23, are captured by the sniffer. The occurrence of 3 such packets could signify the 3-way handshake that occurs during a TCP connection setup.

```
root@1006859:/volumes/sniffer# sniffer_destport23.py
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Destination port: 23
Protocol: 6


Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Destination port: 23
Protocol: 6


Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Destination port: 23
Protocol: 6
```

Capture packets coming from or going to a particular subnet, like 1.1.1.0/24

sniffer_subnet.py is designed to capture all packets coming from and going to the subnet 1.1.1.0/24.

sniffer_subnet.py | icmp.py | tcp.py | tcp_fail.py | tcp_fail_2.py

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

def print_pkt(pkt):
    print("Source IP:", pkt[IP].src)
    print("Destination IP:", pkt[IP].dst)
    print("Protocol:", pkt[IP].proto)
    print("\n")

pkt = sniff(iface="br-ee3f4b5252f1", filter="net 1.1.1.0/24", prn=print_pkt)
```

First, I tried to ping an IP outside of the subnet.

```
root@2ea4542bf776:/home/seed# 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=113 time=11.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=18.0 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 11.948/14.968/17.988/3.020 ms
root@2ea4542bf776:/home/seed#
```

This was indeed not captured on the sniffer.

```
root@1006859:/volumes/sniffer# sniffer_subnet.py
```

Now, I ping the subnet 1.1.1.0/24 in increments of 23. (below shows the start and end of the ping requests)

```
root@2ea4542bf776:/home/seed# 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=113 time=11.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=18.0 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 11.948/14.968/17.988/3.020 ms
root@2ea4542bf776:/home/seed# for ((i=1; i<=254; i+=23)); do ping -c
 1 -W 1 1.1.1.$i & done; wait
[1] 157
[2] 158
[3] 159
[4] 160
[5] 161
[6] 162
[7] 163
[8] 164
[9] 165
[10] 166
[11] 167
[12] 168
PING 1.1.1.208 (1.1.1.208) 56(84) bytes of data.
PING 1.1.1.162 (1.1.1.162) 56(84) bytes of data.
64 bytes from 1.1.1.208: icmp_seq=1 ttl=53 time=12.6 ms

--- 1.1.1.208 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 12.603/12.603/12.603/0.000 ms
64 bytes from 1.1.1.162: icmp_seq=1 ttl=50 time=8.27 ms

--- 1.1.1.162 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.273/8.273/8.273/0.000 ms
64 bytes from 1.1.1.1: icmp_seq=1 ttl=50 time=10.2 ms

--- 1.1.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 10.173/10.173/10.173/0.000 ms
[1]   Done                    ping -c 1 -W 1 1.1.1.$i
[2]   Done                    ping -c 1 -W 1 1.1.1.$i
[3]   Done                    ping -c 1 -W 1 1.1.1.$i
[5]   Done                    ping -c 1 -W 1 1.1.1.$i
[6]   Done                    ping -c 1 -W 1 1.1.1.$i
[8]   Done                    ping -c 1 -W 1 1.1.1.$i
[9]   Done                    ping -c 1 -W 1 1.1.1.$i
[10]   Done                    ping -c 1 -W 1 1.1.1.$i
64 bytes from 1.1.1.139: icmp_seq=1 ttl=50 time=15.7 ms

--- 1.1.1.139 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 15.703/15.703/15.703/0.000 ms
64 bytes from 1.1.1.254: icmp_seq=1 ttl=53 time=8.27 ms

--- 1.1.1.254 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.273/8.273/8.273/0.000 ms
64 bytes from 1.1.1.231: icmp_seq=1 ttl=53 time=8.24 ms

--- 1.1.1.231 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.241/8.241/8.241/0.000 ms
64 bytes from 1.1.1.70: icmp_seq=1 ttl=50 time=14.6 ms

--- 1.1.1.70 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 14.634/14.634/14.634/0.000 ms
[4]   Done                    ping -c 1 -W 1 1.1.1.$i
[7]   Done                    ping -c 1 -W 1 1.1.1.$i
[11]-  Done                    ping -c 1 -W 1 1.1.1.$i
[12]+  Done                    ping -c 1 -W 1 1.1.1.$i
```

These packets were all captured with the sniffer as seen below, as their source/destination IP lies in the 1.1.1.0/24 subnet.

```
root@1006859:/volumes/sniffer# sniffer_subnet.py
Source IP: 10.9.0.5
Destination IP: 1.1.1.208
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.162
Protocol: 1


Source IP: 1.1.1.208
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 1.1.1.162
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.47
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.116
Protocol: 1


Source IP: 1.1.1.47
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.24
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.93
Protocol: 1


Source IP: 1.1.1.185
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.70
Protocol: 1


Source IP: 10.9.0.5
Destination IP: 1.1.1.231
Protocol: 1


Source IP: 1.1.1.254
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 1.1.1.1
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 1.1.1.231
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 1.1.1.139
Destination IP: 10.9.0.5
Protocol: 1


Source IP: 1.1.1.70
Destination IP: 10.9.0.5
Protocol: 1
```

Task 1.2: Spoofing ICMP Packets

icmp.py is designed to spoof ICMP packets from 10.9.0.6 and send it to an arbitrary address. In this case, I have chosen 200.200.200.200.

sniffer_icmp.py | sniffer_destport23.py | icmp.py

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

ip = IP()
ip.dst = "10.9.0.6"
ip.src = "200.200.200.200"
icmp = ICMP()
pkt = ip/icmp

send(pkt)
print("Sent Spoofed ICMP packet from arbitrary 200.200.200.200...")
```

We will then open Wireshark and listen on our network interface, br-ee3f4b5252fl.

Then, we can run icmp.py to send a spoofed ICMP packet from 200.200.200.200 to 10.9.0.6.

```
root@1006859:/volumes/spoofer# icmp.py
.
Sent 1 packets.
Sent Spoofed ICMP packet from arbitrary 200.200.200.200...
```

After this, we can take a look at Wireshark again.

No.	Time	Source	Destination	Protocol	Length	Info
1	2025-09-25 10:3…	02:42:0a:22:28:bb	Broadcast	ARP	42	Who has 10.9.0.6? Tell 10.9.0.1
2	2025-09-25 10:3…	02:42:0a:09:00:06	02:42:0a:22:28:bb	ARP	42	10.9.0.6 is at 02:42:0a:09:00:06
3	2025-09-25 10:3…	200.200.200.200	10.9.0.6	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 4)
4	2025-09-25 10:3…	10.9.0.6	200.200.200.200	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3)
5	2025-09-25 10:3…	02:42:0a:09:00:06	02:42:0a:22:28:bb	ARP	42	Who has 10.9.0.1? Tell 10.9.0.6
6	2025-09-25 10:3…	02:42:0a:22:28:bb	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:0a:22:28:bb

On Wireshark, line 3 shows the ICMP echo request sent from 200.200.200.200 to 10.9.0.6, which means that our spoofed ICMP packet was successfully sent from an arbitrary source IP address.
Line 4 shows the ICMP echo reply sent from 10.9.0.6 to 200.200.200.200, which also means that our spoofed ICMP packet was also accepted by the other VM on the network.

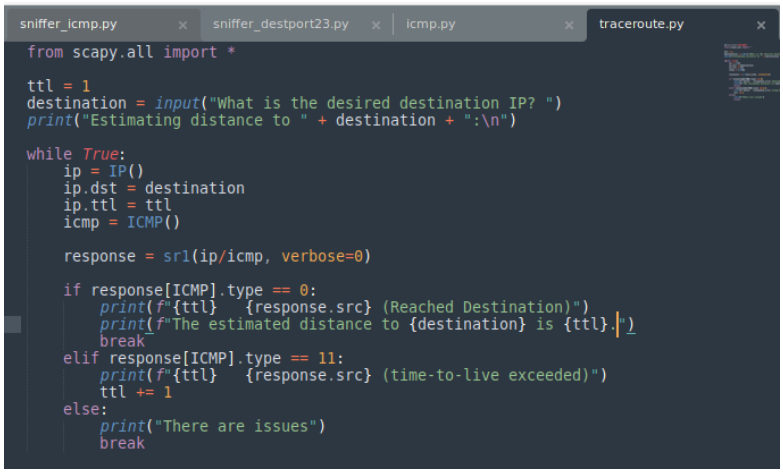
Task 1.3: Traceroute

The “ttl” variable will record the number of “hops” the ICMP request needs in order to reach the destination and will be incremented by 1 each time we receive an ICMP error message.
The user will be prompted to select the desired destination IP, in which I would use 8.8.8.8.

To find out if our ICMP packet reaches the destination, we need to craft the packet, then observe the response.
We can craft the ICMP packet by configuring ip.dst = destination IP, then ip.ttl to be the ttl variable (time-to-leave).
Then, we can send the ICMP packet with sr1(), which basically sends 1 packet and returns the response of the packet.

- I noted that if we reach the destination, we obtain “type: echo-reply” in the ICMP packet, which corresponds to response[ICMP] == 0.
- If we receive “type: time-exceeded” in the ICMP packet, it corresponds to response[ICMP] == 11, which also means that we reached the end of TTL and failed to reach the destination.

Hence, I will break the while loop if I receive response[ICMP].type == 0 to show that we reached the destination, else if I receive response[ICMP].type == 11, I will record the intermediate routers and increase the TTL variable and repeat the above process.



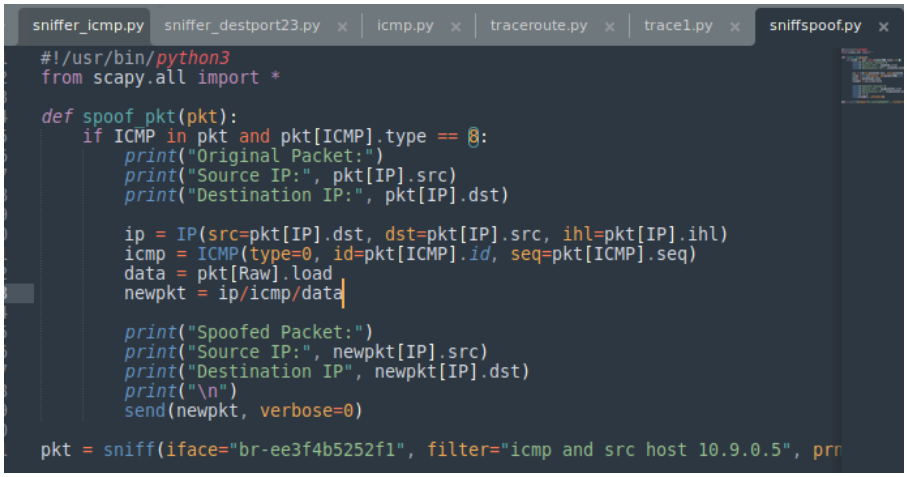
The image below shows the result after running traceroute.py. After choosing 8.8.8.8 as our destination, I show the routers that we will visit based on the TTL (on the left). Ultimately, the distance between 8.8.8.8 and my VM is about 10, in terms of the number of routers.

```
root@1006859:/volumes# traceroute.py
What is the desired destination IP? 8.8.8.8
Estimating distance to 8.8.8.8:

1       10.0.2.2 (time-to-live exceeded)
2       192.168.0.1 (time-to-live exceeded)
3       112.199.128.1 (time-to-live exceeded)
4       202.65.246.6 (time-to-live exceeded)
5       202.65.246.5 (time-to-live exceeded)
6       202.65.245.159 (time-to-live exceeded)
7       72.14.211.106 (time-to-live exceeded)
8       192.178.109.191 (time-to-live exceeded)
9       74.125.251.207 (time-to-live exceeded)
10      8.8.8.8 (Reached Destination)
The estimated distance to 8.8.8.8 is 10.
root@1006859:/volumes#
```

Task 1.4: Sniffing and-then Spoofing

sniffspoof.py is designed to sniff for ICMP requests from 10.9.0.5 (user), and then spoofs an ICMP echo reply packet (swap source IP and destination IP). After that, it sends the spoofed packet back to the user, 10.9.0.5.



Pinging non-existent host on the Internet

On the user machine, I pinged 1.2.3.4, which is a non-existent host on the Internet, but I am receiving ICMP echo replies.

```
seed@2ea4542bf776:/$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=57.9 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=25.7 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=24.8 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=30.6 ms
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
```

Looking at sniffspoof.py, we can see that the ICMP echo reply packets are being spoofed and sent back to the user.

```
root@1006859:/volumes# sniffspoof.py
Original Packet:
Source IP: 10.9.0.5
Destination IP: 1.2.3.4
Spoofed Packet:
Source IP: 1.2.3.4
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 1.2.3.4
Spoofed Packet:
Source IP: 1.2.3.4
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 1.2.3.4
Spoofed Packet:
Source IP: 1.2.3.4
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 1.2.3.4
Spoofed Packet:
Source IP: 1.2.3.4
Destination IP 10.9.0.5
```

The observation is as such because the program will sniff for ICMP echo requests, and send back a spoofed ICMP echo reply to the user, hence ping 1.2.3.4 will make it seem like the user can reach 1.2.3.4.

Pinging non-existent host on the LAN

On the user machine, I pinged 10.9.0.99, which is a non-existent host on the LAN. The results show that 10.9.0.99 is unreachable from the user's machine, which is different from us pinging a non-existent host on the Internet.

```
root@2ea4542bf776:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
From 10.9.0.5 icmp_seq=4 Destination Host Unreachable
From 10.9.0.5 icmp_seq=5 Destination Host Unreachable
From 10.9.0.5 icmp_seq=6 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
7 packets transmitted, 0 received, +6 errors, 100% packet loss, time
 6136ms
pipe 4
```

We can also see that there are no packets being spoofed in the spoofing program.

```
root@1006859:/volumes# sniffspoof.py
```

This observation is due to 10.9.0.99 being located in the LAN. The local subnet range can be determined with ifconfig (with subnet mask). Since 10.9.0.99 lies within the local subnet range, the kernel will send an ARP request locally, broadcasting the request to find out the MAC address that belongs to 10.9.0.99. Since 10.9.0.99 does not actually belong in the LAN, the MAC address will not be resolved, hence the Destination Host is Unreachable.

The sniffer program is only built to sniff ICMP packets, which will not be travelling through the network as ARP is a layer lower compared to ICMP, so we can't sniff it and spoof it.

Pinging existing host on the Internet

On the user machine, I pinged 8.8.8.8. The results show that there are duplicate ICMP echo replies.

```
seed@2ea4542bf776:/$ 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=113 time=8.40 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=64.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=5.38 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=21.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=5.44 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=15.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=113 time=7.22 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=24.9 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, +4 duplicates, 0% packet loss, ti
me 3009ms
rtt min/avg/max/mdev = 5.378/19.025/64.906/18.651 ms
```

The sniffer program also shows that we successfully spoofed ICMP echo replies.

```
root@1006859:/volumes# sniffspoof.py
Original Packet:
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet:
Source IP: 8.8.8.8
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet:
Source IP: 8.8.8.8
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet:
Source IP: 8.8.8.8
Destination IP 10.9.0.5


Original Packet:
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet:
Source IP: 8.8.8.8
Destination IP 10.9.0.5
```

This observation is due to the ICMP packet being passed outside the LAN and onto the Internet. This would be sniffed by the sniffer and spoofer program, hence it will also generate a spoofed ICMP echo reply as shown from the picture above. Since 8.8.8.8 is also a legitimate reachable IP address, the user will also receive the original ICMP echo reply. Hence, there will be duplicates for every ICMP echo request that the user performs.