Task 1.1: Sniffing Packets

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits. In addition, answer any questions if any.

1.1 a) After setting up the environment with Scapy and created the containers, we proceeded with the creation of a python script that calls a function to sniff the packets. This python file is created with root privileges under the attacker container.

```
[02/05/24]seed@VM:~/volumes$ dockps
456591ee3dd0  seed-attacker
1a82cd9c41f0  hostB-10.9.0.6
03bea0aef4bb  hostA-10.9.0.5
[02/05/24]seed@VM:~/volumes$ docksh 45
root@VM:/# cd volumes/
root@VM:/volumes# touch task1.1.py
root@VM:/volumes#
```

In the python file we write the following piece of code. To get the interface, we can use the command **ifconfig**.

```
[02/05/24]seed@VM:~/volumes$ ifconfig
br-11dbd78e08f2: 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::42:3aff:fe7c:81aa  prefixlen 64  scopeid 0x20<link>
        ether 02:42:3a:7c:81:aa  txqueuelen 0  (Ethernet)
        RX packets 288  bytes 17510 (17.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 57  bytes 6404 (6.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
1 #!/usr/bin/env python3
2 from scapy.all import *
3
4
5 def print_pkt(pkt):
6         print('---------------Packet---------------')
7         pkt.show()
8
9 pkt=sniff(iface='br-11dbd78e08f2',filter='icmp',prn=print_pkt)
```

To execute the program, so that the information of the packets gets printed out, we first need to change the permissions of the file. We have also modified the code by printing “packet” before the information is displayed, so we can easily differentiate the packets amongst themselves.

```
root@VM:/volumes# chmod a+x task1.1.py
root@VM:/volumes# ./task1.1.py
--------------Packet--------------
###[ 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        = 30683
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xaeb1
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xb9e8
        id        = 0x2d
        seq       = 0x1
###[ Raw ]###
           load      = '\xe0\xa0\xc0e\x00\x00\x00\x00\xd5\x0f\t\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234
7'

--------------Packet--------------
###[ 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        = 55239
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x8ec5
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
```

Since the filter in the sniff function is 'icmp' the details of the ICMP packet are displayed. However, once we run the program nothing appears on the screen. To get the information of the packets we first have to ping an address from a host. We can open a new window on the terminal write the ping command:

```
[02/05/24]seed@VM:~/volumes$ docksh 1a
root@1a82cd9c41f0:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.108 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.121 ms
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1014ms
rtt min/avg/max/mdev = 0.108/0.114/0.121/0.006 ms
```

If we try to sniff packets from the "seed" account we get the following:

```
root@VM:/volumes# su seed
seed@VM:/volumes$ ./task1_1.py
Traceback (most recent call last):
  File "./task1_1.py", line 7, in <module>
    pkt=sniff(iface='br-11dbd78e08f2',filter='net 128.230.0.0/16',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, 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
seed@VM:/volumes$
```

As we can tell, sniffing packets without root privileges is impossible. That is because all network traffic is documented under the root, where higher privileges are required. If that was not the case, any user could have access to information on packets or data that is transmitted through a network.

We can also change the filter parameter inside the sniff method to “tcp” so that only TCP packets are captured. For testing, we can use telnet since it is commonly used for testing TCP connectivity.

Python script after modifications:

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


def print_pkt(pkt):
        print('---------------Packet---------------')
        pkt.show()

pkt=sniff(iface='br-11dbd78e08f2',filter='tcp && host 10.9.0.6 && dst port 23',prn=print_pkt)
```

Here, the source is host B and the TCP packets are directed to destination port 23.

```
[02/05/24]seed@VM:~/volumes$ docksh hostB-10.9.0.6
root@1a82cd9c41f0:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
03bea0aef4bb 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
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Sun Feb  4 19:17:19 UTC 2024 from 03bea0aef4bb on pts/3
seed@03bea0aef4bb:~$
```

```
seed@VM:/volumes$ exit
exit
root@VM:/volumes# ./task1.1.py
---------------Packet---------------
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 51873
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x5bee
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ TCP ]###
        sport     = 48456
        dport     = telnet
        seq       = 501270057
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x144b
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (3168443312, 0)), ('NOP', None), ('WScale', 7)]

---------------Packet---------------
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 51874
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x5bf5
```

We can also sniff packets from networks that we specify:

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


def print_pkt(pkt):
        print('---------------Packet---------------')
        pkt.show()

pkt=sniff(iface='br-11dbd78e08f2',filter='net
 192.168.0.0/16',prn=print_pkt)
```

```
root@1a82cd9c41f0:/# ping 192.168.1.39
PING 192.168.1.39 (192.168.1.39) 56(84) bytes of data.
64 bytes from 192.168.1.39: icmp_seq=1 ttl=126 time=0.684 ms
64 bytes from 192.168.1.39: icmp_seq=2 ttl=126 time=0.819 ms
^C
--- 192.168.1.39 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1055ms
rtt min/avg/max/mdev = 0.684/0.751/0.819/0.067 ms
root@1a82cd9c41f0:/#
```

```
root@VM:/volumes# ./task1.1.py
--------------Packet--------------
###[ Ethernet ]###
  dst       = 01:00:5e:00:00:fb
  src       = 02:42:8d:16:b5:5f
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 73
     id        = 32047
     flags     = DF
     frag      = 0
     ttl       = 255
     proto     = udp
     chksum    = 0x136f
     src       = 10.9.0.1
     dst       = 224.0.0.251
     \options   \
###[ UDP ]###
        sport     = mdns
        dport     = mdns
        len       = 53
        chksum    = 0xeb4b
###[ DNS ]###
           id        = 0
           qr        = 0
           opcode    = QUERY
           aa        = 0
           tc        = 0
           rd        = 0
           ra        = 0
           z         = 0
           ad        = 0
           cd        = 0
           rcode     = ok
           qdcount   = 2
           ancount   = 0
           nscount   = 0
           arcount   = 0
           \qd        \
            |###[ DNS Question Record ]###
            |  qname     = '_ipps._tcp.local.'
            |  qtype     = PTR
            |  qclass    = IN
            |###[ DNS Question Record ]###
            |  qname     = '_ipp._tcp.local.'
            |  qtype     = PTR
            |  qclass    = IN
           an        = None
           ns        = None
           ar        = None
```

In this scenario, we sniffed packets from a network we defined. We chose the network our physical device belongs to, in order to ensure that the IP address exists within the network the VM is connected to and that the ping will execute successfully with no packet loses. As we can see, after running the python file, we can get information about the UDP packet as well as the DNS server. This proves, that we can sniff packets from any networks.

Task 1.2: Spoofing ICMP Packets

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanations to the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits. In addition, answer any questions if any.

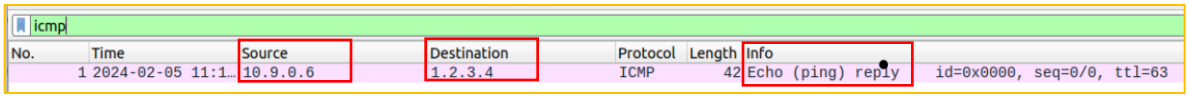
- The following code sends a spoofed ICMP packet. The source IP address is set to **'1.2.3.4'** and the destination IP address is set to **'10.9.0.6'**.

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

print("SENDING SPOOFED ICMP PACKET...")
ip = IP()
ip.src = '1.2.3.4'
ip.dst = '10.9.0.6'
icmp = ICMP()
pkt = ip/icmp
send(pkt)
```

- A spoofed packet has been sent successfully.

```
root@VM:/volumes# python3 task1.2.py
SENDING SPOOFED ICMP PACKET...
.
Sent 1 packets.
```



Upon analyzing the Wireshark capture, we can observe that the echo request was accepted by the receiver, and an echo reply has been sent to the spoofed IP address accordingly.

Task 1.3: Traceroute

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits. In addition, answer any questions if any.

```
1 #!/usr/bin/env python3
2
3 import sys
4 from scapy.all import *
5
6 a = IP()
7 a.dst = '8.8.8.8'
8 a.ttl = int(sys.argv[1])
9 b = ICMP()
10 h = sr1(a/b)
11 print("Router: ", h.src)
```

int(sys.argv[1]): To access the first command-line argument, and then converts it to an integer.

Sends the crafted packet and waits for the first response.

a / b = [IP Header] [ICMP Header] combines the IP and ICMP packets.

```
[01/31/24]seed@VM:~/.../Labsetup$ dockps
4737945c2429  seed-attacker
cea633d7f10a  hostB-10.9.0.6
b40083812bab  hostA-10.9.0.5
```

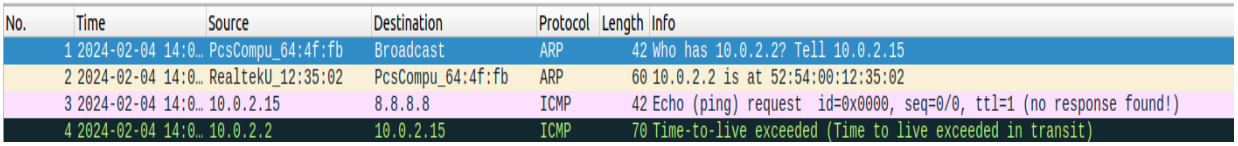
```
[02/04/24]seed@VM:~/.../volumes$ docksh 47
```

```
root@VM:/volumes# ./task1.3.py 1
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Router:  10.0.2.2
```

```
root@VM:/volumes# ./task1.3.py 9
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
Router:  8.8.8.8
```

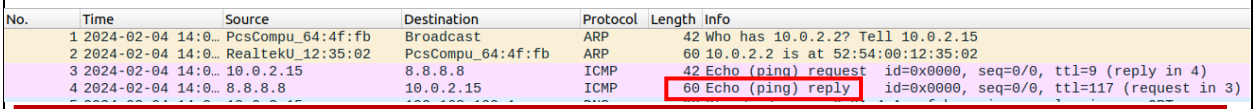
The packet traversed 9 hops before reaching the destination.

```
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::363b:9d0c:1684:93b8  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:64:4f:fb  txqueuelen 1000  (Ethernet)
        RX packets 87622  bytes 122335325 (122.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 19180  bytes 1465983 (1.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```



No.	Time	Source	Destination	Protocol	Length	Info
1	2024-02-04 14:0…	PcsCompu_64:4f:fb	Broadcast	ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
2	2024-02-04 14:0…	RealtekU_12:35:02	PcsCompu_64:4f:fb	ARP	60	10.0.2.2 is at 52:54:00:12:35:02
3	2024-02-04 14:0…	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no response found!)
4	2024-02-04 14:0…	10.0.2.2	10.0.2.15	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)

1)	ARP request sent from the device that has the MAC address “PcsCompu_64:4f:fb” to get the MAC address that is with the IP address 10.0.2.2.
2)	ARP reply
3)	ICMP Echo Request from 10.0.2.15 to 8.8.8.8 with TTL = 1
4)	ICMP Time-to-Live Exceeded indicates that the packet reached the router at 10.0.2.2, however it did not reach the final destination because the TTL expired. And the message is sent by the router to 10.0.2.15.



No.	Time	Source	Destination	Protocol	Length	Info
1	2024-02-04 14:0…	PcsCompu_64:4f:fb	Broadcast	ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
2	2024-02-04 14:0…	RealtekU_12:35:02	PcsCompu_64:4f:fb	ARP	60	10.0.2.2 is at 52:54:00:12:35:02
3	2024-02-04 14:0…	10.0.2.15	8.8.8.8	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=9 (reply in 4)
4	2024-02-04 14:0…	8.8.8.8	10.0.2.15	ICMP	60	Echo (ping) reply id=0x0000, seq=0/0, ttl=117 (request in 3)

successful ICMP Echo Reply, which means that the destination is reachable.

This information is valuable for the attacker as it reveals the IP addresses of routers along the path from the source to the destination. The attacker can carry out a spoofing attack by crafting a malicious packet with source IP address that resembles one of the legitimate routers discovered. The malicious packets sent by the attacker blend in with the normal network traffic. By this attack the attacker aims to deceive network defenses and evade detection.

Task 1.4: Sniffing and-then Spoofing

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanations to the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits. In addition, answer any questions if any.

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


def handle_packet(packet):
    # Check if the packet is an ICMP packet and not from the spoofed source IP
    if ICMP in packet and packet[ICMP].type == 8:
        print("ORIGINAL PACKET")
        print("Source IP:", packet[IP].src)
        print("Destination IP:", packet[IP].dst)

        # SPOOFING
        a = IP(src=packet[IP].dst, dst=packet[IP].src, ihl=packet[IP].ihl)
        a.ttl = 99
        b = ICMP(type=0, id=packet[ICMP].id, seq=packet[ICMP].seq)

        if packet.haslayer(Raw):
                data = packet[Raw].load
                p = a/b/data
        else:
                p = a/b

        print("SPOOFED PACKET")
        print("Source IP:", p[IP].src)
        print("Destination IP:", p[IP].dst)

        # Send the spoofed packet
        send(p, verbose=0)

#a non-existing host on the Internet
#sniff(iface='br-8bab58360523',filter='icmp and host 1.2.3.4', prn=handle_packet)

# a non-existing host on the LAN
#sniff(iface='br-8bab58360523',filter='icmp and host 10.9.0.99', prn=handle_packet)

# an existing host on the Internet
sniff(iface='br-8bab58360523',filter='icmp and host 8.8.8.8', prn=handle_packet)
```

- The code uses the sniff function from the scapy library to monitor network traffic. It is set to filter for ICMP packets to and from specific IP addresses.
- The handle_packet() function is called for every packet that matches the sniffing filter criteria. It first checks if the packet is an ICMP echo request (type 8).
- If an ICMP echo request is detected, the script constructs a new packet with the IP and ICMP layers. It swaps the source and destination IP addresses so that the new packet appears to come from the IP address that was pinged. It also creates a corresponding ICMP echo reply (type 0).
- If the original packet has the payload of the ICMP request, it is copied to the spoofed packet
- Then the spoofed ICMP echo reply packet is sent using the send function.
- 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.

Pinging a non-exsting host on the internet:

```
root@6a84a72a31c3:/# 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=168 ttl=99 time=60.0 ms
64 bytes from 1.2.3.4: icmp_seq=169 ttl=99 time=22.2 ms
64 bytes from 1.2.3.4: icmp_seq=170 ttl=99 time=20.1 ms
64 bytes from 1.2.3.4: icmp_seq=171 ttl=99 time=18.7 ms
64 bytes from 1.2.3.4: icmp_seq=172 ttl=99 time=21.1 ms
64 bytes from 1.2.3.4: icmp_seq=173 ttl=99 time=25.0 ms
64 bytes from 1.2.3.4: icmp_seq=174 ttl=99 time=18.8 ms
64 bytes from 1.2.3.4: icmp_seq=175 ttl=99 time=20.9 ms
64 bytes from 1.2.3.4: icmp_seq=176 ttl=99 time=18.6 ms
64 bytes from 1.2.3.4: icmp_seq=177 ttl=99 time=24.4 ms
64 bytes from 1.2.3.4: icmp_seq=178 ttl=99 time=21.6 ms
64 bytes from 1.2.3.4: icmp_seq=179 ttl=99 time=28.7 ms
64 bytes from 1.2.3.4: icmp_seq=180 ttl=99 time=53.6 ms
^C
--- 1.2.3.4 ping statistics ---
209 packets transmitted, 42 received, 79.9043% packet loss, time 212158ms
rtt min/avg/max/mdev = 14.310/24.809/60.047/8.573 ms
```

Pinging a non-exsting host on the LAN:

```
root@6a84a72a31c3:/# 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
From 10.9.0.5 icmp_seq=7 Destination Host Unreachable
From 10.9.0.5 icmp_seq=8 Destination Host Unreachable
From 10.9.0.5 icmp_seq=9 Destination Host Unreachable
From 10.9.0.5 icmp_seq=10 Destination Host Unreachable
From 10.9.0.5 icmp_seq=11 Destination Host Unreachable
From 10.9.0.5 icmp_seq=12 Destination Host Unreachable
From 10.9.0.5 icmp_seq=13 Destination Host Unreachable
From 10.9.0.5 icmp_seq=14 Destination Host Unreachable
From 10.9.0.5 icmp_seq=15 Destination Host Unreachable
From 10.9.0.5 icmp_seq=16 Destination Host Unreachable
From 10.9.0.5 icmp_seq=17 Destination Host Unreachable
From 10.9.0.5 icmp_seq=18 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
97 packets transmitted, 0 received, +96 errors, 100% packet loss, time 98304ms
pipe 4
```

Pinging an existing host on the Internet

```
root@6a84a72a31c3:/# 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=114 time=22.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=99 time=65.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=99 time=17.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=36.5 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=22.7 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=99 time=26.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=114 time=18.5 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=99 time=22.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=114 time=21.2 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=99 time=25.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=6 ttl=99 time=18.4 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=114 time=21.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=7 ttl=114 time=20.2 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=99 time=22.1 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=8 ttl=99 time=14.5 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=114 time=21.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=9 ttl=114 time=20.5 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=99 time=22.1 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
9 packets transmitted, 9 received, +9 duplicates, 0% packet loss, time 8013ms
rtt min/avg/max/mdev = 14.539/24.355/65.263/10.875 ms
```

Explanations/Observations:

- **Pinging a Non-Existing Host on the Internet**:

  The ARP process for sending a packet to an external IP address involves:

  1. The device recognizes the destination IP as outside the LAN and checks its routing table to direct the packet to the gateway (router).
  2. The packet, containing the destination IP, is sent to the gateway.
  3. The gateway routes the packet through the internet, based on its routing table, until it arrives at the destination network.

  When attempting to ping a non-existent host on the internet (outside the LAN), the packet is routed to the gateway, where an ICMP packet is identified. Despite the absence of the destination host, since the packet is sent out and reaches the router, the python code will detect it and produce a spoofed packet then send it back.

- **Pinging a Non-Existing Host on the LAN**

  We expected the output to be similar to pinging a non-existence host on the internet. Contrary to expectations, the destination is marked unreachable. This behavior can be explained by the ARP process:

  1. The device recognizes the destination is within its network and attempts to retrieve the corresponding MAC address from its MAC table.
  2. If no record exists, an ARP request is broadcasted to get and record the MAC address.

  In this scenario where the LAN host doesn't exist, the MAC table doesn’t have an entry, and no ARP reply is received. Therefor, the packet doesn’t leave the source host, remains undetected, and no spoofed packet is generated.

- **Pinging an Existing Host on the Internet**

  Normally, pinging an address like 8.8.8.8 involves the ICMP echo request traversing the local network, reaching the internet, and being directed to Google's DNS server. If the server is accessible, it responds with an ICMP echo reply. However, when the code is active, as soon as an ICMP echo request is sent to 8.8.8.8, the code generates a spoofed ICMP echo reply. This creates the illusion of an immediate response from 8.8.8.8. Since the server is operational, both legitimate and spoofed reply packets are received, explaining the 'DUP!' found in the output.