ASSIGNMENT # 1 TEMPLATE

Student Name & ID	**Anas Madkoor, 202104114**
Student Name & ID	**Abdulrazzaq Alsiddiq, 202004464**
Student Name & ID	**Omar Amin, 202003122**
Student Name & ID	**Ali Zair, 202109964**
Student Name & ID	**Lance Eric Ruben, 202005801**

Task 1.1: Sniffing Packets

Task 1.1.A (trying to execute the python code from the seed user):

```
root@VM:/volumes# su seed
seed@VM:/volumes$
seed@VM:/volumes$
seed@VM:/volumes$ ./task1.1.py
Traceback (most recent call last):
  File "./task1.1.py", line 15, in <module>
    pkt = sniff(iface='br-681187753cf7', 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, 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$
```

We observe that the user seed is not granted the permission to execute the code but on the other hand when we executed the code using the root user, we were successful. Therefore, by default, we need root privileges to sniff packets on the network using scapy.

Task 1.1B (Capture only the ICMP Packets):

Python code that will be used to perform the task:

```
1 #!/usr/bin/env python3
2
3 # Task 1.1 --------------------------------------------------
4
5 from scapy.all import *
6
7 def print_pkt(pkt):
8    pkt.show()
9
10 # The interface can be found with
11 # 'docker network ls' in the VM or 'ifconfig' in the containner
12
13 #Task 1.1.A and 1.1.B (Capture only the ICMP packet)
14
15 pkt = sniff(iface='br-681187753cf7', filter='icmp', prn=print_pkt)
16
```

Setting the filter to capture icmp packets.

seed@VM: ~/.../volumes

seed@VM: ~/.../Labsetup | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volumes

```
root@VM:/volumes# ./task1.1.py
```

The program is executed now, and is ready to capture the traffic

seed@VM: ~

seed@VM: ~/.../Labsetup | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volumes | seed@VM: ~

```
root@09af0074f7b6:/# 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=0.043 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.116 ms
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1013ms
rtt min/avg/max/mdev = 0.043/0.079/0.116/0.036 ms
root@09af0074f7b6:/#
```

seed@VM: ~/.../volumes

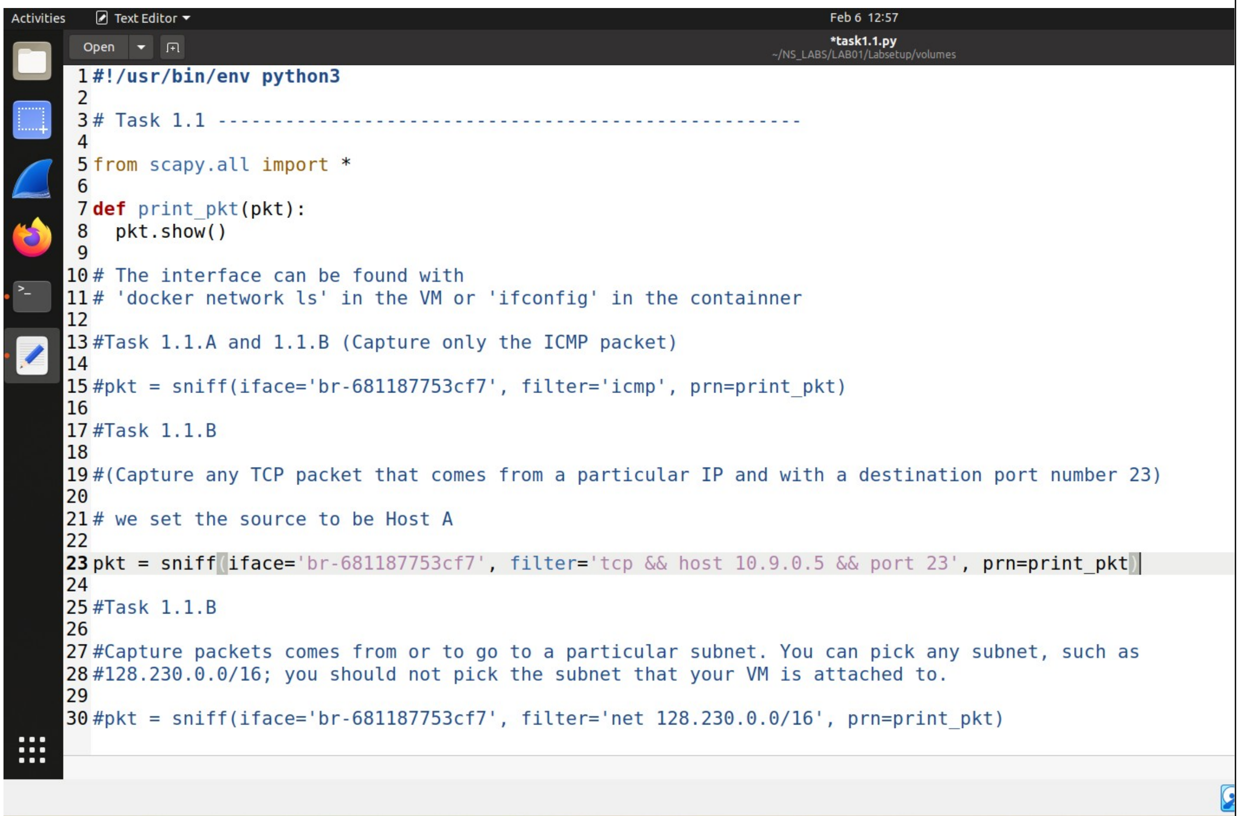
seed@VM: ~/.../Labsetup | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volumes

```
root@VM:/volumes# ./task1.1.py
###[ 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        = 19910
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xd8c6
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x3226
        id        = 0x43
        seq       = 0x1
###[ Raw ]###
           load      = '#z\xc2e\x00\x00\x00\x00\x15\xe3\x0b\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x1
1f !"#$%&\'()*+,-./01234567'
```

Task 1.1.B

(Capture any TCP packet that comes from a particular IP and with a destination port number 23):

Python code that will be used to perform the task:



seed@VM: ~

seed@VM: ~/.../Labsetup | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volumes

```
seed@09af0074f7b6:~$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
09af0074f7b6 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: Tue Feb  6 18:07:43 UTC 2024 from hostA-10.9.0.5.net-10.9.0.0 on pts/1
seed@09af0074f7b6:~$
```

seed@VM: ~/.../volumes

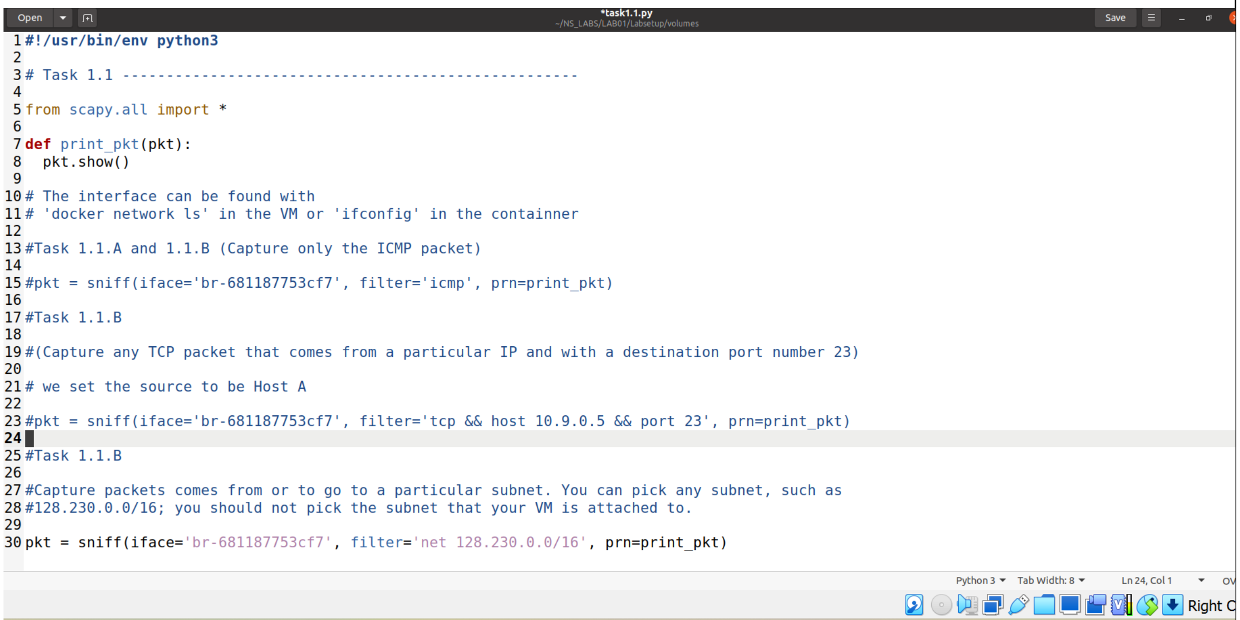
seed@VM: ~/.../Labsetup | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volumes

```
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 376
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x2520
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 47376
        dport     = telnet
        seq       = 158327363
        ack       = 1541739179
        dataofs   = 8
        reserved  = 0
        flags     = A
        window    = 501
        chksum    = 0x1443
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (1670767146, 3877068584))]
```

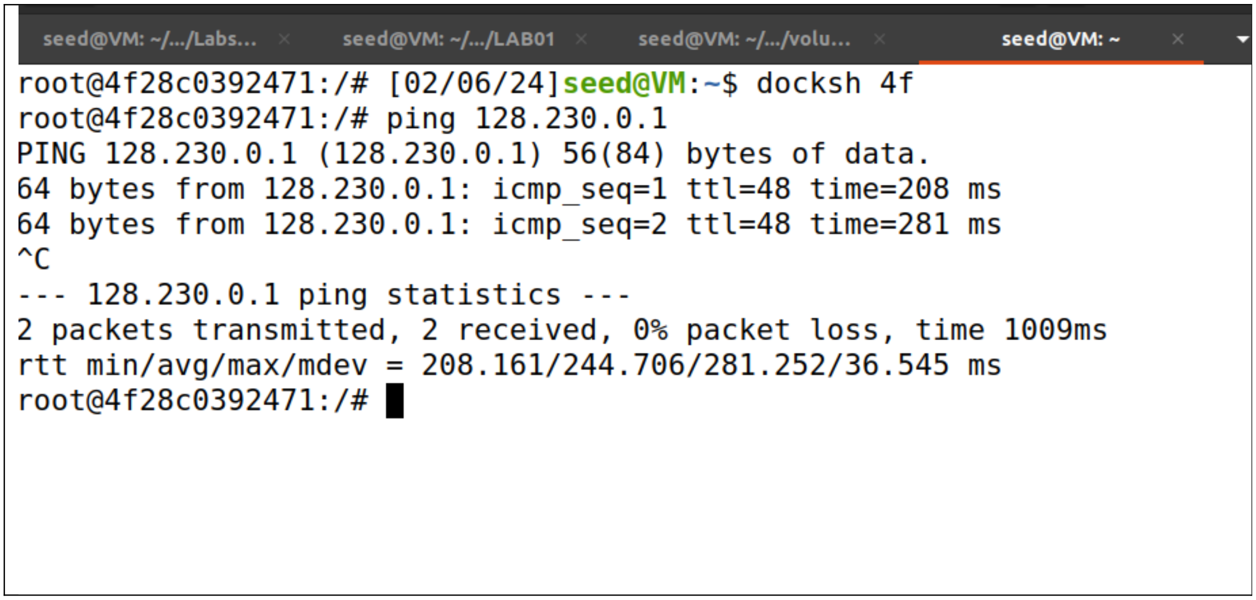
<u>Task 1.1.B</u>

<u>(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):</u>

<u>Python code that will be used to perform the task:</u>




```
#!/usr/bin/env python3

# Task 1.1 --------------------------------------------------

from scapy.all import *

def print_pkt(pkt):
  pkt.show()

# The interface can be found with
# 'docker network ls' in the VM or 'ifconfig' in the containner

#Task 1.1.A and 1.1.B (Capture only the ICMP packet)

#pkt = sniff(iface='br-681187753cf7', filter='icmp', prn=print_pkt)

#Task 1.1.B

#(Capture any TCP packet that comes from a particular IP and with a destination port number 23)

# we set the source to be Host A

#pkt = sniff(iface='br-681187753cf7', filter='tcp && host 10.9.0.5 && port 23', prn=print_pkt)

#Task 1.1.B

#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.

pkt = sniff(iface='br-681187753cf7', filter='net 128.230.0.0/16', prn=print_pkt)
```



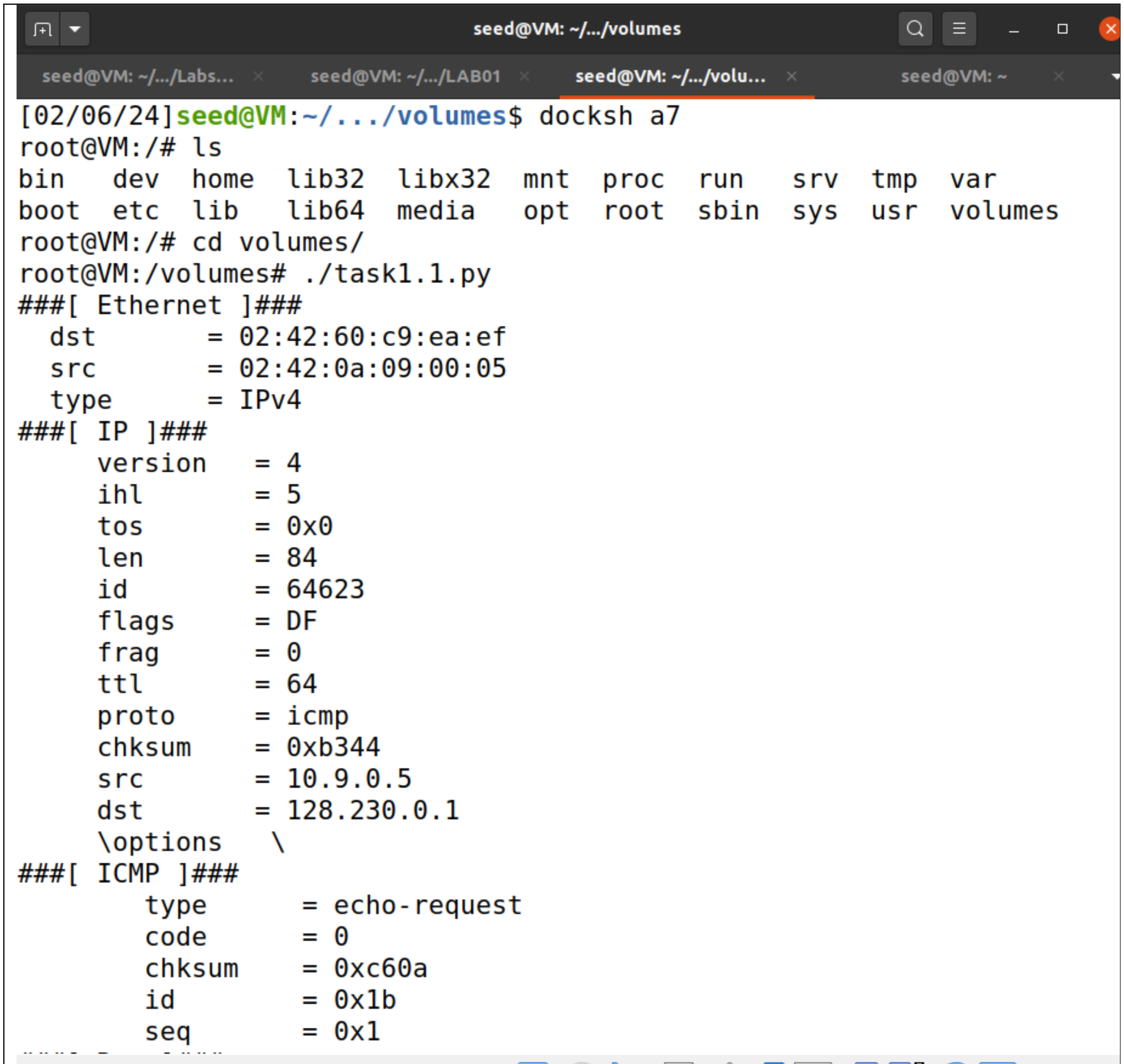

```
root@4f28c0392471:/# [02/06/24]seed@VM:~$ docksh 4f
root@4f28c0392471:/# ping 128.230.0.1
PING 128.230.0.1 (128.230.0.1) 56(84) bytes of data.
64 bytes from 128.230.0.1: icmp_seq=1 ttl=48 time=208 ms
64 bytes from 128.230.0.1: icmp_seq=2 ttl=48 time=281 ms
^C
--- 128.230.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1009ms
rtt min/avg/max/mdev = 208.161/244.706/281.252/36.545 ms
root@4f28c0392471:/# 
```

seed@VM: ~/.../volumes

seed@VM: ~/.../Labs... | seed@VM: ~/.../LAB01 | seed@VM: ~/.../volu... | seed@VM: ~

```
%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:60:c9:ea:ef
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0xa
     len       = 84
     id        = 1041
     flags     =
     frag      = 0
     ttl       = 48
     proto     = icmp
     chksum    = 0xfb99
     src       = 128.230.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xce0a
        id        = 0x1b
        seq       = 0x1
###[ Raw ]###
           load      = 'Dl\xc2e\x00\x00\x00\x00c4\t\x00\x00\x00\x00\x0
0\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$
%&\'()*+,-./01234567'
```

Right Ctrl


seed@VM: ~/.../volumes
seed@VM: ~/.../Labs...
seed@VM: ~/.../LAB01
seed@VM: ~/.../volu...
seed@VM: ~
[02/06/24]seed@VM:~/.../volumes$ docksh a7
root@VM:/# ls
bin dev home lib32 libx32 mnt proc run srv tmp var
boot etc lib lib64 media opt root sbin sys usr volumes
root@VM:/# cd volumes/
root@VM:/volumes# ./task1.1.py
###[Ethernet]###
dst = 02:42:60:c9:ea:ef
src = 02:42:0a:09:00:05
type = IPv4
###[IP]###
version = 4
ihl = 5
tos = 0x0
len = 84
id = 64623
flags = DF
frag = 0
ttl = 64
proto = icmp
chksum = 0xb344
src = 10.9.0.5
dst = 128.230.0.1
\options \
###[ICMP]###
type = echo-request
code = 0
chksum = 0xc60a
id = 0x1b
seq = 0x1

Task 1.2: Spoofing ICMP Packets

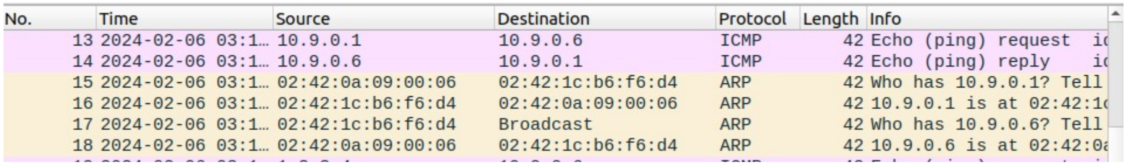
Code of 1.2:

Spoofing involves concealing the true origin of a communication by making it appear as though it comes from a trusted source. In task 1.2, the goal is to employ Scapy, an interactive packet manipulation library in Python, to spoof ICMP echo requests and direct them towards a distinct Virtual Machine within the network.

```
1  from scapy.all import *
2
3  a = IP()
4  a.src = '1.2.3.4'
5  a.dst = '10.9.0.6'
6  b = ICMP()
7  pkt = a/b
8  send(pkt)
9
10 ls(a)
11
```

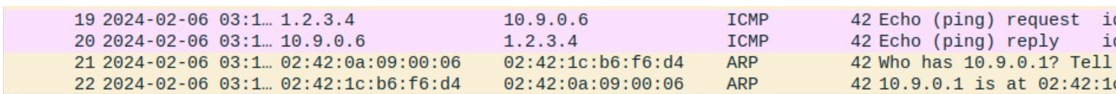
The provided code demonstrates the process of spoofing a packet using Scapy. Firstly, the Scapy library was imported to utilize its features in the code. On Line 3, an IP object was created and assigned to variable A. Variable A was then given a source address (Line 4), which masks the original source address, and a destination address (Line 5). Subsequently, an ICMP object was created and assigned to variable B (Line 6). The ICMP object was encapsulated within the IP object using a "/" (Line 7) and sent to the destination network (Line 8). The `ls(a)` command was employed to list the fields and their descriptions for the specified packet.

Testing of 1.2:



No.	Time	Source	Destination	Protocol	Length	Info
13	2024-02-06 03:1…	10.9.0.1	10.9.0.6	ICMP	42	Echo (ping) request id
14	2024-02-06 03:1…	10.9.0.6	10.9.0.1	ICMP	42	Echo (ping) reply id
15	2024-02-06 03:1…	02:42:0a:09:00:06	02:42:1c:b6:f6:d4	ARP	42	Who has 10.9.0.1? Tell
16	2024-02-06 03:1…	02:42:1c:b6:f6:d4	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:1c
17	2024-02-06 03:1…	02:42:1c:b6:f6:d4	Broadcast	ARP	42	Who has 10.9.0.6? Tell
18	2024-02-06 03:1…	02:42:0a:09:00:06	02:42:1c:b6:f6:d4	ARP	42	10.9.0.6 is at 02:42:0a

Wireshark allows us to visualize the communication between the source and the target destination. In the provided image, the non-spoofed packet is depicted, revealing the message's source as 10.9.0.1 (Line 13).



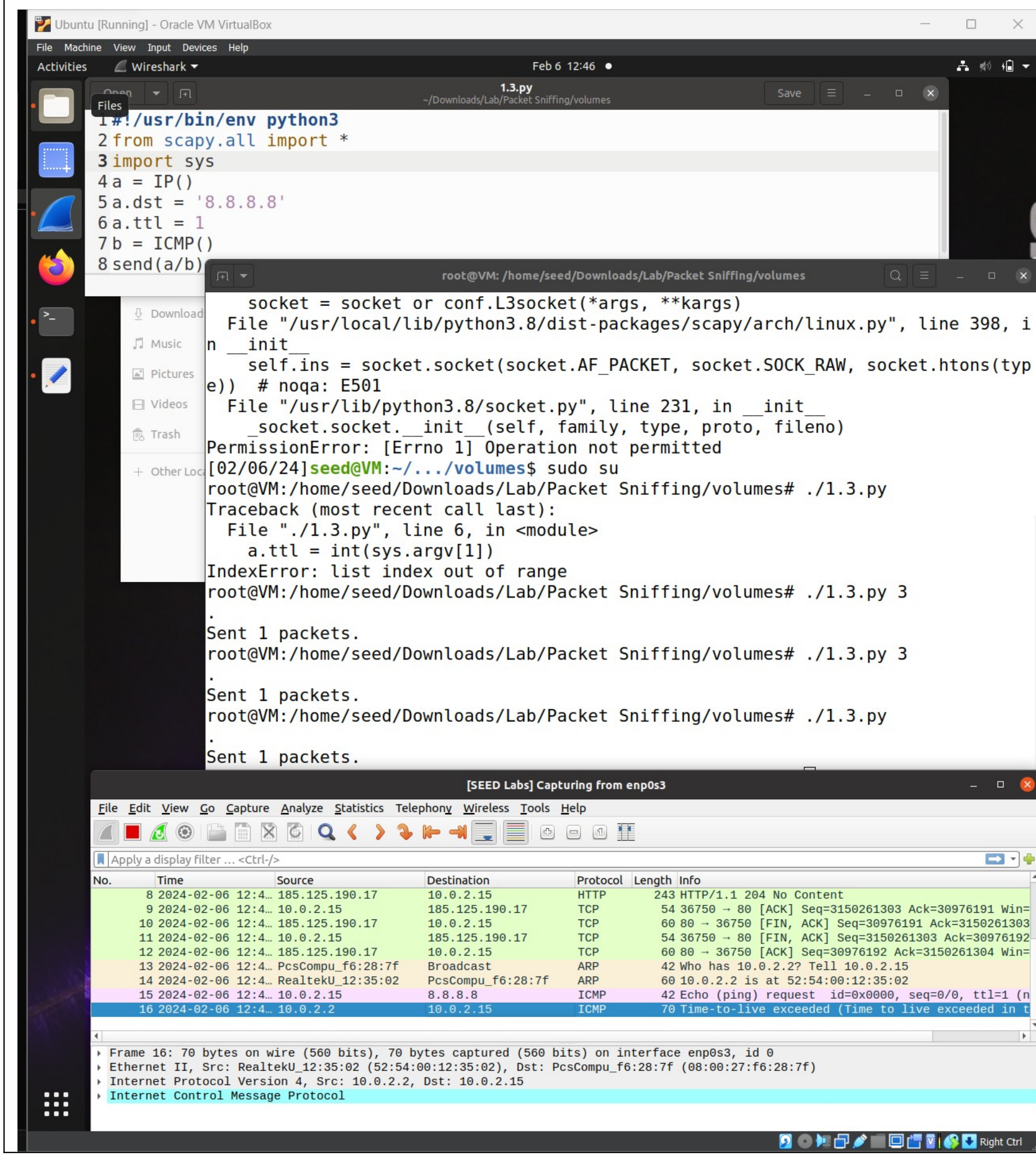
19	2024-02-06 03:1…	1.2.3.4	10.9.0.6	ICMP	42	Echo (ping) request id
20	2024-02-06 03:1…	10.9.0.6	1.2.3.4	ICMP	42	Echo (ping) reply id
21	2024-02-06 03:1…	02:42:0a:09:00:06	02:42:1c:b6:f6:d4	ARP	42	Who has 10.9.0.1? Tell
22	2024-02-06 03:1…	02:42:1c:b6:f6:d4	02:42:0a:09:00:06	ARP	42	10.9.0.1 is at 02:42:1c

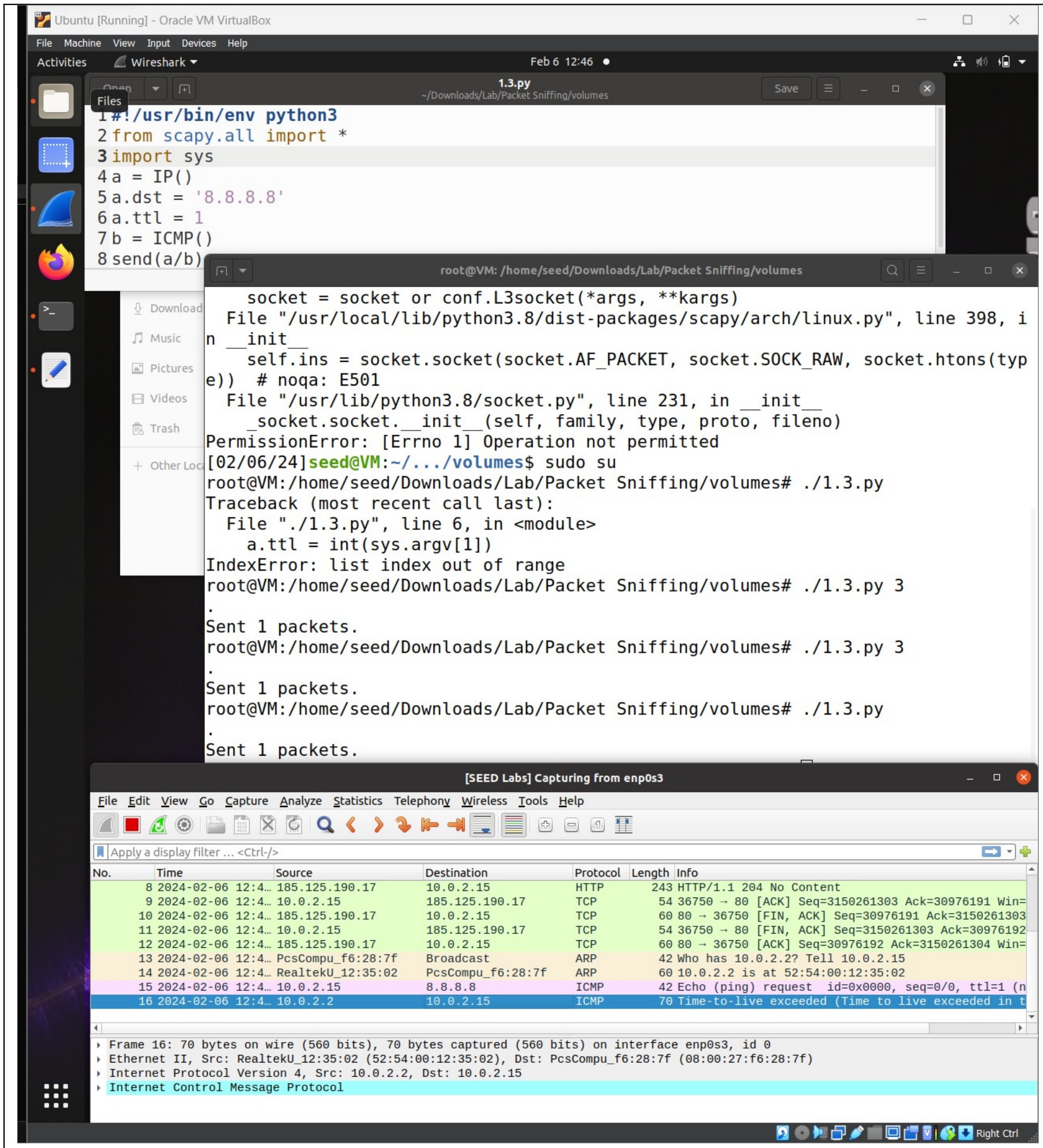
Upon executing the program, the source address was altered to 1.2.3.4 (Line 19), effectively concealing the original source address. It is interesting to see that this manipulation showcases the potential vulnerabilities and security implications associated with spoofing, highlighting the importance of understanding and securing network communications.

Task 1.3: Traceroute

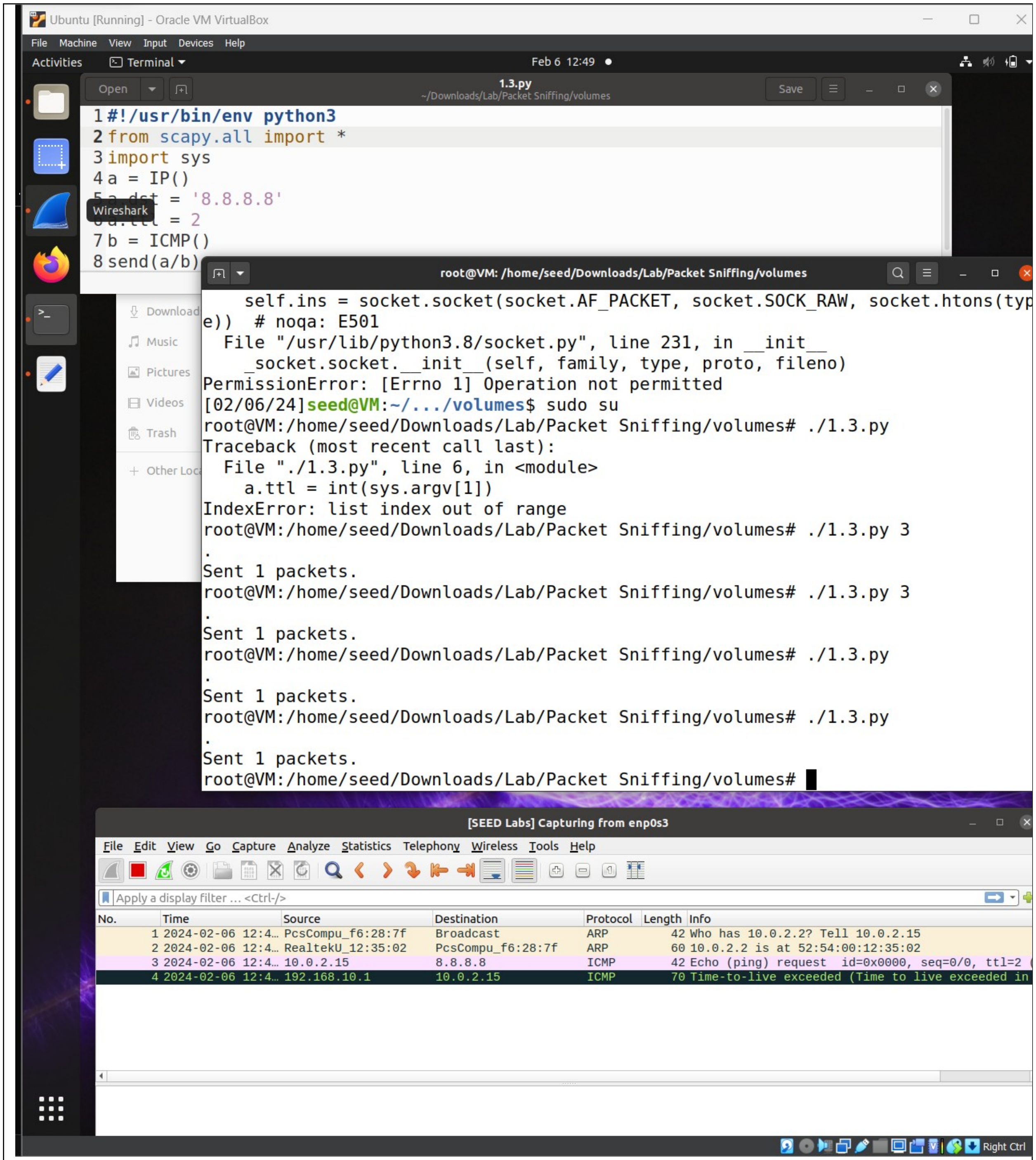
In this task our objective is to estimate the distance between the source (sender) and the destination (receiver), we do this by increasing the TTL (Time To Live) parameter from 1 till the package reach the destination, which will give us the probable route of the packet in the certain time frame and also the estimated distance between the attacking machine and Google's DNS Servers. We will use Wireshark to verify this.

In the figure below, the destination has an IP address of '8.8.8.8'. We used chmod a+x 1.3.py to grant permission for the python file and ./1.3.py to run the file.
While the TTL=1, Wireshark captured the traffic as TTL exceeded which means that the package did not reach the destination and we got a reply from a router.

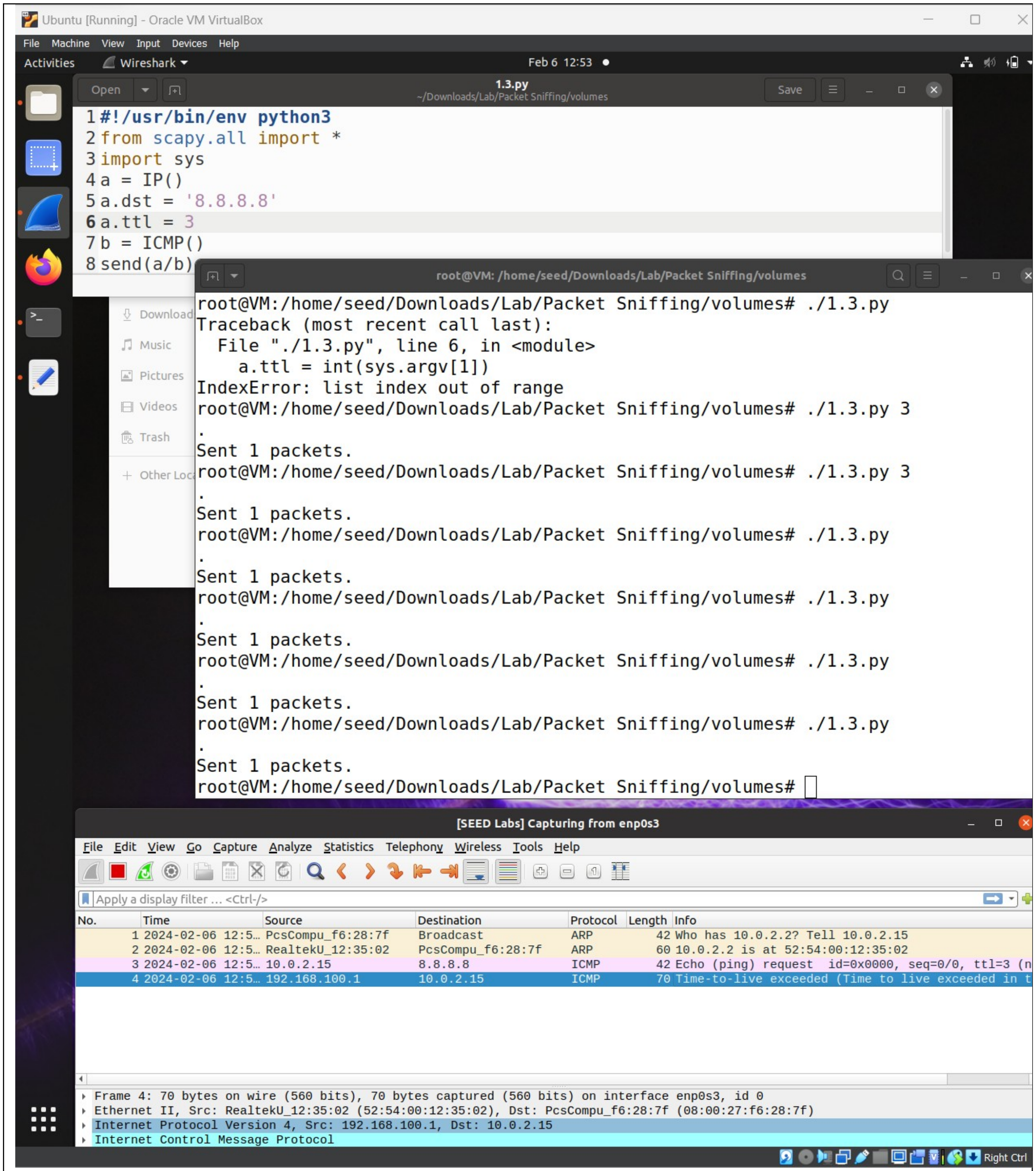




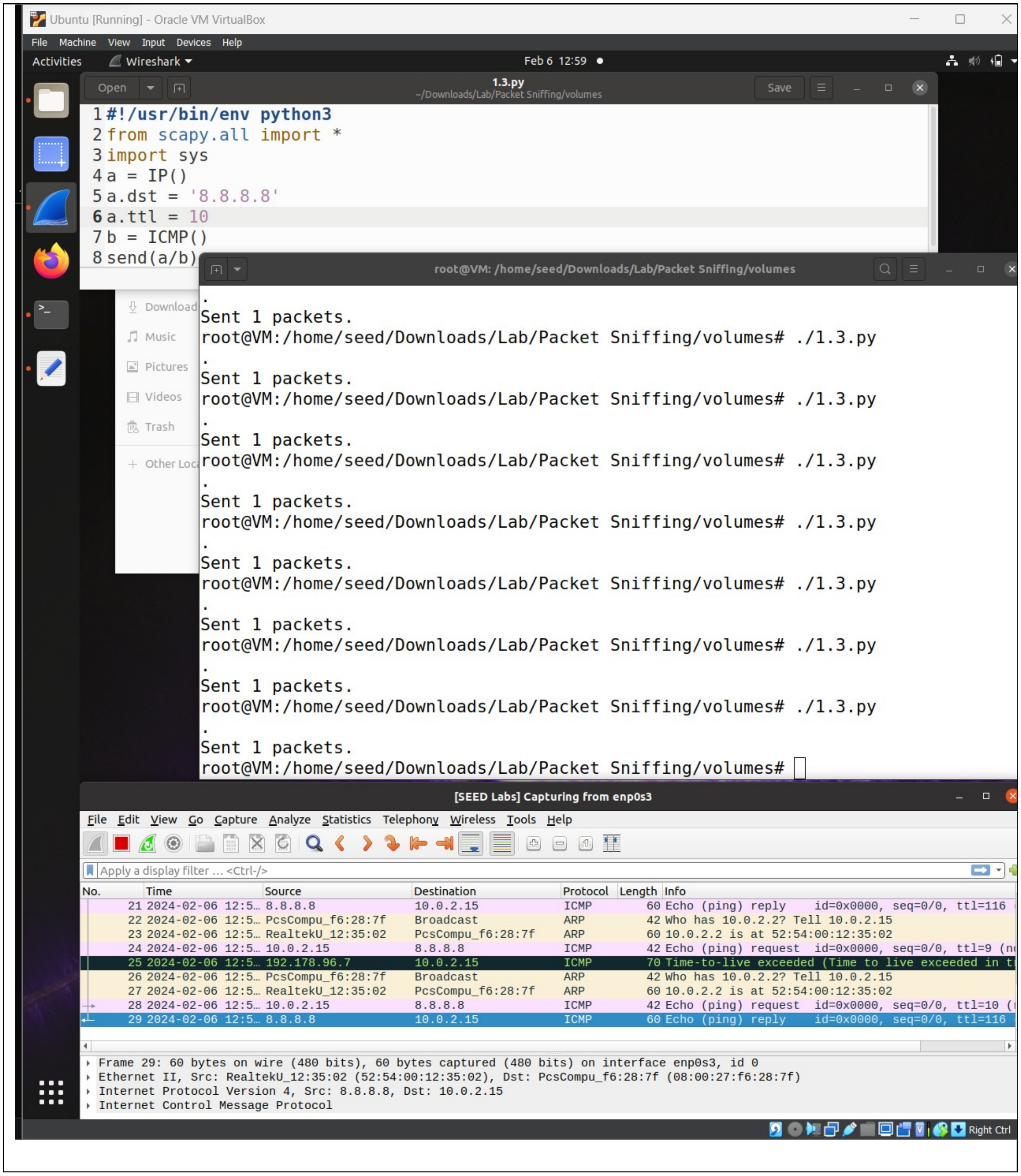
Even after Increasing the TTL to 2 and analyzing the captured network traffic we can see that TTL has exceeded the limit.



Even after Increasing the TTL to 3 and analyzing the captured network traffic we can see that TTL has exceeded the limit.



We keep increasing the TTL till when we get a reply from the source address being the same as the destination “8.8.8.8” (10 in our case), further analysis could be done to get rough estimation on the distance between the two devices.


Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Wireshark
Feb 6 12:59
Open
Save
1.3.py
~/Downloads/Lab/Packet Sniffing/volumes
1 #!/usr/bin/env python3
2 from scapy.all import *
3 import sys
4 a = IP()
5 a.dst = '8.8.8.8'
6 a.ttl = 10
7 b = ICMP()
8 send(a/b)
Download
Music
Pictures
Videos
Trash
+ Other Loc
root@VM: /home/seed/Downloads/Lab/Packet Sniffing/volumes
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes# ./1.3.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Lab/Packet Sniffing/volumes#
[SEED Labs] Capturing from enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
No. Time Source Destination Protocol Length Info
21 2024-02-06 12:5… 8.8.8.8 10.0.2.15 ICMP 60 Echo (ping) reply id=0x0000, seq=0/0, ttl=116
22 2024-02-06 12:5… PcsCompu_f6:28:7f Broadcast ARP 42 Who has 10.0.2.2? Tell 10.0.2.15
23 2024-02-06 12:5… RealtekU_12:35:02 PcsCompu_f6:28:7f ARP 60 10.0.2.2 is at 52:54:00:12:35:02
24 2024-02-06 12:5… 10.0.2.15 8.8.8.8 ICMP 42 Echo (ping) request id=0x0000, seq=0/0, ttl=9
25 2024-02-06 12:5… 192.178.96.7 10.0.2.15 ICMP 70 Time-to-live exceeded (Time to live exceeded in t
26 2024-02-06 12:5… PcsCompu_f6:28:7f Broadcast ARP 42 Who has 10.0.2.2? Tell 10.0.2.15
27 2024-02-06 12:5… RealtekU_12:35:02 PcsCompu_f6:28:7f ARP 60 10.0.2.2 is at 52:54:00:12:35:02
28 2024-02-06 12:5… 10.0.2.15 8.8.8.8 ICMP 42 Echo (ping) request id=0x0000, seq=0/0, ttl=10
29 2024-02-06 12:5… 8.8.8.8 10.0.2.15 ICMP 60 Echo (ping) reply id=0x0000, seq=0/0, ttl=116
Frame 29: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface enp0s3, id 0
Ethernet II, Src: RealtekU_12:35:02 (52:54:00:12:35:02), Dst: PcsCompu_f6:28:7f (08:00:27:f6:28:7f)
Internet Protocol Version 4, Src: 8.8.8.8, Dst: 10.0.2.15
Internet Control Message Protocol
Right Ctrl

Task 1.4: Sniffing and-then Spoofing

In this task, the Sniffing and spoofing techniques have been combined to be able to detect sent packets from a host in the same LAN (in this case the Host is Host A and the packets will be ICMP echo request packets), and then spoof and edit these packets source or destination.

The following Python code was used to detect and spoof the packets, for each IP we had to test for this task we made a customized my_filter variable to add it as a filter to our sniff() function.

```
root@VM:/volumes# cat task1-4.py
#!/bin/env python3
from scapy.all import *
def spoof_pkt(pkt):
        #Sniffing actual packet
        if ICMP in pkt and pkt[ICMP].type==8:
                print("Original packet........")
                print(f"source IP: {pkt[IP].src}")
                print(f"distination IP: {pkt[IP].dst}")
        #Spoofing
        ip= IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
        icmp= ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
        data= pkt[Raw].load
        newpkt= ip/icmp/data
        print("Spoofed paclet....")
        print(f"source IP: {newpkt[IP].src}")
        print(f"distination IP: {newpkt[IP].dst}")
        send(newpkt, verbose=0)
my_filter = 'icmp and host 1.2.3.4'
#my_filter = 'icmp and host 10.9.0.99'
#my_filter = 'icmp and host 8.8.8.8'
pkt= sniff(iface='br-b224dc88f0e9' , filter=my_filter , prn=spoof_pkt )

root@VM:/volumes# 
```

The program starts with a shebang #!/bin/env python3, which specifies the Python 3 interpreter to use.
We defined a spoof_pkt function to handle the incoming packets. It first checks if the packet is an ICMP packet (type 8 is an ICMP echo request). If true, it prints information about the original packet, such as source and destination IP. It then creates a new packet (newpkt) by swapping the source and destination IP addresses, creating a new ICMP packet with type 0 (ICMP echo reply), and copying the payload data from the original packet.
Then information about the spoofed packet is printed and sent using the send function from scapy.
In lines 18- 20. We made packet filter (my_filter) to capture ICMP packets with especific destination IP addresses (1.2.3.4, 10.9.0.99, and 8.8.8.8).
In line 21, We use the sniff function to capture packets on the specified network interface

which is in this case (br-b224dc88f0e9) based on the defined filter. When a packet is captured, the spoof_pkt function is called to process and spoof the packet.

First, through Host A, we try to ping 1.2.3.4, Which is a non-existing host on the internet.

First we run our program in the attacker machine, and ping 1.2.3.4 in the host machine:

```
[02/06/24]seed@VM:~/.../volumes$ dockps
d88ad44f7307  seed-attacker
d5fd3507245a  hostB-10.9.0.6
3c7127f484a7  hostA-10.9.0.5
[02/06/24]seed@VM:~/.../volumes$ docksh 3c7
root@3c7127f484a7:/# 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=54.5 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.8 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=22.1 ms
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 19.842/32.162/54.529/15.842 ms
root@3c7127f484a7:/# 
```

And below we can see the attacker machine sniffing the packets and spoofing them, which explains why host A is receiving the “fake” replies from host 1.2.3.4

```
root@VM:/volumes#  task1-4.py
Original packet........
source IP: 10.9.0.5
distination IP: 1.2.3.4
Spoofed paclet....
source IP: 1.2.3.4
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 1.2.3.4
Original packet........
source IP: 10.9.0.5
distination IP: 1.2.3.4
Spoofed paclet....
source IP: 1.2.3.4
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 1.2.3.4
Original packet........
source IP: 10.9.0.5
distination IP: 1.2.3.4
Spoofed paclet....
source IP: 1.2.3.4
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 1.2.3.4
```

If we observe more, we can see that the requests are being routed through the attacker machine through the ip route get 1.2.3.4 command: (10.9.0.1 is the IP of our attacker machine)

```
root@3c7127f484a7:/# ip route get 1.2.3.4
1.2.3.4 via 10.9.0.1 dev eth0 src 10.9.0.5 uid 0
    cache
```

```
root@VM:/volumes# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:fb:1b:32 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3
       valid_lft 76949sec preferred_lft 76949sec
    inet6 fe80::31a9:3e10:1f64:b336/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: br-b224dc88f0e9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:2f:82:c9:64 brd ff:ff:ff:ff:ff:ff
    inet 10.9.0.1/24 brd 10.9.0.255 scope global br-b224dc88f0e9
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2fff:fe82:c964/64 scope link
       valid_lft forever preferred_lft forever
```

Secondly, we attempt to sniff and spoof a non-existent host in the same LAN, 10.9.0.99
We can see that we couldn’t detect nor spoof the packets.

```
root@3c7127f484a7:/# 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 ---
8 packets transmitted, 0 received, +6 errors, 100% packet loss, time 7175ms
pipe 4
root@3c7127f484a7:/# arp -a
VM (10.9.0.1) at 02:42:2f:82:c9:64 [ether] on eth0
hostB-10.9.0.6.net-10.9.0.0 (10.9.0.6) at 02:42:0a:09:00:06 [ether] on eth0
? (10.9.0.99) at <incomplete> on eth0
root@3c7127f484a7:/#
```

Since both Host A and Host 10.9.0.99 should exist in the same LAN (in theory), the protocol used will be ARP, and since host 10.9.0.99 does not have a MAC address – the missing MAC address is shown as <incomplete>. Therefore, the arp prevents the packets from being sent through the LAN.

Finally, we try to ping a real existing host, 8.8.8.8, which is the primary DNS server for Google DNS.
Before we show what happens after running task1-4.py, we will first ping 8.8.8.8 normally.

```
root@3c7127f484a7:/# 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=56 time=20.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=20.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=22.8 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=56 time=18.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=56 time=31.0 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4014ms
rtt min/avg/max/mdev = 18.887/22.763/30.981/4.293 ms
root@3c7127f484a7:/# ip route get 8.8.8.8
8.8.8.8 via 10.9.0.1 dev eth0 src 10.9.0.5 uid 0
    cache
root@3c7127f484a7:/#
```

Now again with the sniff and spoof program running:

```
root@3c7127f484a7:/# 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=56 time=27.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=63.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=19.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=24.6 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=19.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=22.3 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=26.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=56 time=73.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=56 time=17.9 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=64 time=20.2 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=6 ttl=64 time=26.9 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=56 time=27.7 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=7 ttl=56 time=20.7 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=64 time=32.0 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 7 received, +7 duplicates, 0% packet loss, time 6022ms
rtt min/avg/max/mdev = 17.860/30.095/73.894/16.330 ms
root@3c7127f484a7:/# ip show route get 8.8.8.8
Object "show" is unknown, try "ip help".
root@3c7127f484a7:/# ip route get 8.8.8.8
8.8.8.8 via 10.9.0.1 dev eth0 src 10.9.0.5 uid 0
    cache
root@3c7127f484a7:/#
```

```
root@VM:/volumes# task1-4.py
Original packet........
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 8.8.8.8
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Original packet........
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 8.8.8.8
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Original packet........
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 8.8.8.8
distination IP: 10.9.0.5
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Spoofed paclet....
source IP: 10.9.0.5
distination IP: 8.8.8.8
Original packet........
```

We notice that there are duplicated reply packets, indicated by the (DUP!) at the end of some of the packets received. This happens when 2 packets with the same sequence number arrive at the same host. The most likely reason for this is that the destination exists and is also receiving the packets and sending echo replies just like our attacker machine. This might also be the case because in both cases, the packets are routed through our attacker machine (10.9.0.1) as shown through the ip route get 8.8.8.8 command.