CS 255 Computer Security

LAB 3: Packet Sniffing and Spoofing

Devasheesh Vaid
SID: 862395097
dvaid003@ucr.edu

Environment Setup using Containers:

$ dcbuild

All images are already being used in the containers; hence this step is skipped, as shown below.

After that we run *dcup* command, which creates two hosts and one seed attacker and starts those docker containers as shown in the screenshot below.

$ dcup

```
[11/23/23]seed@VM:~/.../Labsetup$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
[11/23/23]seed@VM:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Pulling attacker (handsonsecurity/seed-ubuntu:large)...
large: Pulling from handsonsecurity/seed-ubuntu
da7391352a9b: Pull complete
14428a6d4bcd: Pull complete
2c2d948710f2: Pull complete
b5e99359ad22: Pull complete
3d2251ac1552: Pull complete
1059cf087055: Pull complete
b2afee800091: Pull complete
c2ff2446bab7: Pull complete
4c584b5784bd: Pull complete
Digest: sha256:41efab02008f016a7936d9cadfbe8238146d07c1c12b39cd63c3e73a0297c07a
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large
Creating hostB-10.9.0.6 ... done
Creating hostA-10.9.0.5 ... done
Creating seed-attacker  ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostA-10.9.0.5 |  * Starting internet superserver inetd                 [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

After starting the containers and running the below command we see the list of active running containers.

$ dockps

```
[11/23/23]seed@VM:~$ dockps
b0607c98989d  seed-attacker
95c5fa082414  hostB-10.9.0.6
f58d70b67ac2  hostA-10.9.0.5
[11/23/23]seed@VM:~$
```

The IP address assigned to our VM is 10.9.0.1. We need to find the name of the corresponding network interface on our VM, because we need to use it in our programs. The interface name is the concatenation of *br-* and the ID of the network created by Docker. We use *ifconfig* to list network interfaces as below.

$ ifconfig

```
[11/23/23]seed@VM:~$ ifconfig
br-8a184f62243c: 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:f5ff:fe52:373  prefixlen 64  scopeid 0x20<link>
        ether 02:42:f5:52:03:73  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 41  bytes 4987 (4.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ca:97:32:7e  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
```

Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

Task 1.1: Sniffing Packets

First, I create a *sniffer.py* script with the below code. Then I update the interface with the interface id which we got with *ifconfig* command previously.

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-8a184f62243c', filter='icmp', prn=print_pkt)
```

Task 1.1 A

On running the above *sniffer.py* with root privileges, we are sniffing for ICMP packets. Since *ping* command sends ICMP *echo-request* and receives ICMP *echo-reply* packets, I ping one of the hosts, Host A in our case, at 10.9.0.5 to look for ICMP packets.

On pinging the host A, I see ICMP packets in the terminal window where we are sniffing for packets. Below is the *echo-request* packet from source (*src*) 10.9.0.1 to destination (*dst*) 10.9.0.5.

```
root@VM: /home/seed/Downloads/Labsetup
[11/25/23]seed@VM:~/.../Labsetup$ sudo su
root@VM:/home/seed/Downloads/Labsetup# ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:44:d9:2a:c0
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 22994
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xccbf
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x2bca
        id        = 0x1
        seq       = 0x1
###[ Raw ]###
           load      = '\xc8\xa8be\x00\x00\x00\x00\xe1R\x01\x00\x00\x00\x00\x
00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./01234567'
```

```
seed@VM: ~
[11/25/23]seed@VM:~$ 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=3.33 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.146 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.721 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.135 ms
^Z
[1]+  Stopped                 ping 10.9.0.5
[11/25/23]seed@VM:~$
```

The *echo-reply* packet which is the response packet from 10.9.0.5 to 10.9.0.1 as shown below.

```
root@VM: /home/seed/Downloads/Labsetup
00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:44:d9:2a:c0
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 55245
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x8ec4
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x33ca
        id        = 0x1
        seq       = 0x1
###[ Raw ]###
           load      = '\xc8\xa8be\x00\x00\x00\x00\xe1R\x01\x00\x00\x00\x00\x
00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./01234567'
```

```
seed@VM: ~
[11/25/23]seed@VM:~$ 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=3.33 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.146 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.721 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.135 ms
^Z
[1]+  Stopped                 ping 10.9.0.5
[11/25/23]seed@VM:~$
```

When I run the same sniffer.py script without the root privileges, we get the following *PermissionError*. Thus, root privileges are required for packet sniffing.

```
[11/25/23]seed@VM:~/.../Labsetup$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface='br-8a184f62243c', 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
[11/25/23]seed@VM:~/.../Labsetup$
```

Task 1.1B.

- **Capture only the ICMP packet:**

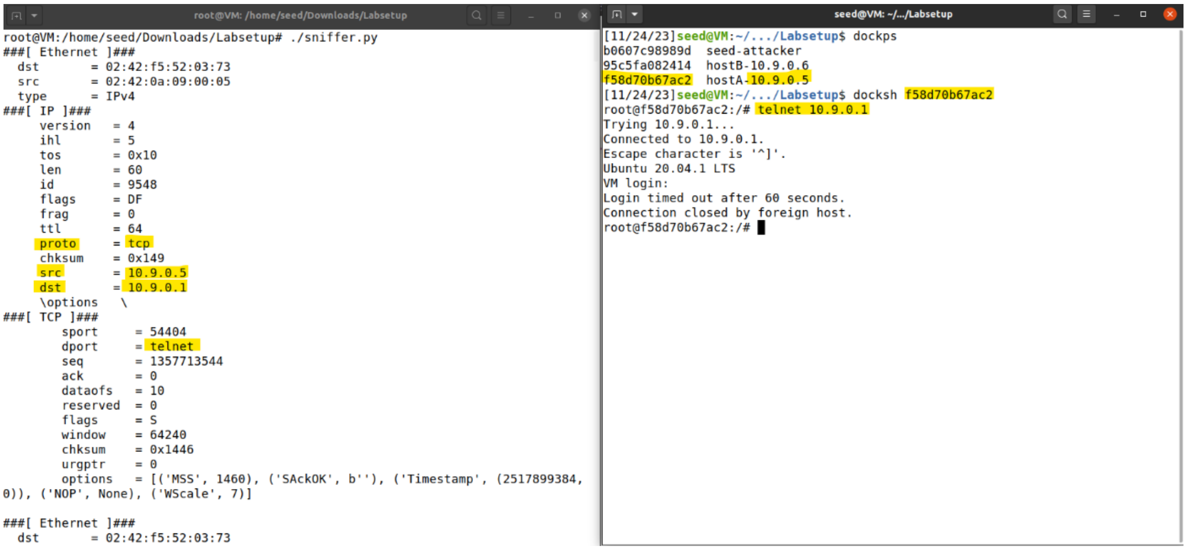
  This is done in the previous task- Task 1.1A

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

  For this I update the *sniffer.py* file and modify the *filter* attribute of the *sniff()* as below. Therefore, it will sniff for any packet using TCP and coming from host 10.9.0.5 (one of the host container IP) with a destination port 23.

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff(iface='br-8a184f62243c', filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
~
```

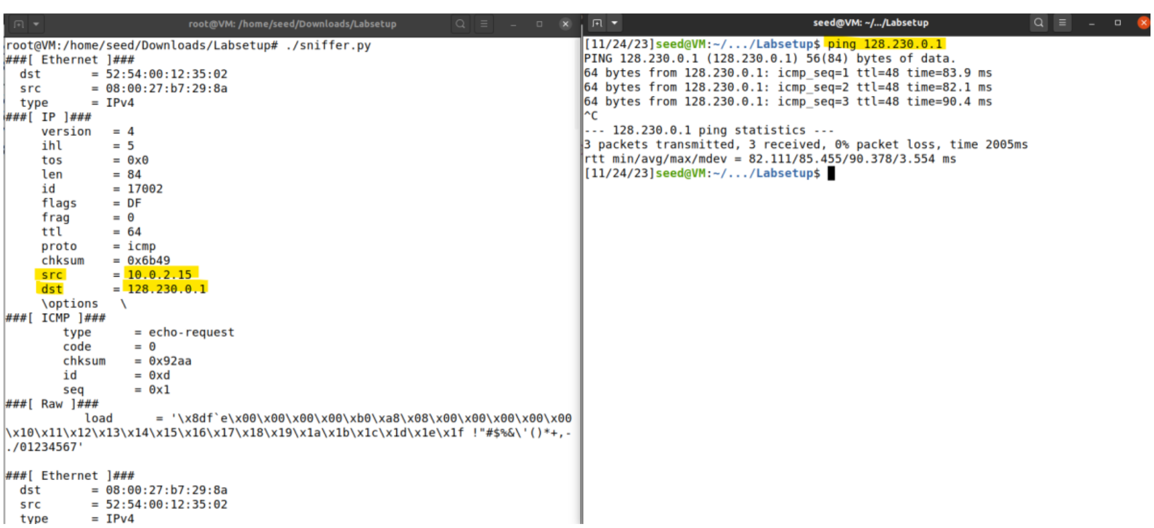
Since the destination port mentioned is 23, it is used by *TELNET*, for remote access. So, to do that, I log in to the host A docker container, with IP 10.9.0.5, and do a *telnet 10.9.0.1* to our VM as shown below.



- **Capture packets come from or to go to a particular subnet.**
  Again I start with modifying modify the *filter* attribute of the *sniff()* as below. I chose the subnet as *128.230.0.0/16* to monitor. We can pick any subnet, apart from the subnet that our VM is attached to. Below is the modified code.

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

def print_pkt(pkt):
    pkt.show()

pkt = sniff( filter='net 128.230.0.0/16', prn=print_pkt)
~
```

  So, when we *ping* 128.230.0.1, we capture the ICMP packets destined for 128.230.0.1 as shown below, thus sniffing the packets going to the subnet 128.230.0.0/16.



Task 1.2: Spoofing ICMP Packets

First, I create spoof.py file with the below contents. Here *a.src* is the spoofed IP and the packet is sent to *a.dst.* So, the destination IP would think that the packet is sent from the mentioned source 50.1.2.3, where it was actually sent by attacker at 10.9.0.1.

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

a = IP()

a.src='50.1.2.3'
a.dst = '10.9.0.5'

b = ICMP()

p = a/b
send(p)
~
```

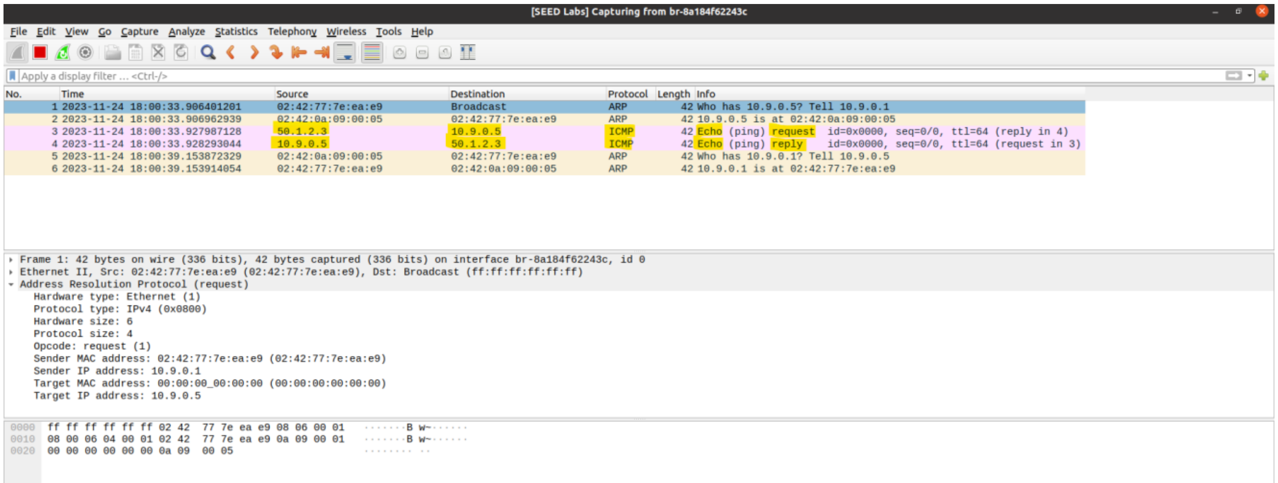
If I run the above code in python IDLE and see the *ls(a)* output, we see the source IP is modified, as below.

```
root@VM: /home/seed
>>>
>>> a = IP()
>>>
>>> a.src='50.1.2.3'
>>> a.dst = '10.9.0.5'
>>>
>>> b = ICMP()
>>>
>>> p = a/b
>>> ls(a)
version    : BitField  (4 bits)              = 4               (4)
ihl        : BitField  (4 bits)              = None            (None)
tos        : XByteField                      = 0               (0)
len        : ShortField                      = None            (None)
id         : ShortField                      = 1               (1)
flags      : FlagsField  (3 bits)            = <Flag 0 ()>     (<Flag 0 ()>
)
frag       : BitField  (13 bits)             = 0               (0)
ttl        : ByteField                       = 64              (64)
proto      : ByteEnumField                   = 0               (0)
chksum     : XShortField                     = None            (None)
src        : SourceIPField                   = '50.1.2.3'      (None)
dst        : DestIPField                     = '10.9.0.5'      (None)
options    : PacketListField                 = []              ([])
>>> send(p)
.
Sent 1 packets.
>>> send(p)
.
Sent 1 packets.
>>>
```

When I run this *spoof.py* script it sends out one packet, as shown below.

```
root@VM:/home/seed/Downloads/Labsetup# ls
docker-compose.yml  sniffer.py  spoof.py  volumes
root@VM:/home/seed/Downloads/Labsetup# cat spoof.py
#!/usr/bin/env python3
from scapy.all import *

a = IP()

a.src='50.1.2.3'
a.dst = '10.9.0.5'

b = ICMP()

p = a/b
send(p)
root@VM:/home/seed/Downloads/Labsetup# ./spoof.py
.
Sent 1 packets.
root@VM:/home/seed/Downloads/Labsetup#
```

We observe the activity on wireshark for the interface starting with '*br-…*' and see that there was indeed an ICMP *echo-request* packet sent from 50.1.2.3 to 10.9.0.5, which was later accepted with an ICMP *echo-reply* packet (row 3 & 4 below), where it was actually sent by attacker at 10.9.0.1.



Task 1.3: Traceroute

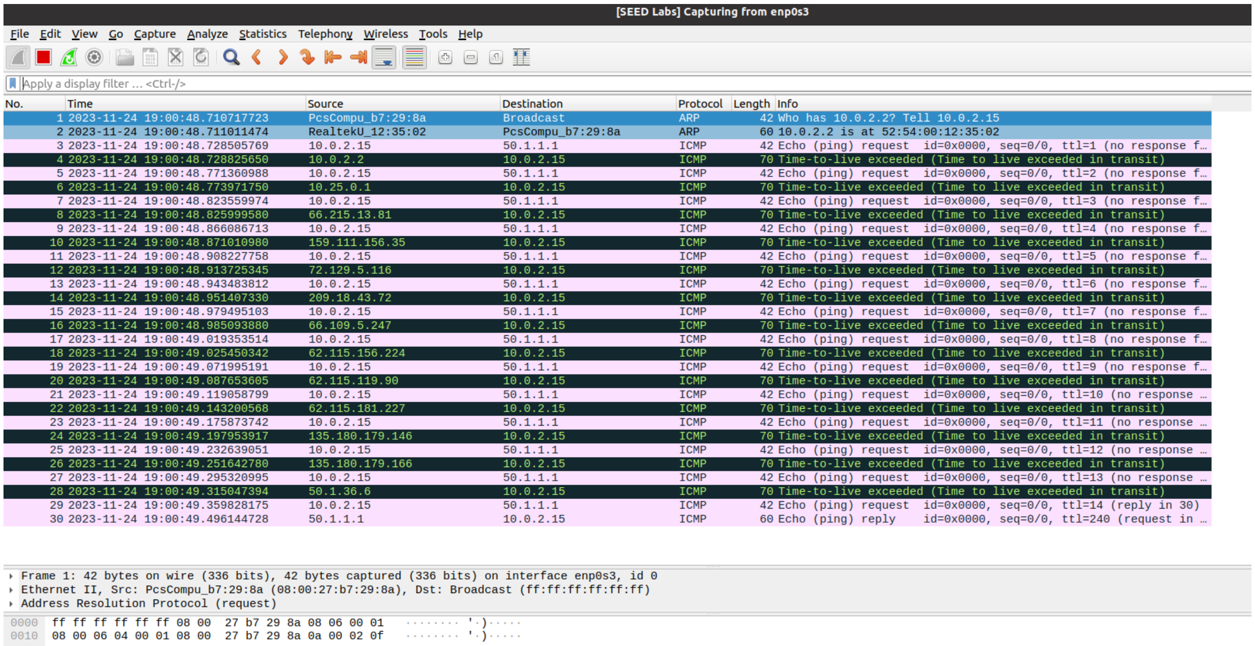
In this task we use Scapy to estimate the distance, in terms of number of routers, between the VM and a selected destination. In other words, we have to replicate the traceroute tool. So, I created the below traceroute.py script. In which the TTL starts from 1 and goes up till 64, which is the commonly used upper limit. I have chosen the destination as 50.1.1.1. We send the packet and receive the reply using scapy's *sr1()*. If the reply is not null, the router IP is printed and in case if the IP matches with the destination IP, we print the total hops and break out of the loop and exit the program.

```
#!/usr/bin/env python3
from scapy.all import *
a=IP()
a.dst = '50.1.1.1'
b=ICMP()

for i in range(1,65):
        a.ttl = i
        reply=sr1(a/b,verbose=False)
        if reply!=None:
                print(str(a.ttl)+'-> '+reply.src)
                if(a.dst==reply.src):
                        print("Total TTL or HOPs= ",i)
                        break
```

Below is the output of the program. It prints all the routers' IP addresses the packet goes through, on its way to the destination.

```
root@VM:/home/seed/Downloads/Labsetup# ./traceroute.py
1-> 10.0.2.2
2-> 10.25.0.1
3-> 66.215.13.81
4-> 159.111.156.35
5-> 72.129.5.116
6-> 209.18.43.72
7-> 66.109.5.247
8-> 62.115.156.224
9-> 62.115.119.90
10-> 62.115.181.227
11-> 135.180.179.146
12-> 135.180.179.166
13-> 50.1.36.6
14-> 50.1.1.1
Total TTL or HOPs=  14
root@VM:/home/seed/Downloads/Labsetup# 
```

The IP addresses can also be collected from the wireshark tool, as at each hop when the TTL expires, the router sends back an ICMP *TTL expire* packet back to the sender, along with its own IP. The wireshark output is as below.



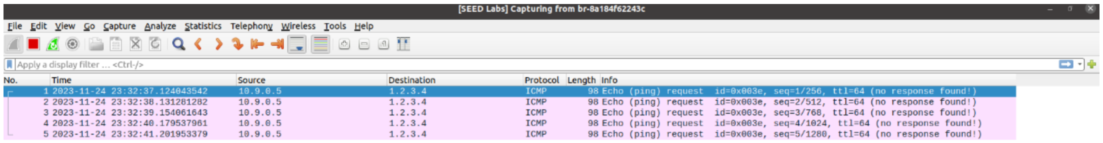
Task 1.4: Sniffing and-then Spoofing

For this task I created the below *sniffAndSpoof.py* script which sniffs ICMP packets and then constructs echo reply packets using fields information from the request packets. The code is shown below.

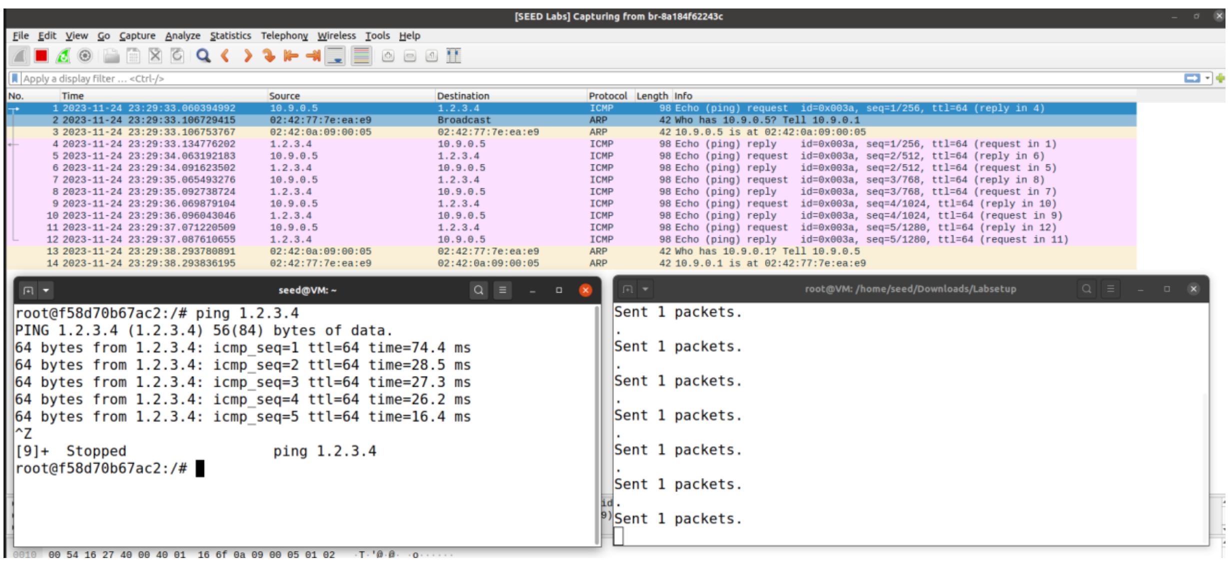
```
#!/usr/bin/env python3
from scapy.all import*

def print_pkt(pkt):
    a=IP(src=pkt[IP].dst, dst=pkt[IP].src)
    b=ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
    c=pkt[Raw].load
    reply=a/b/c
    send(reply)


pkt = sniff(iface='br-8a184f62243c', filter='icmp', prn=print_pkt)

~
```

- **ping 1.2.3.4 # a non-existing host on the Internet**

  Since this is a non- existing host on the Internet, when we first try to ping this IP without running our *sniffAndSpoof.py* script we get no reply from the destination as shown below.

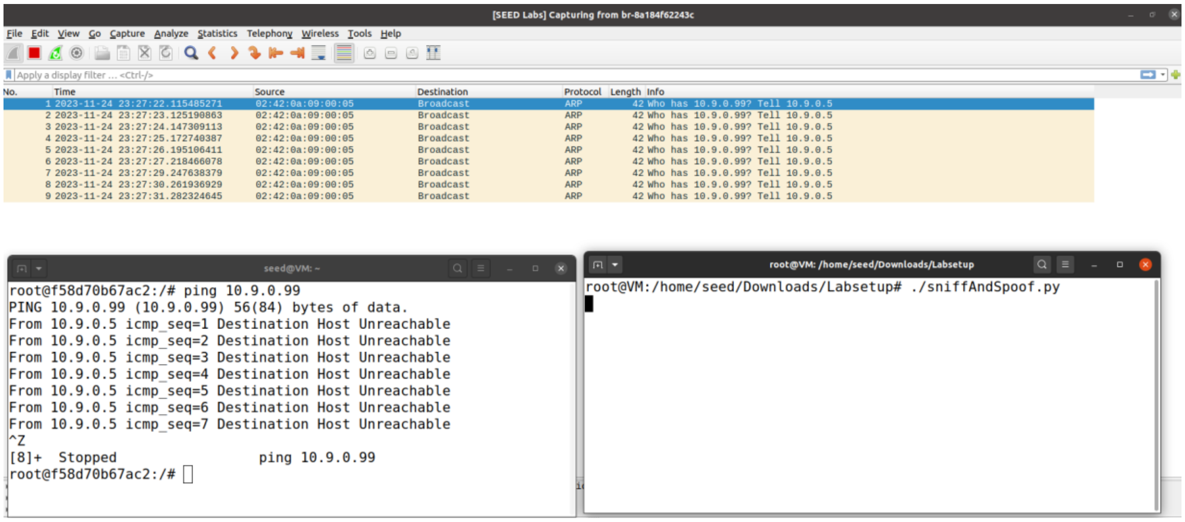


But when we again ping the same IP, but this time running our *sniffAndSpoof.py* script we see that to our *echo-request* packet, we get back *echo-response*. This response is not from the actual destination, since it is non existing, but spoofed from our attacker VM i.e. from 10.9.0.1. The terminal and wireshark results are shown below.



- **ping 10.9.0.99 # a non-existing host on the LAN**

  Since this is a nonexistent host on the same LAN, the ARP request is not resolved for this IP since no MAC address is attached to this IP. Hence, we get no ICMP echo request/response. Below is the screenshot.



- **ping 8.8.8.8 # an existing host on the Internet**

  Since this is an existing host on the Internet, we get a response back from 8.8.8.8 to our ping request, as shown below. But since the attacker at 10.9.0.1 is also sending ICMP *echo-reply* packets, for each *echo-request* we get two *echo-reply* packet as shown in wireshark screenshot below. Also, for the duplicate responses we see *(DUP!)* messages, indicating duplicate response, in the terminal window, as below.