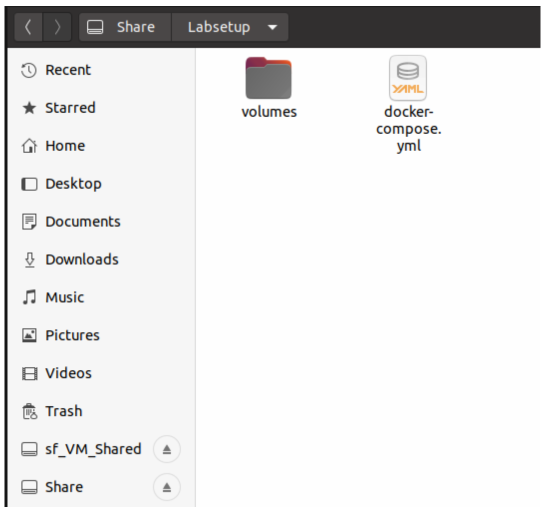
Computer Security

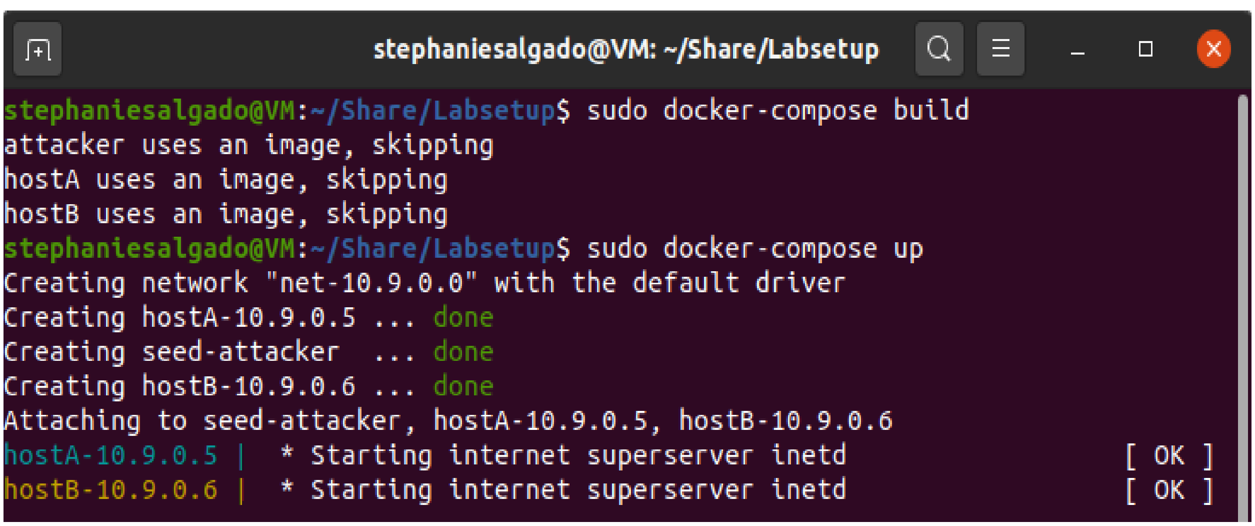
Lab 5 Demo: Packet Sniffing and Spoofing

Stephanie Salgado

Set up:



Located lab setup files in shared folder.



I began by navigating to project files and using “sudo docker-compose build” then “sudo docker-compose up” to build and start the container.

```
stephaniesalgado@VM:~/Share/Labsetup$ sudo docker ps
CONTAINER ID   IMAGE                               COMMAND                  CREATED          STATUS          PORTS     NAMES
f589e678e850   handsonsecurity/seed-ubuntu:large   "bash -c ' /etc/init…"   16 minutes ago   Up 16 minutes             hostB-10.9.0.6
d658205a3f98   handsonsecurity/seed-ubuntu:large   "/bin/sh -c /bin/bash"   16 minutes ago   Up 16 minutes             seed-attacker
2364903ac6b4   handsonsecurity/seed-ubuntu:large   "bash -c ' /etc/init…"   16 minutes ago   Up 16 minutes             hostA-10.9.0.5
stephaniesalgado@VM:~/Share/Labsetup$ sudo docker exec -it seed-attacker bash
root@VM:/#
```

In another terminal, I used “sudo docker ps” to find out the ID of the container. Then, “sudo docker exec -it seed-attacker bash” to start a shell on the container.

```
stephaniesalgado@VM: ~/Share/Labsetup
root@VM:/# ifconfig
br-05ff62bc2439: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.100.0.1  netmask 255.255.255.0  broadcast 10.100.0.255
        ether 02:42:d3:e3:a3:a0  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

br-0ebf74fb72d4: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.23.0.1  netmask 255.255.0.0  broadcast 172.23.255.255
        ether 02:42:f2:e6:0b:07  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

br-204b65107357: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.151.0.1  netmask 255.255.255.0  broadcast 10.151.0.255
        ether 02:42:c9:36:6c:a1  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

br-3a4b4583bca0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.153.0.1  netmask 255.255.255.0  broadcast 10.153.0.255
        ether 02:42:f5:7d:53:95  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

br-9a8a3d9d41a2: 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::42:94ff:fe6c:f61d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:94:6c:f6:1d  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 49  bytes 6026 (6.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

br-aeac8ab296fe: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.152.0.1  netmask 255.255.255.0  broadcast 10.152.0.255
        ether 02:42:7d:05:23:d1  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
```

I used “ifconfig” to find the name of the corresponding network interface on my VM. I made sure to look for IP address 10.9.0.1. That’s where I got, “br-9a8a3d9d41a2”.

```
stephaniesalgado@VM:~/Share/Labsetup$ sudo docker network ls
NETWORK ID     NAME                          DRIVER    SCOPE
2241646630cb   bridge                        bridge    local
b3581338a28d   host                          host      local
f438ea825ed2   internet-nano_default         bridge    local
204b65107357   internet-nano_net_151_net0    bridge    local
aeac8ab296fe   internet-nano_net_152_net0    bridge    local
3a4b4583bca0   internet-nano_net_153_net0    bridge    local
05ff62bc2439   internet-nano_net_ix_ix100    bridge    local
0ebf74fb72d4   map_default                   bridge    local
9a8a3d9d41a2   net-10.9.0.0                  bridge    local
77acecccbe26   none                          null      local
```

I made sure by using the “sudo docker network ls” command.

Task 1.1: Sniffing Packets

```
stephaniesalgado@VM: ~/Share/Labsetup
root@VM:/volumes# nano task1.1.py
root@VM:/volumes# cat task1.1.py
#!/usr/bin/env python3
from scapy.all import *

print("SNIFFING PACKETS.........")

def print_pkt(pkt):
  pkt.show()

pkt = sniff(iface='br-9a8a3d9d41a2', filter='icmp', prn=print_pkt)
root@VM:/volumes# chmod a+x task1.1.py
root@VM:/volumes# ./task1.1.py
SNIFFING PACKETS.........
```

I navigated into “volumes” on the attack container then created the “task1.1.py” file. After that, using “chmod a+x task1.1.py” I made the file executable. Since I’m already “root” by default in the attack container, I only had to use “./task1.1.py” to run the file.

```
stephaniesalgado@VM:~/Share/Labsetup$ sudo docker exec -it hostA-10.9.0.5 bash
root@2364903ac6b4:/# 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=0.139 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.050 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.062 ms
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.052 ms
64 bytes from 10.9.0.6: icmp_seq=6 ttl=64 time=0.052 ms
64 bytes from 10.9.0.6: icmp_seq=7 ttl=64 time=0.064 ms
```

I created network traffic by going to “hostA” and pinging “hostB” with command “ping 10.9.0.6”, where “10.9.0.6” is the IP corresponding to “hostB”. I let that run for a while.

```
stephaniesalgado@VM: ~/Share/Labsetup
64 bytes from 10.9.0.6: icmp_seq=95 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=96 ttl=64 time=0.067 ms
64 bytes from 10.9.0.6: icmp_seq=97 ttl=64 time=0.046 ms
64 bytes from 10.9.0.6: icmp_seq=98 ttl=64 time=0.044 ms
64 bytes from 10.9.0.6: icmp_seq=99 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=100 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=101 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=102 ttl=64 time=0.045 ms
64 bytes from 10.9.0.6: icmp_seq=103 ttl=64 time=0.093 ms
64 bytes from 10.9.0.6: icmp_seq=104 ttl=64 time=0.064 ms
64 bytes from 10.9.0.6: icmp_seq=105 ttl=64 time=0.053 ms
64 bytes from 10.9.0.6: icmp_seq=106 ttl=64 time=0.045 ms
64 bytes from 10.9.0.6: icmp_seq=107 ttl=64 time=0.053 ms
64 bytes from 10.9.0.6: icmp_seq=108 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=109 ttl=64 time=0.043 ms
64 bytes from 10.9.0.6: icmp_seq=110 ttl=64 time=0.060 ms
64 bytes from 10.9.0.6: icmp_seq=111 ttl=64 time=0.045 ms
64 bytes from 10.9.0.6: icmp_seq=112 ttl=64 time=0.048 ms
64 bytes from 10.9.0.6: icmp_seq=113 ttl=64 time=0.045 ms
^C
--- 10.9.0.6 ping statistics ---
113 packets transmitted, 113 received, 0% packet loss, time 114741ms
rtt min/avg/max/mdev = 0.041/0.055/0.139/0.016 ms
root@2364903ac6b4:/#
```

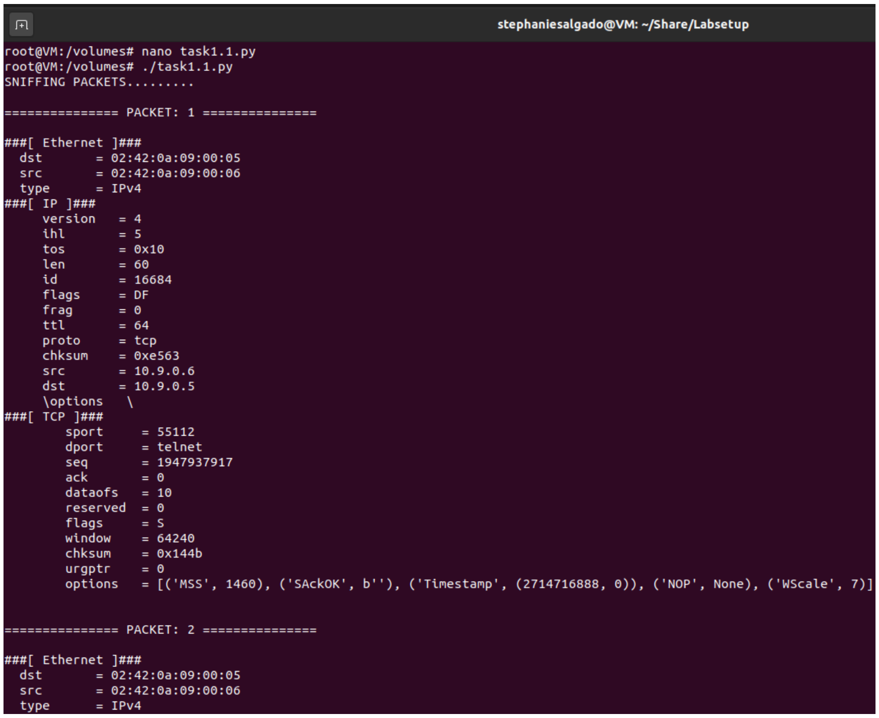
I stopped after 113 packets.

```
root@VM:/volumes# ./task1.1.py
SNIFFING PACKETS.........
###[ 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        = 21823
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xd14d
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x80cb
        id        = 0x1
        seq       = 0x1
###[ Raw ]###
           load      = '\xf0o\xd9e\x00\x00\x00\x00\xee\x89\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x1
8\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

The results looked like this in the attack container. Since the lab asked to capture only the ICMP packet, any TCP packet that comes from a particular IP and with a destination port number 23, and packets from or to a particular subnet, I modified “task.1.1.py”.

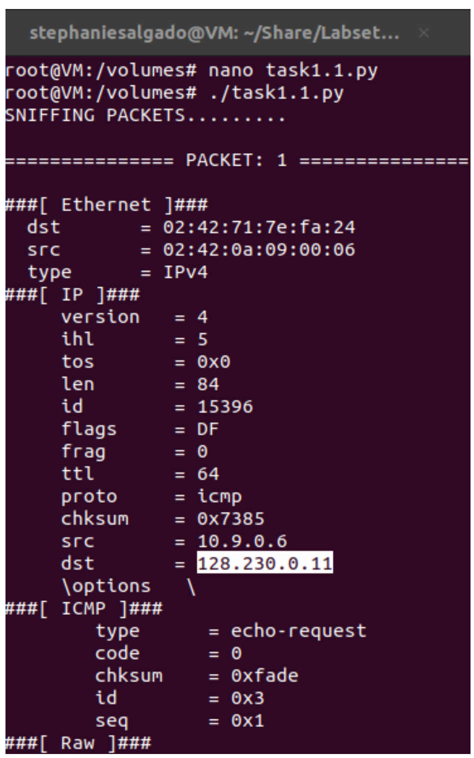
```
stephaniesalgado@VM: ~/Share/Labsetup
root@VM:/volumes# nano task1.1.py
root@VM:/volumes# ./task1.1.py
SNIFFING PACKETS.........
Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Protocol: 1


Source IP: 10.9.0.6
Destination IP: 10.9.0.5
Protocol: 1


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


Source IP: 10.9.0.5
Destination IP: 10.9.0.6
Protocol: 1


Source IP: 10.9.0.6
```

This was the output when I used the provided code.



```
stephaniesalgado@VM: ~/Share/Labsetup
root@VM:/volumes# nano task1.1.py
root@VM:/volumes# ./task1.1.py
SNIFFING PACKETS.........

============== PACKET: 1 ==============

###[ 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        = 16684
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xe563
     src       = 10.9.0.6
     dst       = 10.9.0.5
     \options   \
###[ TCP ]###
        sport     = 55112
        dport     = telnet
        seq       = 1947937917
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x144b
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (2714716888, 0)), ('NOP', None), ('WScale', 7)]


============== PACKET: 2 ==============

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
```

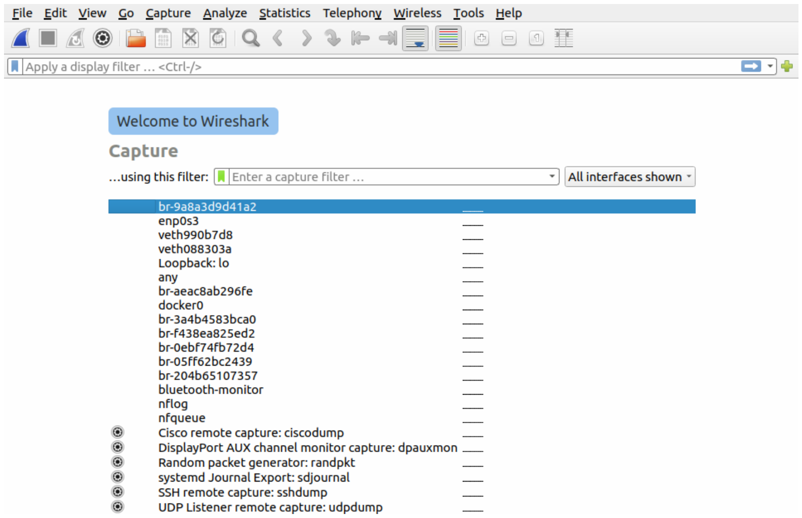
This is the output I got after filtering with “filter='tcp && src host 10.9.06 && dst port 23'”.



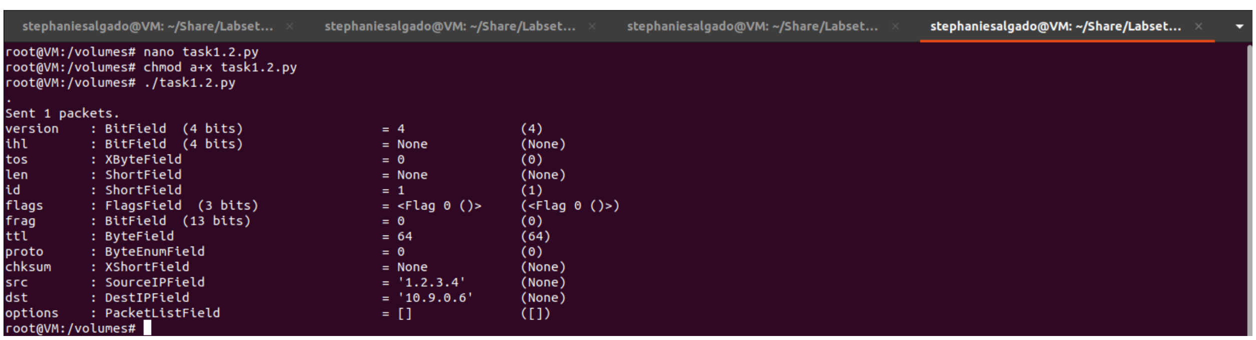
```
stephaniesalgado@VM: ~/Share/Labset...
root@VM:/volumes# nano task1.1.py
root@VM:/volumes# ./task1.1.py
SNIFFING PACKETS.........

============== PACKET: 1 ==============

###[ Ethernet ]###
  dst       = 02:42:71:7e:fa:24
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 15396
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7385
     src       = 10.9.0.6
     dst       = 128.230.0.11
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xfade
        id        = 0x3
        seq       = 0x1
###[ Raw ]###
```

This was the output after filtering with “filter='net 128.230.0.0/16'” for subnet.

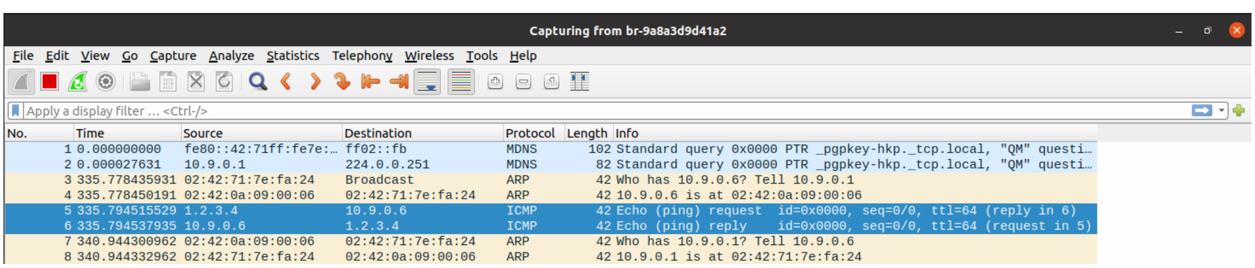
Task 1.2: Spoofing ICMP Packets



For this task, we will be using Wireshark to observe whether our request will be accepted by the receiver. I began by making sure I selected the correct interface (br-9a8a3d9d41a2).



I modified the provided task 1.2 code then made it executable and ran it.



Then I went to Wireshark to check if the request was accepted. I verified an echo reply packet was sent.

Task 1.3: Traceroute

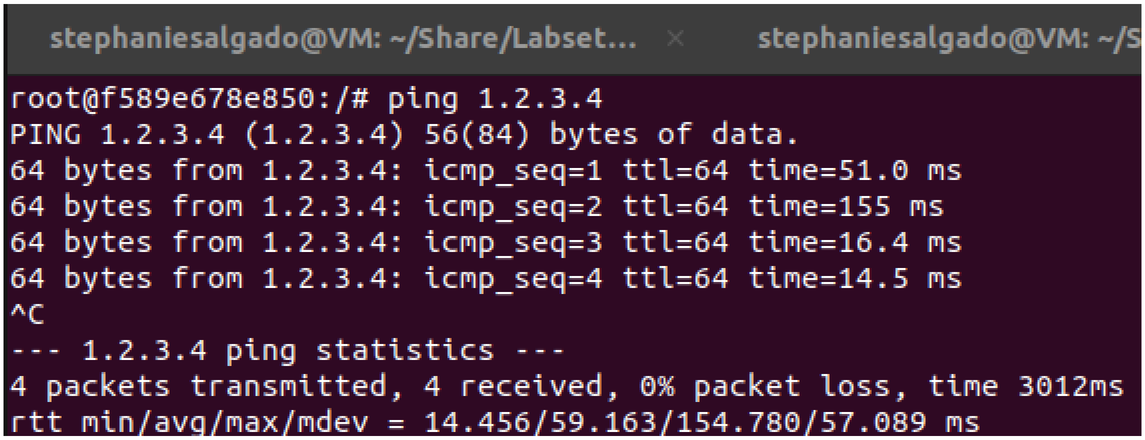
```
stephaniesalgado@VM: ~/Share/Labset...
root@VM:/volumes# touch task1.3.py
root@VM:/volumes# nano task1.3.py
root@VM:/volumes# chmod a+x task1.3.py
root@VM:/volumes# ./task1.3.py
1 10.0.2.2
2 192.168.1.1
3 70.94.136.1
4 24.28.133.233
5 24.175.34.16
6 24.175.32.149
7 24.175.32.156
8 131.150.63.17
9 108.170.225.147
10 142.251.253.25
11 8.8.4.4
root@VM:/volumes#
```

For this task, I asked for the destination IP, “8.8.4.4” which is the secondary DNS server for Google Public DNS. To accomplish this, I used the “sr1()” method from Scapy which listens and waits for a packet response. Eleven routers were reached before making it to the IP address destination.

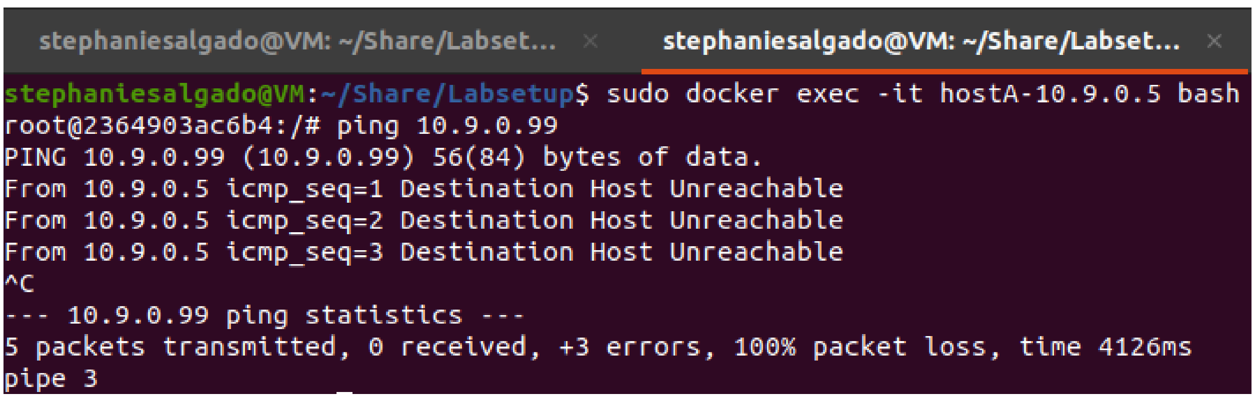
Task 1.4: Sniffing and-then Spoofing

```
stephaniesalgado@VM: ~/Share/Labset...
root@VM:/volumes# nano task1.4.py
root@VM:/volumes# chmod a+x task1.4.py
root@VM:/volumes# ./task1.4.py
Original Packet.........
Source IP:  10.9.0.6
Destination IP:  1.2.3.4
Spoofed Packet.........
Source IP: 1.2.3.4
Destination IP: 10.9.0.6
Spoofed Packet.........
Source IP: 10.9.0.6
Destination IP: 1.2.3.4
Original Packet.........
Source IP:  10.9.0.6
Destination IP:  1.2.3.4
```

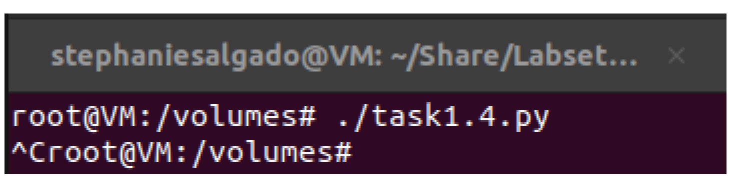
I start editing the “task1.4.py” file, then making it executable and running it. I leave it running then go to “hostB”.



From “hostB”, I “ping 1.2.3.4” which should be a non-existing host on the Internet. Packets were still transmitted/received. As can be seen on the other screenshot, the output was simply flipped IPs from “Original Packet” to “Spoofed Packet”.



I used “hostA” this time. Notice that there was “100% packet loss”.



The output was blank. This is because since the host does not exist on LAN, the packet can’t even be created.

```
stephaniesalgado@VM: ~/Share/Labset...
root@VM:/volumes# ./task1.4.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
Spoofed Packet.........
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
Spoofed Packet.........
Source IP: 10.9.0.5
Destination IP: 8.8.8.8
```

Once again, after modifying the file, I ran it and then went to “hostA” to “ping 8.8.8.8”, which is the existing host on the internet. The output seemed similar to the output I got for the non-existing host on the internet, in which the IPs on the original and spoofed packets switched.

```
stephaniesalgado@VM: ~/Share/Labset...    stephaniesalgado@VM: ~/Share/Labset...
root@2364903ac6b4:/# 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=33.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=40.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=18.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=35.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=17.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=38.2 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +3 duplicates, 0% packet loss, time 2007ms
rtt min/avg/max/mdev = 17.277/30.322/39.993/9.083 ms
```

Unlike when pinging “1.2.3.4”, which is the host that doesn’t exist on the internet, pinging “8.8.8.8” shows “DUP !” (duplicates). This basically serves as a response from the real host.

Summary:

Once again, this lab made good use of containers. I had never used Wireshark before so that's at least one take away from the lab. Scapy seems like a very valuable tool when working on things like this. I found it interesting how you can make a program and use the library to filter for specific sources and destinations. I was surprised by the fact that a library such as Scapy can be used to implement your own versions of traceroute. Furthermore, being able to tell whether a host is existing on the internet or LAN based off of the output from Task 1.4, means that using similar sniffing and spoofing programs can be incredibly valuable to attackers.