Scapy is a packet manipulation tool. Craft and send packets.

It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies.

Code:

```
init.py

#!/bin/bin/python
from scapy.all import *
a = IP()
a.show()
```

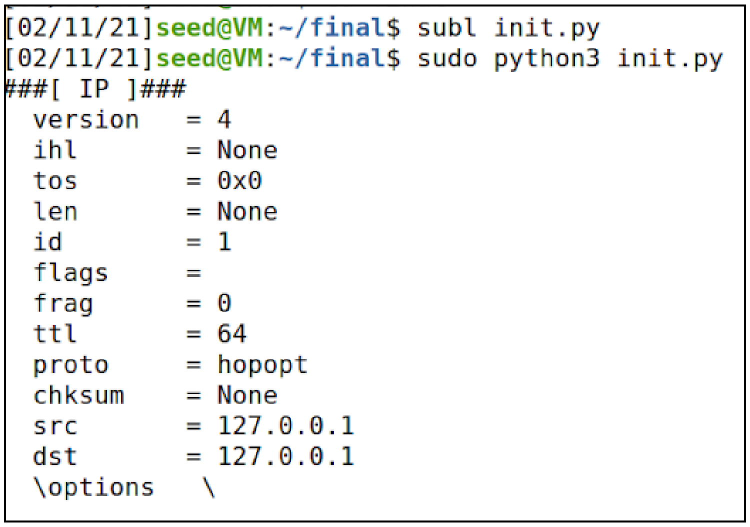
Explanation:

In this code (*init.py*), the `IP()` method creates and returns a new IP default and empty packet.

If we execute the `show()` command it will display the contents of the packet.

I used '*sudo*' the root privilege because it's required for packet manipulations.

Screenshot:



```
[02/11/21]seed@VM:~/final$ subl init.py
[02/11/21]seed@VM:~/final$ sudo python3 init.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   \
```

Task 1.1A.
In the above program, for each captured packet, the callback function print pkt() will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

Code:

```
sniffer.py
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-5b2e5b5961ac','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

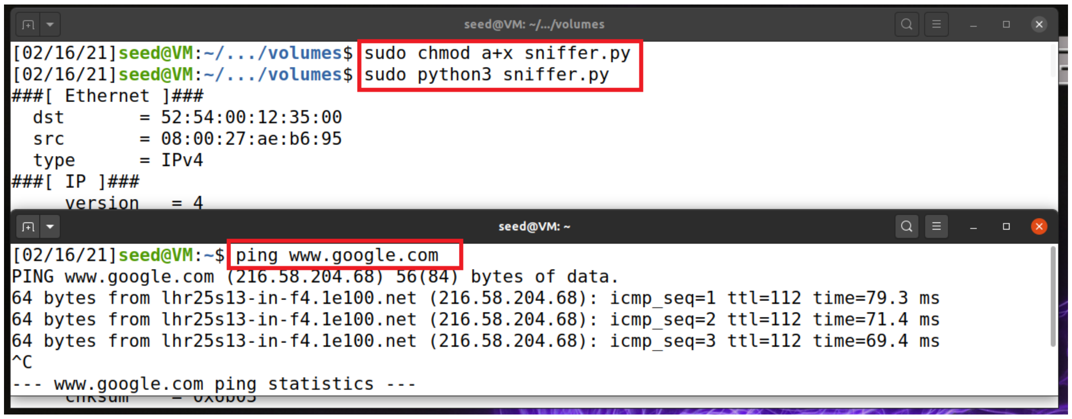
1. **About the code**: I picked these interfaces using the command 'ifconfig' in terminal and mentioned them in a list of the interfaces I wish to sniff the packets from. Using the filter, Scapy will show us only ICMP packets.
2. **About the question**: In order to run the program with the root privilege, I made this possible with the help of the following command: `'sudo chmod a+x sniffer.py'`, the 'a+x' means: execute + all. I used the 'ping' command with 'google.com' for ICMP echo request and reply packets. (We can see this case at 'Screenshot 1' below). I ran the program again, but this time, without using the root privilege. As we know, the 'sudo' method (superuser do) requires that the user be set up with the power to run "as root". So this time, we will run the program without 'sudo'. ('Screenshot 2' below).

Why did this happen?

Running the program using sudo allows us to see the whole network traffic in our given interfaces. As we can see in 'Screenshot 2', we got a **PermissionError - "operation not permitted"**.

So, if we want to sniff packets, we need root privileges to see the traffic and to capture the relevant packets.

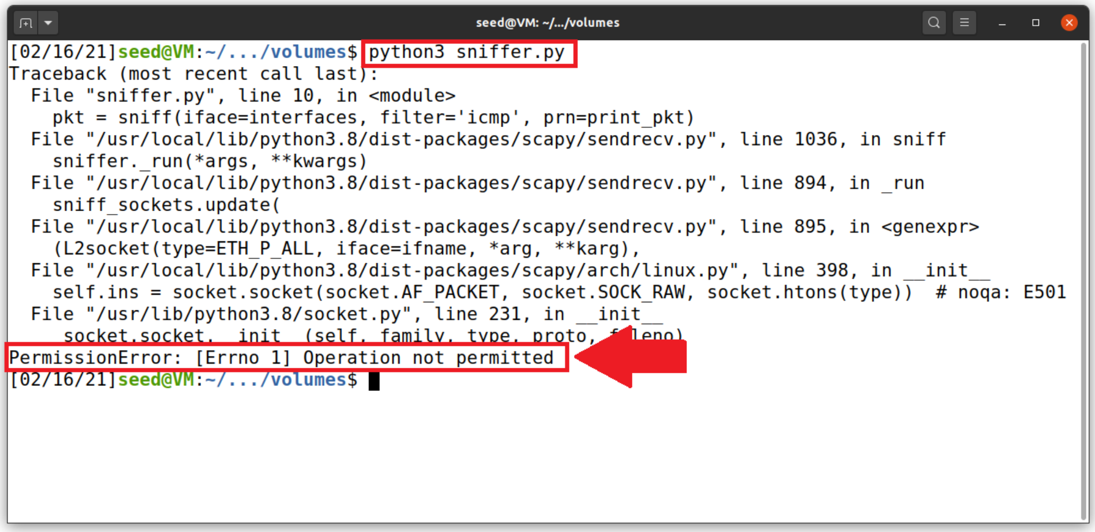
Screenshot 1:



```
[02/16/21]seed@VM:~/.../volumes$ sudo chmod a+x sniffer.py
[02/16/21]seed@VM:~/.../volumes$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ae:b6:95
  type      = IPv4
###[ IP ]###
     version   = 4
```

```
[02/16/21]seed@VM:~$ ping www.google.com
PING www.google.com (216.58.204.68) 56(84) bytes of data.
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=1 ttl=112 time=79.3 ms
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=2 ttl=112 time=71.4 ms
64 bytes from lhr25s13-in-f4.1e100.net (216.58.204.68): icmp_seq=3 ttl=112 time=69.4 ms
^C
--- www.google.com ping statistics ---
```

Screenshot 2:



```
[02/16/21]seed@VM:~/.../volumes$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 10, in <module>
    pkt = sniff(iface=interfaces, 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 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, 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
[02/16/21]seed@VM:~/.../volumes$
```

Task 1.1B.
Usually, when we sniff packets, we are only interested in certain types of packets.
We can do that by setting filters in sniffing.
Scapy's filter use the BPF (Berkeley Packet Filter) syntax;
you can find the BPF manual from the Internet.

Task 1.1B. - Capture only the ICMP packet.

Code:

sniff_only_icmp.py

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):

      if pkt[ICMP] is not None:
            if pkt[ICMP].type == 0 or pkt[ICMP].type == 8:
                  print("ICMP Packet=====")
                  print(f"\tSource: {pkt[IP].src}")
                  print(f"\tDestination: {pkt[IP].dst}")

                  if pkt[ICMP].type == 0:
                        print(f"\tICMP type: echo-reply")

                  if pkt[ICMP].type == 8:
                        print(f"\tICMP type: echo-request")

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

Explanation:

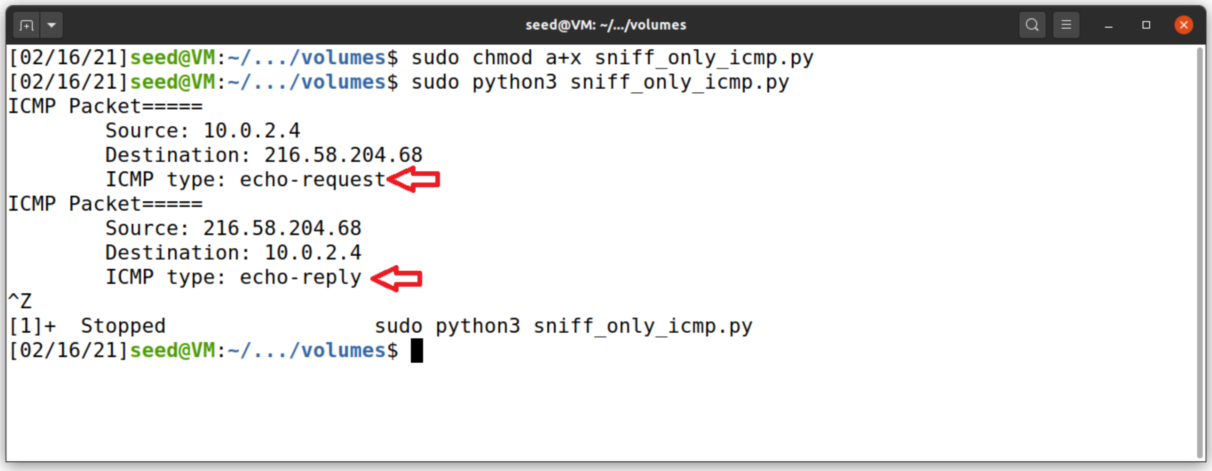
1. **About the code**: I used the same filter like the last code 'sniffer.py' which accepts the syntax of Berkeley Packet Filter. I could have used the method 'show()' but I wanted to print only the relevant details.
   So I printed only the source and destination IP and the ICMP type.

2. **About the question:** I sent a ping an IP of X='www.google.com'.
   This will generate an ICMP echo request packet. If X is alive, the program will receive an echo reply, and print out the response.

Wireshark:

A recording capture file named '*sniff_only_icmp.pdf*' is attached.

Screenshot:



```
[02/16/21]seed@VM:~/.../volumes$ sudo chmod a+x sniff_only_icmp.py
[02/16/21]seed@VM:~/.../volumes$ sudo python3 sniff_only_icmp.py
ICMP Packet=====
        Source: 10.0.2.4
        Destination: 216.58.204.68
        ICMP type: echo-request
ICMP Packet=====
        Source: 216.58.204.68
        Destination: 10.0.2.4
        ICMP type: echo-reply
^Z
[1]+  Stopped                 sudo python3 sniff_only_icmp.py
[02/16/21]seed@VM:~/.../volumes$ 
```

Task 1.1B. - Capture any TCP packet that comes from a particular IP and with a destination port number 23.

Code:

tcp_sniffer.py

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
	if pkt[TCP] is not None:
		print("TCP Packet=====")
		print(f"\tSource: {pkt[IP].src}")
		print(f"\tDestination: {pkt[IP].dst}")
		print(f"\tTCP Source port: {pkt[TCP].sport}")
		print(f"\tTCP Destination port: {pkt[TCP].dport}")

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='tcp port 23 and src host 10.0.2.4',
prn=print_pkt)
```

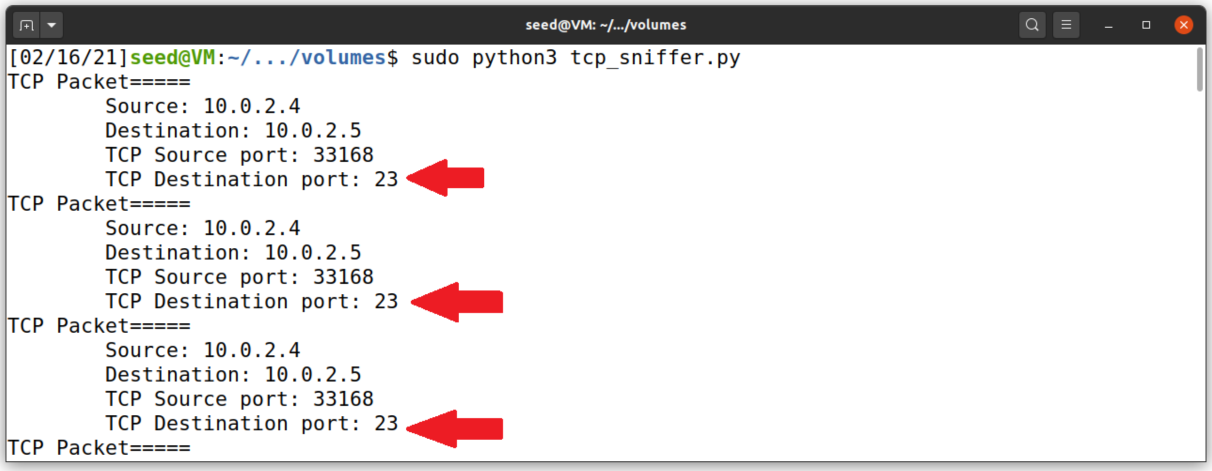
Explanation:

1. **About the code**: I filtered this time with `'tcp port 23 and src host 10.0.2.4'`.
   The syntax I used for this filter is from BPF syntax website.
   And again like the previous code, I printed only the relevant data for this question. That's why I didn't use 'show()'.
2. **About the question**:
   My default VM's IP is '10.0.2.4' and I sent it to '10.0.2.5' which is another VM that I used. This way, I sent a command 'telnet 10.0.2.5' from my regular VM to the second one, and the program 'tcp_sniffer.py' sniffed the TCP packets.
   Why telnet? - this protocol is used to establish a connection to TCP port number 23.

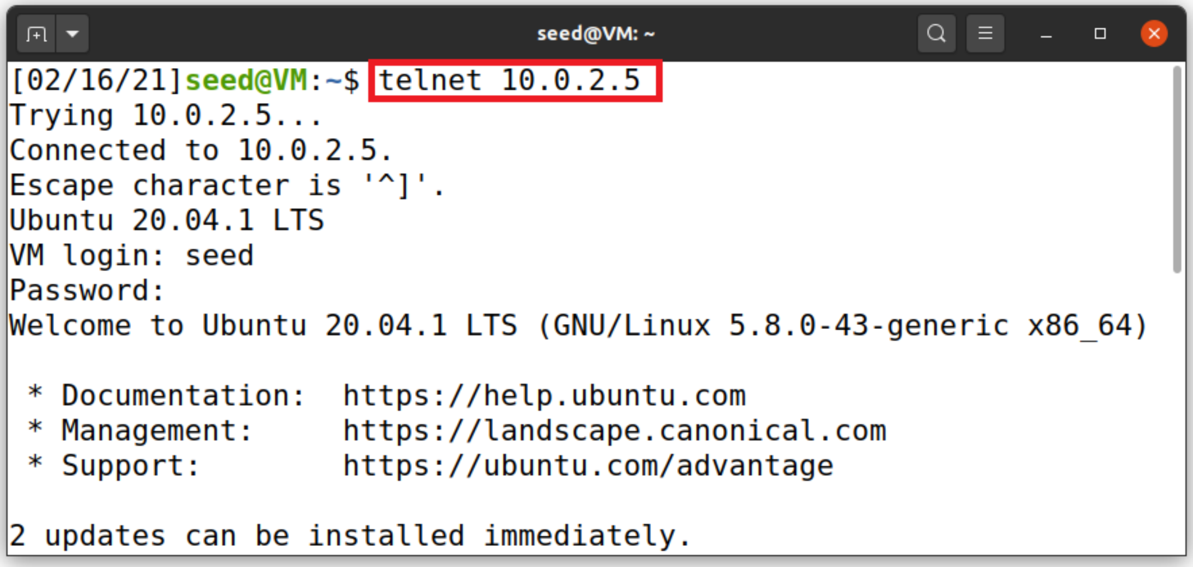
Wireshark:

A recording capture file named '*sniffer_tcp.pdf*' is attached.

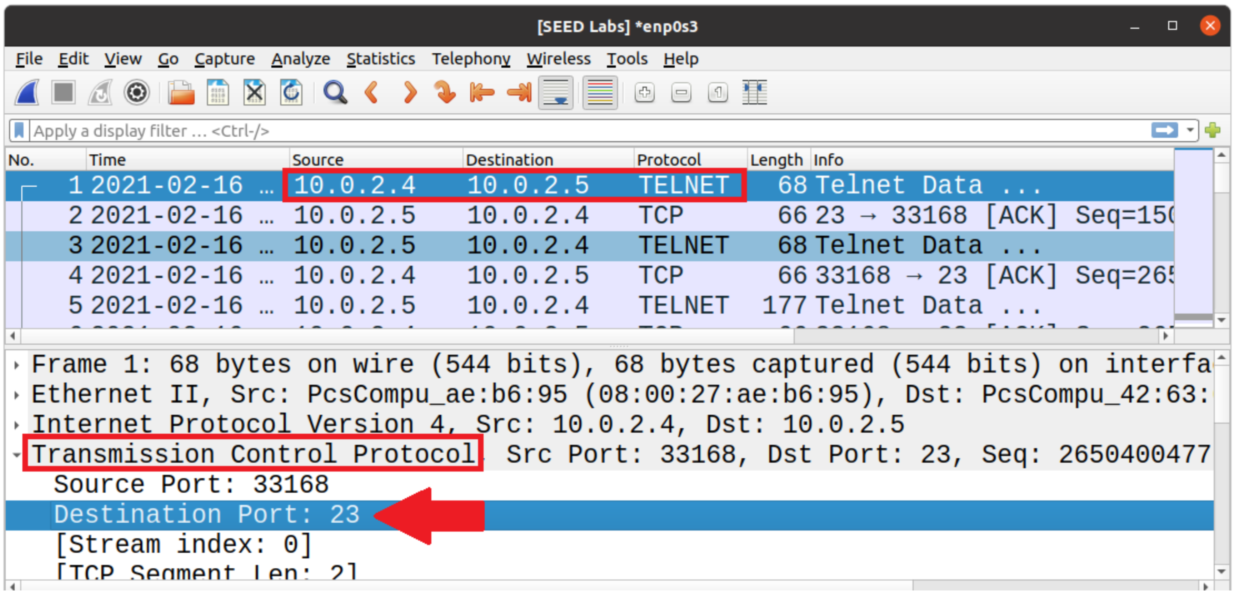
Screenshots:



```
[02/16/21]seed@VM:~/.../volumes$ sudo python3 tcp_sniffer.py
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
        Source: 10.0.2.4
        Destination: 10.0.2.5
        TCP Source port: 33168
        TCP Destination port: 23
TCP Packet=====
```



```
[02/16/21]seed@VM:~$ telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
VM login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.8.0-43-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

2 updates can be installed immediately.
```


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-02-16 …	10.0.2.4	10.0.2.5	TELNET	68	Telnet Data ...
2	2021-02-16 …	10.0.2.5	10.0.2.4	TCP	66	23 → 33168 [ACK] Seq=15(
3	2021-02-16 …	10.0.2.5	10.0.2.4	TELNET	68	Telnet Data ...
4	2021-02-16 …	10.0.2.4	10.0.2.5	TCP	66	33168 → 23 [ACK] Seq=265
5	2021-02-16 …	10.0.2.5	10.0.2.4	TELNET	177	Telnet Data ...

```
› Frame 1: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interfa
› Ethernet II, Src: PcsCompu_ae:b6:95 (08:00:27:ae:b6:95), Dst: PcsCompu_42:63:
› Internet Protocol Version 4, Src: 10.0.2.4, Dst: 10.0.2.5
⌄ Transmission Control Protocol, Src Port: 33168, Dst Port: 23, Seq: 2650400477
    Source Port: 33168
    Destination Port: 23
    [Stream index: 0]
    [TCP Segment Len: 2]
```

Task 1.1B. - Capture packets comes from or to go to a particular subnet.
You can pick any subnet, such as 128.230.0.0/16;
you should not pick the subnet that your VM is attached to.

Code:

subnet_sniffer.py

```python
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    pkt.show()

interfaces = ['br-e12cb9117793','enp0s3','lo']
pkt = sniff(iface=interfaces, filter='dst net 128.230.0.0/16', prn=print_pkt)
```

send_subnet_packet.py

```python
from scapy.all import *
ip=IP()
ip.dst='128.230.0.0/16'
send(ip,4)
```

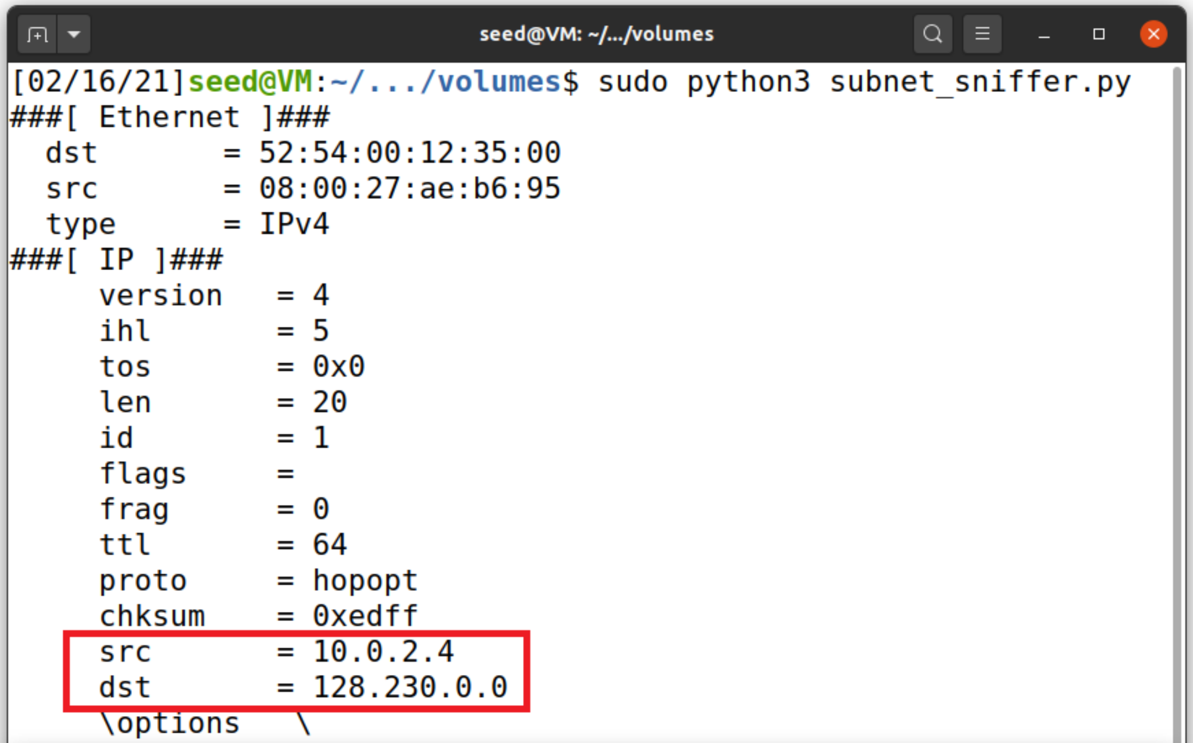
Explanation:

1. **About the code**: in 'subnet_sniffer.py' I picked the filter using Berkeley Packet Filter syntax: `'dst net 128.230.0.0/16'` .
   'dst' means a possible direction.
   'net' returns true if there is a possible type of net, in this case, a subnet which represented in the task details. I wrote the program 'send_subnet_packet.py' the same like the example shown at page 5 of the task file.
2. **About the question**:
   A packet was sent to a particular subnet and the program sniffed only packets that were sent from source '10.0.2.4' to destination IP from the other subnet.

Wireshark:

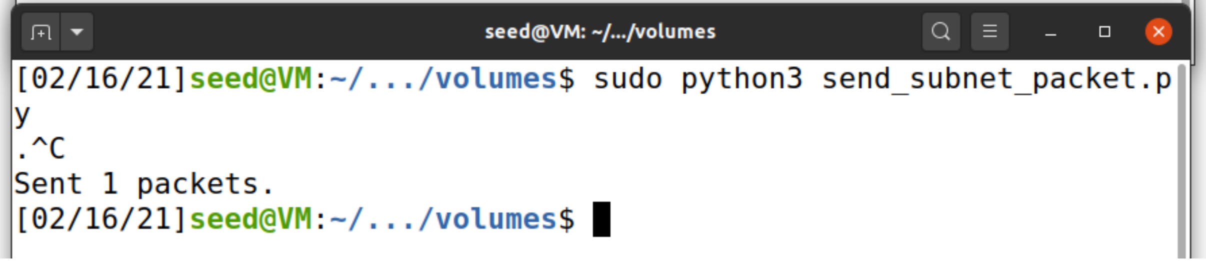
A recording capture file named '*subnet_sniffer.pdf*' is attached.

Screenshots:



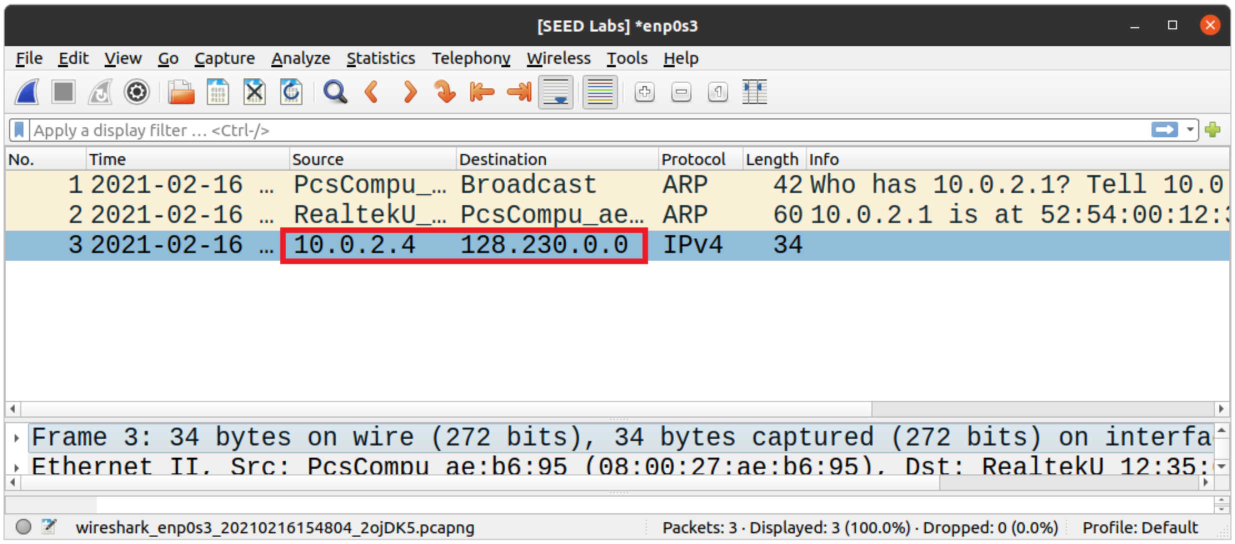
seed@VM: ~/.../volumes

```
[02/16/21]seed@VM:~/.../volumes$ sudo python3 subnet_sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ae:b6:95
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 20
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = hopopt
     chksum    = 0xedff
     src       = 10.0.2.4
     dst       = 128.230.0.0
     \options   \
```



seed@VM: ~/.../volumes

```
[02/16/21]seed@VM:~/.../volumes$ sudo python3 send_subnet_packet.p
y
.^C
Sent 1 packets.
[02/16/21]seed@VM:~/.../volumes$
```



[SEED Labs] *enp0s3

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.  Time            Source         Destination    Protocol Length Info
   1 2021-02-16 …  PcsCompu_…   Broadcast       ARP        42 Who has 10.0.2.1? Tell 10.0
   2 2021-02-16 …  RealtekU_…   PcsCompu_ae…    ARP        60 10.0.2.1 is at 52:54:00:12:
   3 2021-02-16 …  10.0.2.4     128.230.0.0     IPv4       34
```

```
› Frame 3: 34 bytes on wire (272 bits), 34 bytes captured (272 bits) on interfa
› Ethernet II, Src: PcsCompu_ae:b6:95 (08:00:27:ae:b6:95), Dst: RealtekU_12:35:
```

wireshark_enp0s3_20210216154804_2ojDK5.pcapng   Packets: 3 · Displayed: 3 (100.0%) · Dropped: 0 (0.0%)   Profile: Default

Task 1.2: Spoofing ICMP Packets
Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

What is **packet spoofing**? - When a normal user sends a packet, the OS usually does not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). The OS 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.**

Code:

icmp_spoofing.py

```
from scapy.all import *

a = IP()
a.src = '1.2.3.4'
a.dst = '10.0.2.6'
send(a/ICMP())
ls(a)
```

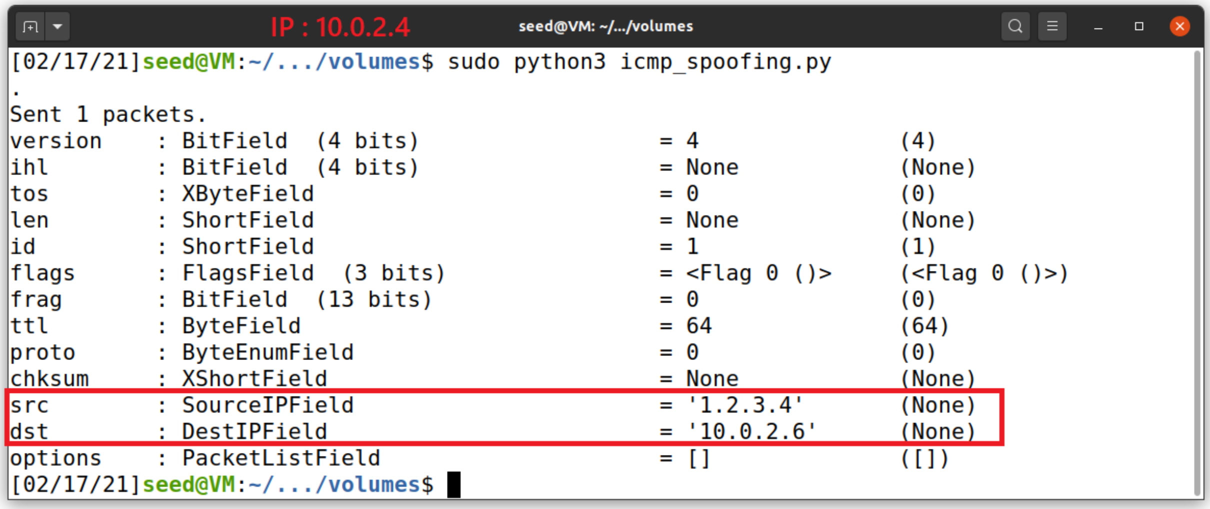
Explanation:

1. **About the code**: I used another VM (IP destination '10.0.2.6') and sent an ICMP packet using a random IP source '1.2.3.4'.
2. **About the question**: I spoofed ICMP echo request packet, and sent it to another VM on the same subnet.
   I used Wireshark to observe whether our request will be accepted by the receiver.
   Using scapy library, the source ip was overwritten with our own ip: 1.2.3.4 and sent the packet to destination 10.0.2.6; the packet was received by 10.0.2.6 and sent an echo reply back to 1.2.3.4.

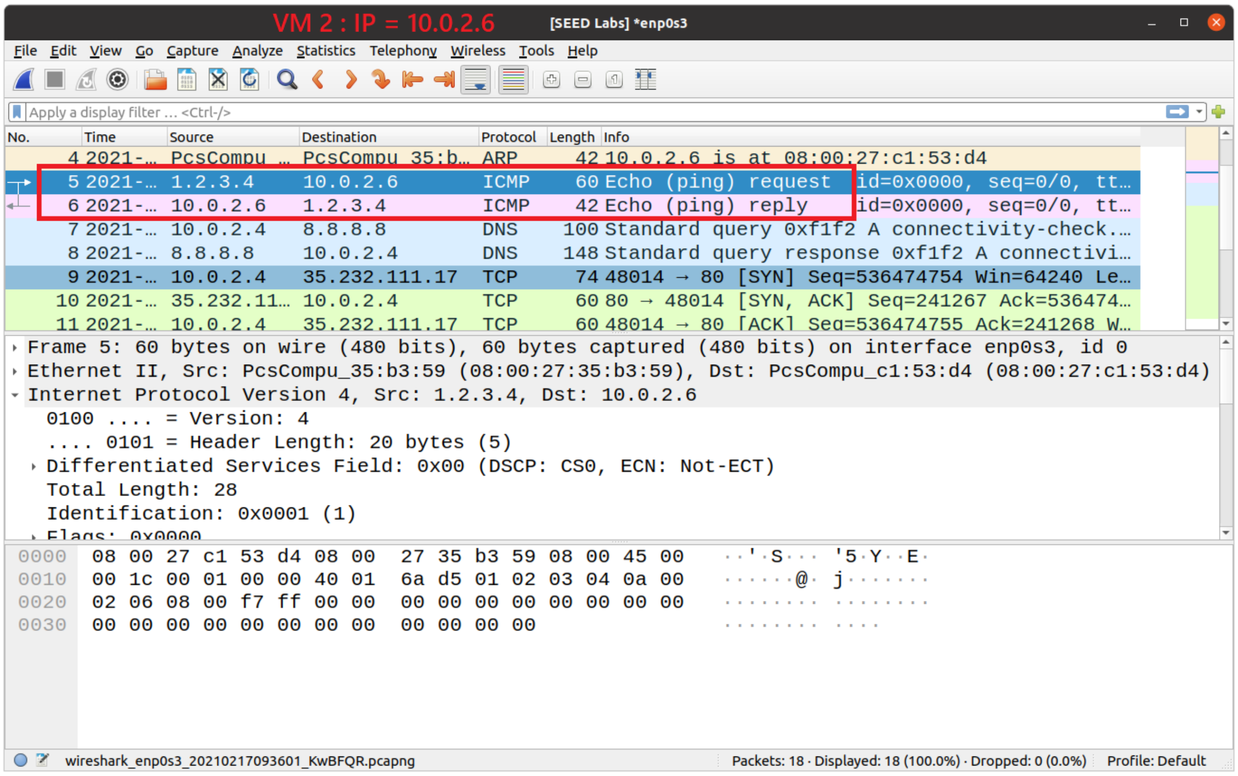
Wireshark:

A recording capture file named '*icmp_spoofing.pdf*' is attached.

Screenshot:



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 icmp_spoofing.py
.
Sent 1 packets.
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                     = '1.2.3.4'     (None)
dst        : DestIPField                       = '10.0.2.6'    (None)
options    : PacketListField                   = []            ([])
[02/17/21]seed@VM:~/.../volumes$
```


VM 2 : IP = 10.0.2.6
[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.  Time      Source          Destination       Protocol Length Info
   4 2021-…  PcsCompu …    PcsCompu_35:b…    ARP        42 10.0.2.6 is at 08:00:27:c1:53:d4
   5 2021-…  1.2.3.4       10.0.2.6          ICMP       60 Echo (ping) request  id=0x0000, seq=0/0, tt…
   6 2021-…  10.0.2.6      1.2.3.4           ICMP       42 Echo (ping) reply    id=0x0000, seq=0/0, tt…
   7 2021-…  10.0.2.4      8.8.8.8           DNS       100 Standard query 0xf1f2 A connectivity-check.…
   8 2021-…  8.8.8.8       10.0.2.4          DNS       148 Standard query response 0xf1f2 A connectivi…
   9 2021-…  10.0.2.4      35.232.111.17     TCP        74 48014 → 80 [SYN] Seq=536474754 Win=64240 Le…
  10 2021-…  35.232.11…    10.0.2.4          TCP        60 80 → 48014 [SYN, ACK] Seq=241267 Ack=536474…
  11 2021-…  10.0.2.4      35.232.111.17     TCP        60 48014 → 80 [ACK] Seq=536474755 Ack=241268 W…

› Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface enp0s3, id 0
› Ethernet II, Src: PcsCompu_35:b3:59 (08:00:27:35:b3:59), Dst: PcsCompu_c1:53:d4 (08:00:27:c1:53:d4)
⌄ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.6
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  › Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 28
    Identification: 0x0001 (1)
  › Flags: 0x0000

0000  08 00 27 c1 53 d4 08 00  27 35 b3 59 08 00 45 00   ··'·S···  '5·Y··E·
0010  00 1c 00 01 00 00 40 01  6a d5 01 02 03 04 0a 00   ······@·  j·······
0020  02 06 08 00 f7 ff 00 00  00 00 00 00 00 00 00 00   ········  ········
0030  00 00 00 00 00 00 00 00  00 00 00 00               ········  ····

wireshark_enp0s3_20210217093601_KwBFQR.pcapng    Packets: 18 · Displayed: 18 (100.0%) · Dropped: 0 (0.0%)    Profile: Default
```

Task 1.3: Traceroute
write your tool to perform the entire procedure automatically.

What is **traceroute**? - traceroute is a computer network diagnostic command for displaying possible routes and measuring transit delays of packets across an Internet Protocol network.

Code:

my_traceroute.py

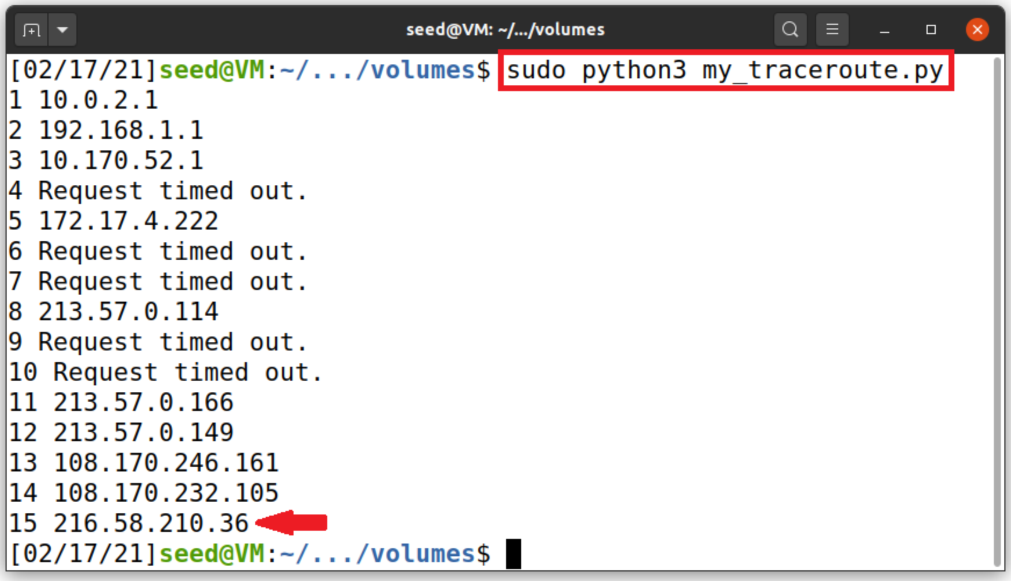
```python
from scapy.all import *

inRoute = True
i = 1
while inRoute:
	a = IP(dst='216.58.210.36', ttl=i)
	response = sr1(a/ICMP(),timeout=7,verbose=0)

	if response is None:
		print(f"{i} Request timed out.")
	elif response.type == 0:
		print(f"{i} {response.src}")
		inRoute = False
	else:
		print(f"{i} {response.src}")

	i = i + 1
```

Explanation:

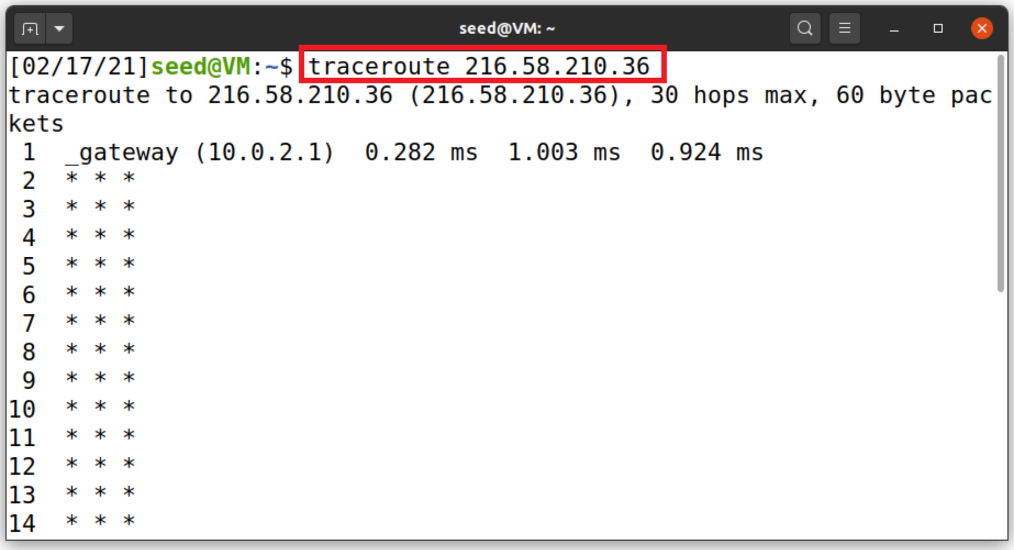
1. **About the code**: I programmed my own traceroute using Scapy library.
   I asked for the destination IP ‘216.58.210.36’ which is Google LLC.
   The flag ‘ttl’ of the packet is increasing by one in each given packet.
   I used a while-loop which will keep iterating as long as it’s routing.
   The sr1() method of Scapy is a method which will listen and wait for a packet response. (timeout = time limit for response, verbose = ignore printing unnecessary details).

2. **About the question**: This program figures out how many routers (hops) it takes to send out that packet all the way to the IP address destination.
   Each line at the display is a different router. The time- to-live is used to return an error of each hop till the destination, this way we can print each IP router till it stops. In this case we reached 15 different routers, 5 of them are timed out.
   A "Request timed out" message at the beginning/middle of a traceroute is very common and can be ignored. This is typically a device that doesn't respond to ICMP or traceroute requests.

Screenshots:



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 my_traceroute.py
1 10.0.2.1
2 192.168.1.1
3 10.170.52.1
4 Request timed out.
5 172.17.4.222
6 Request timed out.
7 Request timed out.
8 213.57.0.114
9 Request timed out.
10 Request timed out.
11 213.57.0.166
12 213.57.0.149
13 108.170.246.161
14 108.170.232.105
15 216.58.210.36
[02/17/21]seed@VM:~/.../volumes$
```



```
[02/17/21]seed@VM:~$ traceroute 216.58.210.36
traceroute to 216.58.210.36 (216.58.210.36), 30 hops max, 60 byte pac
kets
 1  _gateway (10.0.2.1)  0.282 ms  1.003 ms  0.924 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  * * *
```

Task 1.4: Sniffing and-then Spoofing.
you will combine the sniffing and spoofing techniques to implement the following sniff-and then- spoof program.

What is an **ARP** protocol? - a communication protocol used for discovering the link layer address, such as a MAC address, associated with a given internet layer address, typically an IPv4 address. Such a request consists of special packets commonly known as 'who-has' packets. 'Who-has' packets are packets that the IP system broadcasts to all devices on a network (or VLAN), to determine the owner of a specific IP address.

Code:

sniffing_and_spoofing.py

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

def send_packet(pkt):

      if(pkt[2].type == 8):
            src=pkt[1].src
            dst=pkt[1].dst
            seq = pkt[2].seq
            id = pkt[2].id
            load=pkt[3].load
            print(f"Flip: src {src} dst {dst} type 8 REQUEST")
            print(f"Flop: src {dst} dst {src} type 0 REPLY\n")
            reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load
            send(reply,verbose=0)

interfaces = ['enp0s3','lo']
pkt = sniff(iface=interfaces, filter='icmp', prn=send_packet)
```

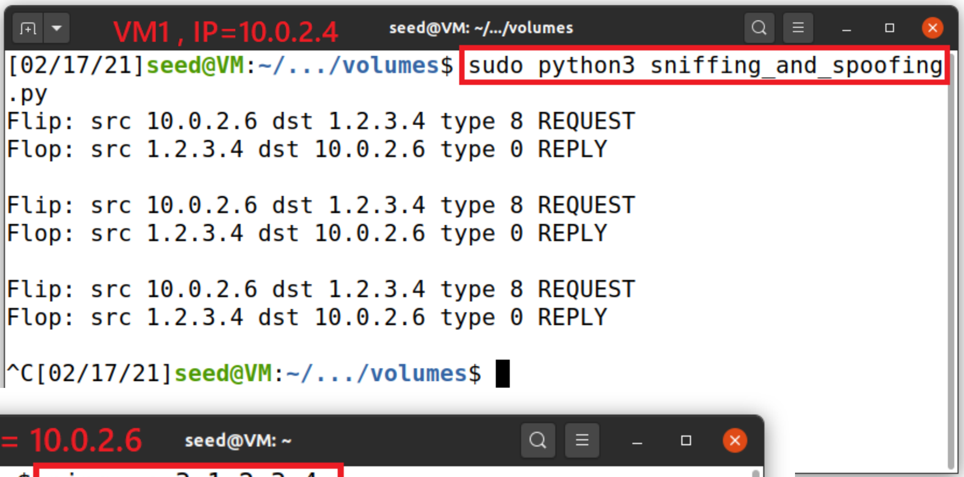
Explanation:

1. **About the code**: the 'if' block is checking if an ICMP is a request.
   If it's true, the reply packet will be based on details derived from the original packet, but it will flip dst and src so whenever it sees an ICMP echo request, regardless of what the target IP address is, the program should immediately

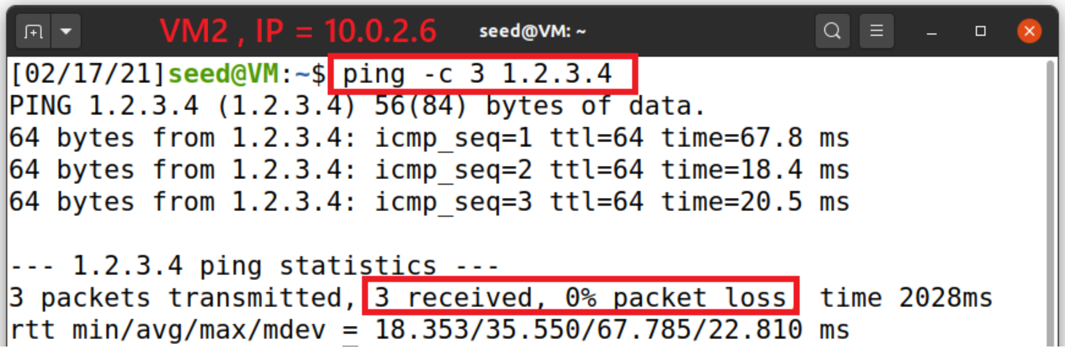
send out an echo reply using this packet spoofing technique.
The pkt[Raw].load is used to store the original packet data payload this way it Will return properly to the sender.

2. **About the question (explanation with screenshots):**
   In this task I tested 3 scenarios of 3 different IPs to ping.
   The program 'sniffing_and_spoofing.py' will sniff for any ICMP packets in the subnet, and when it catches one, the program will return to the sender an ICMP reply packet back. So even if the IP echo request isn't available at all, the program will always return to the sender a reply packet.
   I used two VMs for this task.
   My original VM will be known as 'VM1' with IP address of '10.0.2.4' and the other one will be 'VM2' with IP address of '10.0.2.5'.

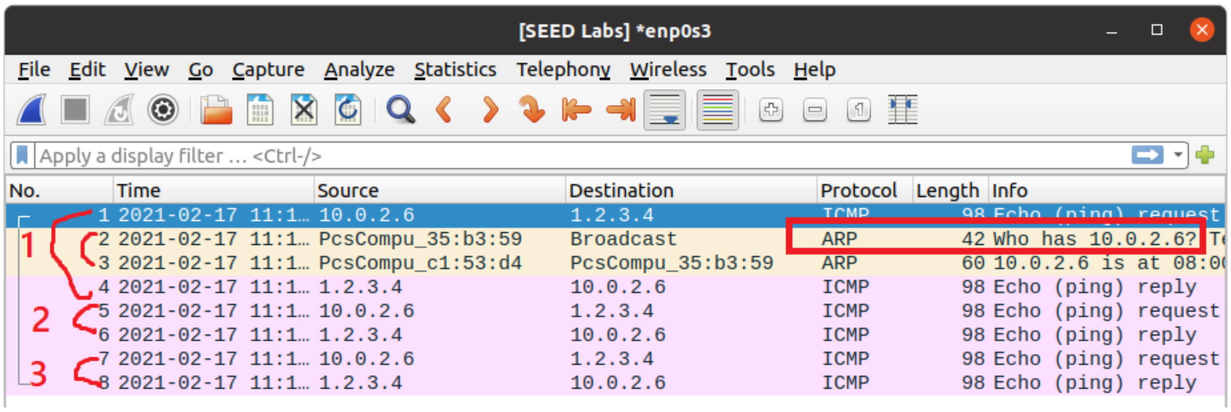
At the first scenario VM2 sends a ping to '1.2.3.4' which is a non-existing host on the Internet. Without the program we will get a 100% packet loss because it will never return to the source. We can see in the screenshot of the wireshark that the ARP protocol is asking for 1.2.3.4 and asking on the network who has that IP destination? So the attacker (my program on VM1) returns with an answer to it and the ICMP packet reply is coming back to VM2.(A wireshark capture is attached: 'snifsnof1.pdf').



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 1.2.3.4 type 8 REQUEST
Flop: src 1.2.3.4 dst 10.0.2.6 type 0 REPLY

^C[02/17/21]seed@VM:~/.../volumes$ 
```

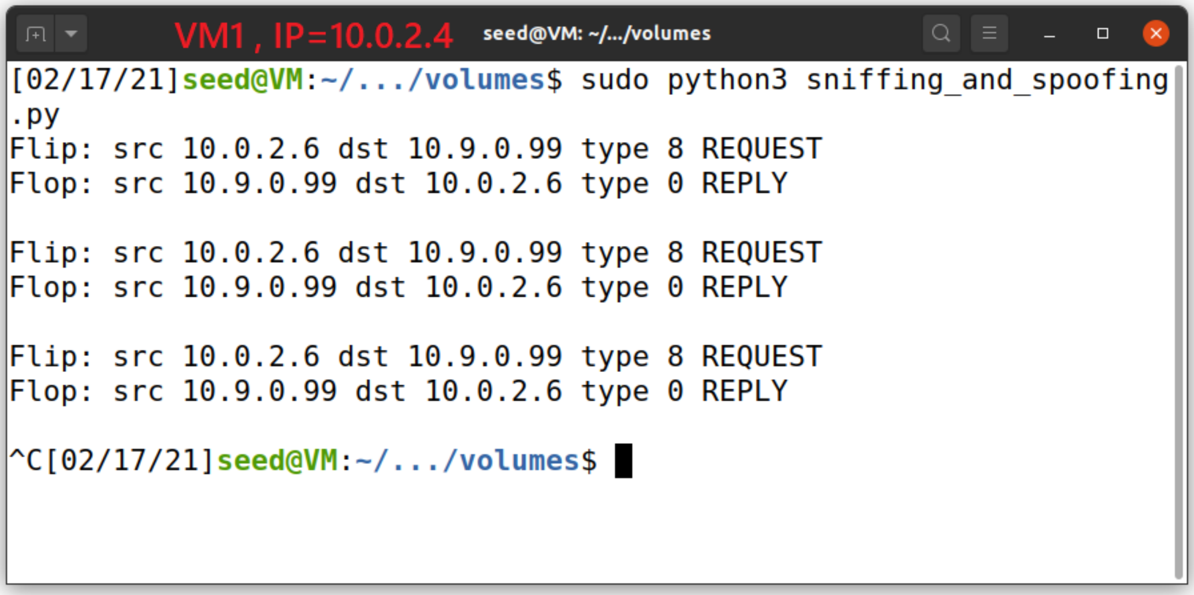


```
[02/17/21]seed@VM:~$ ping -c 3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=67.8 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=18.4 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=20.5 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2028ms
rtt min/avg/max/mdev = 18.353/35.550/67.785/22.810 ms
```

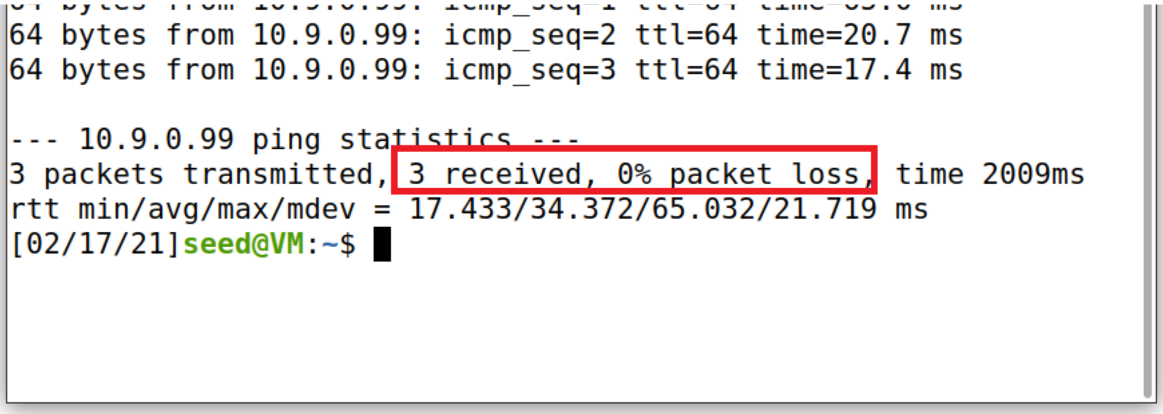

[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
1
2
3

```
No.   Time                 Source              Destination         Protocol  Length Info
  1 2021-02-17 11:1… 10.0.2.6            1.2.3.4             ICMP          98 Echo (ping) request
  2 2021-02-17 11:1… PcsCompu_35:b3:59   Broadcast           ARP           42 Who has 10.0.2.6? Te
  3 2021-02-17 11:1… PcsCompu_c1:53:d4   PcsCompu_35:b3:59   ARP           60 10.0.2.6 is at 08:0
  4 2021-02-17 11:1… 1.2.3.4             10.0.2.6            ICMP          98 Echo (ping) reply
  5 2021-02-17 11:1… 10.0.2.6            1.2.3.4             ICMP          98 Echo (ping) request
  6 2021-02-17 11:1… 1.2.3.4             10.0.2.6            ICMP          98 Echo (ping) reply
  7 2021-02-17 11:1… 10.0.2.6            1.2.3.4             ICMP          98 Echo (ping) request
  8 2021-02-17 11:1… 1.2.3.4             10.0.2.6            ICMP          98 Echo (ping) reply
```

At the second scenario, VM2 sends a ping to ‘10.9.0.99’ which is a non-existing host on the LAN. In this scenario we’re getting the same concept with the same idea of the ARP protocol. Even though the host doesn't even exist. The program from VM1 will send back an ICMP response packet. (A wireshark capture is attached: ‘snifsnof2.pdf’).

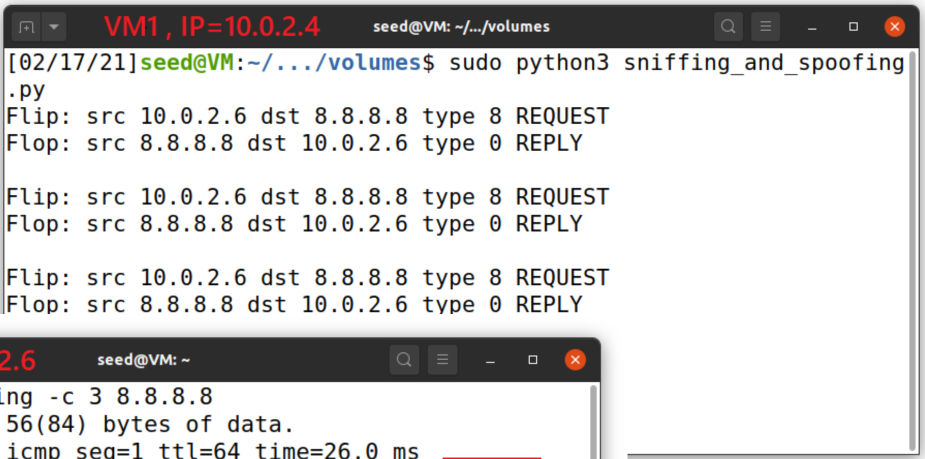


```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 10.9.0.99 type 8 REQUEST
Flop: src 10.9.0.99 dst 10.0.2.6 type 0 REPLY

^C[02/17/21]seed@VM:~/.../volumes$
```

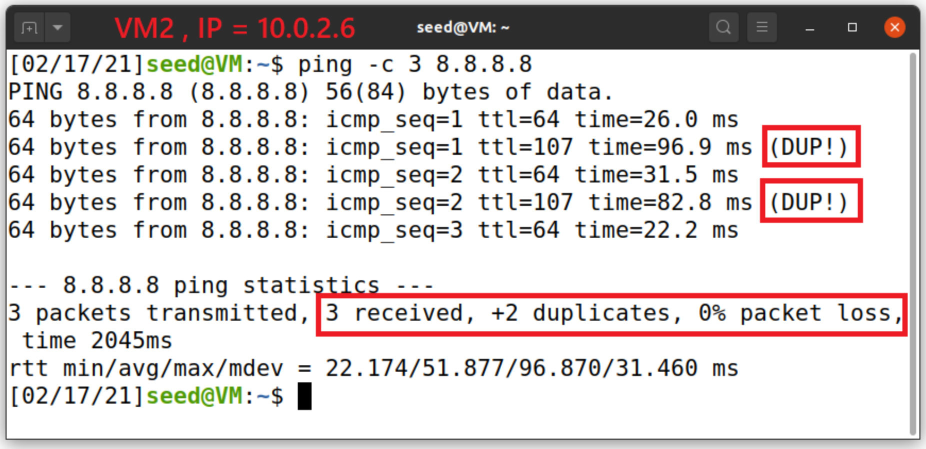


```
64 bytes from 10.9.0.99: icmp_seq=2 ttl=64 time=20.7 ms
64 bytes from 10.9.0.99: icmp_seq=3 ttl=64 time=17.4 ms

--- 10.9.0.99 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2009ms
rtt min/avg/max/mdev = 17.433/34.372/65.032/21.719 ms
[02/17/21]seed@VM:~$
```

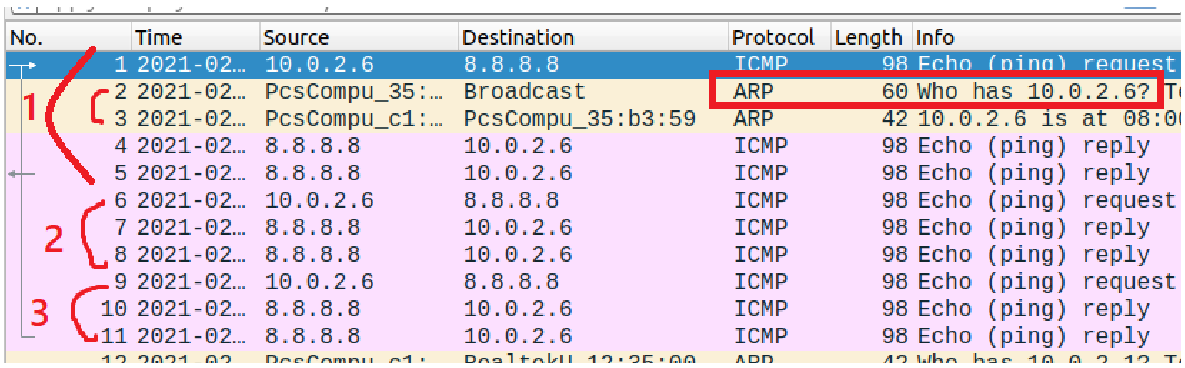
At the third scenario, VM2 sends a ping to '8.8.8.8' which is an existing host on the Internet. This scenario is different from the others because it really exists on the net. So in this case we're getting duplicate responses, that's because the real destination is responding to the source, but my program is also responding to the source.
We can see it very clear in the screenshots and in the wireshark recording.
(A wireshark capture is attached: 'snifsnof3.pdf').



```
[02/17/21]seed@VM:~/.../volumes$ sudo python3 sniffing_and_spoofing
.py
Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY

Flip: src 10.0.2.6 dst 8.8.8.8 type 8 REQUEST
Flop: src 8.8.8.8 dst 10.0.2.6 type 0 REPLY
```



```
[02/17/21]seed@VM:~$ 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=64 time=26.0 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=107 time=96.9 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=31.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=107 time=82.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=22.2 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, +2 duplicates, 0% packet loss,
 time 2045ms
rtt min/avg/max/mdev = 22.174/51.877/96.870/31.460 ms
[02/17/21]seed@VM:~$
```



No.	Time	Source	Destination	Protocol	Length	Info
1	2021-02...	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
2	2021-02...	PcsCompu_35:...	Broadcast	ARP	60	Who has 10.0.2.6? T
3	2021-02...	PcsCompu_c1:...	PcsCompu_35:b3:59	ARP	42	10.0.2.6 is at 08:0
4	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
5	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
6	2021-02...	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
7	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
8	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
9	2021-02...	10.0.2.6	8.8.8.8	ICMP	98	Echo (ping) request
10	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
11	2021-02...	8.8.8.8	10.0.2.6	ICMP	98	Echo (ping) reply
12	2021-02...	PcsCompu_c1:...	RealtekU_12:35:00	ARP	42	Who has 10.0.2.12 T

Task 2.1: Writing Packet Sniffing Program

What is **pcap**? - **pcap** is an application programming interface (API) for capturing network traffic.

Code:

sniffer.c

```
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char
*packet){
  struct ethheader *eth = (struct ethheader *)packet;

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IPv4 type
    struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));

    printf("Source: %s   ", inet_ntoa(ip->iph_sourceip));
    printf("Destination: %s\n", inet_ntoa(ip->iph_destip));
  }
}

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 enp0s3
  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);   //Close the handle
  return 0;
}
```