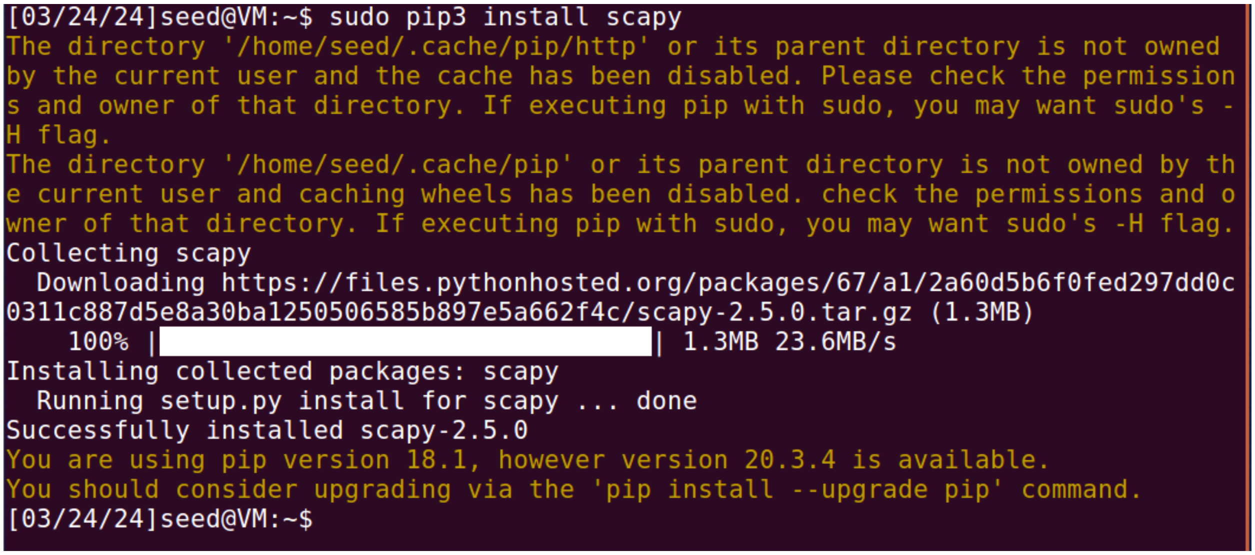
SEED Lab Report: Sniffing and Spoofing
Andrew Simon
N00695969

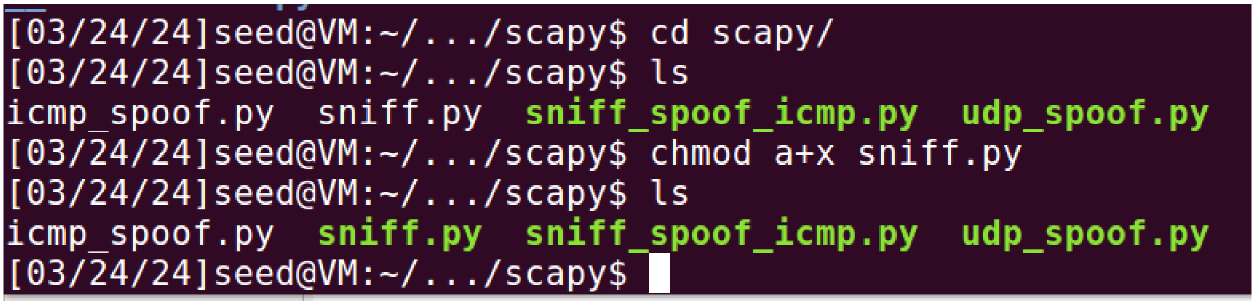
Task 1.1: Sniffing Packets

The first set of tasks will require scapy to be used as a tool for sniffing and spoofing. I first made sure it was installed for python3 on my machine:



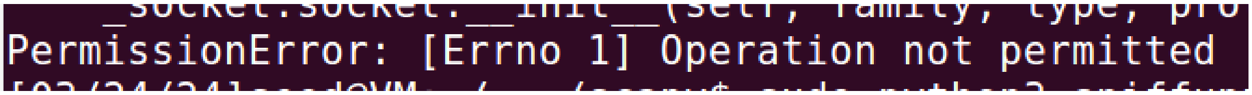
```
[03/24/24]seed@VM:~$ sudo pip3 install scapy
The directory '/home/seed/.cache/pip/http' or its parent directory is not owned
by the current user and the cache has been disabled. Please check the permission
s and owner of that directory. If executing pip with sudo, you may want sudo's -
H flag.
The directory '/home/seed/.cache/pip' or its parent directory is not owned by th
e current user and caching wheels has been disabled. check the permissions and o
wner of that directory. If executing pip with sudo, you may want sudo's -H flag.
Collecting scapy
  Downloading https://files.pythonhosted.org/packages/67/a1/2a60d5b6f0fed297dd0c
0311c887d5e8a30ba1250506585b897e5a662f4c/scapy-2.5.0.tar.gz (1.3MB)
    100% |████████████████████████████████| 1.3MB 23.6MB/s
Installing collected packages: scapy
  Running setup.py install for scapy ... done
Successfully installed scapy-2.5.0
You are using pip version 18.1, however version 20.3.4 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
[03/24/24]seed@VM:~$
```

Once installed, I needed to make the provided python3 code executable:



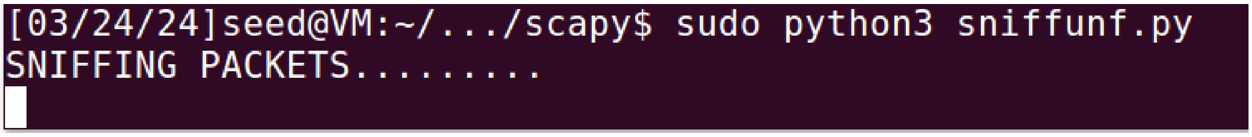
```
[03/24/24]seed@VM:~/.../scapy$ cd scapy/
[03/24/24]seed@VM:~/.../scapy$ ls
icmp_spoof.py  sniff.py  sniff_spoof_icmp.py  udp_spoof.py
[03/24/24]seed@VM:~/.../scapy$ chmod a+x sniff.py
[03/24/24]seed@VM:~/.../scapy$ ls
icmp_spoof.py  sniff.py  sniff_spoof_icmp.py  udp_spoof.py
[03/24/24]seed@VM:~/.../scapy$ 
```

When attempting to run the program without sudo, I get the following error:



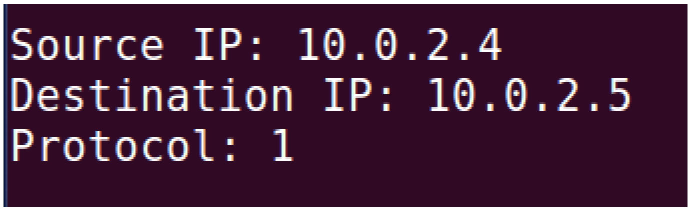
```
    _socket.socket.__init__(self, family, type, pro
PermissionError: [Errno 1] Operation not permitted
```

After running the program with sudo, I get a successful execution, and the program begins to sniff packets:



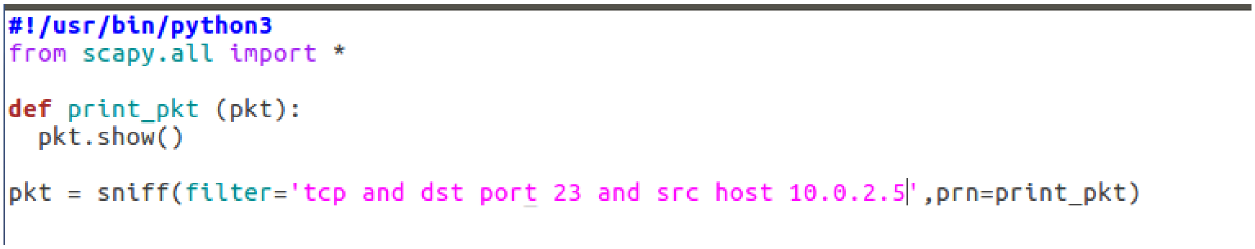
```
[03/24/24]seed@VM:~/.../scapy$ sudo python3 sniffunf.py
SNIFFING PACKETS.........
```

When pinging from another machine, I get the following output from the program:



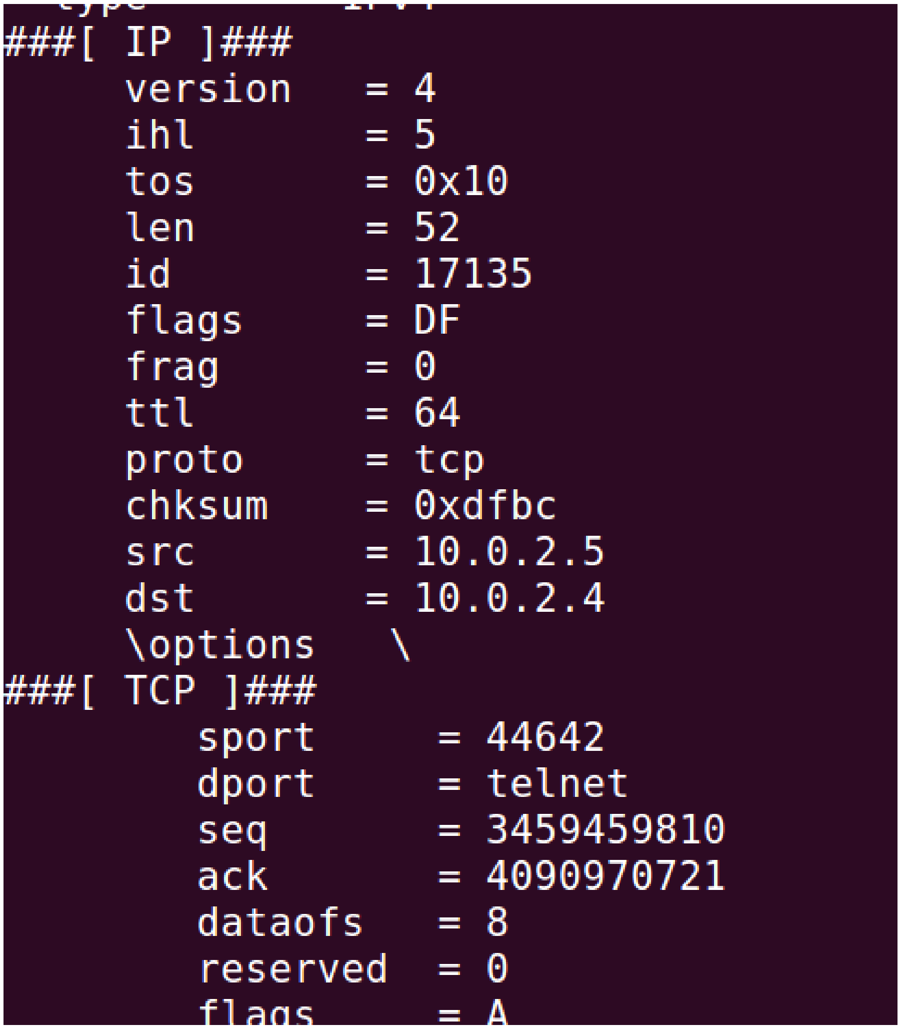
```
Source IP: 10.0.2.4
Destination IP: 10.0.2.5
Protocol: 1
```

In order to get tcp packets, I made the following change to a stripped-down version of the program (only using pkt.show()):



```python
#!/usr/bin/python3
from scapy.all import *

def print_pkt (pkt):
  pkt.show()

pkt = sniff(filter='tcp and dst port 23 and src host 10.0.2.5',prn=print_pkt)
```

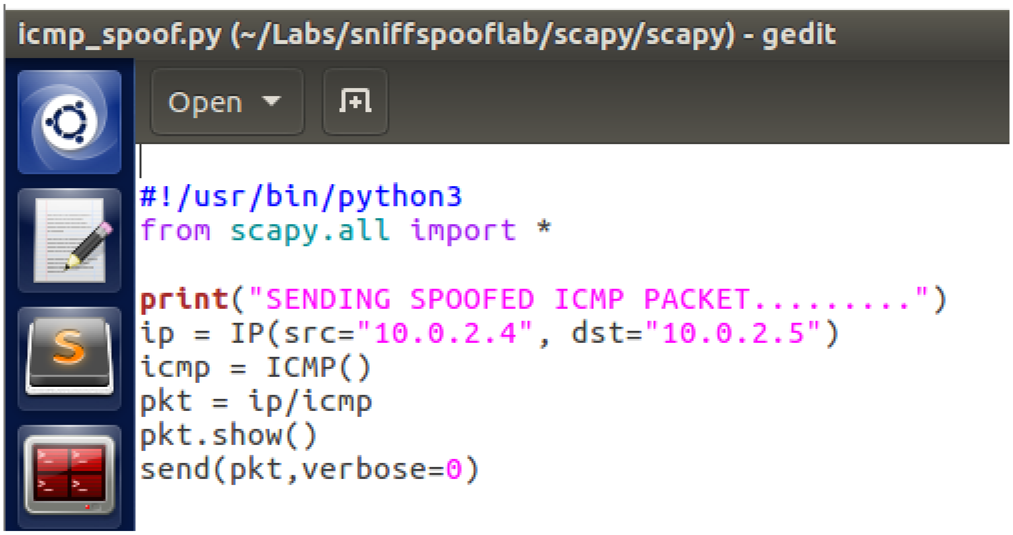
Running telnet 10.0.2.4 from my other machine during the program's execution, I was provided the following sniffed information:



```
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 52
     id        = 17135
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xdfbc
     src       = 10.0.2.5
     dst       = 10.0.2.4
     \options   \
###[ TCP ]###
        sport     = 44642
        dport     = telnet
        seq       = 3459459810
        ack       = 4090970721
        dataofs   = 8
        reserved  = 0
        flags     = A
```

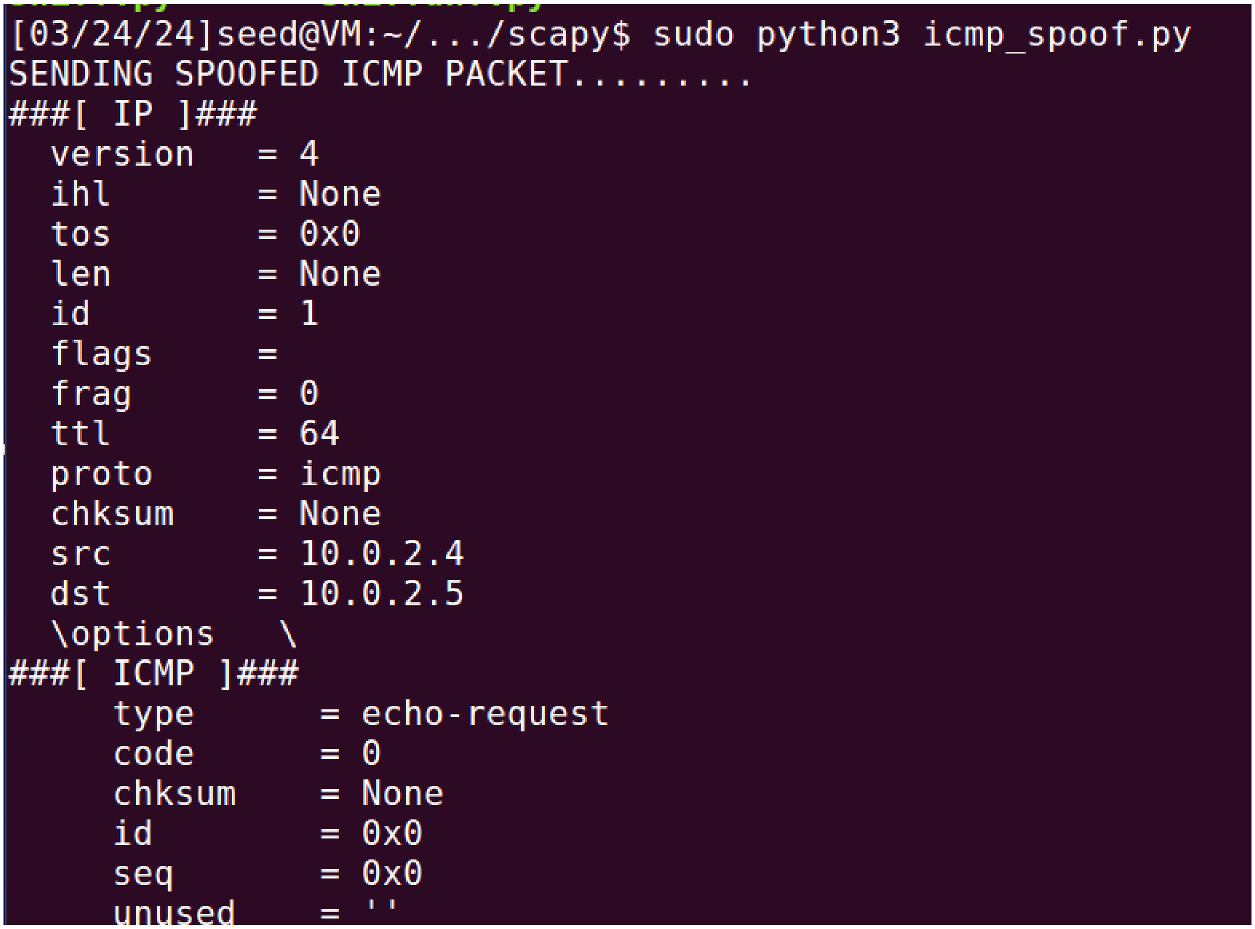
Task 1.2: Spoofing ICMP Packets

When looking to spoof icmp packets, I needed to start by changing the IP addresses in the provided lab code to the addresses of the machines I was using:

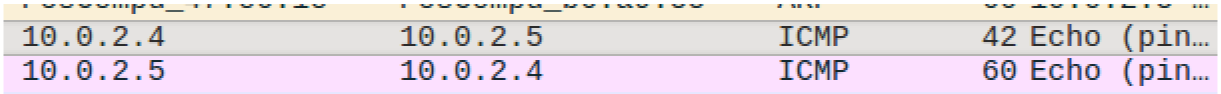


```
#!/usr/bin/python3
from scapy.all import *

print("SENDING SPOOFED ICMP PACKET.........")
ip = IP(src="10.0.2.4", dst="10.0.2.5")
icmp = ICMP()
pkt = ip/icmp
pkt.show()
send(pkt,verbose=0)
```

When executing this code, I was able to properly send a spoofed packet:



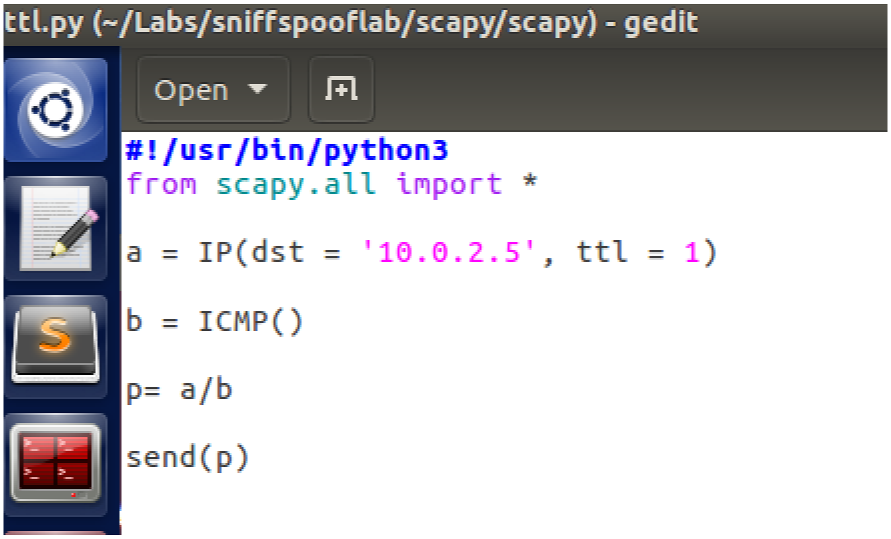
```
[03/24/24]seed@VM:~/.../scapy$ sudo python3 icmp_spoof.py
SENDING SPOOFED ICMP PACKET.........
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = None
  src       = 10.0.2.4
  dst       = 10.0.2.5
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = None
     id        = 0x0
     seq       = 0x0
     unused    = ''
```



```
10.0.2.4        10.0.2.5        ICMP    42 Echo (pin…
10.0.2.5        10.0.2.4        ICMP    60 Echo (pin…
```

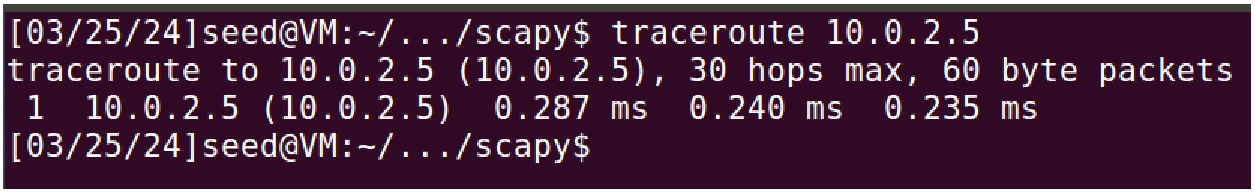
Task 1.3: Traceroute

The next task was aimed at estimating the distance between my VM and the given destination. I made the following program in Python to test a similar functionality of **traceroute**:



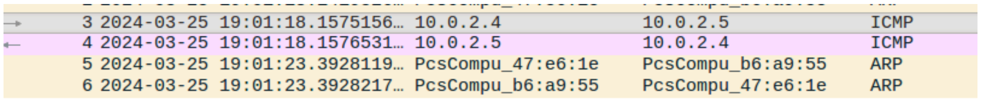
```
#!/usr/bin/python3
from scapy.all import *

a = IP(dst = '10.0.2.5', ttl = 1)

b = ICMP()

p= a/b

send(p)
```

For a proper test, I used **traceroute** to measure the distance between my two machines:

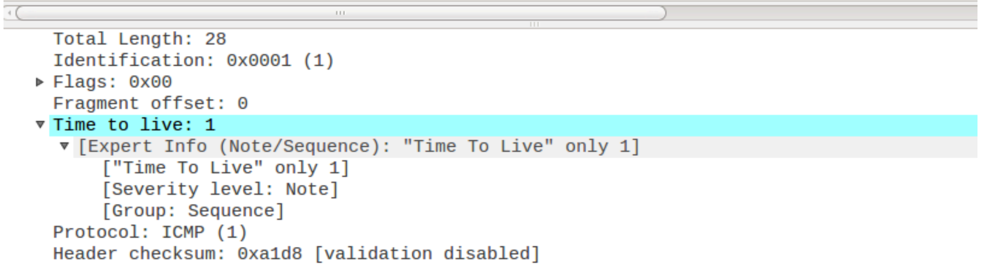


```
[03/25/24]seed@VM:~/.../scapy$ traceroute 10.0.2.5
traceroute to 10.0.2.5 (10.0.2.5), 30 hops max, 60 byte packets
 1  10.0.2.5 (10.0.2.5)  0.287 ms  0.240 ms  0.235 ms
[03/25/24]seed@VM:~/.../scapy$
```

When setting the TTL variable as only 1, I get the following response in Wireshark:

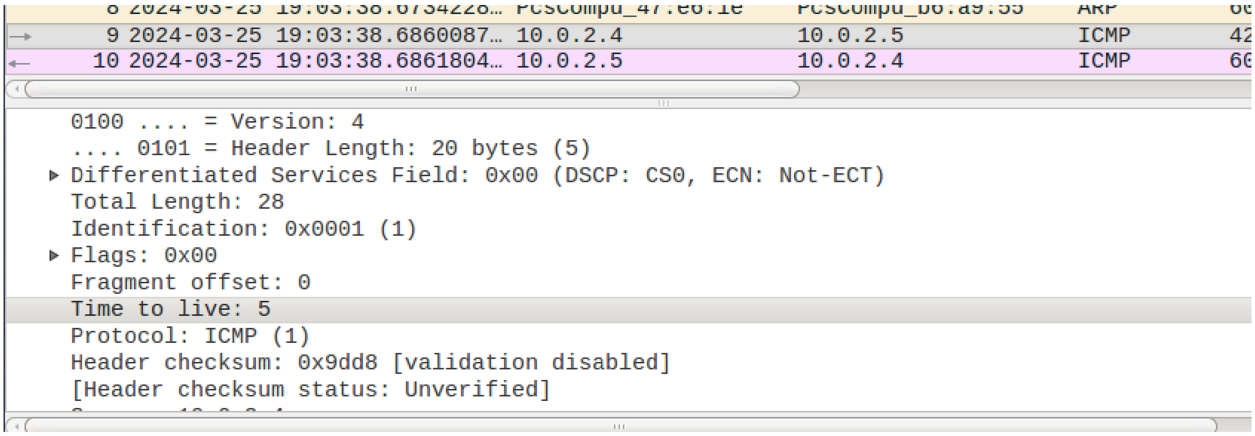


```
3 2024-03-25 19:01:18.1575156… 10.0.2.4           10.0.2.5           ICMP
4 2024-03-25 19:01:18.1576531… 10.0.2.5           10.0.2.4           ICMP
5 2024-03-25 19:01:23.3928119… PcsCompu_47:e6:1e  PcsCompu_b6:a9:55  ARP
6 2024-03-25 19:01:23.3928217… PcsCompu_b6:a9:55  PcsCompu_47:e6:1e  ARP
```



```
    Total Length: 28
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
  ▾ Time to live: 1
    ▾ [Expert Info (Note/Sequence): "Time To Live" only 1]
        ["Time To Live" only 1]
        [Severity level: Note]
        [Group: Sequence]
    Protocol: ICMP (1)
    Header checksum: 0xa1d8 [validation disabled]
```

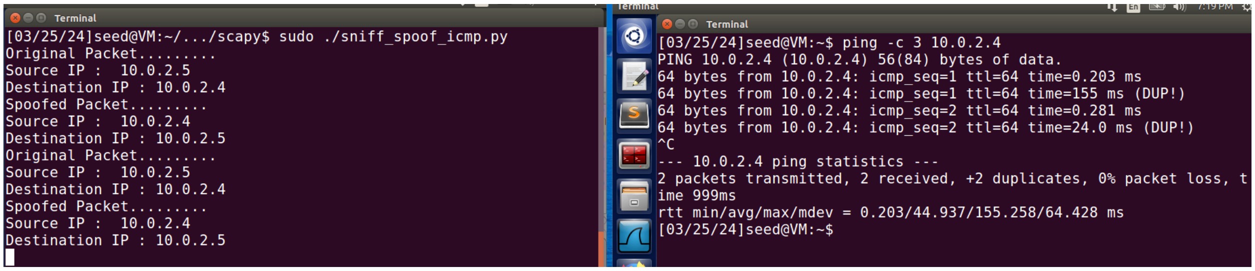
When raising the time to live up to 5 in the program, that issue is resolved as shown in Wireshark:



```
     8 2024-03-25 19:03:38.6734228… PcsCompu_47:e6:1e     PcsCompu_b6:a9:55     ARP         60
     9 2024-03-25 19:03:38.6860087… 10.0.2.4              10.0.2.5              ICMP        42
    10 2024-03-25 19:03:38.6861804… 10.0.2.5              10.0.2.4              ICMP        60

    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: 0x00
    Fragment offset: 0
    Time to live: 5
    Protocol: ICMP (1)
    Header checksum: 0x9dd8 [validation disabled]
    [Header checksum status: Unverified]
```

Task 1.4: Sniffing, Then Spoofing

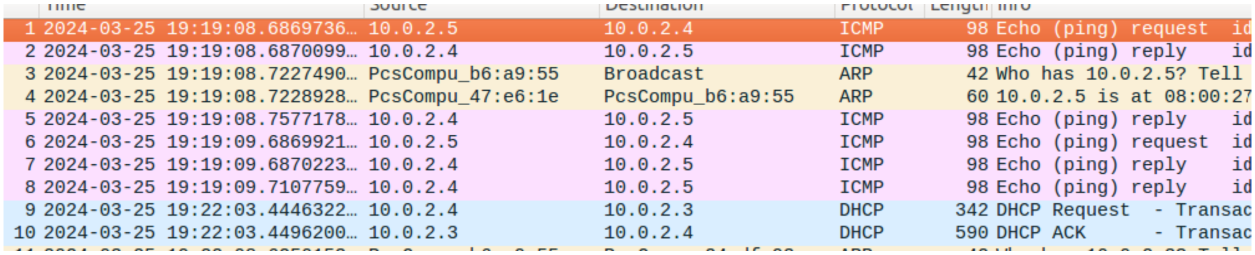
For this task we are aiming to combine sniffing and spoofing. After altering the provided program to reach the IP address of my second machine, I received the following feedback when pinging the machine:



```
[03/25/24]seed@VM:~/.../scapy$ sudo ./sniff_spoof_icmp.py
Original Packet.........
Source IP :  10.0.2.5
Destination IP : 10.0.2.4
Spoofed Packet.........
Source IP :  10.0.2.4
Destination IP : 10.0.2.5
Original Packet.........
Source IP :  10.0.2.5
Destination IP : 10.0.2.4
Spoofed Packet.........
Source IP :  10.0.2.4
Destination IP : 10.0.2.5
```

```
[03/25/24]seed@VM:~$ ping -c 3 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=0.203 ms
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=155 ms (DUP!)
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.281 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=24.0 ms (DUP!)
^C
--- 10.0.2.4 ping statistics ---
2 packets transmitted, 2 received, +2 duplicates, 0% packet loss, t
ime 999ms
rtt min/avg/max/mdev = 0.203/44.937/155.258/64.428 ms
[03/25/24]seed@VM:~$
```

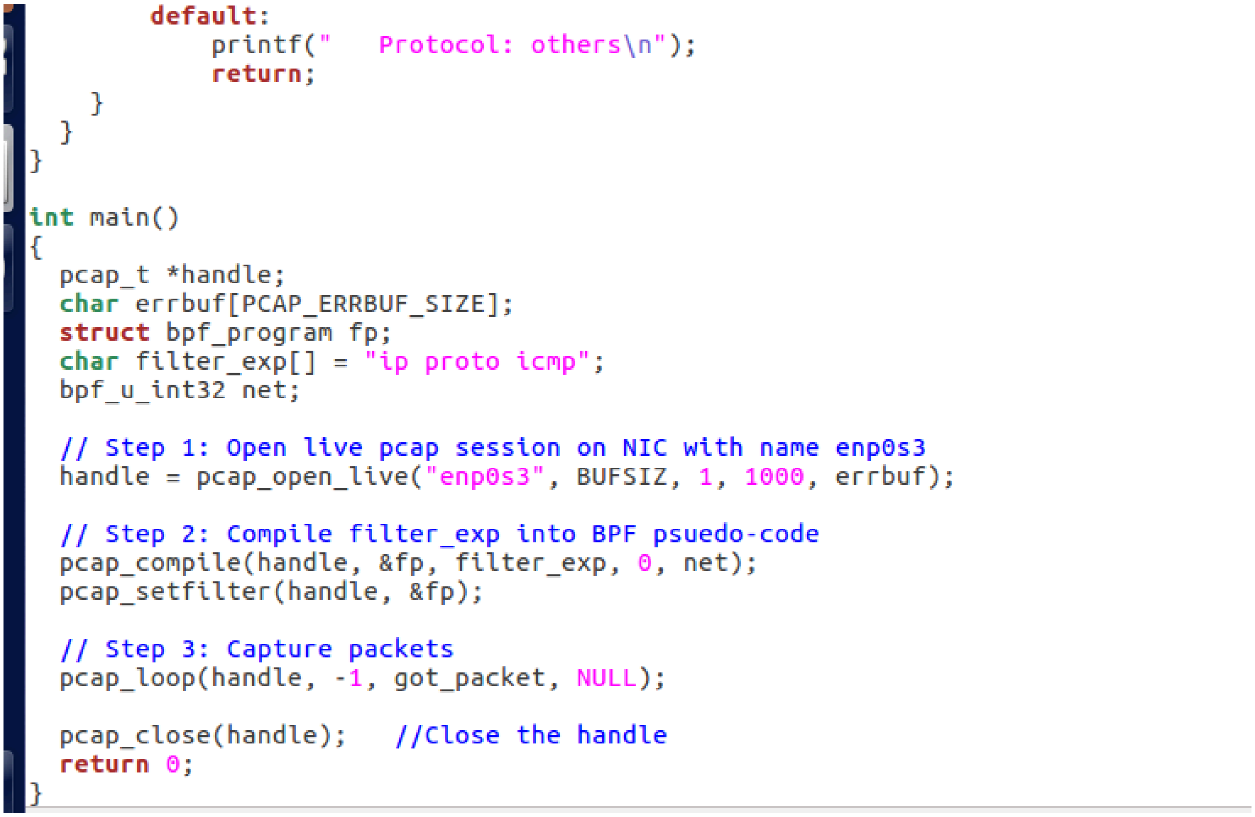
The ICMP Echo packets caught in Wireshark backup the result of the program’s execution:



```
    Time                             Source                Destination           Protocol Length Info
 1 2024-03-25 19:19:08.6869736… 10.0.2.5              10.0.2.4              ICMP         98 Echo (ping) request  id
 2 2024-03-25 19:19:08.6870099… 10.0.2.4              10.0.2.5              ICMP         98 Echo (ping) reply    id
 3 2024-03-25 19:19:08.7227490… PcsCompu_b6:a9:55     Broadcast             ARP          42 Who has 10.0.2.5? Tell
 4 2024-03-25 19:19:08.7228928… PcsCompu_47:e6:1e     PcsCompu_b6:a9:55     ARP          60 10.0.2.5 is at 08:00:27
 5 2024-03-25 19:19:08.7577178… 10.0.2.4              10.0.2.5              ICMP         98 Echo (ping) reply    id
 6 2024-03-25 19:19:09.6869921… 10.0.2.5              10.0.2.4              ICMP         98 Echo (ping) request  id
 7 2024-03-25 19:19:09.6870223… 10.0.2.4              10.0.2.5              ICMP         98 Echo (ping) reply    id
 8 2024-03-25 19:19:09.7107759… 10.0.2.4              10.0.2.5              ICMP         98 Echo (ping) reply    id
 9 2024-03-25 19:22:03.4446322… 10.0.2.4              10.0.2.3              DHCP        342 DHCP Request  - Transac
10 2024-03-25 19:22:03.4496200… 10.0.2.3              10.0.2.4              DHCP        590 DHCP ACK      - Transac
```

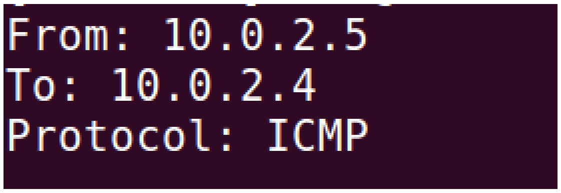
Task 2.1: Writing a Packet Sniffing Program

When working with the sniffer program, I first wanted to check that the pcap name was appropriate for my machine to capture:



```
          default:
              printf("   Protocol: others\n");
              return;
      }
    }
}

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;
}
```

When pinging from my second machine, I got the following style of responses from my sniffer program's output, letting me know that it was capturing packets:



```
From: 10.0.2.5
To: 10.0.2.4
Protocol: ICMP
```

Question 1: Please use in your own words to describe the sequence of the library calls that are essential for sniffer programs

Response: A sniffer program typically starts by opening a live capture on a network interface (referenced in the code's line "pcap_open_live"). Then, it compiles a filter expression into BPF pseudo-code (referenced in the code's line "pcap_compile"), specifying which packets to capture with "pcap_setfilter." It does this process in a loop with "pcap_loop" using a callback function and closes the session with "pcap_close" when done.

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?
Response: You need to have root privileges when using a sniffer program primarily because it requires access to raw network packets. Due to their sensitive nature, these packets are usually only available to privileged users. Without this privilege, the program would fail to access the necessary network interfaces when attempting to interact with these packets. It will not be able to run the “pcap_open_live” command, as the creation of this session would not be permitted.

Question 3: Please turn on and off promiscuous mode in your sniffer program. Can you demonstrate the difference between when this mode is on or off? Please describe how you can demonstrate this.
Response: In the “pcap_open_live” line of the program, the third parameter allows us to turn promiscuous mode on and off. When set to 0, it will be OFF, and when set to anything other than 0, it will be ON. With the mode ON, the program will sniff all traffic on the line no matter who it is intended for. With the mode OFF, the program will only sniff traffic directly intended for the host machine.

Question 4: Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
Response: Yes, you can, but the value will be corrected to its original size

Question 5: Using the raw socket programming, do you have to calculate the checksum for the IP header?
Response: Yes, you are responsible for calculating the checksum, but you can make the kernel calculate it. In the IP header fields, you need to set the ip_check field equal to 0 for the kernel to calculate it.

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?
Response: The primary reason root privileges are needed for these sorts of programs is that nonprivileged users are not permitted to make changes to the fields in the protocol headers and cannot access the raw sockets. The program will fail when attempting to setup the raw sockets.