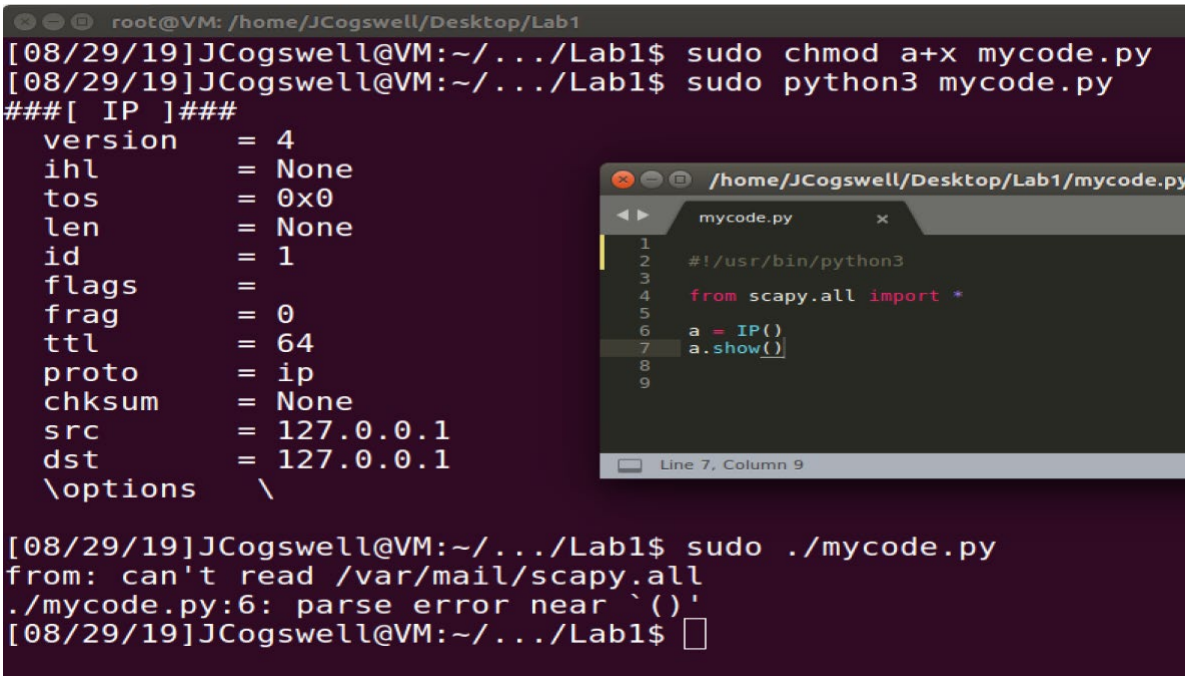
Task 1: Using Tools to Sniff and Spoof Packets

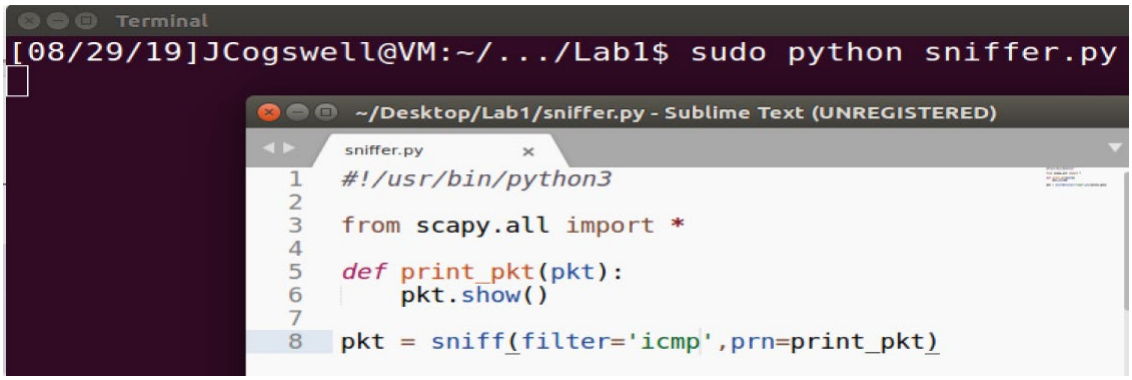
Initially, I used the program provided by SEEDLabs and executed it with and without root privileges. The outcome and the program used are shown below.



The difference between running the program with root privileges and without root privileges is that root privileges are required for spoofing packets, and therefore the program cannot execute the line that calls the scapy modules.

Task 1.1: Sniffing Packets

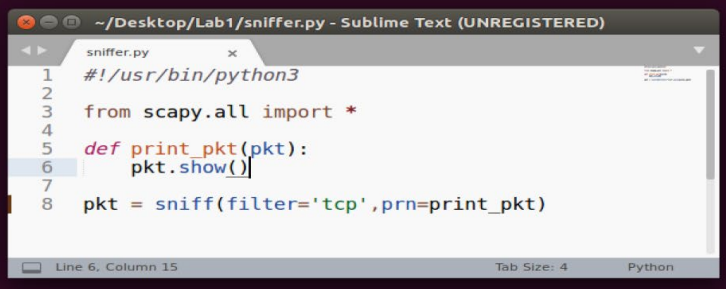
In this task I wasted a lot of time trying to figure out why nothing would happen when I executed the sniffer.py program that I created. Then I looked up when ICMP packets occur and realized that was the reason nothing was showing up. I changed the filter to TCP and then the program worked.



I opened another terminal and pinged google. Then the program worked as written. Examples of this and the TCP adjustment are shown below.

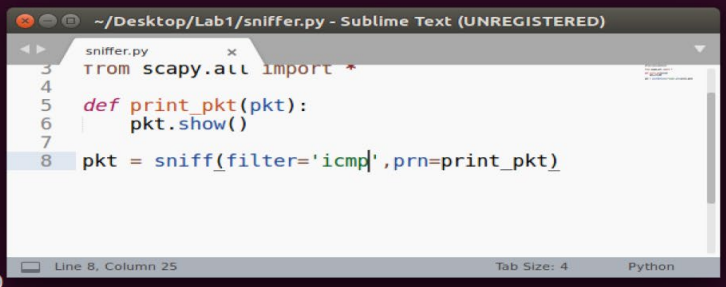
TCP →

```
###[ Ethernet ]###
  dst       = 00:0c:29:a0:52:98
  src       = 00:50:56:ef:b2:f8
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 1500
     id        = 26599
     flags     =
     frag      = 0
     ttl       = 128
     proto     = tcp
     chksum    = 0xbfed
     src       = 172.217.13.67
     dst       = 192.168.146.130
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 35836
        seq       = 356392649
        ack       = 1392971056
        dataofs   = 5
        reserved  = 0
        flags     = A
        window    = 64240
        chksum    = 0x2b89
        urgptr    = 0
        options   = []
###[ Raw ]###
           load      = '2\x89\x81\xd1E\xc5\xc4\x8b\xdb\xf9\x011\x8c\xe3\xbe\x05=
\xd6\xac\xd4\xdd\xb5y!\xf1AHZy~o\xf3\x8cA<\x9efa \x90\xbe\xbc\xe9\xc3\x7f\x8b[\x
fd\xc7\xfc\xec\x17\x8c\xb6\x1dF\xa3\xfa\x02\xde\xb8\xa7\x83\xfb\x18\x0b\xf3\xd5\
xfa\xee:X\xfb\xe3\xa0\xe1\xab\xb49l;J\xae\xa5\xaa\x13t\xbb\xf3^<\x19\xbcHt\x88M\
xb1H>\xcd\xda\xb7!\x8fz{\x963\xa86\xaf3\xf9\xde\xa5\x03Q\xa4\x89 \xb1d\xe91;\xbe
f\xe33\xbf\x8f\xefTF\xce\xccb\x99\x11\x1f\x9a\xdf\x1ef\xf8e\xda10.\x0c\x0c\x9e"\
x8e\x11\xe1\x88\xb1\xc9\xd8\x15_\xbe@-5%Py70\x1c\xbf\x08\xba/\x8e\xf4n.\xb1Z\xc6
```



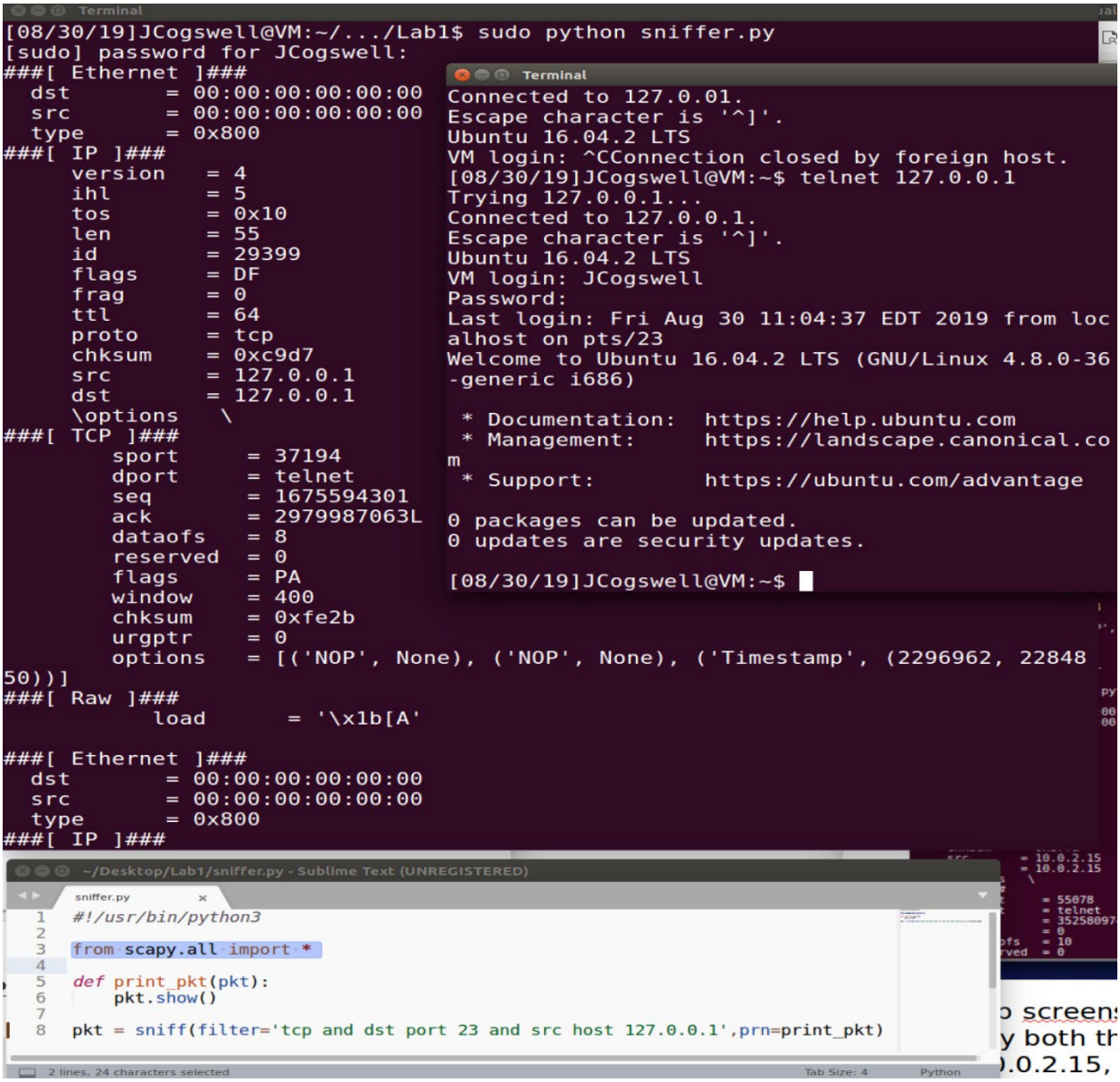
ICMP →

```
[08/29/19]JCogswell@VM:~/.../Lab1$ sudo python sniffer.py
###[ Ethernet ]###
  dst       = 00:50:56:ef:b2:f8
  src       = 00:0c:29:a0:52:98
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 27055
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xc727
     src       = 192.168.146.130
     dst       = 172.217.9.206
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x1ec
        id        = 0x2410
        seq       = 0x1
###[ Raw ]###
           load      = '80h]?r\x07\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13
\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

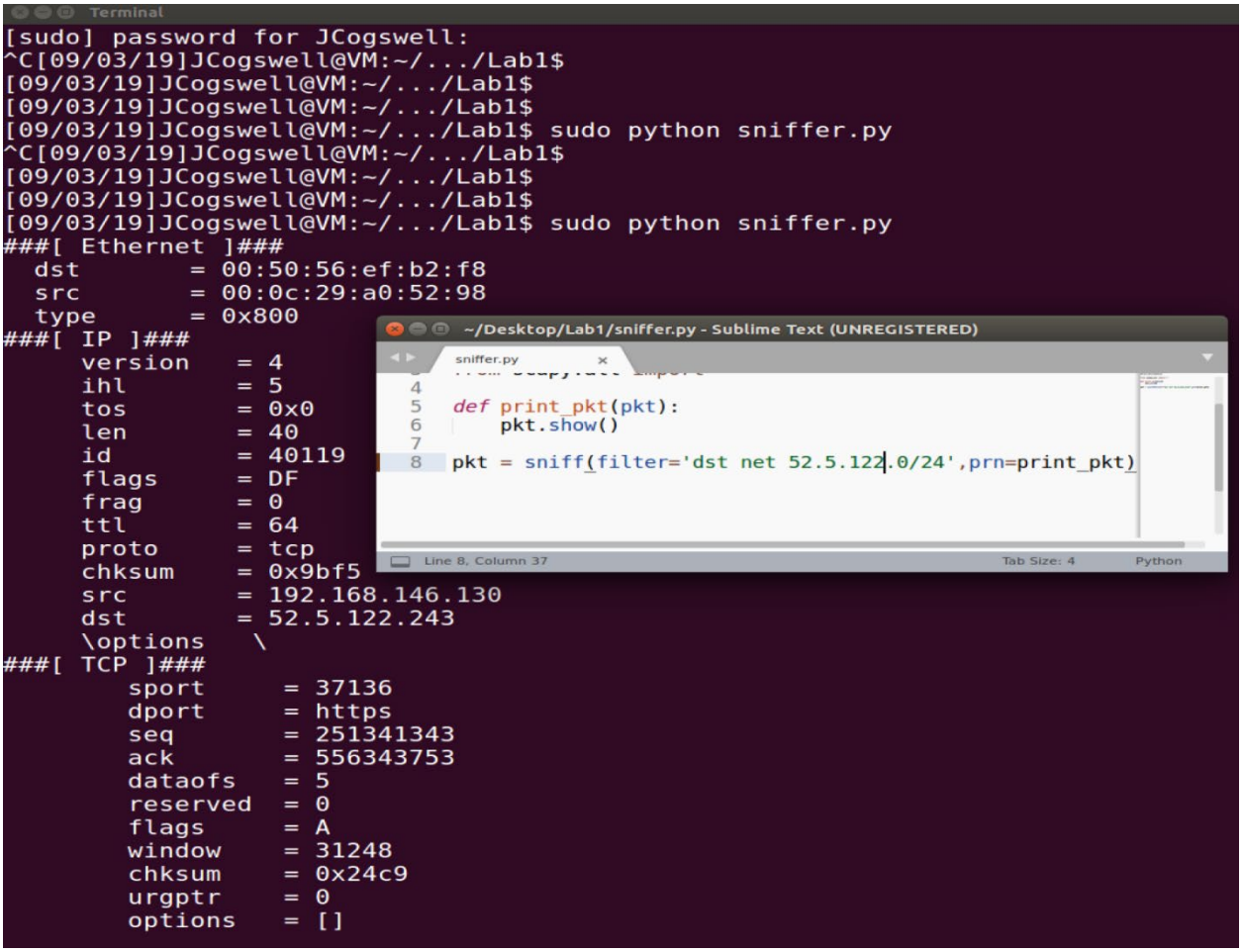



```
Terminal
[08/29/19]JCogswell@VM:~$ ping google.com
PING google.com (172.217.9.206) 56(84) bytes of data.
^C
--- google.com ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1013ms
```

In this portion of task 1.1, I edited the sniffer.py code to sniff out TCP packets that have a specific source address and a specific destination port (23 – Telnet). To do this, I needed to create a Telnet connection with another terminal (shown in the screenshot below).

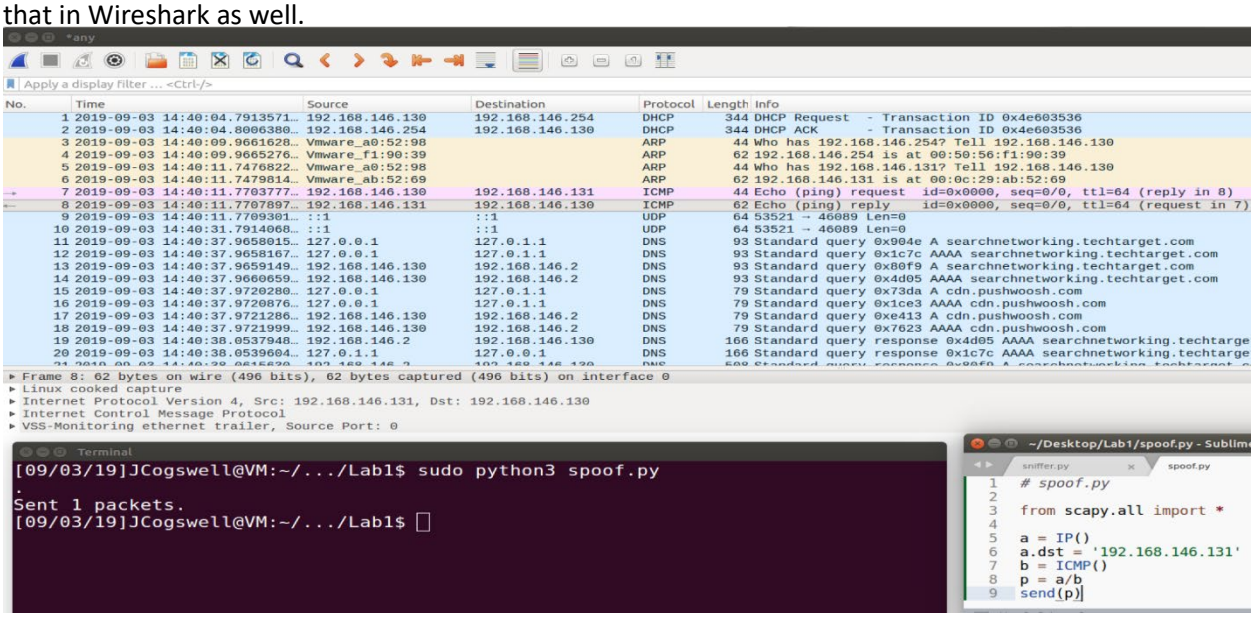


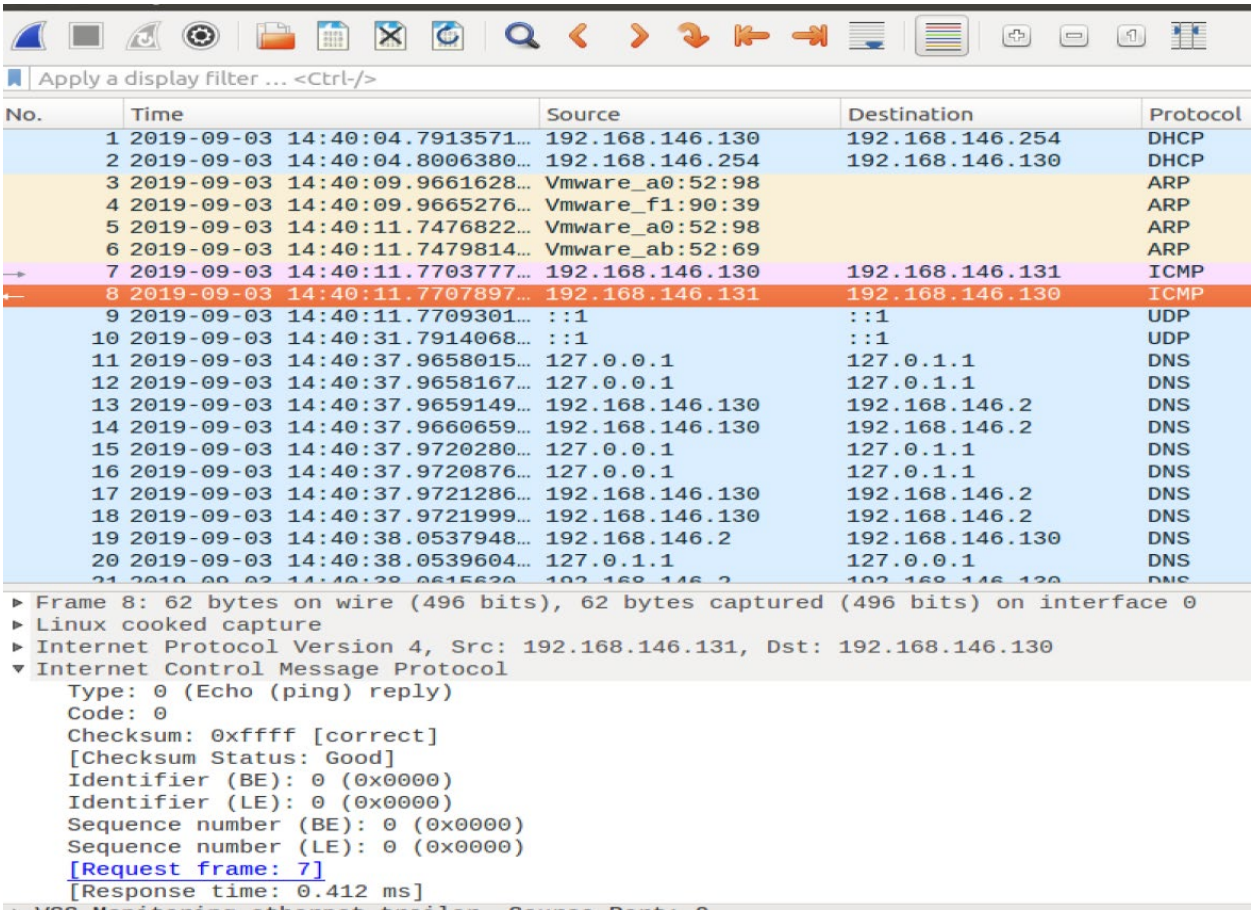
In the last part of Task 1.1b I sniffed all packets from the subnet 52.5.122.0/24 as shown in the screenshot below.



Task 1.2: Spoofing ICMP Packets

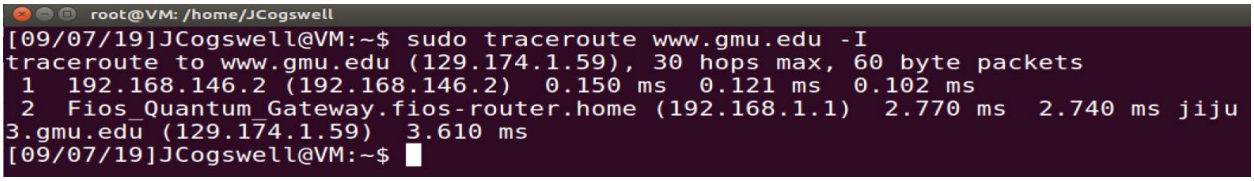
In this task I used scapy to spoof an ICMP packet to send a request to a separate remote VM and, as shown in the screen shot of Wireshark, the echo reply came back from the remote VM. This is evidence that the spoofed ICMP packet was successful. If I added a.src='**My IP Address**' then it would reflect that in Wireshark as well.

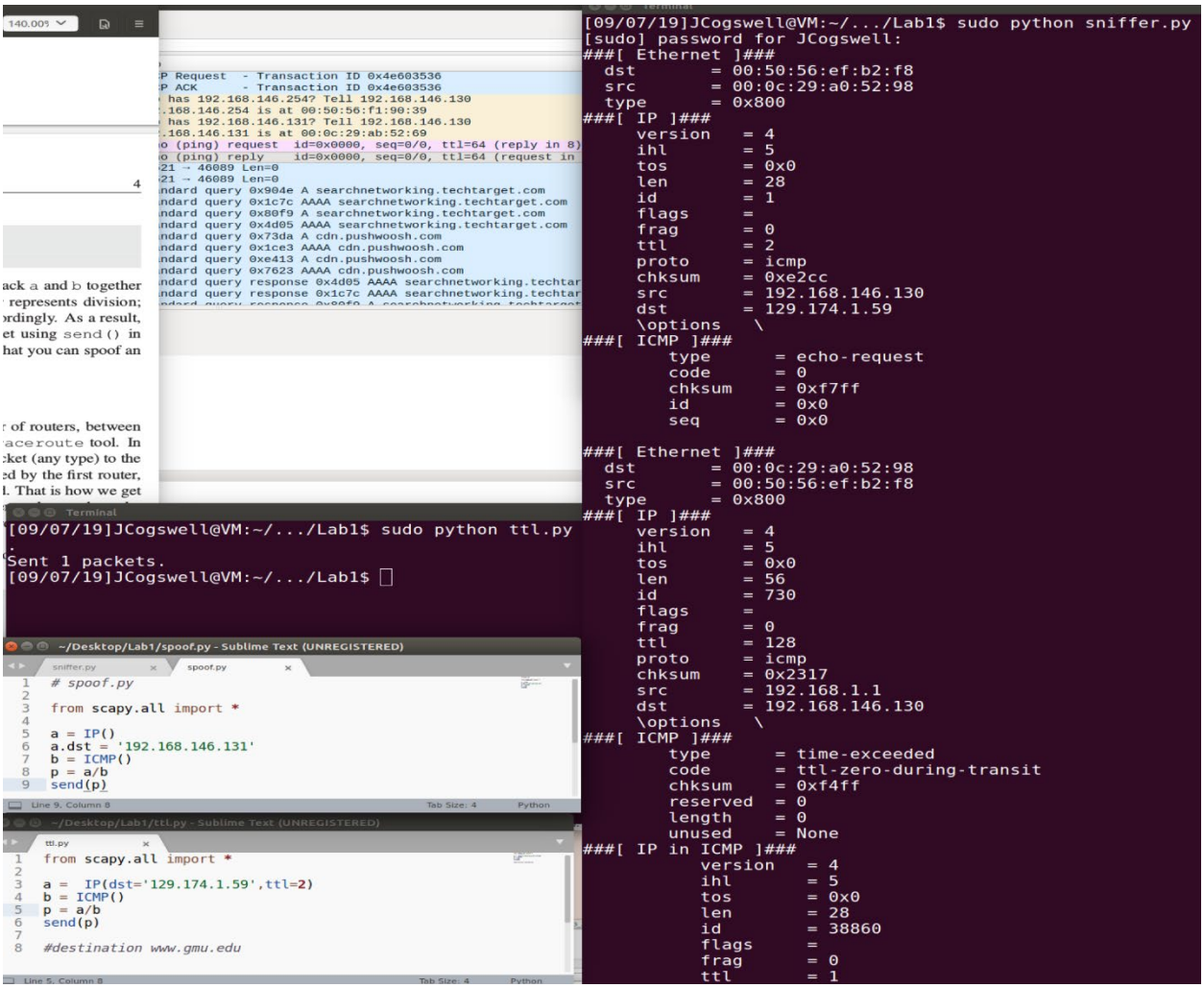


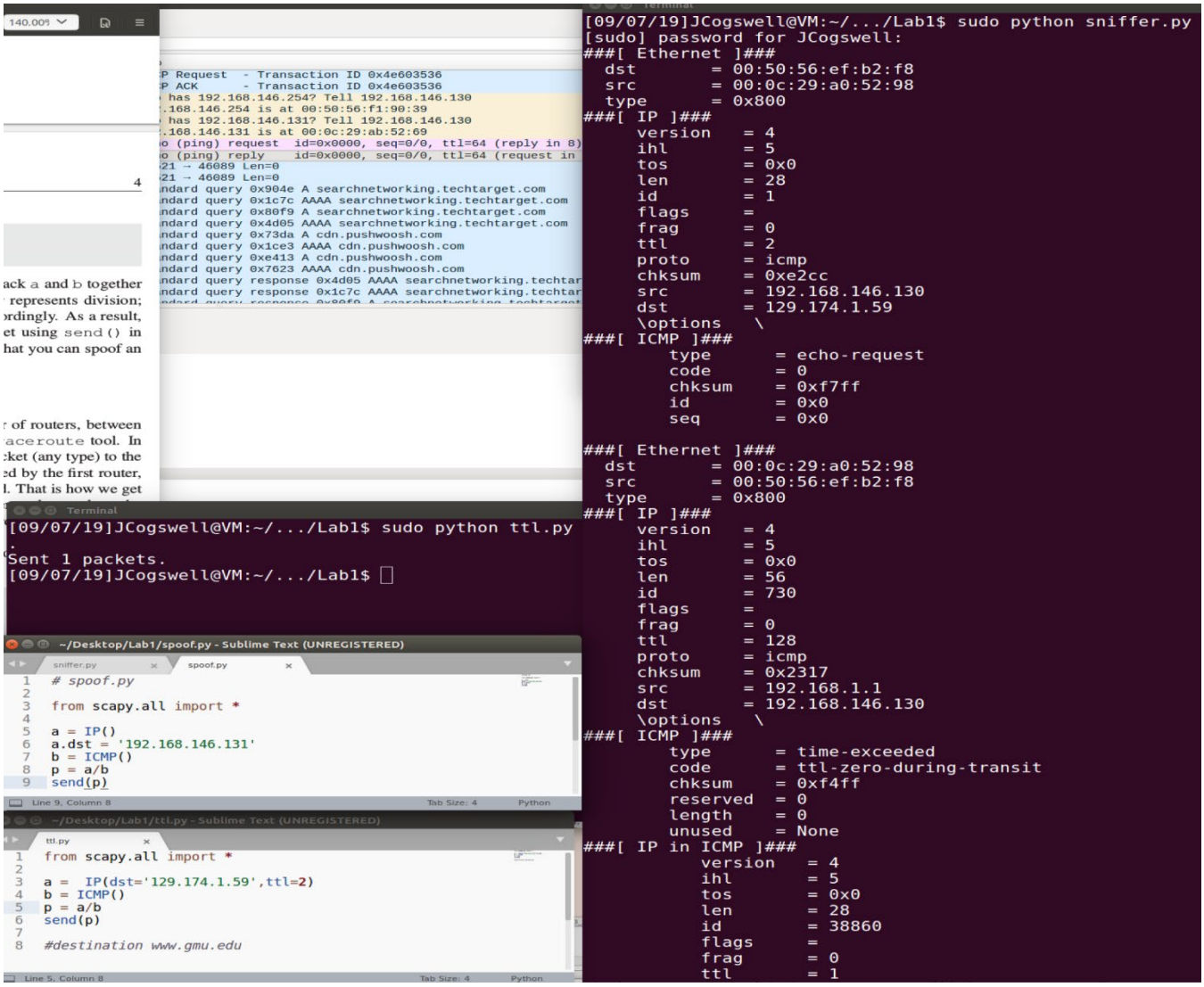


Task 1.3: Traceroute

In this task I used traceroute to see how many hops it would take to reach the destination (www.gmu.edu), and as shown in the top screenshot, it took three hops to reach the destination. However, the python script has the ttl set to 2 hops, hence why you see "time-exceeded" in the ICMP portion of the sniffed information (terminal on the right side shows this). The bottom screenshot shows the correct script with ttl set to 3.





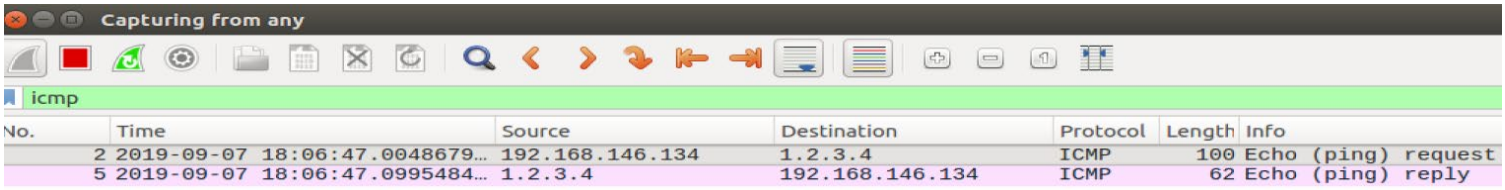

[09/07/19]JCogswell@VM:~/.../Lab1$ sudo python sniffer.py
[sudo] password for JCogswell:
###[Ethernet]###
dst = 00:50:56:ef:b2:f8
src = 00:0c:29:a0:52:98
type = 0x800
###[IP]###
version = 4
ihl = 5
tos = 0x0
len = 28
id = 1
flags =
frag = 0
ttl = 2
proto = icmp
chksum = 0xe2cc
src = 192.168.146.130
dst = 129.174.1.59
\options \
###[ICMP]###
type = echo-request
code = 0
chksum = 0xf7ff
id = 0x0
seq = 0x0
###[Ethernet]###
dst = 00:0c:29:a0:52:98
src = 00:50:56:ef:b2:f8
type = 0x800
###[IP]###
version = 4
ihl = 5
tos = 0x0
len = 56
id = 730
flags =
frag = 0
ttl = 128
proto = icmp
chksum = 0x2317
src = 192.168.1.1
dst = 192.168.146.130
\options \
###[ICMP]###
type = time-exceeded
code = ttl-zero-during-transit
chksum = 0xf4ff
reserved = 0
length = 0
unused = None
###[IP in ICMP]###
version = 4
ihl = 5
tos = 0x0
len = 28
id = 38860
flags =
frag = 0
ttl = 1
[09/07/19]JCogswell@VM:~/.../Lab1$ sudo python ttl.py
.
Sent 1 packets.
[09/07/19]JCogswell@VM:~/.../Lab1$
~/Desktop/Lab1/spoof.py - Sublime Text (UNREGISTERED)
sniffer.py
spoof.py
# spoof.py
from scapy.all import *
a = IP()
a.dst = '192.168.146.131'
b = ICMP()
p = a/b
send(p)
Line 9, Column 8
Tab Size: 4
Python
~/Desktop/Lab1/ttl.py - Sublime Text (UNREGISTERED)
ttl.py
from scapy.all import *
a = IP(dst='129.174.1.59',ttl=2)
b = ICMP()
p = a/b
send(p)
#destination www.gmu.edu
Line 5, Column 8
Tab Size: 4
Python

Task 1.4: Sniffing and-then Spoofing

In this task I used my victim VM to ping an arbitrary IP address. I then created a new program called spoofsniff.py that includes the code from both the spoof.py and sniff.py programs. It sniffs for ICMP packets and once it finds one, it sends a spoofed echo reply packet back to the source from its intended destination.

```
Terminal
CogswellJames@ATTACKER-Machine:~$ sudo python spoofsniff.py
###[ Ethernet ]###
  dst       = 00:50:56:ef:b2:f8
  src       = 00:0c:29:df:59:96
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 36651
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x5449
     src       = 192.168.146.134
     dst       = 1.2.3.4
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xa94c
        id        = 0x1a15
        seq       = 0x1
###[ Raw ]###
           load      = '\xf7)t]\xde\x12\x00\x00\x08\t\n\x0b\x
1\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#
67'

.
Sent 1 packets.
CogswellJames@ATTACKER-Machine:~$
```

```
~/Desktop/Lab1/spoofsniff.py - Sublime Text (UNREGISTERED)
ttl.py | spoof.py | sniffer.py | spoofsniff.py

def print_pkt(pkt):
    pkt.show()

    a = IP()
    a.dst = '192.168.146.134'
    a.src = '1.2.3.4'
    b = ICMP()
    b.type = 'echo-reply'
    b.seq = 1
    p = a/b
    send(p)

pkt = sniff(filter='icmp',count=1,prn=print_pkt)
```



I think that the task wanted us to somehow pull the information from the sniffed packet automatically and send a reply, but after about 4-5 hours of trying to figure it out, I gave up and stuck with the manual source and destination IP entry. The result of this is that the ping still eventually times out because it doesn't recognize the reply as a reply to its original ping request.

Task 2.1: Writing Packet Sniffing Program

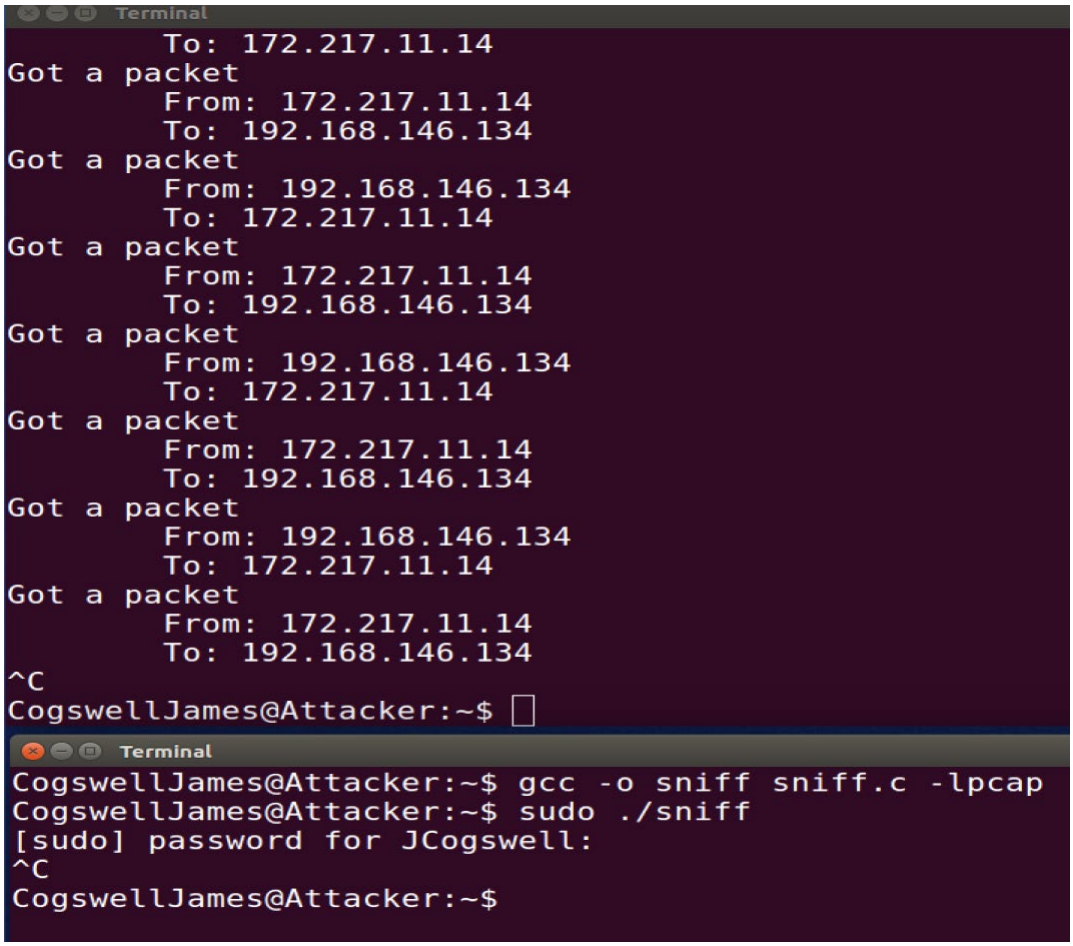
Task 2.1A: Understanding How a Sniffer Works

```
Terminal
CogswellJames@Attacker:~$ gcc -o sniff sniff.c -lpcap
CogswellJames@Attacker:~$ sudo ./sniff
Got a packet
Got a packet
        From: 192.168.146.130
        To: 192.168.146.2
Got a packet
        From: 192.168.146.2
        To: 192.168.146.130
Got a packet
        From: 192.168.146.130
        To: 172.217.10.78
Got a packet
        From: 172.217.10.78
        To: 192.168.146.130
Got a packet
        From: 192.168.146.130
        To: 192.168.146.2
Got a packet
        From: 192.168.146.2
        To: 192.168.146.130
Got a packet
Got a packet
^C
Terminal
CogswellJames@Attacker:~$ ping google.com
PING google.com (172.217.10.78) 56(84) bytes of data.
64 bytes from lga34s14-in-f14.1e100.net (172.217.10.78): icmp_seq=1 ttl=128 time
=14.0 ms
^C
--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 14.069/14.069/14.069/0.000 ms
CogswellJames@Attacker:~$
```

Question 1: The first library call is to set the device by looking up the name using ifconfig. Then I change it in the pcap_open_live() function. The pcap_open_live() function is used to open the device for sniffing by changing the 3rd argument to 1 to turn on promiscuous mode. Next, we need us use pcap_compile to compile the bpf program and then call pcap_setfilter() to filter out the types of packets we want to sniff. Next, we call pcap_loop() to actually sniff the packets. Lastly, we use pcap_close() to close the session and halt the sniffing.

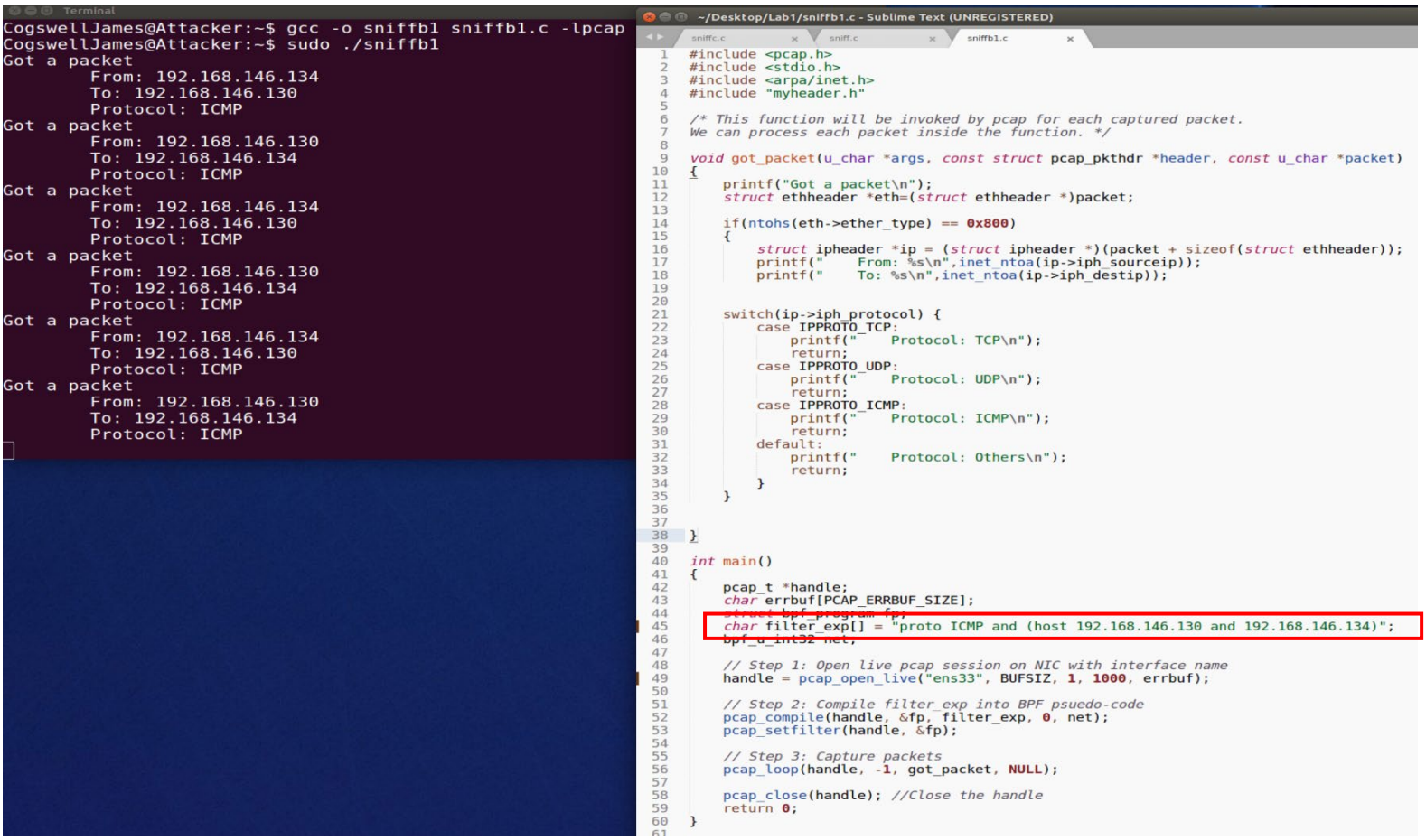
Question 2: We need root privilege to run a sniffer program because the operating system won't allow general users to manipulate and sniff packets very much. With root privilege you can utilize the commands necessary to spoof and sniff full packets. This program will fail when calling the pcap_open_live() function since it turns on promiscuous mode.

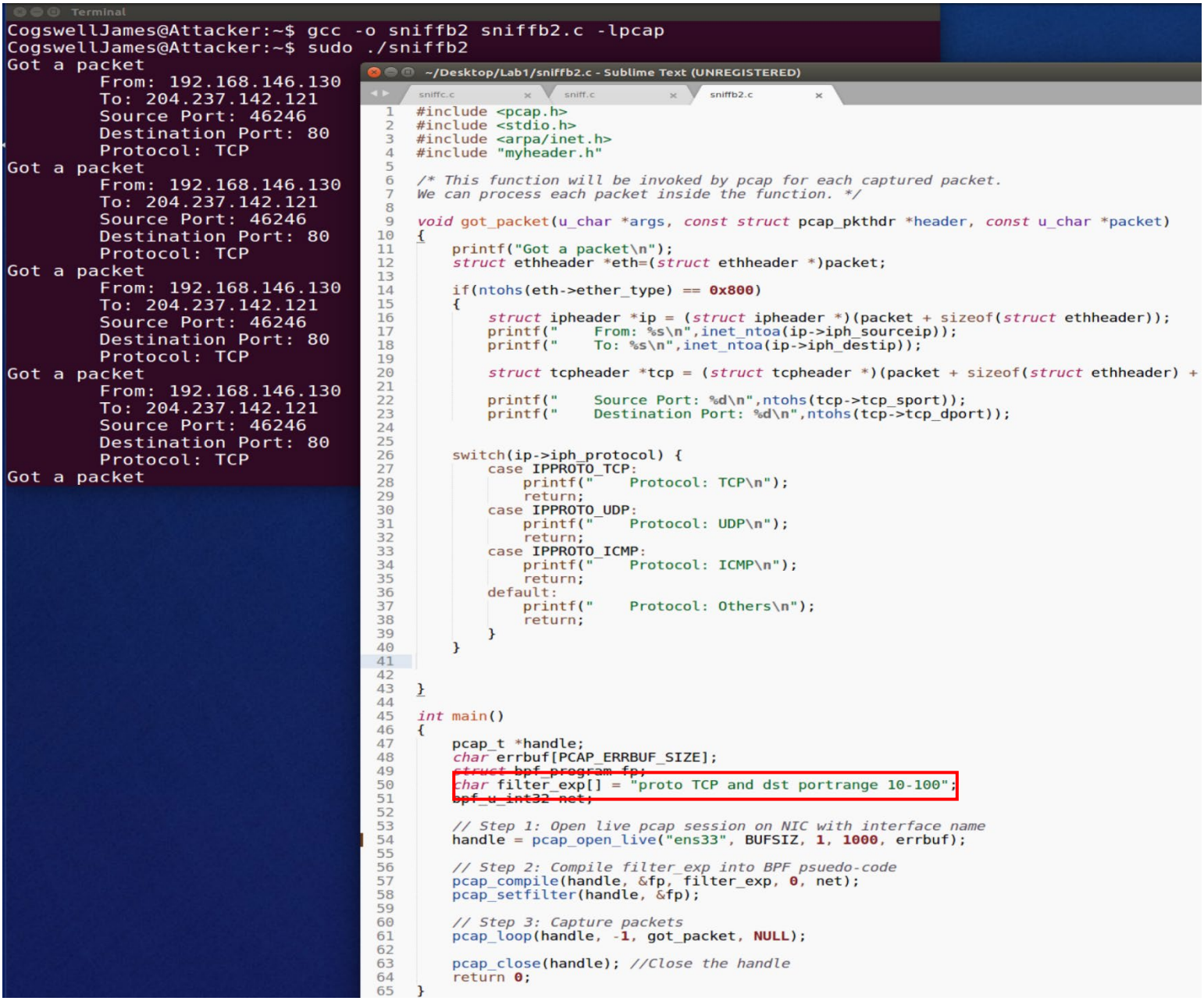
Question 3: The difference between promiscuous and non-promiscuous modes are that with promiscuous mode enabled, the program will sniff out all traffic on the network, whereas with promiscuous mode disabled, it will only sniff traffic being sent directly to you. You can demonstrate this by pinging an external IP from a separate VM in the same network and run the program once with promiscuous mode on and one time when it's off. Screenshots are below. Top screenshot is with it enabled. Bottom is with it disabled.



Task 2.1B: Writing Filters

In this part of task 2 I needed to use filters to capture specific types of traffic. First, I captured ICMP packets between two specific addresses (my two VMs) and this is represented in the first screenshot. The screenshot below shows the result of changing the filter to capture TCP packets with a destination port between 10-100. To produce TCP traffic that have ports between 10-100, I simply opened my web browser to the home page which is TCP port 80.





```
Terminal
CogswellJames@Attacker:~$ gcc -o sniffb2 sniffb2.c -lpcap
CogswellJames@Attacker:~$ sudo ./sniffb2
Got a packet
        From: 192.168.146.130
        To: 204.237.142.121
        Source Port: 46246
        Destination Port: 80
        Protocol: TCP
Got a packet
        From: 192.168.146.130
        To: 204.237.142.121
        Source Port: 46246
        Destination Port: 80
        Protocol: TCP
Got a packet
        From: 192.168.146.130
        To: 204.237.142.121
        Source Port: 46246
        Destination Port: 80
        Protocol: TCP
Got a packet
        From: 192.168.146.130
        To: 204.237.142.121
        Source Port: 46246
        Destination Port: 80
        Protocol: TCP
Got a packet
```

~/Desktop/Lab1/sniffb2.c - Sublime Text (UNREGISTERED)

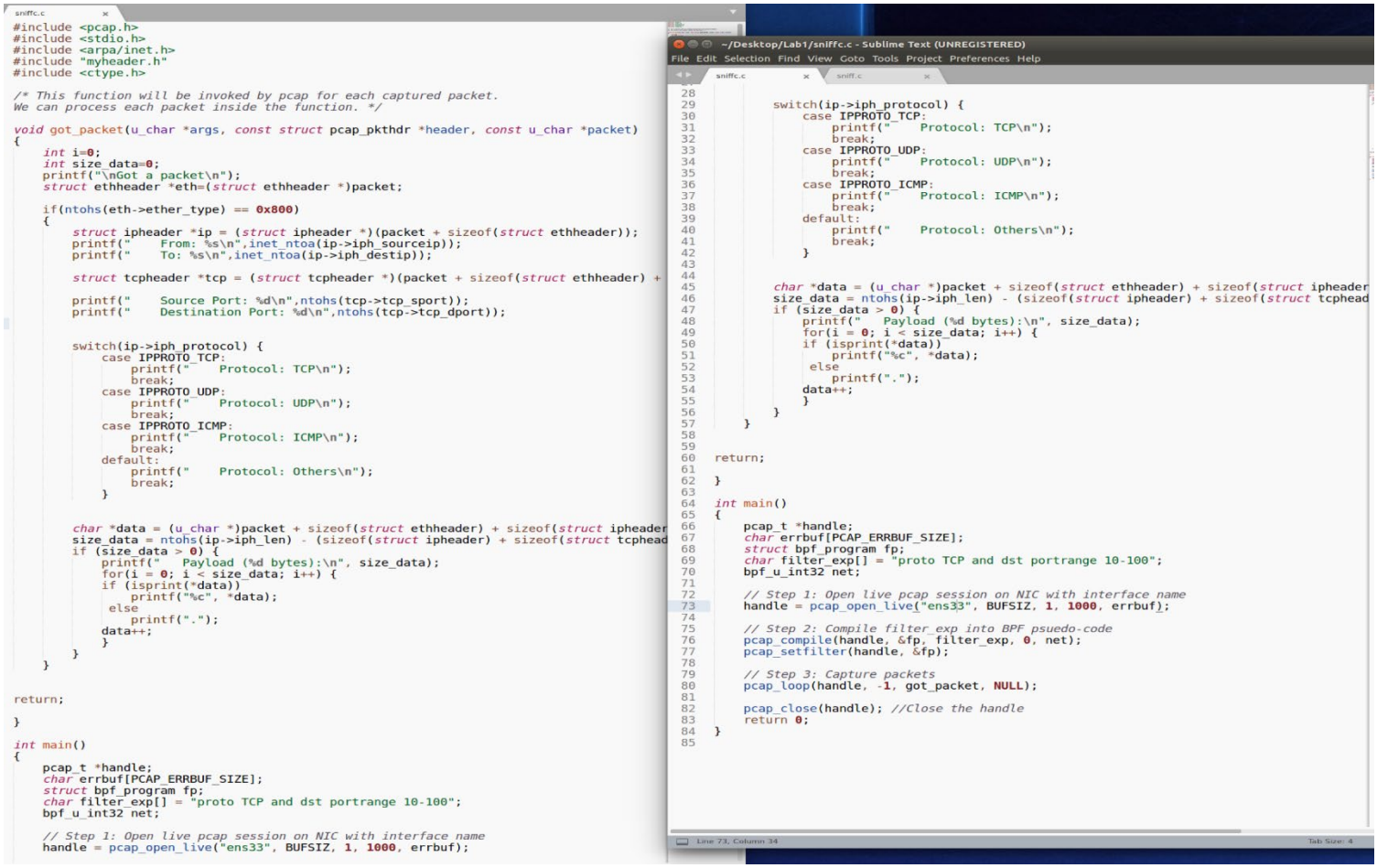
sniffc.c | sniff.c | sniffb2.c

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

/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function. */

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    printf("Got a packet\n");
    struct ethheader *eth=(struct ethheader *)packet;

    if(ntohs(eth->ether_type) == 0x800)
    {
        struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("    From: %s\n",inet_ntoa(ip->iph_sourceip));
        printf("    To: %s\n",inet_ntoa(ip->iph_destip));

        struct tcpheader *tcp = (struct tcpheader *)(packet + sizeof(struct ethheader) +

        printf("    Source Port: %d\n",ntohs(tcp->tcp_sport));
        printf("    Destination Port: %d\n",ntohs(tcp->tcp_dport));


    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("    Protocol: TCP\n");
            return;
        case IPPROTO_UDP:
            printf("    Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("    Protocol: ICMP\n");
            return;
        default:
            printf("    Protocol: Others\n");
            return;
        }
    }


}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto TCP and dst portrange 10-100";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with interface name
    handle = pcap_open_live("ens33", 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;
}
```

Task 2.1C: Sniffing Passwords

In this task I used a separate VM to telnet to my attacker VM and used the sniffing program to sniff the telnet traffic and show me the password. As shown in the screenshots, I have the complete program code with my device name 'ens33' as well as the output of the code which shows the password in plain text. It can be hard to see the plaintext password and username so I will highlight it with blue boxes.

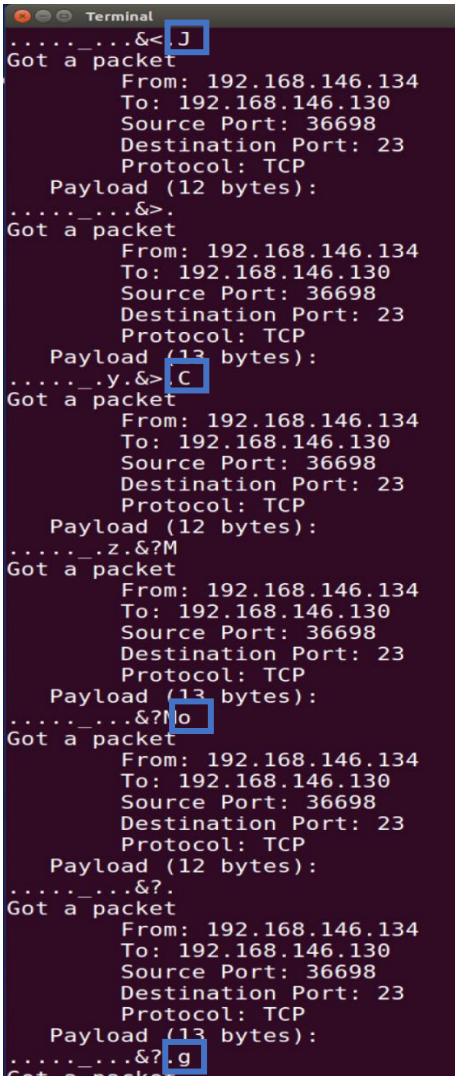
This is the sniffing code.

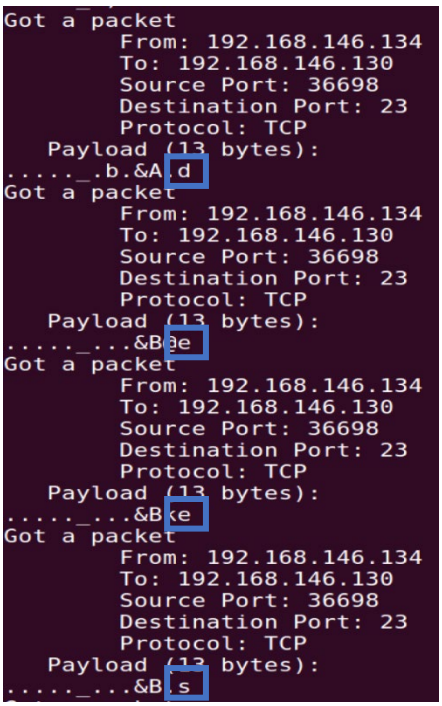


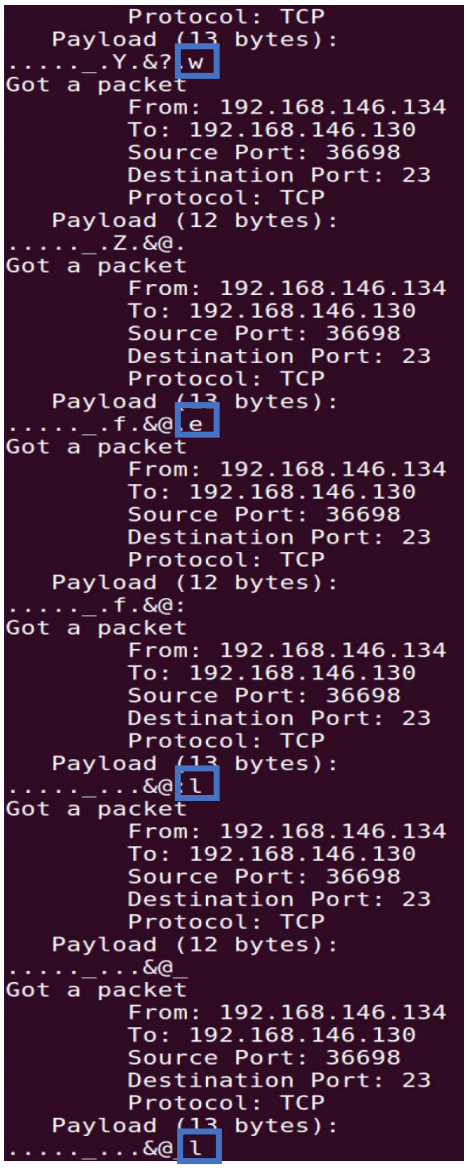
```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include "myheader.h"
#include <ctype.h>

/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function. */

void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    int i=0;
    int size_data=0;
    printf("\nGot a packet\n");
    struct ethheader *eth=(struct ethheader *)packet;

    if(ntohs(eth->ether_type) == 0x800)
    {
        struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
        printf("    From: %s\n",inet_ntoa(ip->iph_sourceip));
        printf("    To: %s\n",inet_ntoa(ip->iph_destip));

        struct tcpheader *tcp = (struct tcpheader *)(packet + sizeof(struct ethheader) +

        printf("    Source Port: %d\n",ntohs(tcp->tcp_sport));
        printf("    Destination Port: %d\n",ntohs(tcp->tcp_dport));


        switch(ip->iph_protocol) {
            case IPPROTO_TCP:
                printf("    Protocol: TCP\n");
                break;
            case IPPROTO_UDP:
                printf("    Protocol: UDP\n");
                break;
            case IPPROTO_ICMP:
                printf("    Protocol: ICMP\n");
                break;
            default:
                printf("    Protocol: Others\n");
                break;
            }


        char *data = (u_char *)packet + sizeof(struct ethheader) + sizeof(struct ipheader
        size_data = ntohs(ip->iph_len) - (sizeof(struct ipheader) + sizeof(struct tcphead
        if (size_data > 0) {
            printf("   Payload (%d bytes):\n", size_data);
            for(i = 0; i < size_data; i++) {
            if (isprint(*data))
                printf("%c", *data);
             else
                printf(".");
            data++;
            }
        }
    }


return;

}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto TCP and dst portrange 10-100";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with interface name
    handle = pcap_open_live("ens33", BUFSIZ, 1, 1000, errbuf);
```

```
~/Desktop/Lab1/sniffc.c - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
```

```c
        switch(ip->iph_protocol) {
            case IPPROTO_TCP:
                printf("    Protocol: TCP\n");
                break;
            case IPPROTO_UDP:
                printf("    Protocol: UDP\n");
                break;
            case IPPROTO_ICMP:
                printf("    Protocol: ICMP\n");
                break;
            default:
                printf("    Protocol: Others\n");
                break;
            }

        char *data = (u_char *)packet + sizeof(struct ethheader) + sizeof(struct ipheader
        size_data = ntohs(ip->iph_len) - (sizeof(struct ipheader) + sizeof(struct tcphead
        if (size_data > 0) {
            printf("   Payload (%d bytes):\n", size_data);
            for(i = 0; i < size_data; i++) {
            if (isprint(*data))
                printf("%c", *data);
             else
                printf(".");
            data++;
            }
        }
    }


return;

}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "proto TCP and dst portrange 10-100";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with interface name
    handle = pcap_open_live("ens33", 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;
}
```







Username: JCogswell

Password: dees

Note: I only have screenshots of JCogwell. I did not want to take a whole new screenshot just to capture the ‘s’ but it’s there.

Task 2.2: A) Write a spoofing program. B) Spoof an ICMP Echo Request.

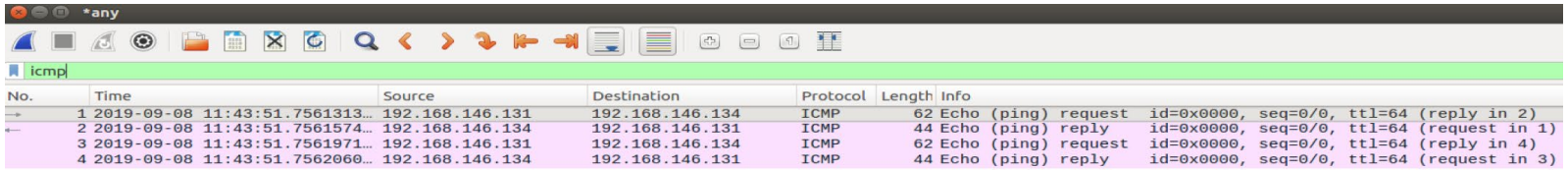
In these tasks, I needed to use the provided icmpspoof.c program and edit it for my network to work. I then sent the packet from my attacker VM-A to the victim VM-B with a spoofed source IP of VM-C which is a live IP address. The screenshots show the wireshark capture from the victim VM-B, the code (two parts because it's a long program, and finally, the output of the program from sending the packet. These tasks involve the same thing, so I combined them into one. I have the screenshots showing the spoofing program and the screenshots of the wireshark capturing the ICMP spoof. In the wireshark capture screenshot, you can see that the source IP address is the IP address of VM-C (192.168.146.131).

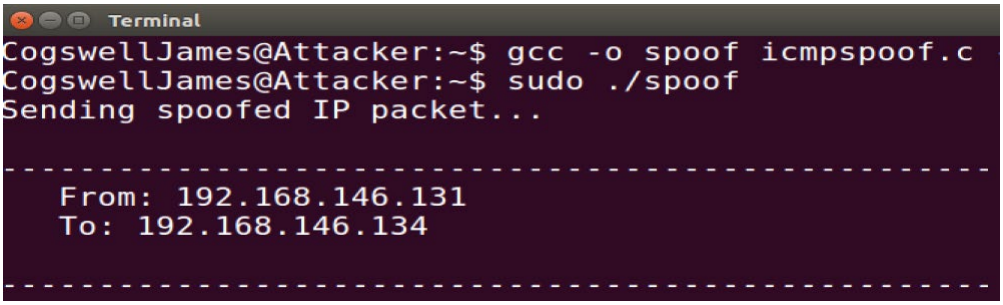
~/Desktop/Lab1/icmpspoof.c - Sublime Text (UNREGISTERED)

icmpspoof.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#include "myheader.h"

#define SRC_IP    "192.168.146.131"
#define DEST_IP   "192.168.146.134"


unsigned short in_cksum(unsigned short *buf,int length);
void send_raw_ip_packet(struct ipheader* ip);



/******************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
******************************************************************/
int main() {
   char buffer[PACKET_LEN];

   memset(buffer, 0, PACKET_LEN);

   /*********************************************************
      Step 1: Fill in the ICMP header.
    ********************************************************/
   struct icmpheader *icmp = (struct icmpheader *) (buffer +
                                                    sizeof(struct ipheader));
   icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                sizeof(struct icmpheader));

   /*********************************************************
      Step 2: Fill in the IP header.
    ********************************************************/
   struct ipheader *ip = (struct ipheader *) buffer;
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_tos = 16;
   ip->iph_ident = htons(54321);
   ip->iph_ttl = 64;
   ip->iph_sourceip.s_addr = inet_addr(SRC_IP);
   ip->iph_destip.s_addr = inet_addr(DEST_IP);
   ip->iph_protocol = IPPROTO_ICMP; // The value is 1, representing ICMP.
   ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));

   // No need to set the following fileds, as they will be set by the system.
   // ip->iph_chksum = ...;

   /*********************************************************
      Step 3: Finally, send the spoofed packet
    ********************************************************/
   send_raw_ip_packet (ip);

   return 0;
}
```

```
/*****************************************************************************
  Given an IP packet, send it out using raw socket.
*****************************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Create a raw network socket, and set its options.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

    // Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Send the packet out.
    printf("Sending spoofed IP packet...\n");
    sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_i
    if(sendto(sock,ip,ntohs(ip->iph_len),0,(struct sockaddr *)&dest_info,sizeof(dest_inf
    {
        perror("PACKET NOT SENT\n");
        return;
    }
    else {
        printf("\n-----------------------------------------------------\n");
        printf("    From: %s\n",inet_ntoa(ip->iph_sourceip));
        printf("    To: %s\n",inet_ntoa(ip->iph_destip));
        printf("\n-----------------------------------------------------\n");
    }
    close(sock);
}


unsigned short in_cksum(unsigned short *buf,int length)
{
        unsigned short *w = buf;
        int nleft = length;
        int sum = 0;
        unsigned short temp=0;

        /*
        * The algorithm uses a 32 bit accumulator (sum), adds
        * sequential 16 bit words to it, and at the end, folds back all the
        * carry bits from the top 16 bits into the lower 16 bits.
        */
        while (nleft > 1)  {
                sum += *w++;
                nleft -= 2;
        }

        /* treat the odd byte at the end, if any */
        if (nleft == 1) {
                *(u_char *)(&temp) = *(u_char *)w ;
                sum += temp;
        }

        /* add back carry outs from top 16 bits to low 16 bits */
        sum = (sum >> 16) + (sum & 0xffff);     // add hi 16 to low 16
        sum += (sum >> 16);                     // add carry
        return (unsigned short)(~sum);
}
```


*any
icmp
No. Time Source Destination Protocol Length Info
1 2019-09-08 11:43:51.7561313… 192.168.146.131 192.168.146.134 ICMP 62 Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 2)
2 2019-09-08 11:43:51.7561574… 192.168.146.134 192.168.146.131 ICMP 44 Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 1)
3 2019-09-08 11:43:51.7561971… 192.168.146.131 192.168.146.134 ICMP 62 Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 4)
4 2019-09-08 11:43:51.7562060… 192.168.146.134 192.168.146.131 ICMP 44 Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 3)


Terminal
CogswellJames@Attacker:~$ gcc -o spoof icmpspoof.c
CogswellJames@Attacker:~$ sudo ./spoof
Sending spoofed IP packet...
From: 192.168.146.131
To: 192.168.146.134

2: Ping Scans

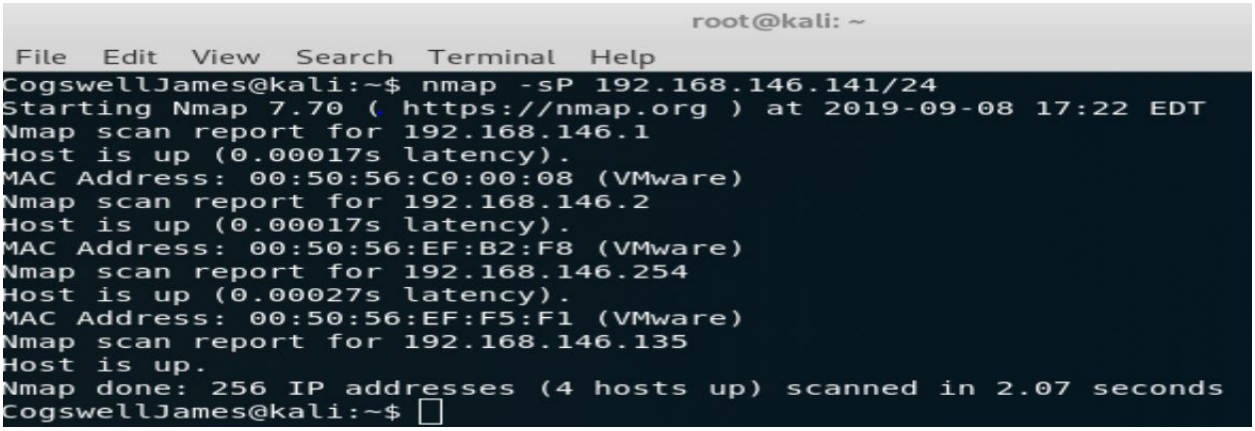
We will run nmap in the terminal window. Find your IP address before proceeding. Your IP address will be used in this and later sections of the lab. IP: 192.168.146.141

2.1: Use a ping scan to check which of the 256 IP addresses in a /24 address space around your IP address are up. How many IP addresses are up in this range? Don't count manually. Pipe the output of nmap to a command that will count for you.

```
# nmap -sP YOUR_IP_ADDRESS/24
```

4 hosts up.

Screenshot of the execution is below. You can see at the bottom where it says (4 hosts up).

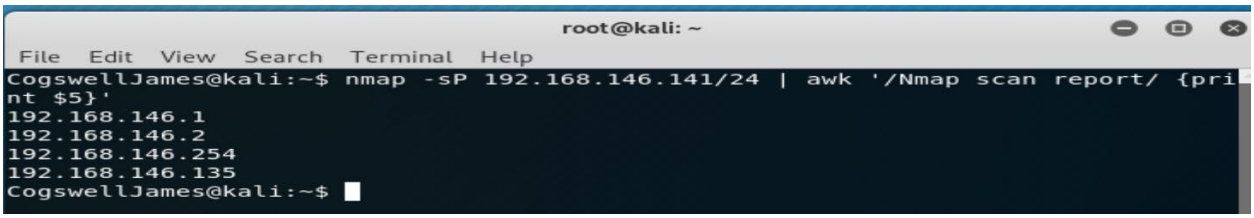


2.2: List the highest and lowest IP addresses that were shown to be up by the ping scan. The awk command is useful for parsing nmap output, as it can find the lines that you want using a pattern in //'s and then select the column of output that contains the information you want to retrieve.

```
# nmap -sP YOUR_IP_ADDRESS/24 | awk '/Nmap scan report/ {print $5}'
```

Low: 192.168.146.1 High: 192.168.146.254

As shown in the screenshot, the largest and smallest IPs that were shown to be up by the ping scan were printed to the terminal. We used the awk command to only show the IPs from the original nmap scan report (the unparsed nmap scan is shown in the screenshot from the first question).



3: Port Scans

In this section, we will scan TCP ports in several different ways to gather information on an example target system. We will not scan UDP ports, as we do not have enough time in lab to wait for the 15-20 minutes a UDP scan can take.

3.1: What ports are open on the scanme.nmap.org test server? Use a TCP connect scan. Your answer must include only the port numbers. Do not include other parts of nmap output.

```
# nmap -sT scanme.nmap.org
```

22, 80, 9929, 31337

The open ports shown when scanning scanme.nmap.org are 22, 80, 9929, and 31337 as shown in the screenshot below.

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
CogswellJames@kali:~$ nmap -sT scanme.nmap.org
Starting Nmap 7.70 ( https://nmap.org ) at 2019-09-08 17:34 EDT
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.053s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2
f
Not shown: 996 filtered ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
9929/tcp  open  nping-echo
31337/tcp open  Elite

Nmap done: 1 IP address (1 host up) scanned in 15.73 seconds
CogswellJames@kali:~$
```

3.2: Using a TCP SYN scan, what ports do you find open on scanme.nmap.org?

```
# nmap -sS scanme.nmap.org
```

22, 80, 514, 9929, 31337

3.3: Looking at the output of the two scans outside the ports listed, what differences do you find between the TCP connect and SYN scans? If there is no difference, then just write "None" below.

514

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
CogswellJames@kali:~$ nmap -sS scanme.nmap.org
Starting Nmap 7.70 ( https://nmap.org ) at 2019-09-08 17:38 EDT
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (2.4s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2
f
Not shown: 995 closed ports
PORT      STATE    SERVICE
22/tcp    open     ssh
80/tcp    open     http
514/tcp   filtered shell
9929/tcp  open     nping-echo
31337/tcp open     Elite

Nmap done: 1 IP address (1 host up) scanned in 93.64 seconds
CogswellJames@kali:~$
```

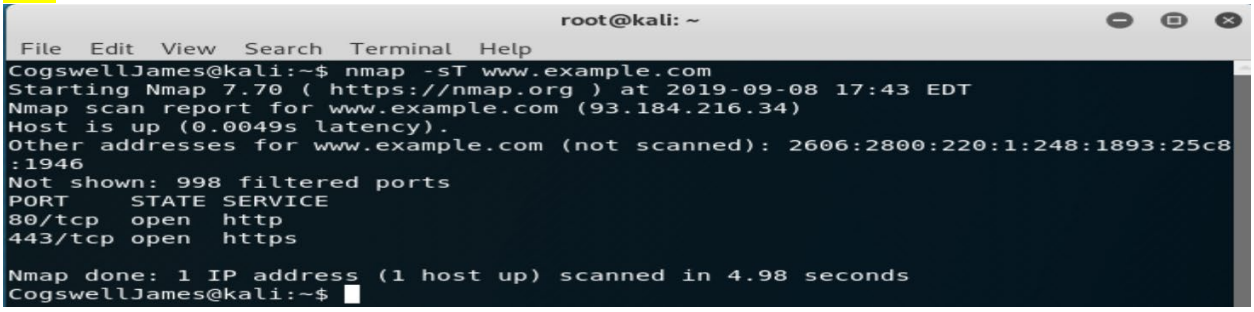
The differences between the two scans are listed above. Ports 25 and 54 are the only differences.

3.4: Some machines are behind a firewall, which filters connections to some ports, preventing nmap from receiving any response from those ports. Blocked ports may be listed as either "filtered" or "closed". To see an example of such a scan, perform a TCP connect scan on www.example.com. Your answer must include port numbers for both closed and open ports. Do not include other parts of nmap output.

nmap -sT www.example.com

80, 443

The screenshot shows that the two open ports from the nmap scan of www.exacmple.com are 80 and 443.

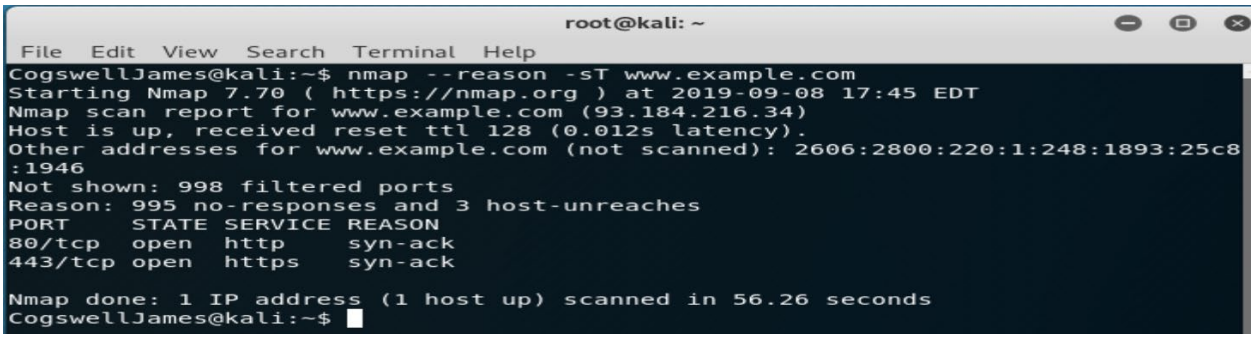


3.5: To determine why a scan returns the results that it does, use the —reason option. Explain the reasons that ports are listed as open, closed, or filtered in the scan of www.example.com.

nmap —reason -sT www.example.com

998 filtered ports → Reason: No-responses. 80 & 443 are open because it's a website so it has to have HTTP & HTTPS open.

As stated in the screenshot above, the reasons that 998 filtered ports aren't shown are because of no-response and 3 of the hosts were unable to be reached. The open ports are 'open' because it received a 3-way TCP handshake (syn-ack). These open ports are open because it's a website and need to allow HTTP (80) and HTTPS (443).



3.6: To see every packet sent by a scan, use the —packet-trace option. We will save this output in a file for further analysis using I/O redirection. We do not redirect STDERR, so we will still see error output on the screen.

In the box below, explain what packet is sent first in the scan and count how many packets are sent in total. Use a command to do the counting for you, but be sure not to count regular nmap output (what you would see without the trace option).

```
# nmap --packet-trace -sS scanme.nmap.org >trace.out
# less trace.out
```

First packet is ICMP Echo Request.
1 packet.

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
CogswellJames@kali:~$ nmap --packet-trace -sS scanme.nmap.org >trace.out
NSOCK INFO [0.1400s] nsock_iod_new2(): nsock_iod_new (IOD #1)
NSOCK INFO [0.1410s] nsock_connect_udp(): UDP connection requested to 192.168.14
6.2:53 (IOD #1) EID 8
NSOCK INFO [0.1410s] nsock_read(): Read request from IOD #1 [192.168.146.2:53] (
timeout: -1ms) EID 18
NSOCK INFO [0.1410s] nsock_write(): Write request for 43 bytes to IOD #1 EID 27
[192.168.146.2:53]
NSOCK INFO [0.1410s] nsock_trace_handler_callback(): Callback: CONNECT SUCCESS f
or EID 8 [192.168.146.2:53]
NSOCK INFO [0.1410s] nsock_trace_handler_callback(): Callback: WRITE SUCCESS for
 EID 27 [192.168.146.2:53]
NSOCK INFO [0.1470s] nsock_trace_handler_callback(): Callback: READ SUCCESS for
EID 18 [192.168.146.2:53] (72 bytes): ............156.32.33.45.in-addr.arpa....
..............scanme.nmap.org.
NSOCK INFO [0.1470s] nsock_read(): Read request from IOD #1 [192.168.146.2:53] (
timeout: -1ms) EID 34
NSOCK INFO [0.1470s] nsock_iod_delete(): nsock_iod_delete (IOD #1)
NSOCK INFO [0.1470s] nevent_delete(): nevent_delete on event #34 (type READ)
CogswellJames@kali:~$
File  Edit  View  Search  Terminal  Help
Starting Nmap 7.70 ( https://nmap.org ) at 2019-09-08 17:46 EDT
SENT (0.1017s) ICMP [192.168.146.135 > 45.33.32.156 Echo request (type=8/code=0)
 id=55054 seq=0] IP [ttl=38 id=8414 iplen=28 ]
SENT (0.1018s) TCP 192.168.146.135:44572 > 45.33.32.156:443 S ttl=59 id=47387 ip
len=44  seq=1099095721 win=1024 <mss 1460>
SENT (0.1019s) TCP 192.168.146.135:44572 > 45.33.32.156:80 A ttl=44 id=21021 ipl
en=40  seq=0 win=1024
SENT (0.1019s) ICMP [192.168.146.135 > 45.33.32.156 Timestamp request (type=13/c
ode=0) id=63071 seq=0 orig=0 recv=0 trans=0] IP [ttl=55 id=44234 iplen=40 ]
RCVD (0.1026s) TCP 45.33.32.156:80 > 192.168.146.135:44572 R ttl=128 id=33128 ip
len=40  seq=1099095721 win=32767
SENT (0.1891s) TCP 192.168.146.135:44828 > 45.33.32.156:3389 S ttl=46 id=60844 i
plen=44  seq=3988187500 win=1024 <mss 1460>
SENT (0.1892s) TCP 192.168.146.135:44828 > 45.33.32.156:22 S ttl=40 id=12572 ipl
en=44  seq=3988187500 win=1024 <mss 1460>
SENT (0.1892s) TCP 192.168.146.135:44828 > 45.33.32.156:139 S ttl=57 id=6950 ipl
en=44  seq=3988187500 win=1024 <mss 1460>
SENT (0.1892s) TCP 192.168.146.135:44828 > 45.33.32.156:25 S ttl=43 id=49179 ipl
en=44  seq=3988187500 win=1024 <mss 1460>
SENT (0.1893s) TCP 192.168.146.135:44828 > 45.33.32.156:8888 S ttl=43 id=26505 i
plen=44  seq=3988187500 win=1024 <mss 1460>
SENT (0.1893s) TCP 192.168.146.135:44828 > 45.33.32.156:1025 S ttl=45 id=18111 i
plen=44  seq=3988187500 win=1024 <mss 1460>
trace.out
File  Edit  View  Search  Terminal  Help
7 iplen=40  seq=1500782244 win=64240
RCVD (11.7686s) TCP 45.33.32.156:9595 > 192.168.146.135:44828 RA ttl=128 id=3514
8 iplen=40  seq=563938413 win=64240
RCVD (11.7697s) TCP 45.33.32.156:13456 > 192.168.146.135:44828 RA ttl=128 id=351
49 iplen=40  seq=5526116 win=64240
RCVD (11.7716s) TCP 45.33.32.156:3905 > 192.168.146.135:44829 RA ttl=128 id=3515
0 iplen=40  seq=85612620 win=64240
SENT (11.7746s) TCP 192.168.146.135:44846 > 45.33.32.156:80 A ttl=54 id=34127 ip
len=40  seq=0 win=1024
RCVD (11.7747s) TCP 45.33.32.156:80 > 192.168.146.135:44846 R ttl=128 id=35151 i
plen=40  seq=3853967724 win=32767
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.023s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2
f
Not shown: 996 filtered ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
9929/tcp  open  nping-echo
31337/tcp open  Elite

Nmap done: 1 IP address (1 host up) scanned in 11.81 seconds
(END)
```

I'm not sure how to tell have it count the amount of packets sent automatically, but I used the provided command and this is what I got. The first packet sent was an ICMP echo request. Given how long it took me to scroll to the bottom of trace.out, I would say that there were about 1000 sent packets.

3.7: The nmap scanner can return additional information, including service versions, OS identification, and tracerouting. The -A option will perform all of these tests. The -T4 option tells nmap to use aggressive packet timing, which can be dangerous as it can cause some older machines to crash. However, scanme.nmap.org is configured so that it will have no problems with the -T4 option. Even with the faster speed, this scan will take longer than previous ones due to the large number of tests performed. We add the -v option so that you can see the scan in progress

```
# nmap -v -A -T4 scanme.nmap.org | less
```

Based on the output of the scan above, answer the following questions:

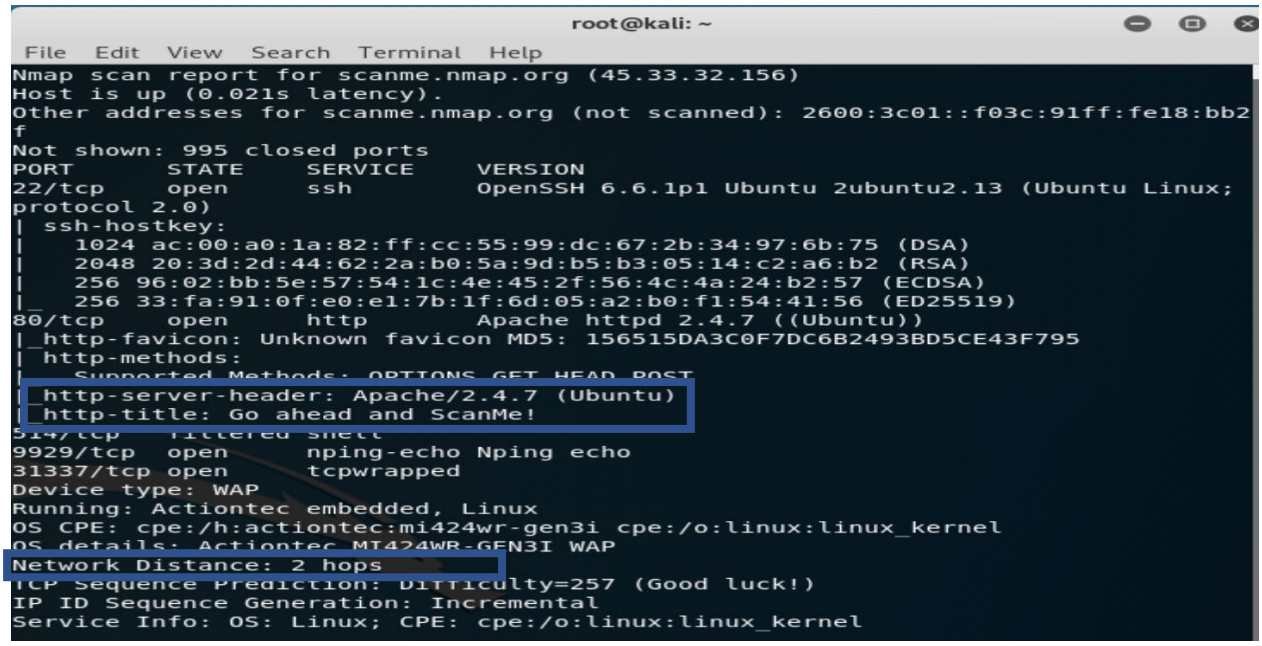
1. What is the server software name and version for each of the ports?
2. What title would you see in the top of your web browser if you contacted the web server at scanme.nmap.org?
3. How many network hops does it take to reach scanme.nmap.org from your VM?

1. ~~OpenSSH 6.6.1p1~~ 80 http Apache httpd 2.4.7 (Ubuntu)

2. Go ahead and ScanMe!

3. 2 hops

Screenshot of execution below. Answer above.



3.8: UDP scans are much slower than TCP scans due to the unreliability of UDP, so the scan in this question will require the longest amount of time. Write the list of open UDP ports in the box below.

```
# nmap -v -sU scanme.nmap.org
```

ntp

123

The execution is shown below. The answer is port 123 being the only open UDP port.