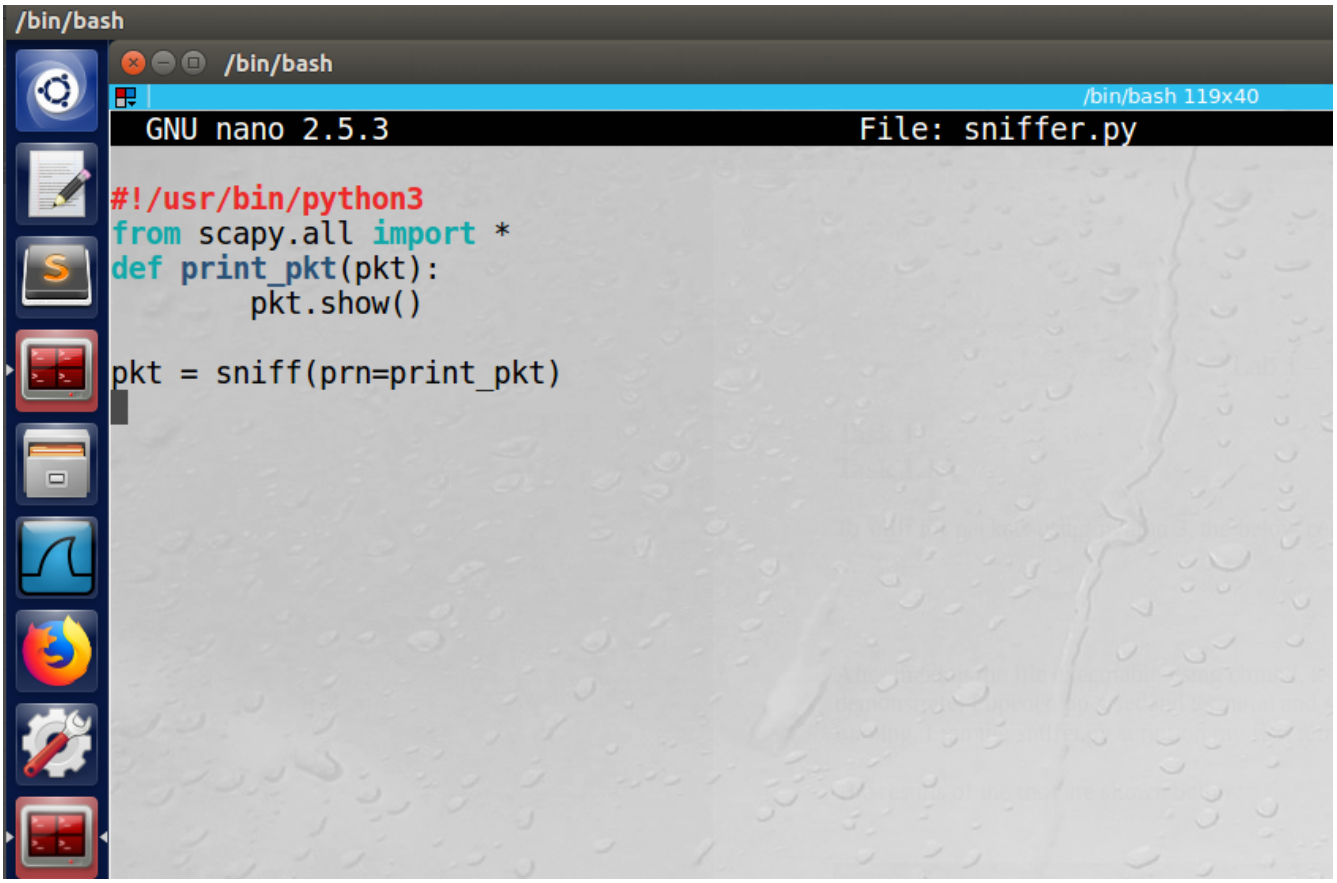
Lab 1 – Kevin Martin

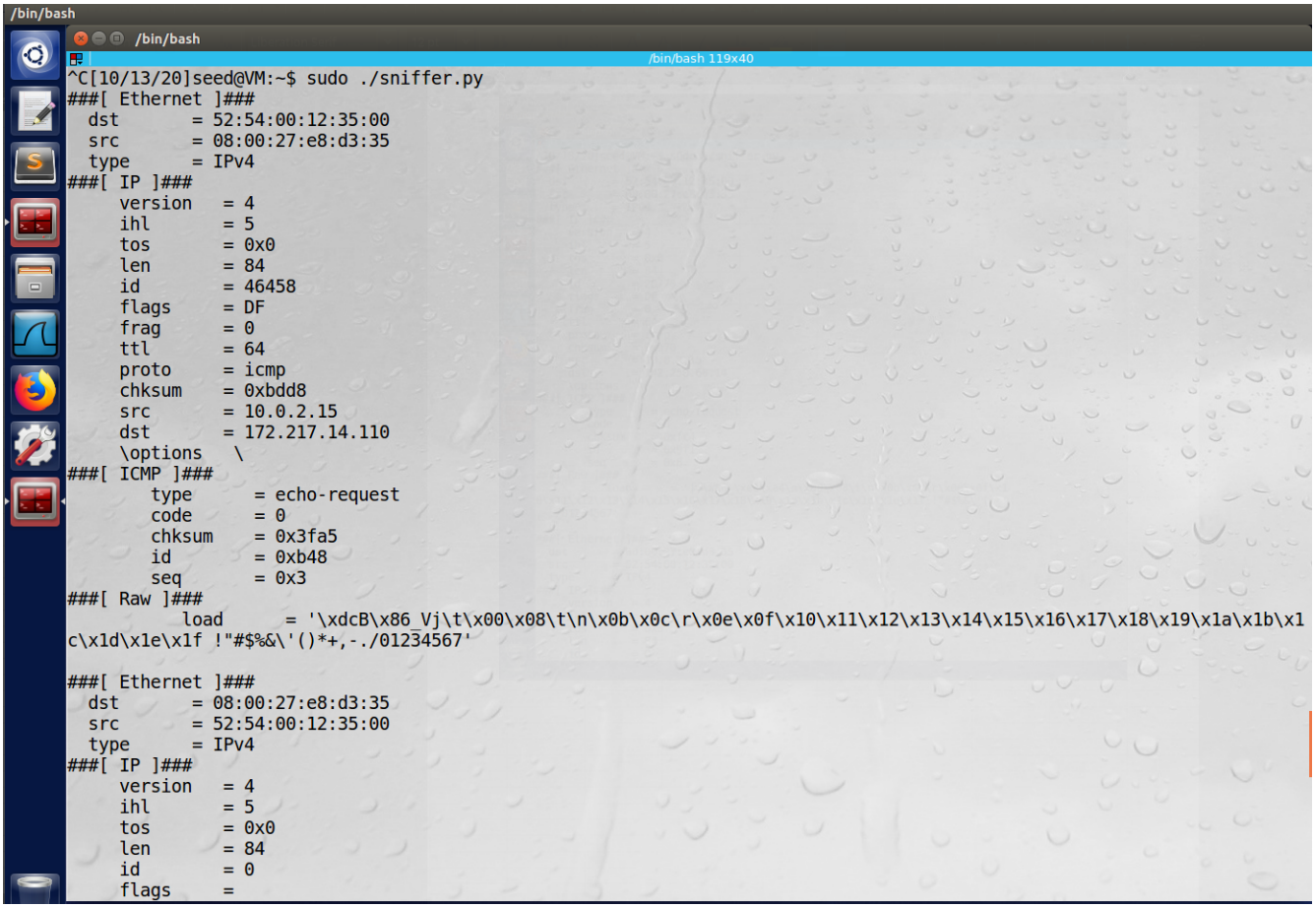
Task 1
Task 1.1A

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

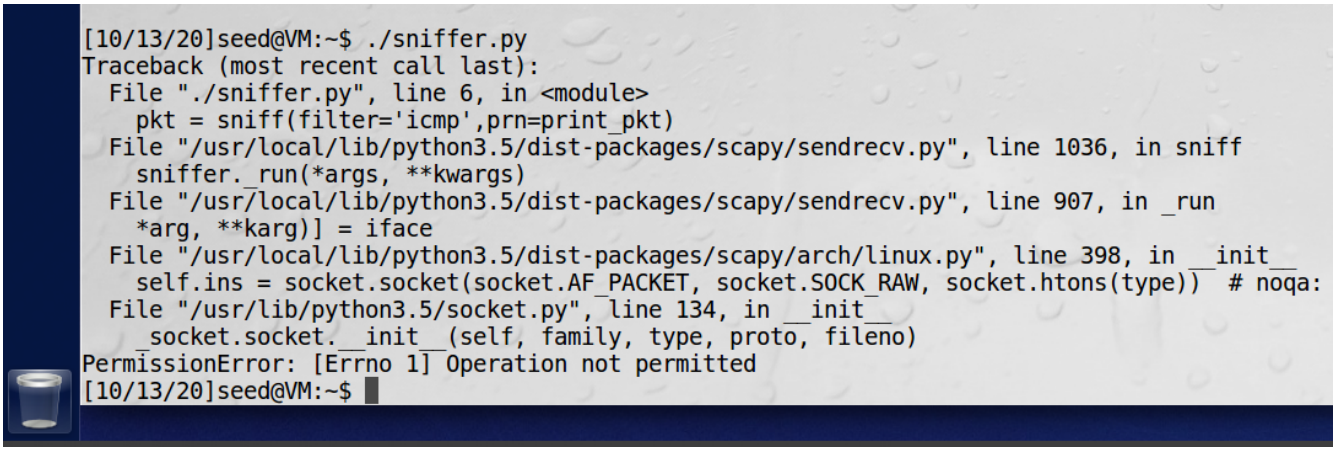


After making the file executable using chmod, it was run both in root and non-not root. In order to demonstrate, I opened up a second terminal and simply entered “ping google.com”. While that was running, I ran the sniffer.py script on my first terminal.

The results of the root are shown below:



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

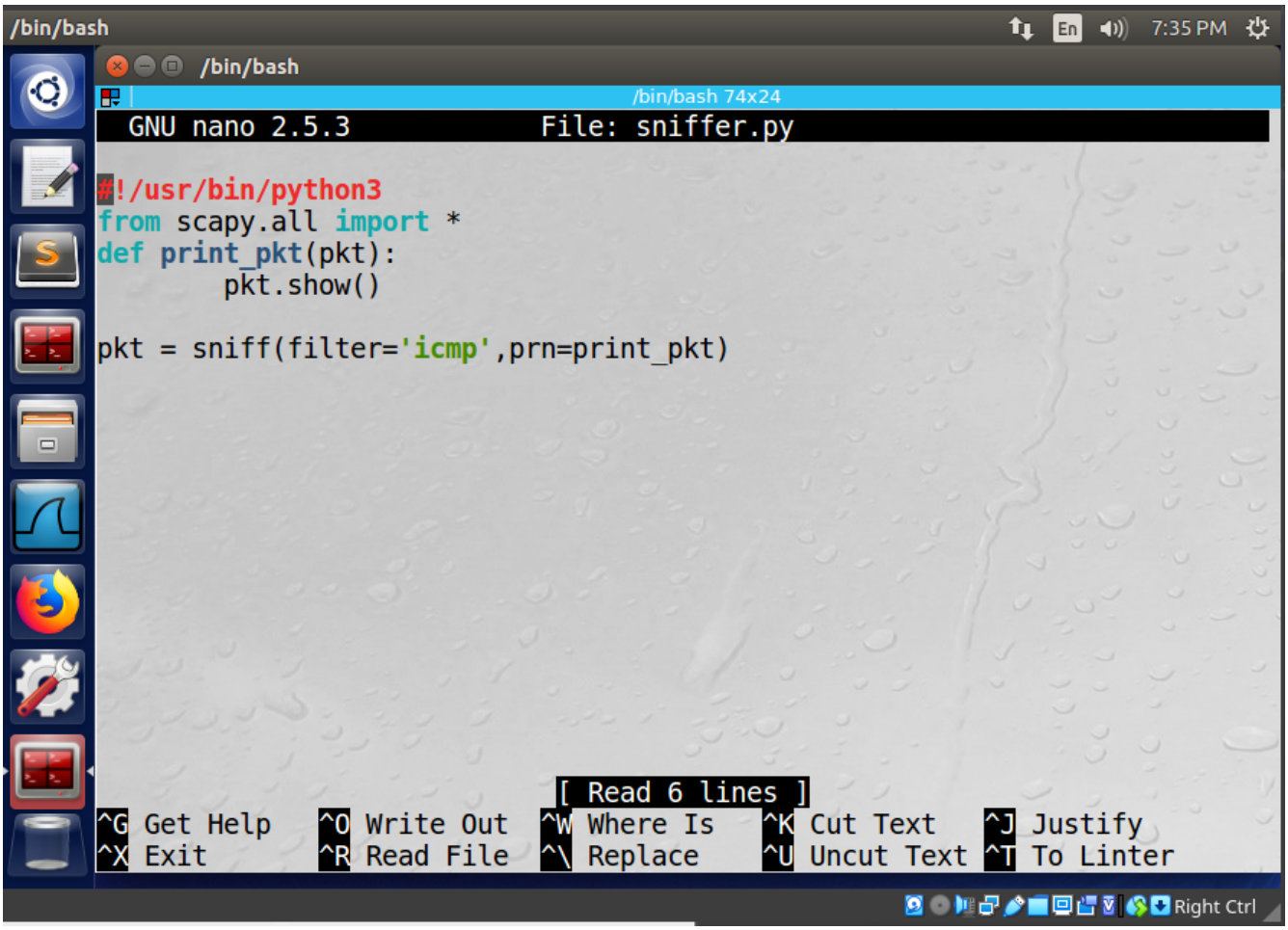


Observation: Running the program in root, we see that we are indeed caputring network traffic using sniffing. Note the source (src) is the host machine's IP address, and the destination (dest) is that of this particular Google server.

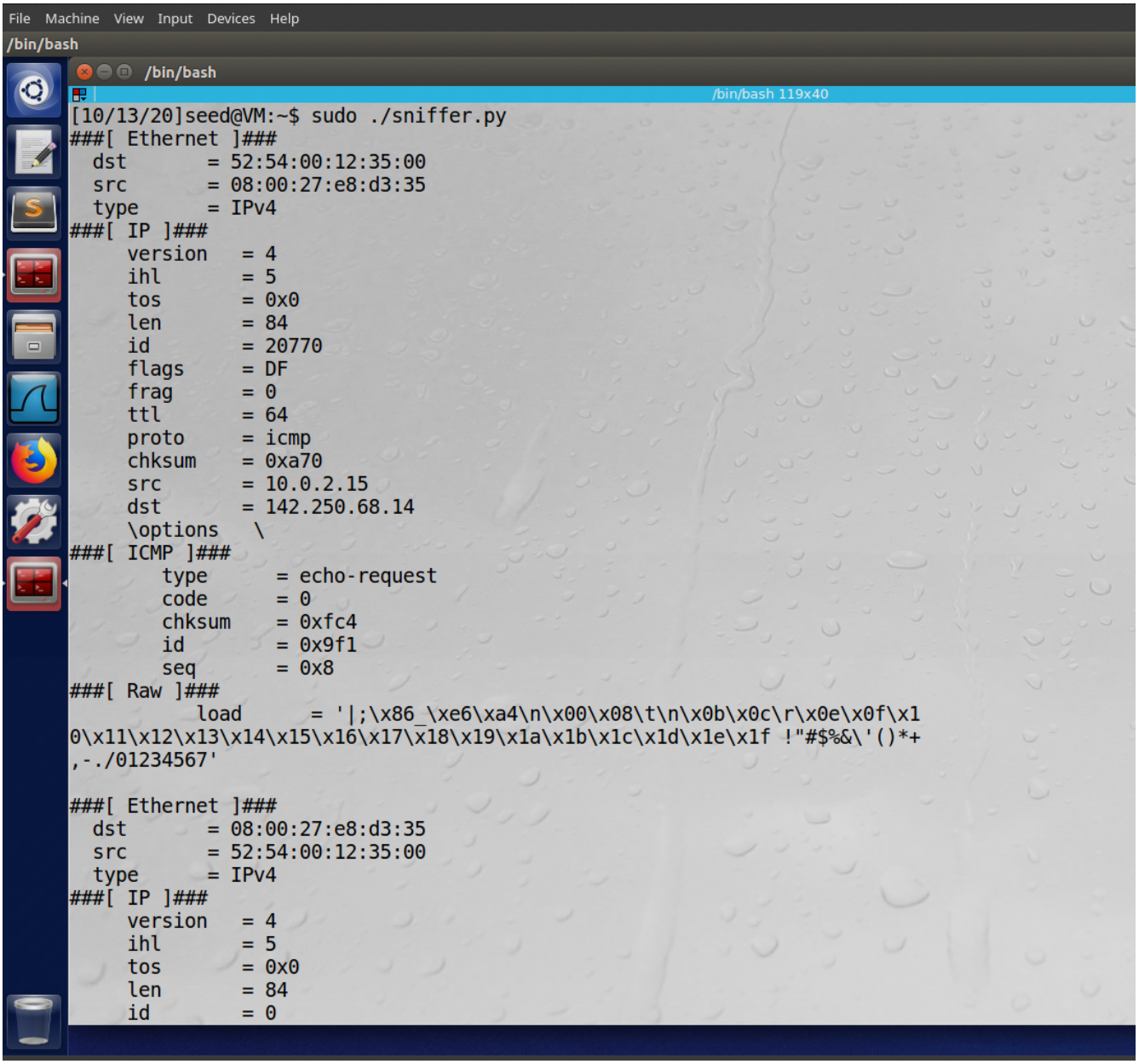
Explanation: The program needs to be run as root in order to allow for promiscuous mode. As this is a non-typical operation and can lead to nefarious activity, the root permission is required.

Task 1.1B

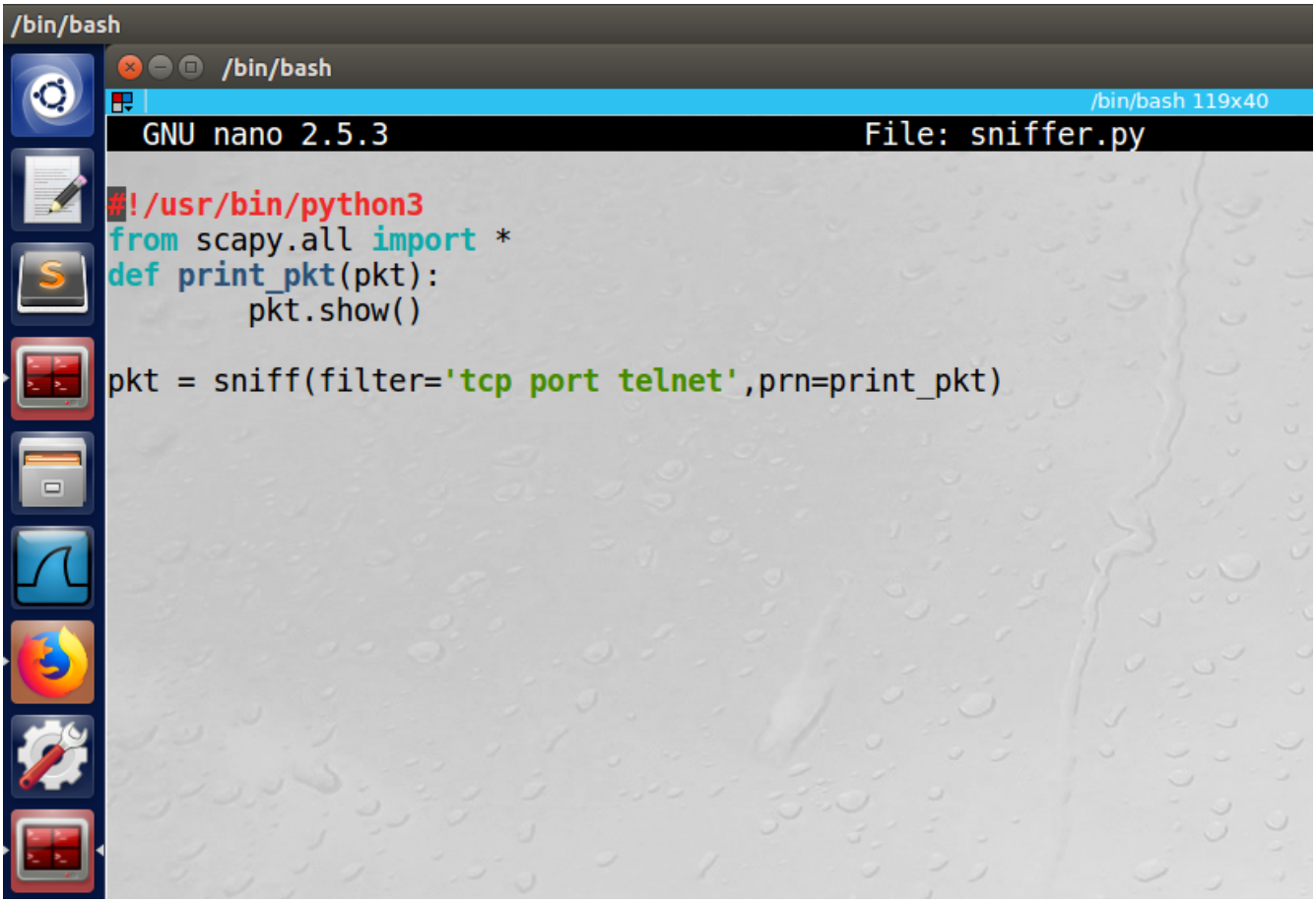
First, to capture just the ICMP packets, the below code will be used:



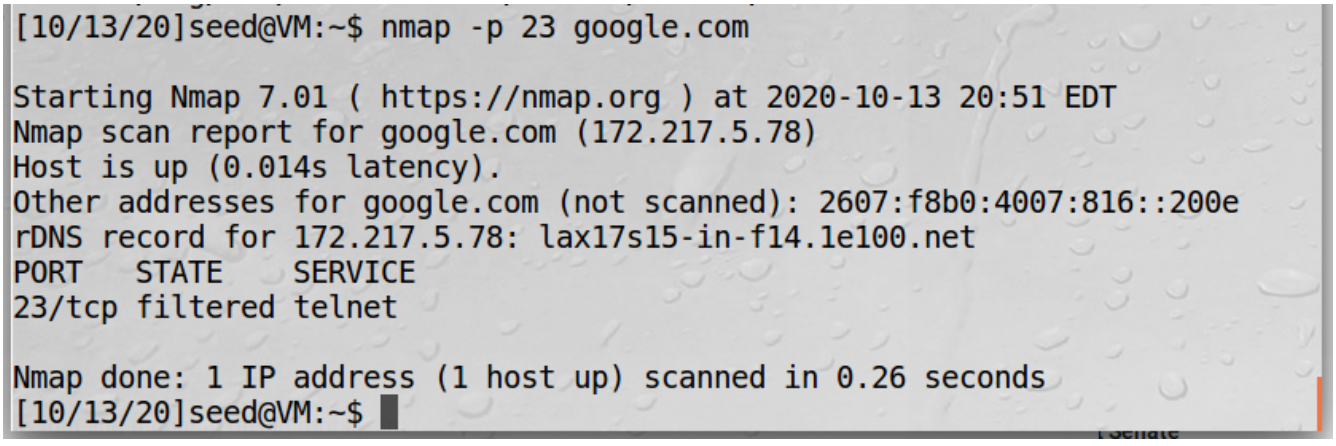
Note that the filter has now been added to include 'icmp'. The results of pinging google.com (while root only):



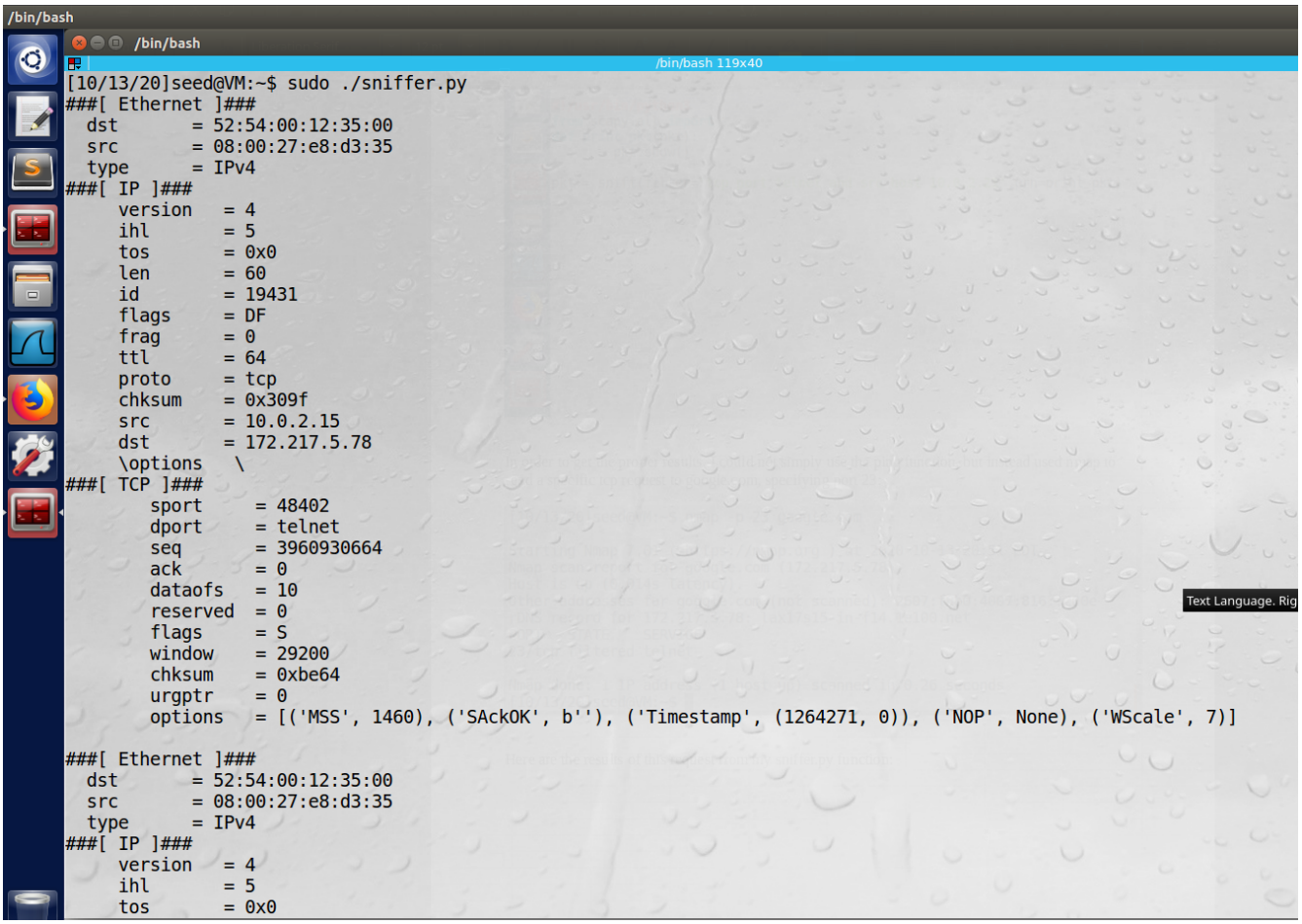
Next, to caputre the TCP packets from a particular IP and a destination port number of 23, the following will be used. Note that port 23 is the port for telnet, which is explicitly stated, and the first argument has been changed to “tcp”:



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:

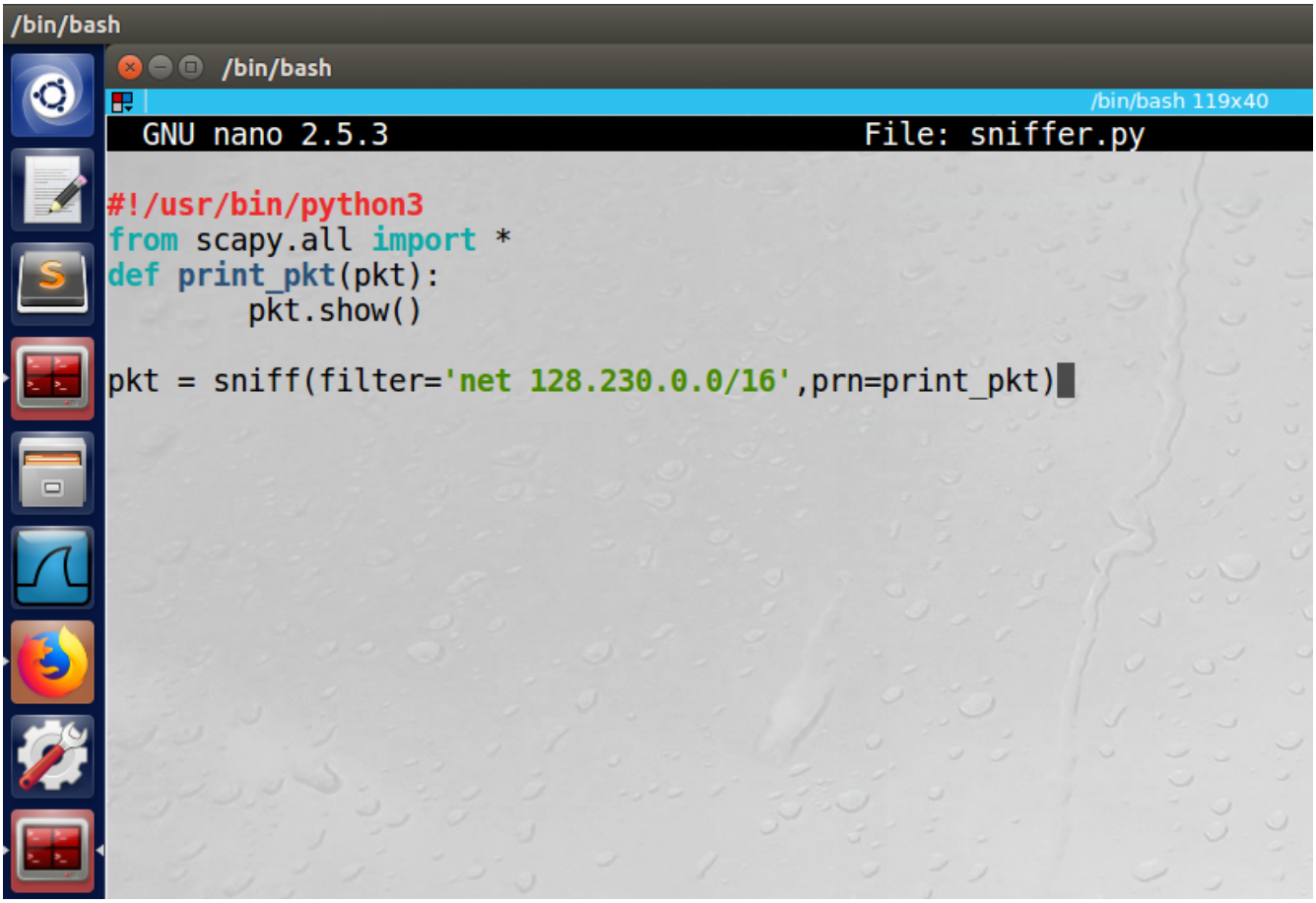


Here are the results of this request from my sniffer.py function:

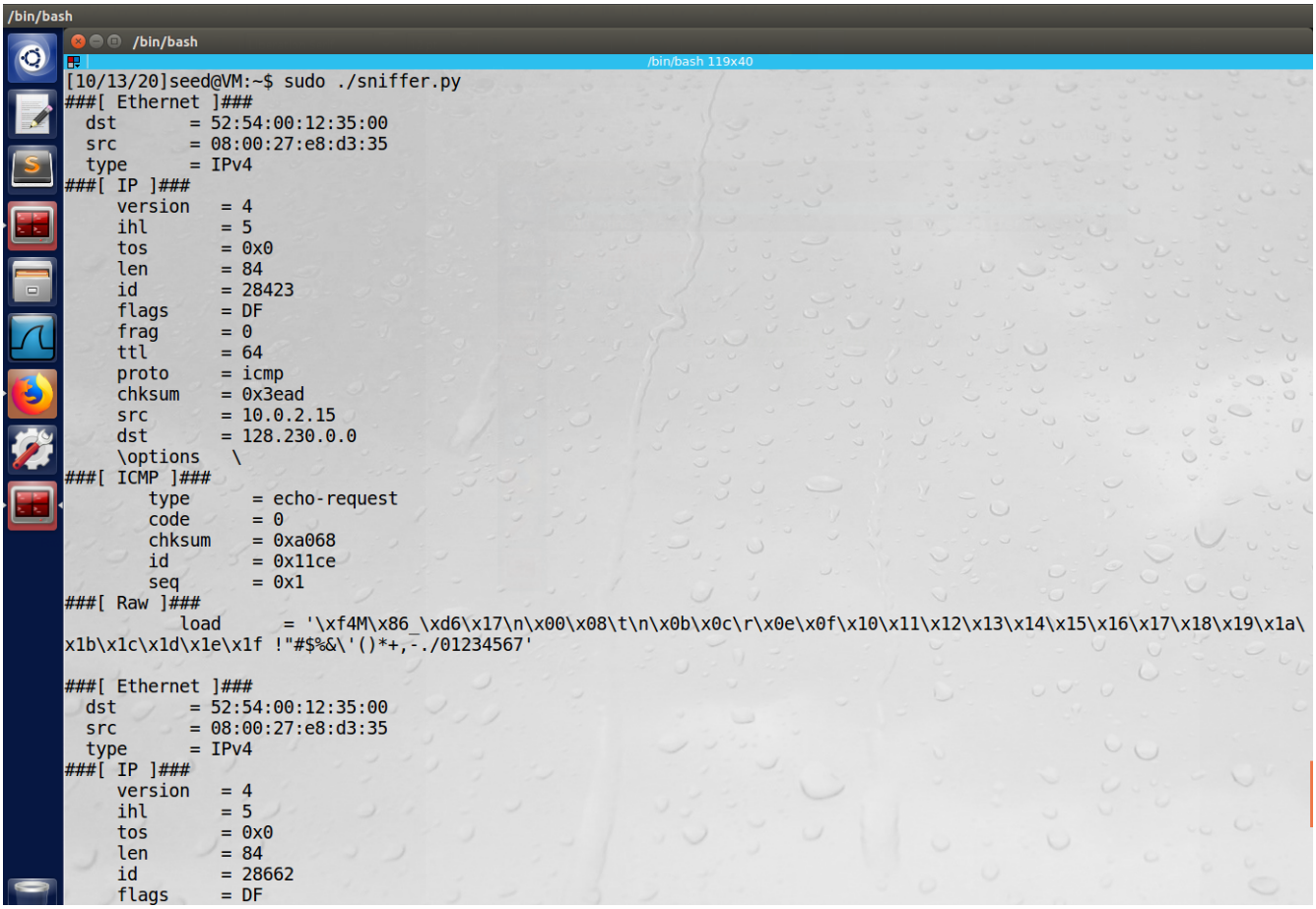


Note the “dport” field under the TCP heading is listed as telnet, matching our intended port 23.

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



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

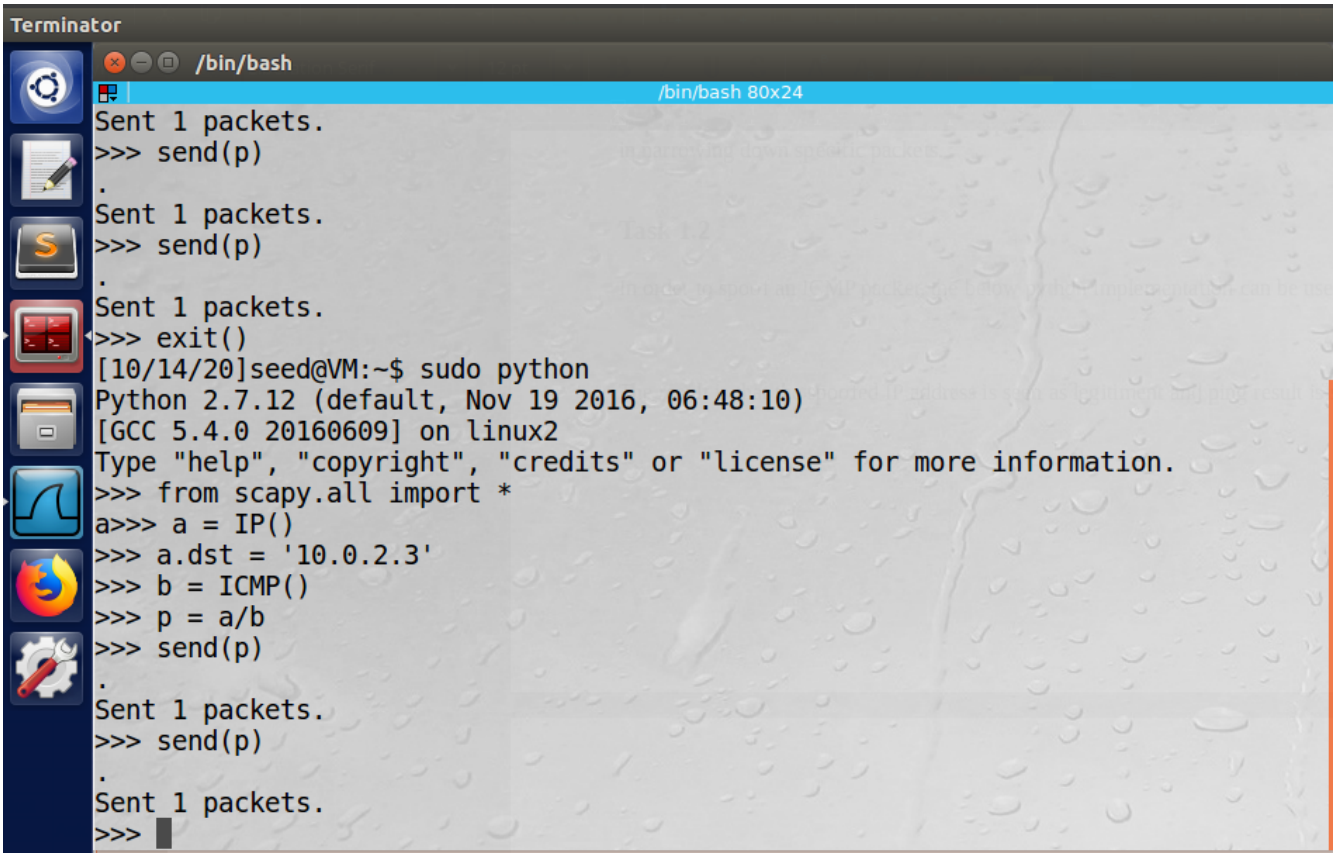


Observation: By simply manipulating the filter argument, we can caputre different packets. In this case, we checked ICMP, then TCP with a destination port of 23 (telnet), and finally a specific subnet.

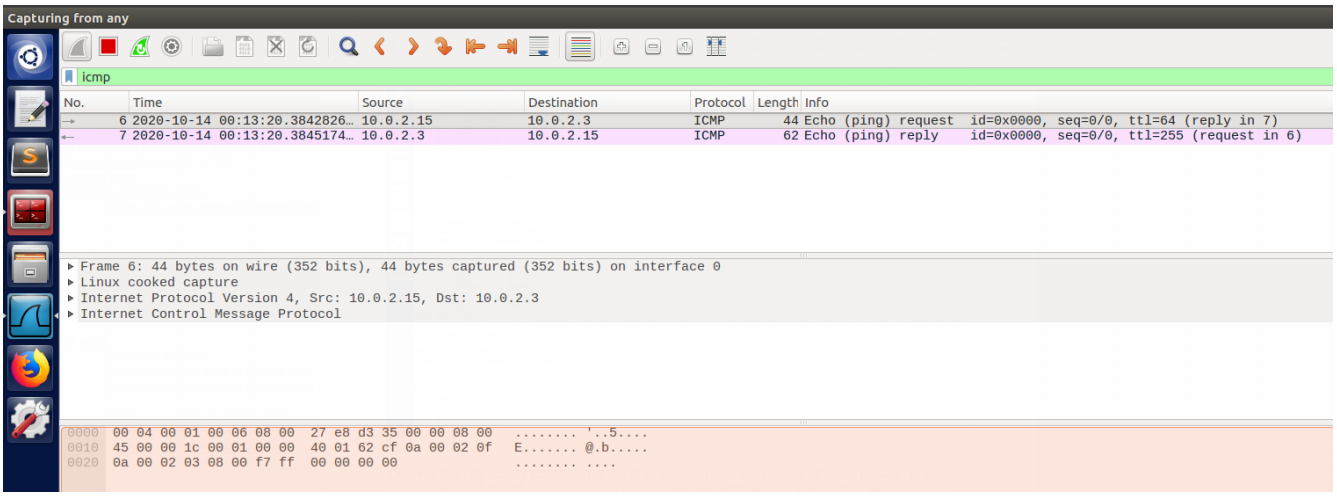
Explanation: The filter argument is very powerful and allows for very flexible requests. This is helpful in narrowing down specific packets.

Task 1.2

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



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

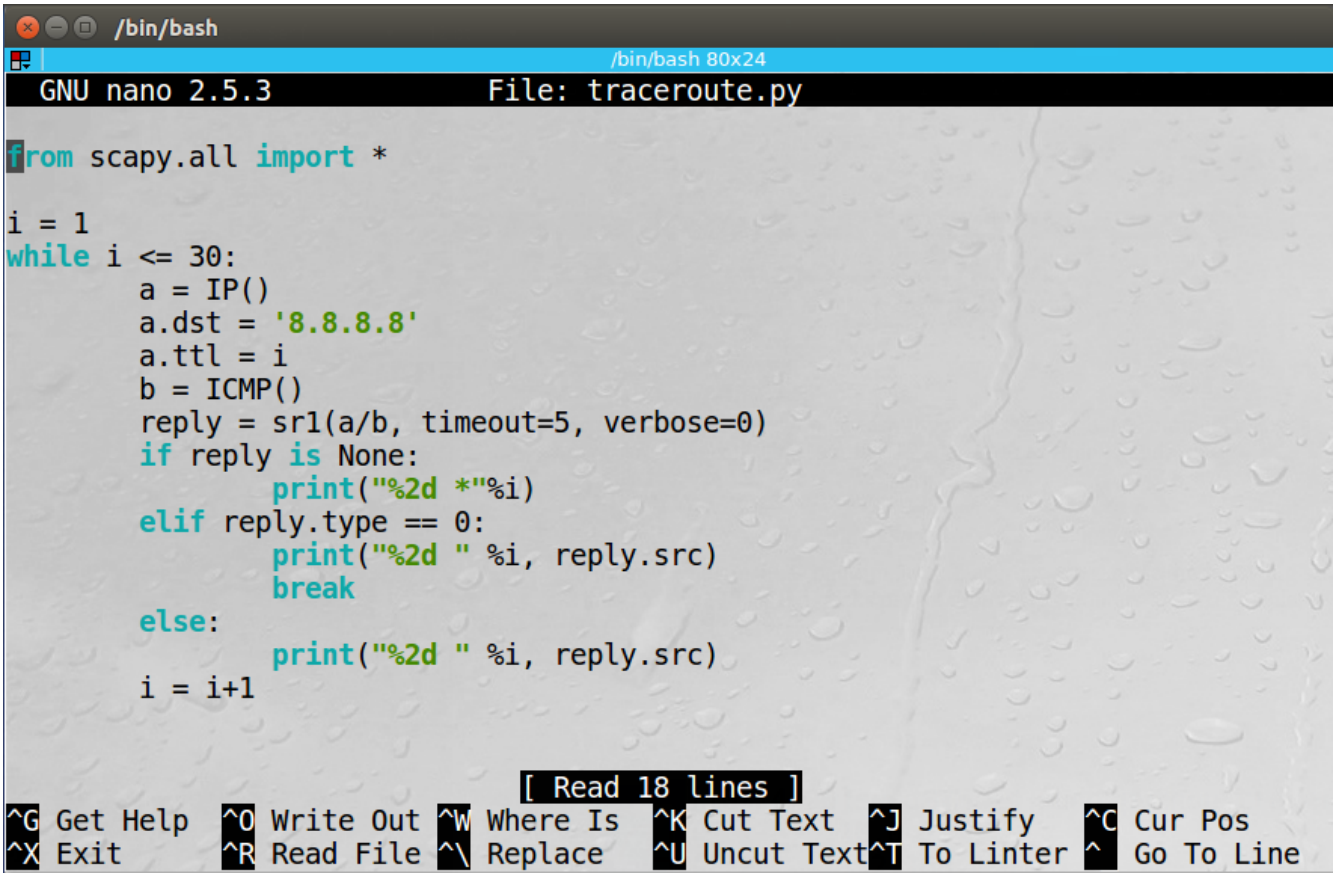


Observation: Note how the source was the VM's IP address (10.0.2.15) and the destination was the spoofed one (10.0.2.3) entered by the user. Also note that this was attempt number 6/7 as I didn't have the Wireshark filter configured right the first few attempts.

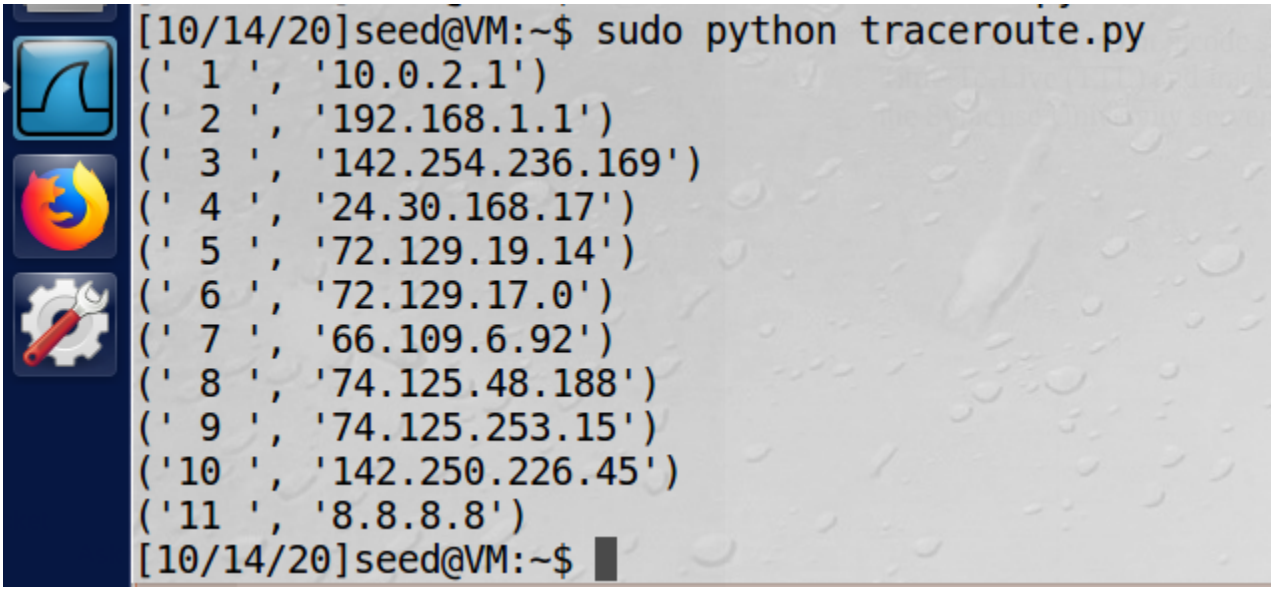
Explanation: Packets can be easily spoofed over the ICMP protocol using the Scapy API. Additionally, the user can specifiy the spoofied IP address to anything valid.

Task 1.3

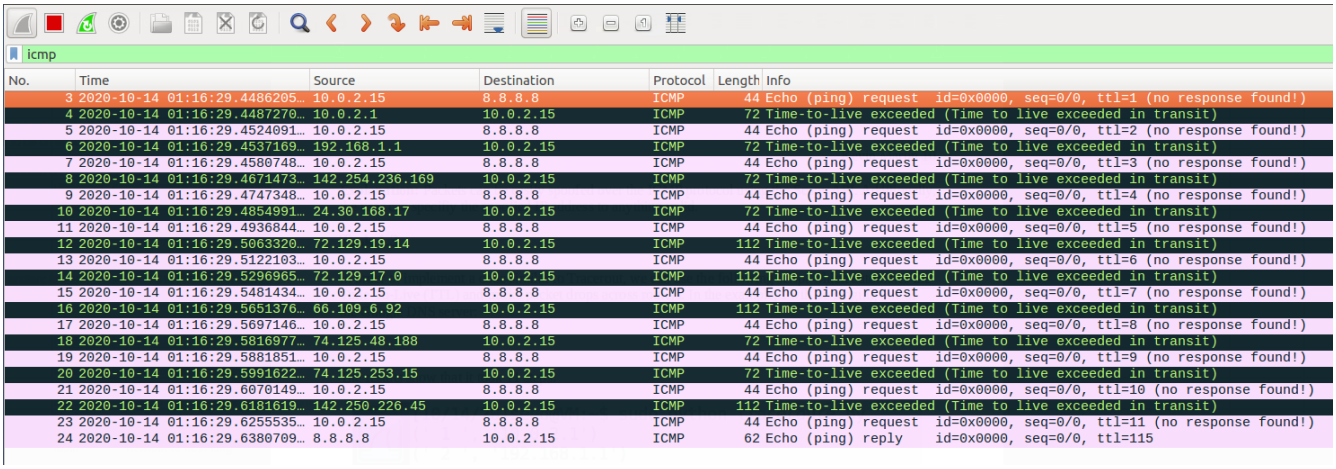
In order to implement a code similar to Tracerout, we can use the following script which automates the Time-To-Live (TTL) and tracks packet drops across routers. In the example, I will be pinging one of the Google DNS server:



The results show that it took 11 steps:



And the Wireshark output:



Observation: The loop worked and 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.

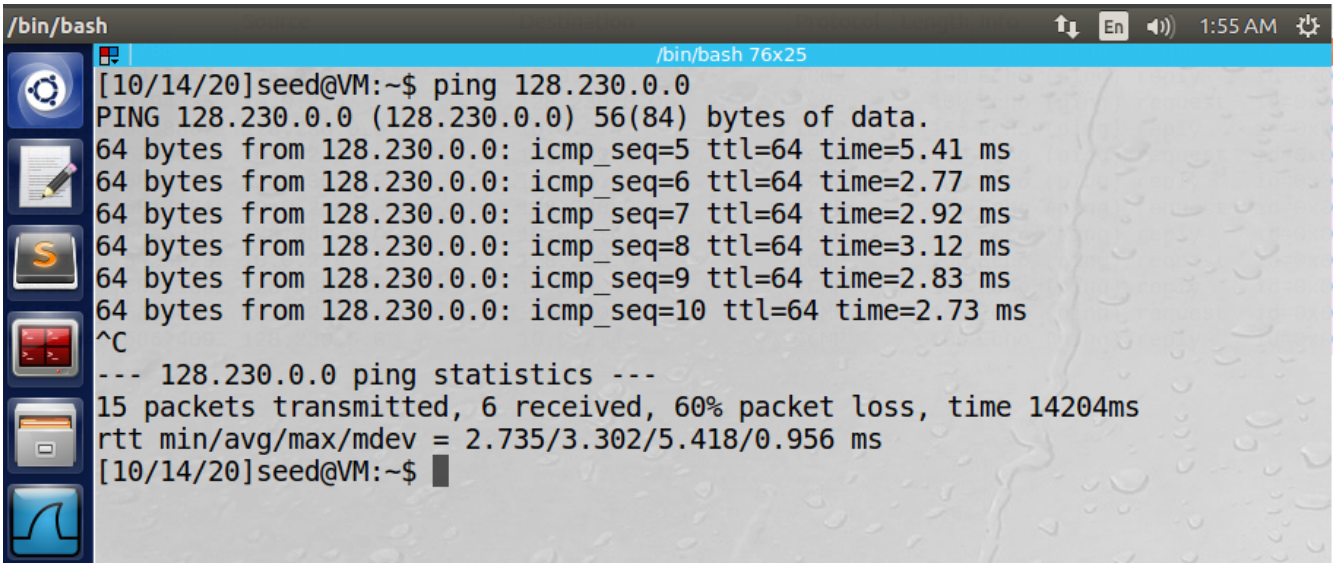
Explanation: Once again, we see the ease of implementation and flexibility from Scapy. Here we could track the routers with a very simple loop.

Task 1.4

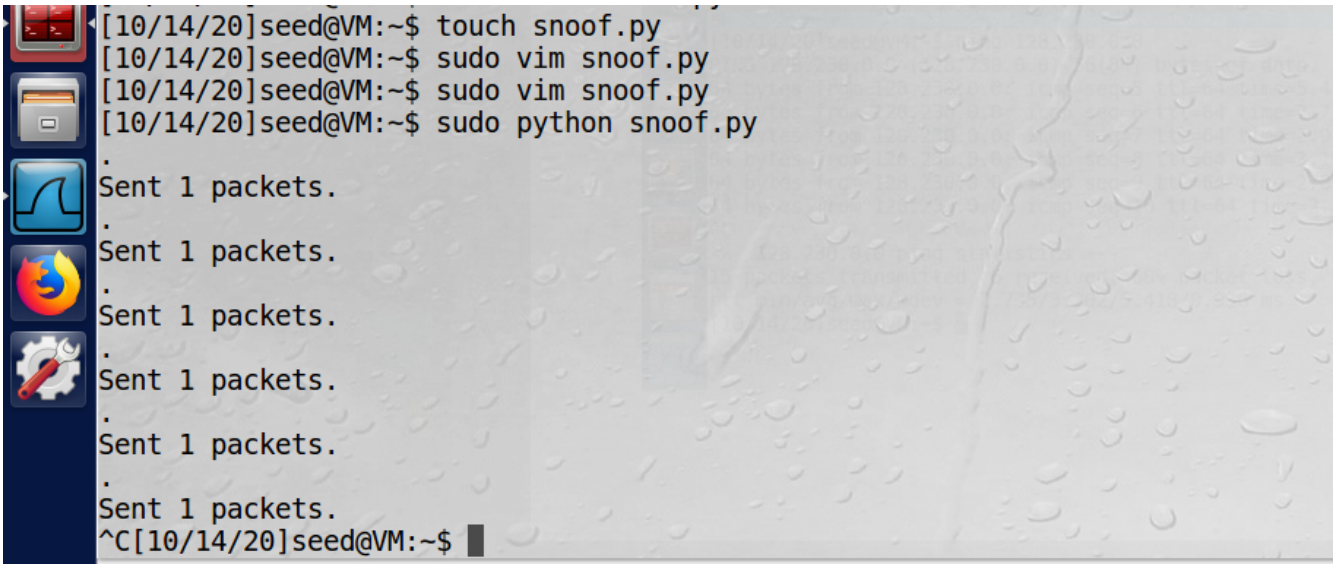
In order to run the sniff and then spoofing (snoofing) program, the following code is used. Here we take in (snifff) the source and destination IP addresses and flip them (spoof) to create an authentic

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):

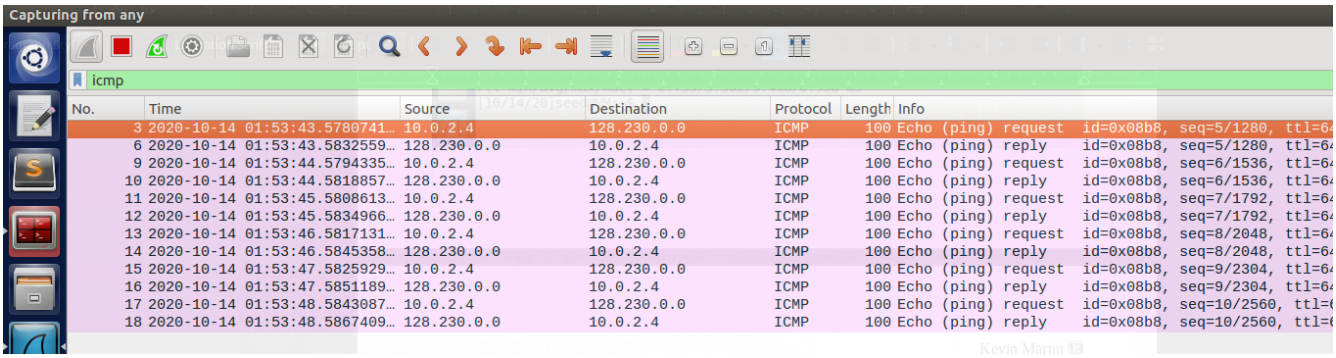
VM A pinging a random IP (128.230.0.0):



VM B implementing the snoof.py script:



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 snoofing.

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):

sniffing.c

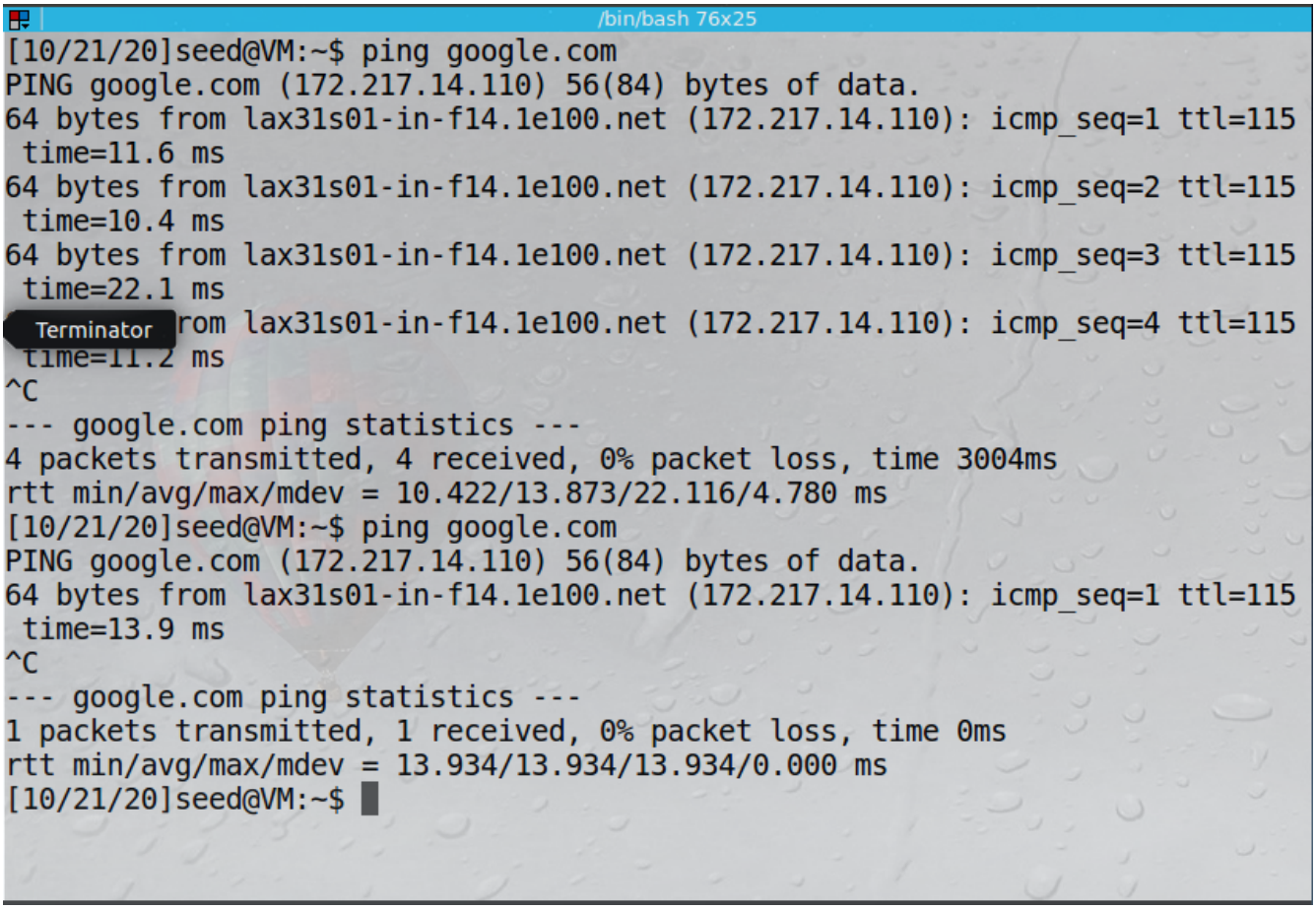
```c
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>

/* Ethernet header */
struct ethheader {
  u_char  ether_dhost[6]; /* destination host address */
  u_char  ether_shost[6]; /* source host address */
  u_short ether_type;     /* protocol type (IP, ARP, RARP, etc) */
};

/* IP Header */
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification
  unsigned short int iph_flag:3, //Fragmentation flags
                     iph_offset:13; //Flags offset
  unsigned char      iph_ttl; //Time to Live
  unsigned char      iph_protocol; //Protocol type
  unsigned short int iph_chksum; //IP datagram checksum
  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

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

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    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));

    /* determine protocol */
    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;
    }
  }
}

```

```
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);
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
```

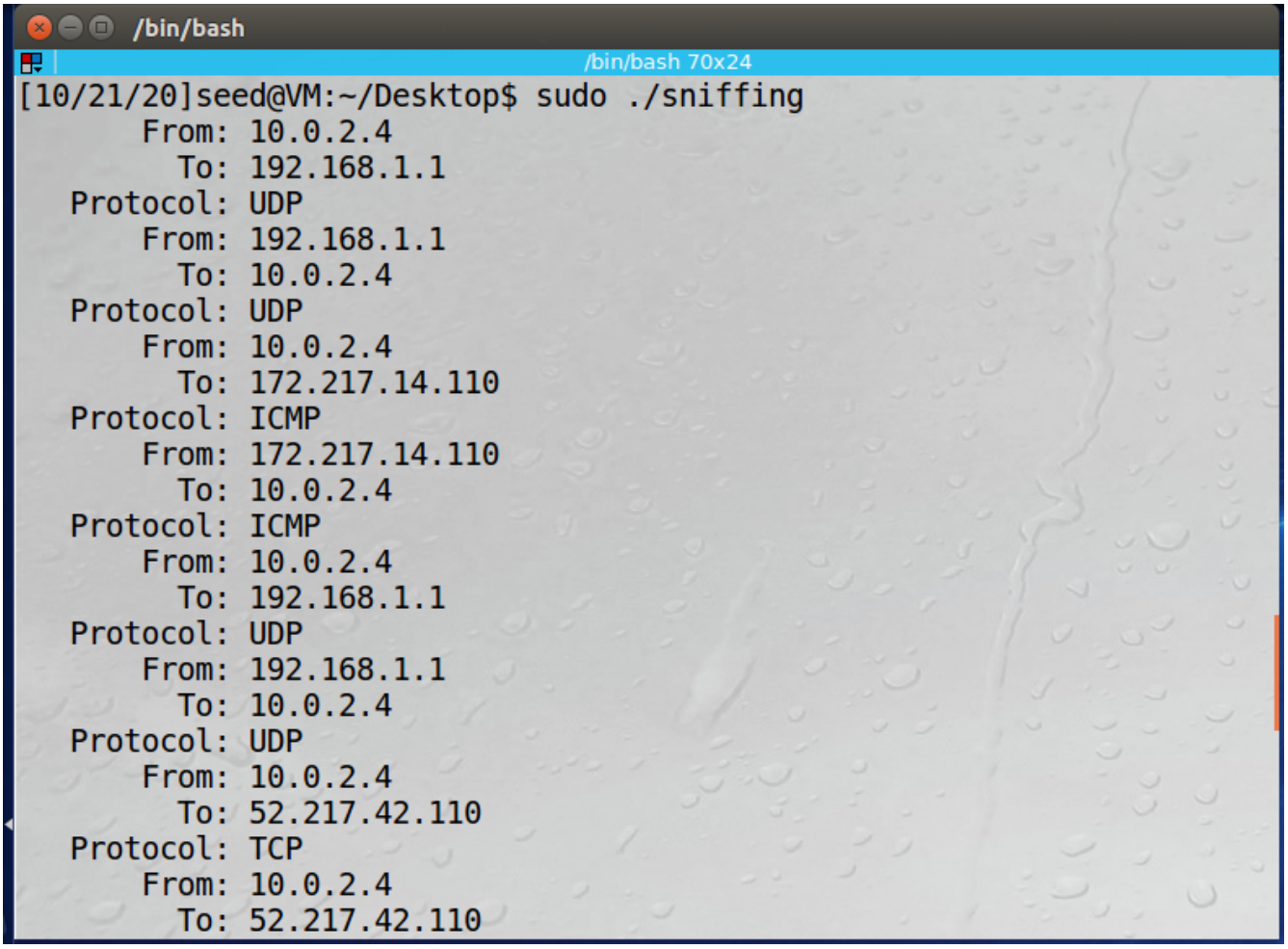
To compile the code, I had to run the following command:

```
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:



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



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.

Questin 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 monitering. Finally, the program is closed in an elgant way using the built in pcap_close.

Question 2: Root prvilege is needed to run a sniffing program because in order to sniff, the NIC card must be set to promiscusous 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 implment filters for the sniffing program, the filter_exp[] field can be filed 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:

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 ./sniffig
       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  }
81
82
83
```

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:

```
[10/21/20]seed@VM:~/Desktop$ sudo ./sniffig
       From: 10.0.2.4
         To: 142.250.72.238
   Protocol: TCP
       From: 10.0.2.4
         To: 142.250.72.238
   Protocol: TCP
       From: 10.0.2.4
         To: 142.250.72.238
   Protocol: TCP
       From: 10.0.2.4
         To: 142.250.72.238
   Protocol: TCP
       From: 10.0.2.4
         To: 142.250.72.238
   Protocol: TCP
^C
[10/21/20]seed@VM:~/Desktop$
```

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.

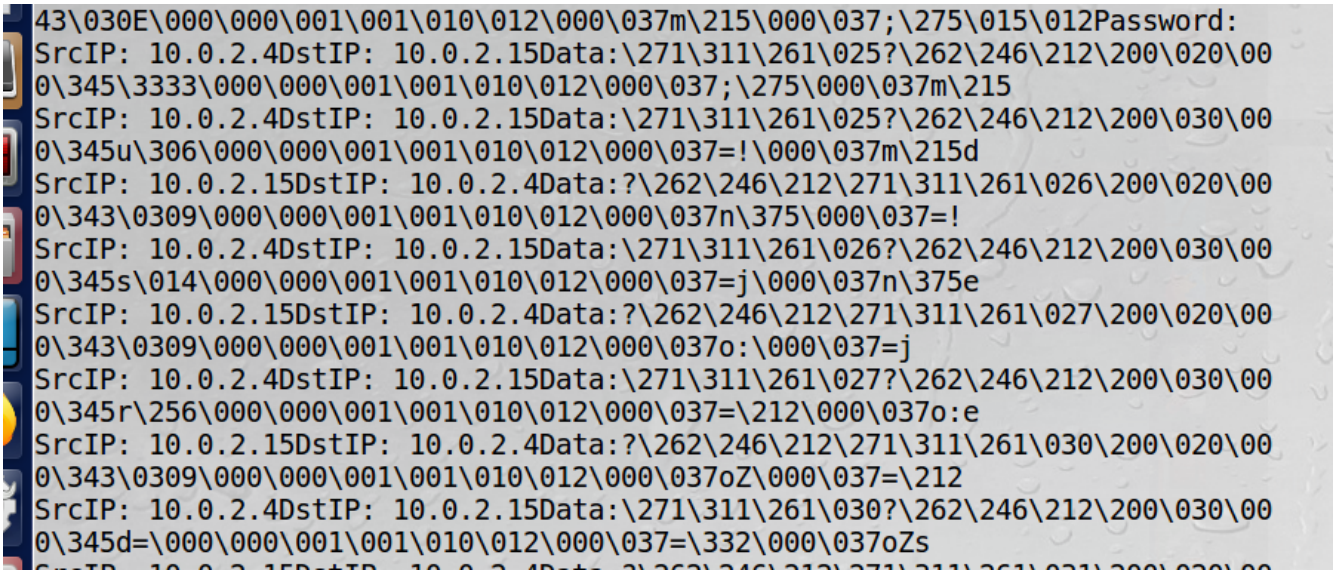
Task 2.1C – Sniffing Passwords

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

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
  const u_char *packet)
{
  struct ipheader * ip = (struct ipheader *)(packet + sizeof(struct ethheader));
  unsigned short pktlen = ntohs(ip->iph_len);
  struct tcpheader *tcp = (struct tcpheader *)((u_char*)ip + sizeof(struct ipheader));
  u_char dataoffset = 4;
  if((pktlen - sizeof(struct ipheader))> dataoffset)
  {
    printf("SrcIP: %s", inet_ntoa(ip->iph_sourceip));
    printf("DstIP: %s", inet_ntoa(ip->iph_destip));
    printf("Data:");
    u_char* data = (u_char*)tcp + dataoffset;
    for(unsigned short s = 0;s < (ntohs(ip->iph_len) - (sizeof(struct ipheader) + dataoffset));s++)
    {
      if (isprint(*data) != 0)
      {
        printf("%c", *data);
      }
      else
      {
      printf("\\%.3hho", *data);
      }
  data++;
}
printf("\n");
}
}
```

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

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



Observation: the password, in order, after the “Password” promt 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:

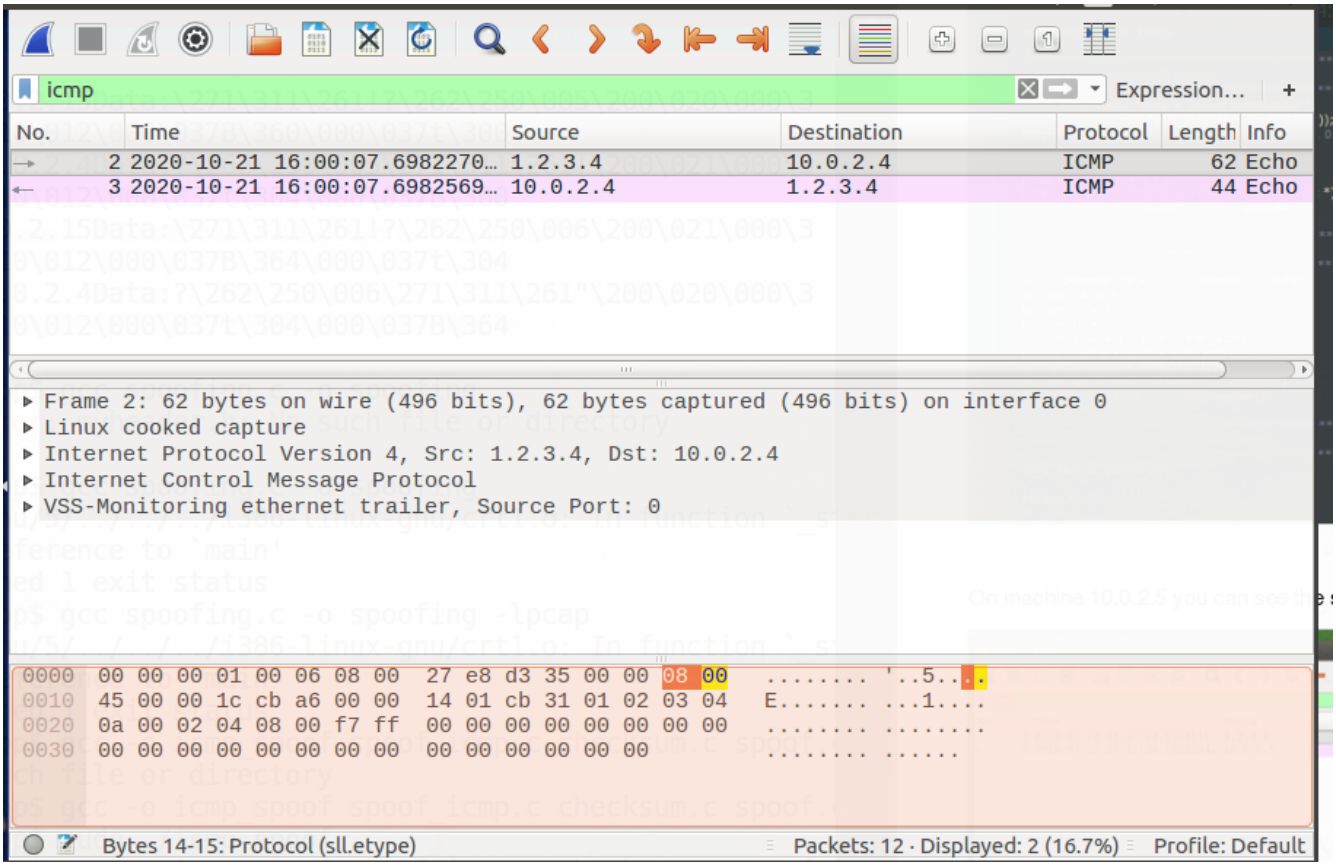
spoof.c | myheader.h | checksum.c | spoof_icmp.c

```
/*****************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
*****************************************************************/
int main() {
   char buffer[1500];

   memset(buffer, 0, 1500);

   /*********************************************************
      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_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.4");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));

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

   return 0;
}
```

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:

spoof.c | myheader.h | checksum.c | spoof_icmp.c | sniffing.c

```c
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <pcap.h>

#include "myheader.h"

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

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,
                        &enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->iph_destip;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->iph_len), 0,
           (struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}

void send_echo_reply(struct ipheader * ip)
{
  int ip_header_len = ip->iph_ihl * 4;
  const char buffer[1500];
  memset((char*)buffer, 0, 1500);
  memcpy((char*)buffer, ip, ntohs(ip->iph_len));
  struct ipheader* newip=(struct ipheader*)buffer;
  struct icmpheader* newicmp=(struct icmpheader*)(buffer + ip_header_len);
  newip->iph_sourceip=ip->iph_destip;
  newip->iph_destip=ip->iph_sourceip;
```

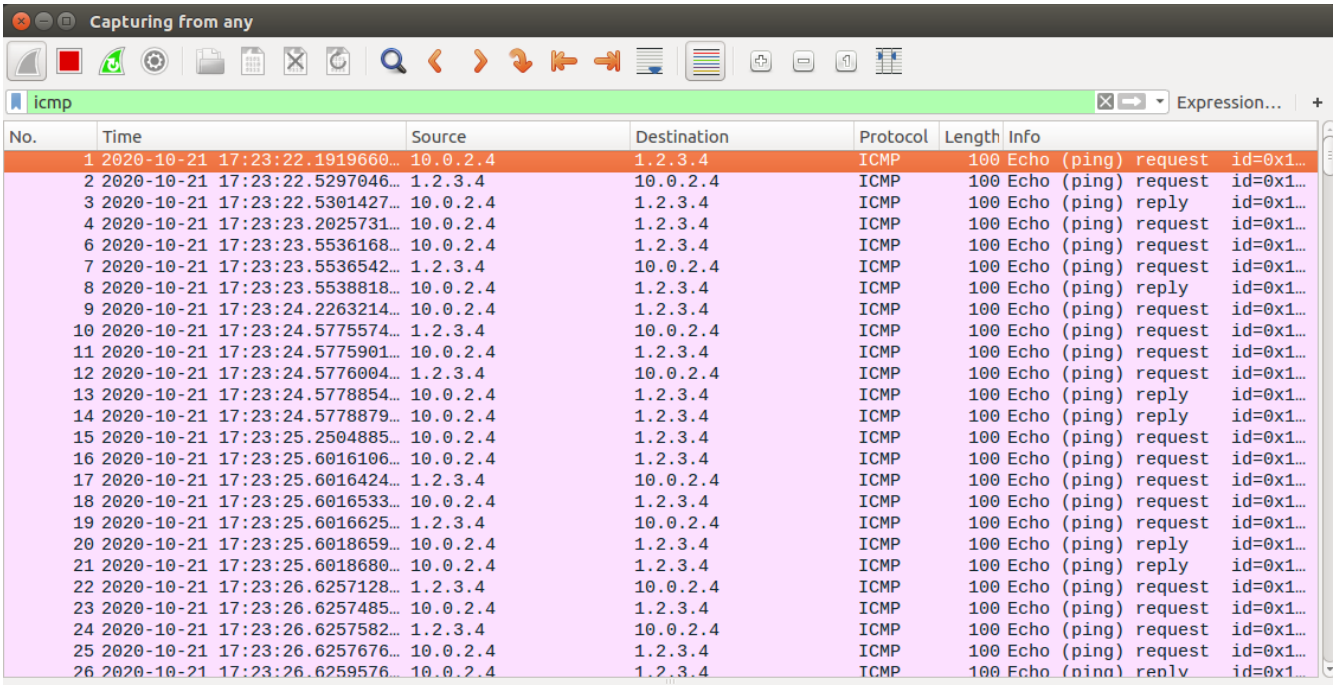
spoof.c | myheader.h | checksum.c | spoof_icmp.c | sniffing.c

```
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;
```

spoof.c | myheader.h | checksum.c | spoof_icmp.c | sniffing.c

```
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 resposne 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:



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 entended, supplying fake replys from an erroneous IP address. The program is also sophisticated enough to not need the fake IP address, but can read and suppy it back from the original sending computer. Note that 1.2.3.4 does not appear anywhere in the snoof code.