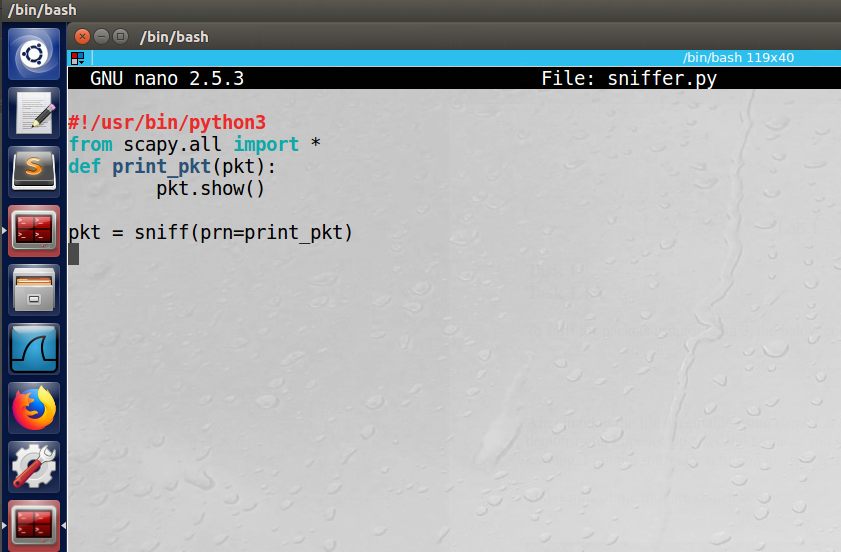
Lab 1 – Kevin Martin

**Task 1**

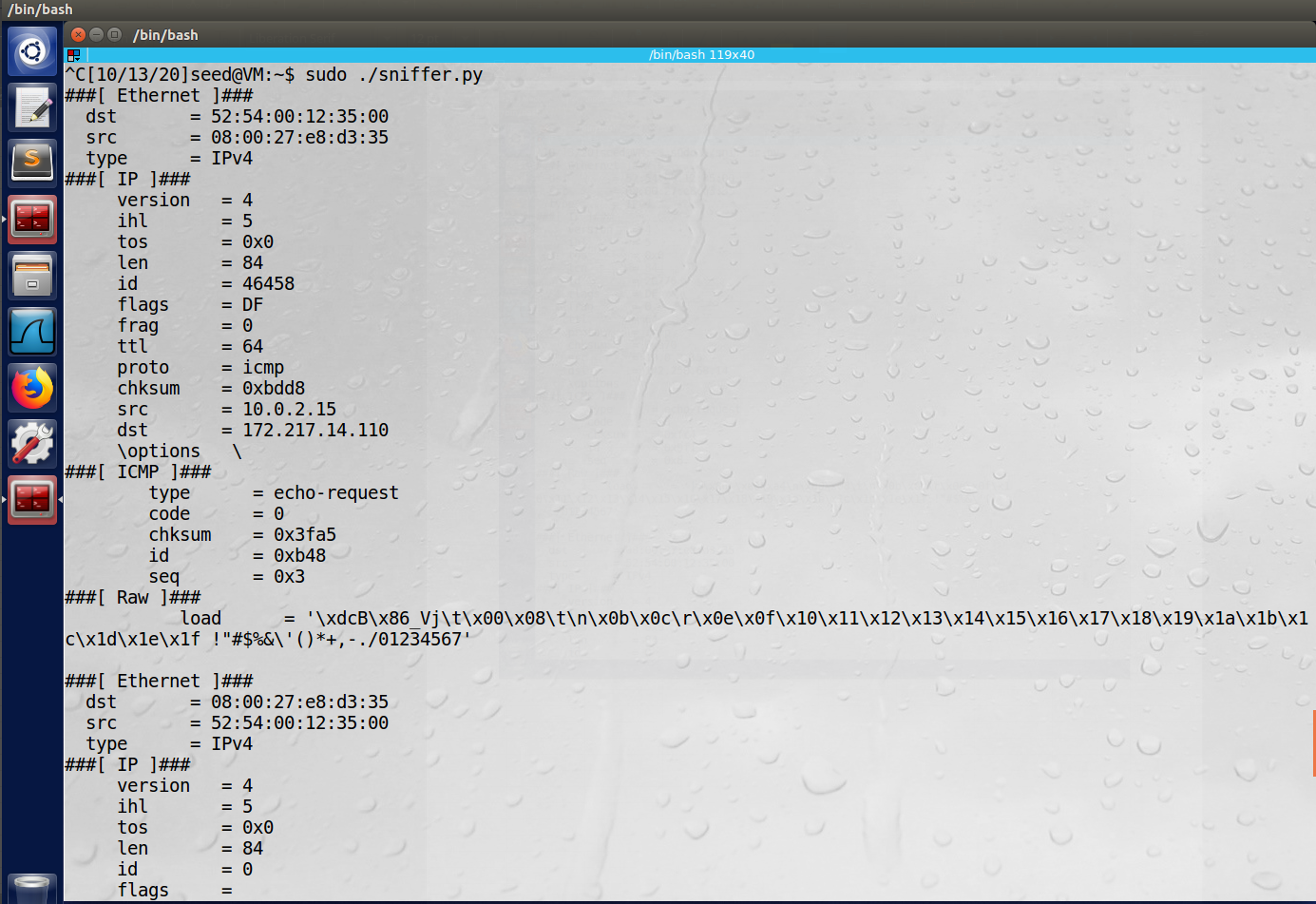
**Task 1.1A**

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

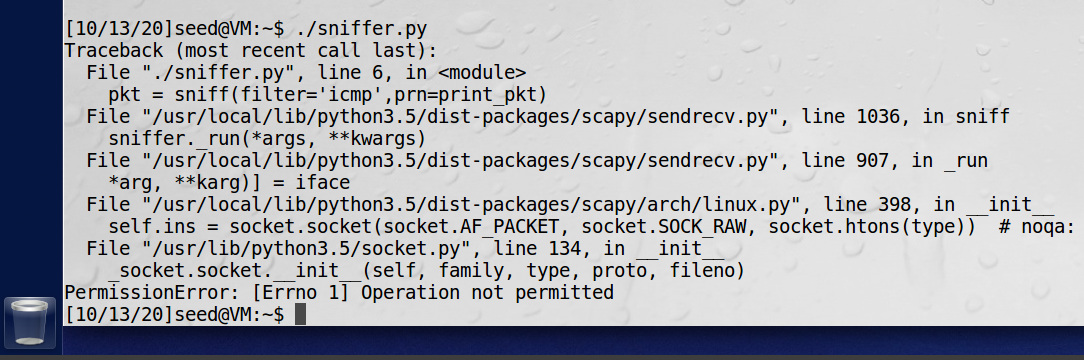


After making the file executable using chmod, it was run both in root and non-not root. In order to demonstrate, I opened up a second terminal and simply entered “ping google.com”. While that was running, I ran the sniffer.py script on my first terminal.

The results of the root are shown below:



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

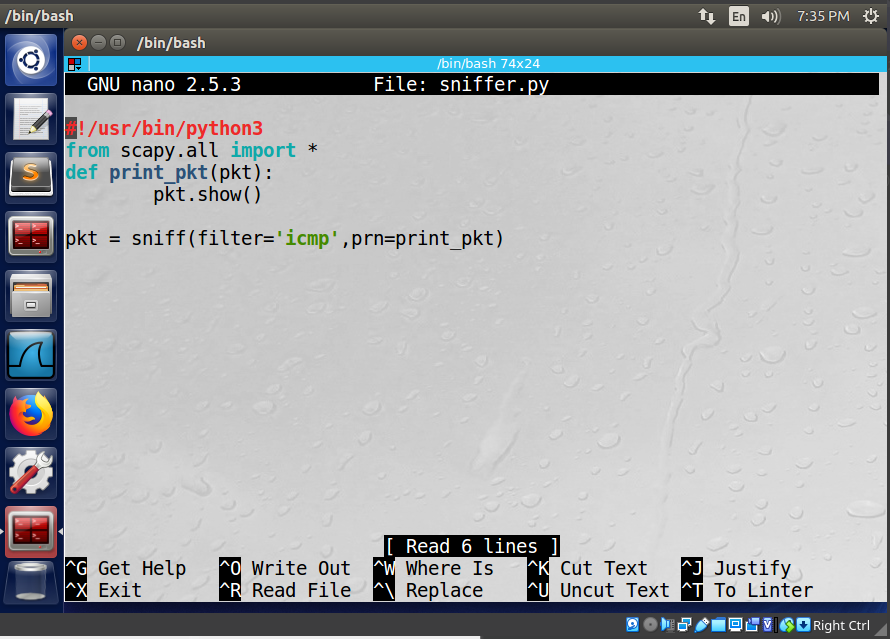


**Observation**: Running the program in root, we see that we are indeed caputring network traffic using sniffing. Note the source (src) is the host machine’s IP address, and the destination (dest) is that of this particular Google server.

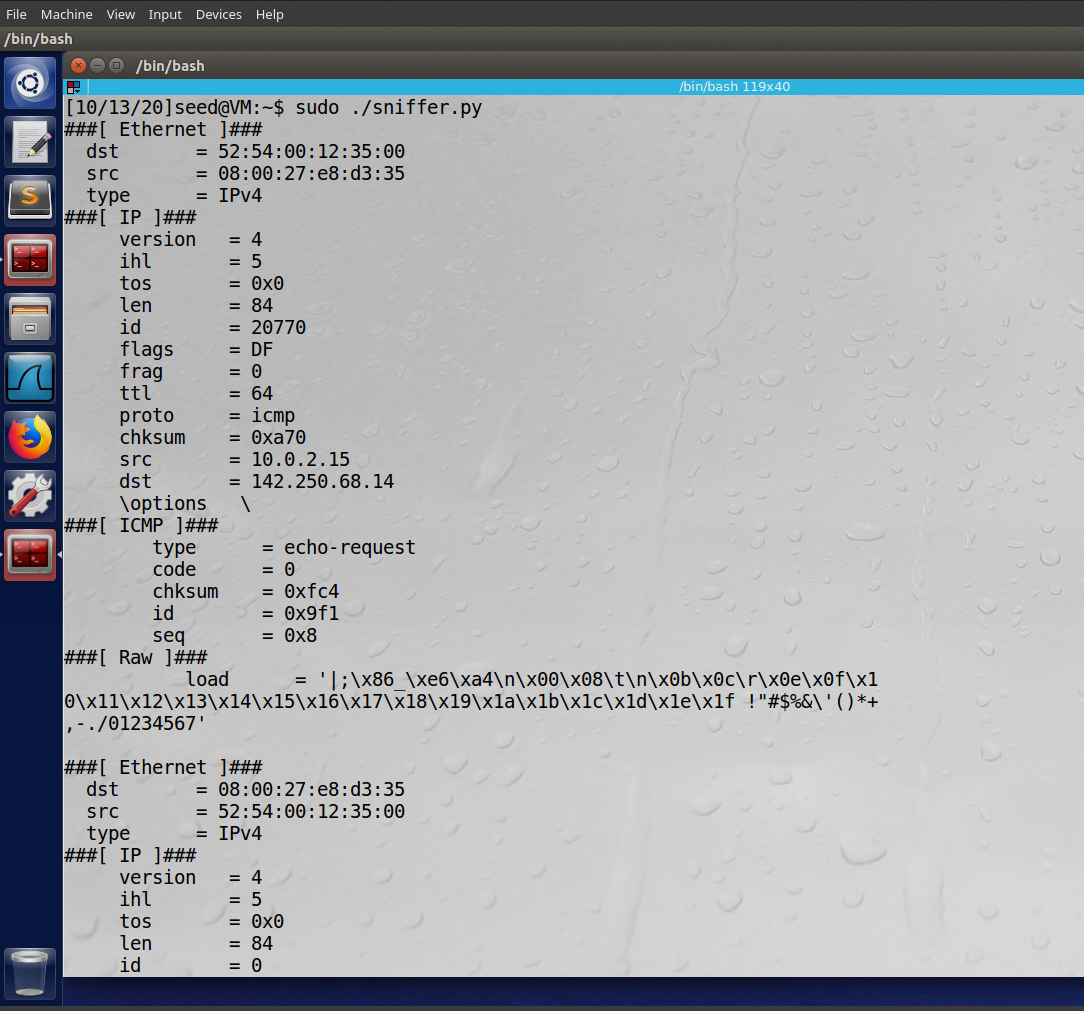
**Explanation:** The program needs to be run as root in order to allow for promiscuous mode. As this is a non-typical operation and can lead to nefarious activity, the root permission is required.

**Task 1.1B**

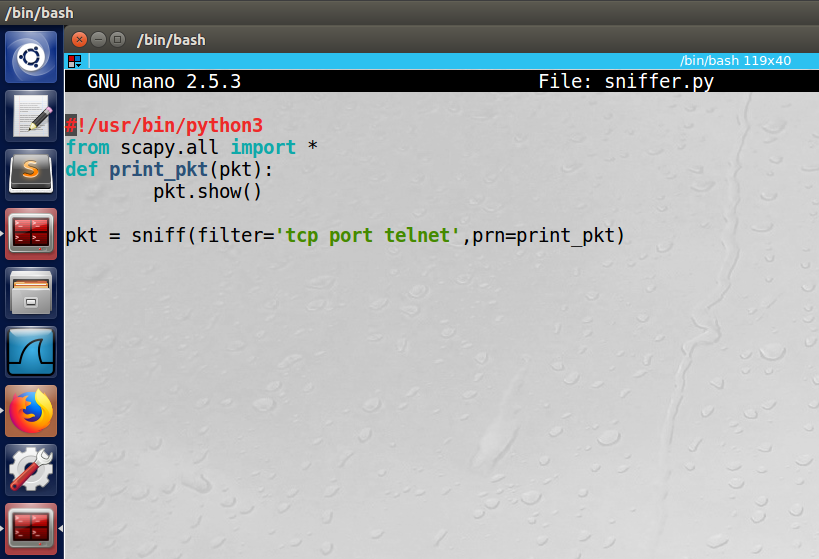
First, to capture just the ICMP packets, the below code will be used:

****

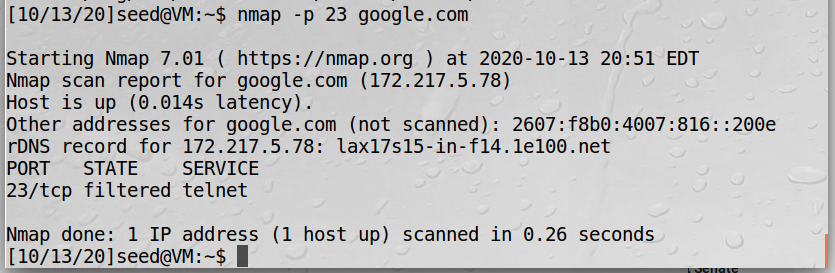
Note that the filter has now been added to include ‘icmp’. The results of pinging google.com (while root only):

****

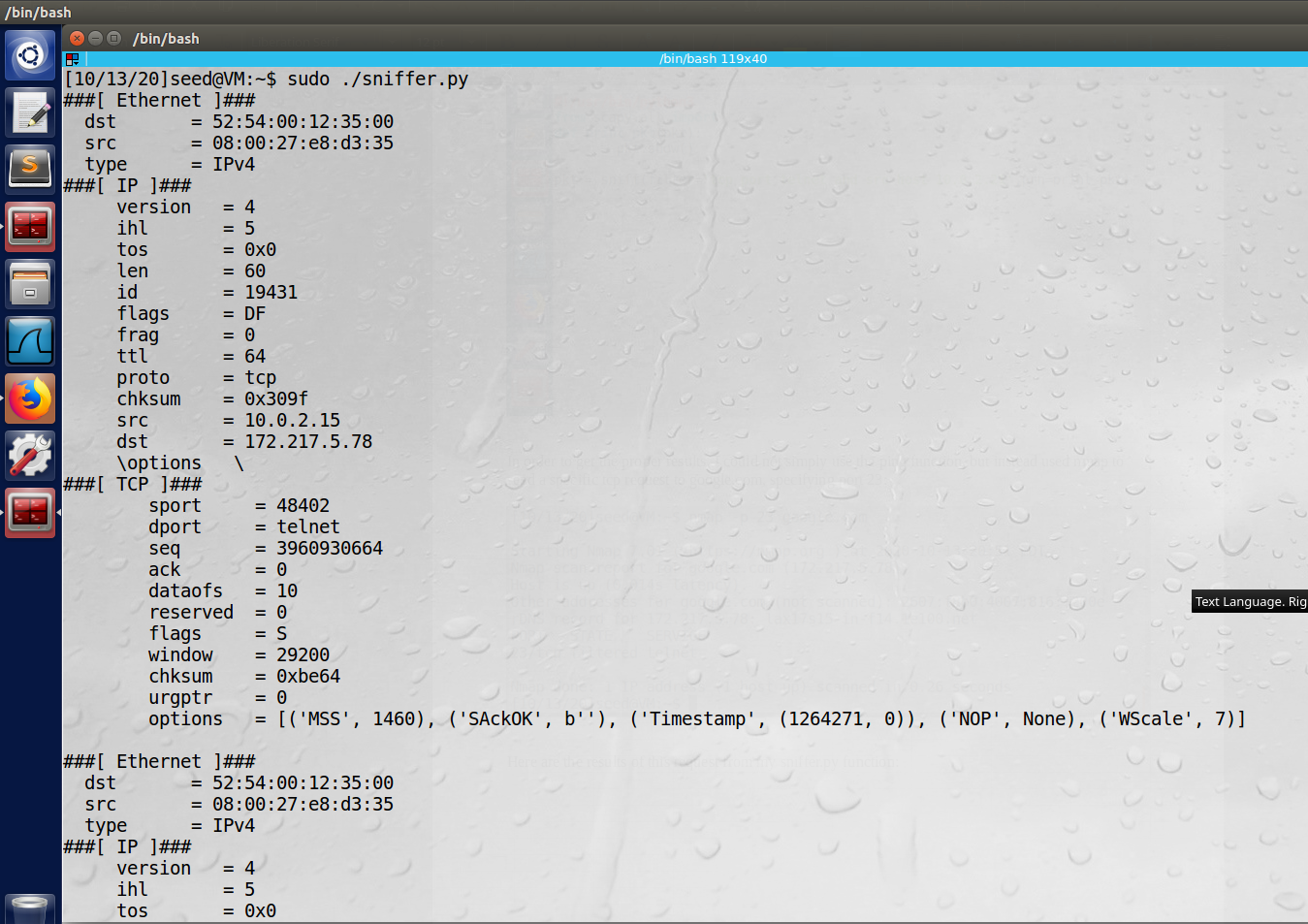
Next, to caputre the TCP packets from a particular IP and a destination port number of 23, the following will be used. Note that port 23 is the port for telnet, which is explicitly stated, and the first argument has been changed to “tcp”:

****

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:

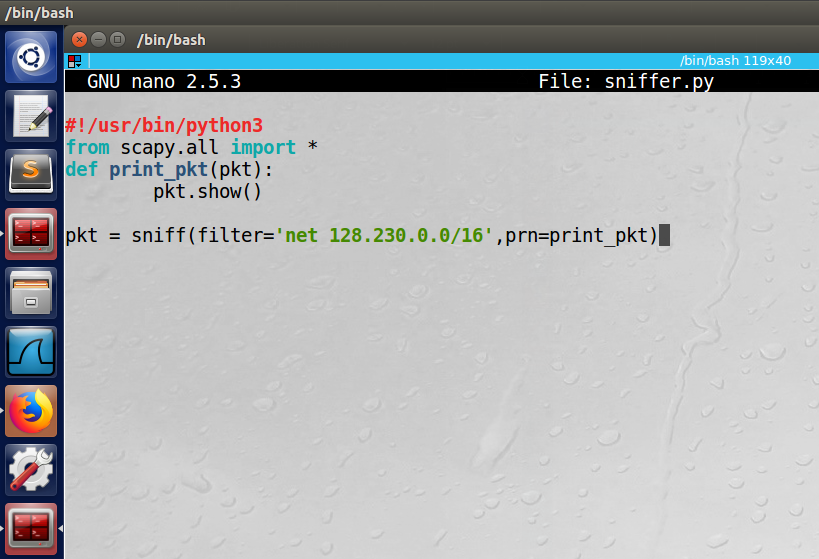


Here are the results of this request from my sniffer.py function:

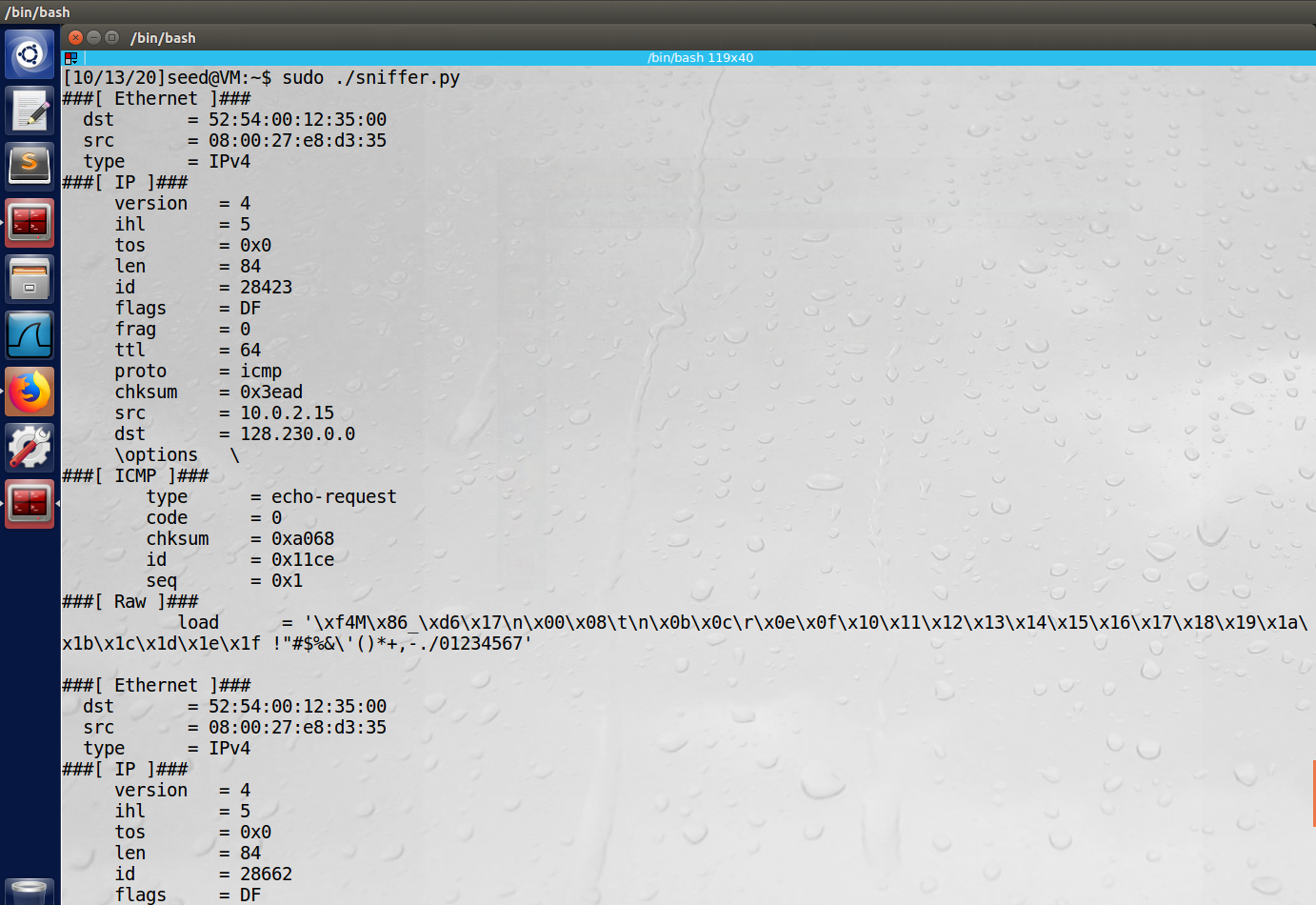
****

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:

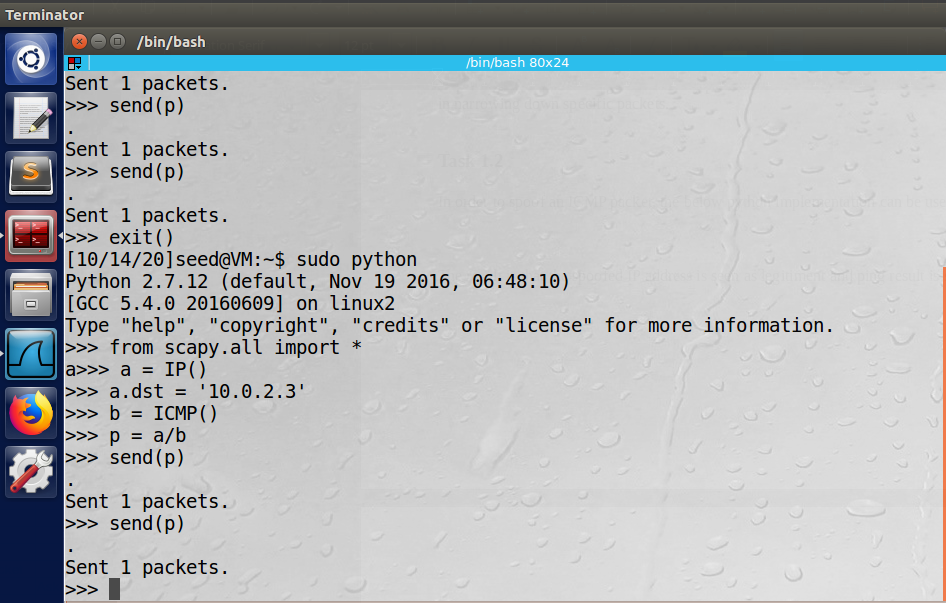
****

**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.

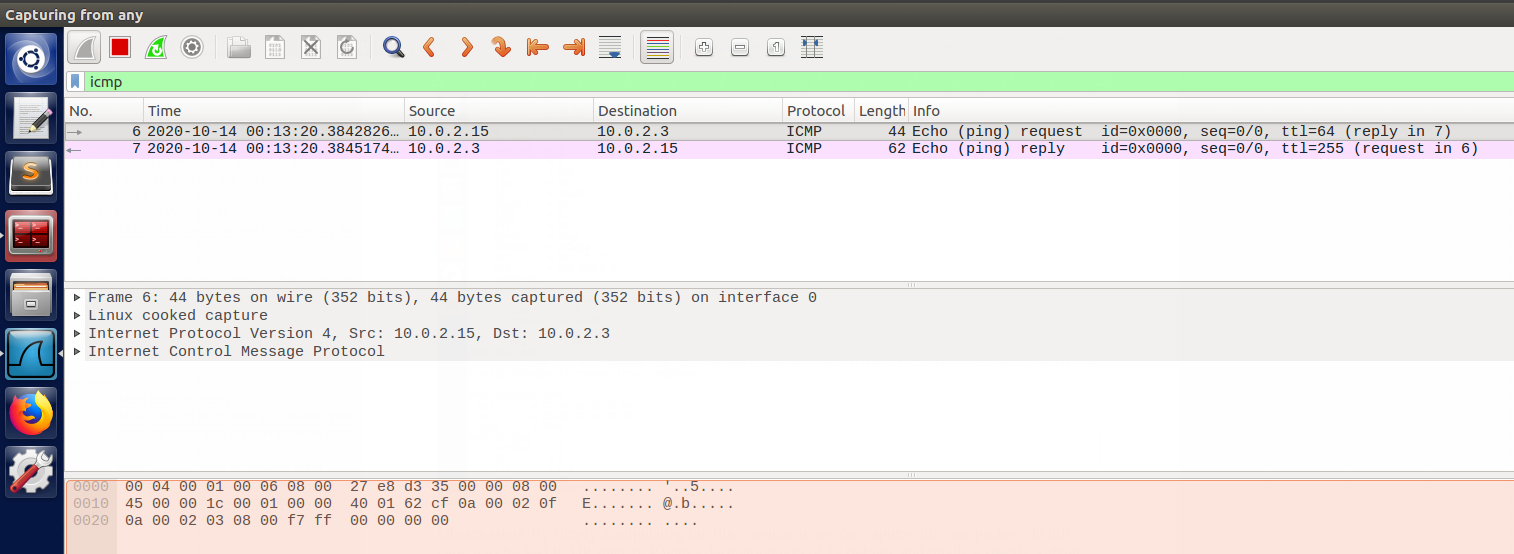
**Explanation:** The filter argument is very powerful and allows for very flexible requests. This is helpful in narrowing down specific packets.

**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:

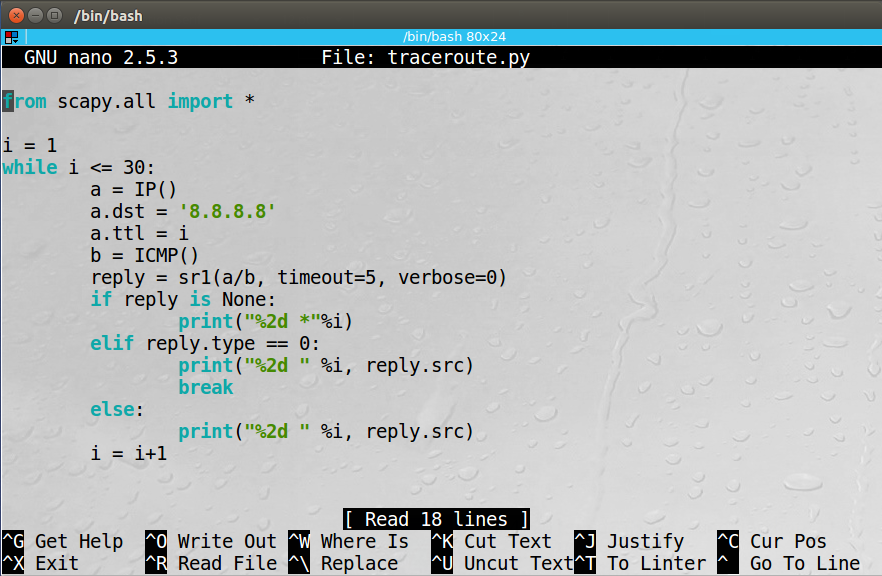


**Observation:** Note how the source was the VM’s IP address (10.0.2.15) and the destination was the spoofed one (10.0.2.3) entered by the user. Also note that this was attempt number 6/7 as I didn’t have the Wireshark filter configured right the first few attempts.

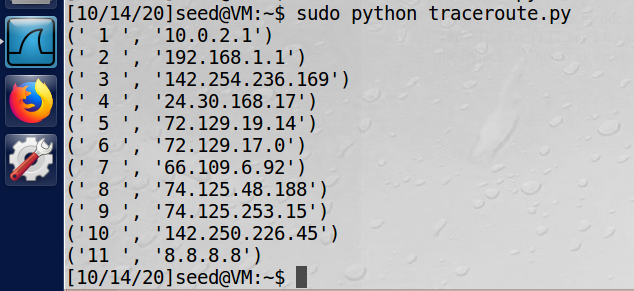
**Explanation:** Packets can be easily spoofed over the ICMP protocol using the Scapy API. Additionally, the user can specifiy the spoofied IP address to anything valid.

**Task 1.3**

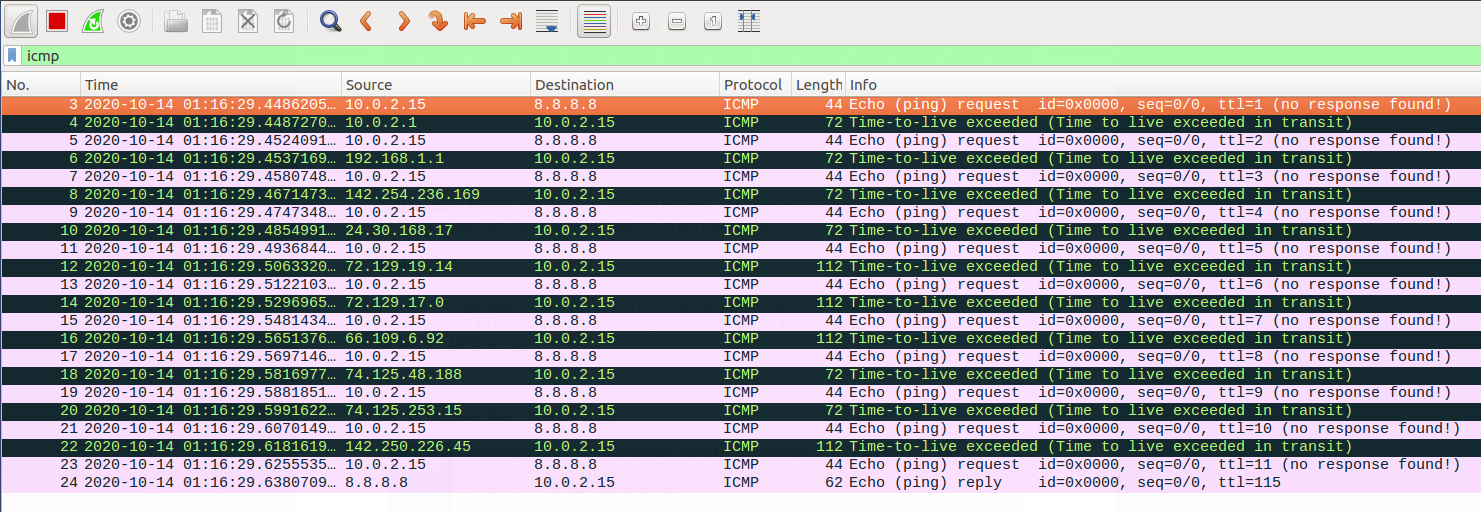
In order to implement a code similar to Tracerout, we can use the following script which automates the Time-To-Live (TTL) and tracks packet drops across routers. In the example, I will be pinging one of the Google DNS server:



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.

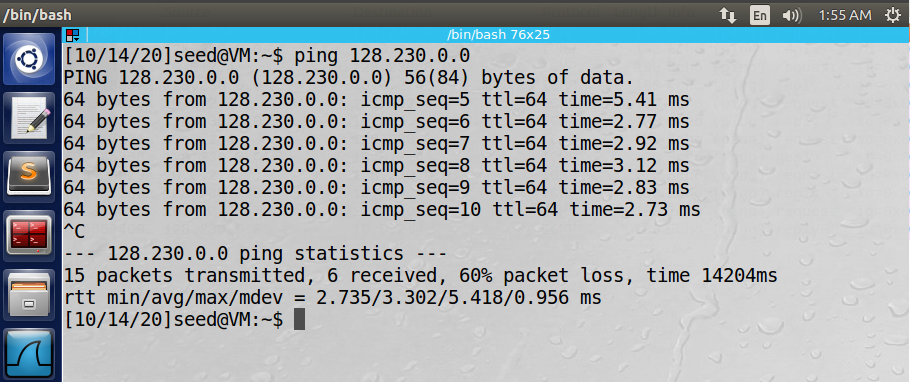
**Explanation:** Once again, we see the ease of implementation and flexibility from Scapy. Here we could track the routers with a very simple loop.

**Task 1.4**

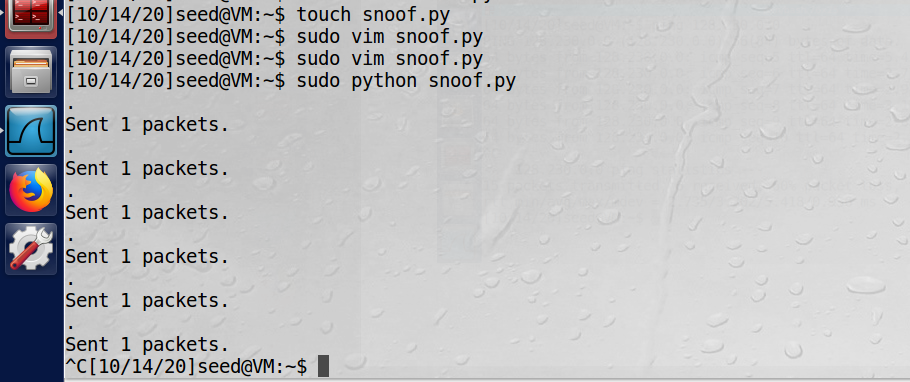
In order to run the sniff and then spoofing (snoofing) program, the following code is used. Here we take in (snifff) the source and destination IP addresses and flip them (spoof) to create an authentic looking response. The first VM, VM A, will ping an IP address, and the second VM, VM B, will run this script (and also Wireshark in the background):

****

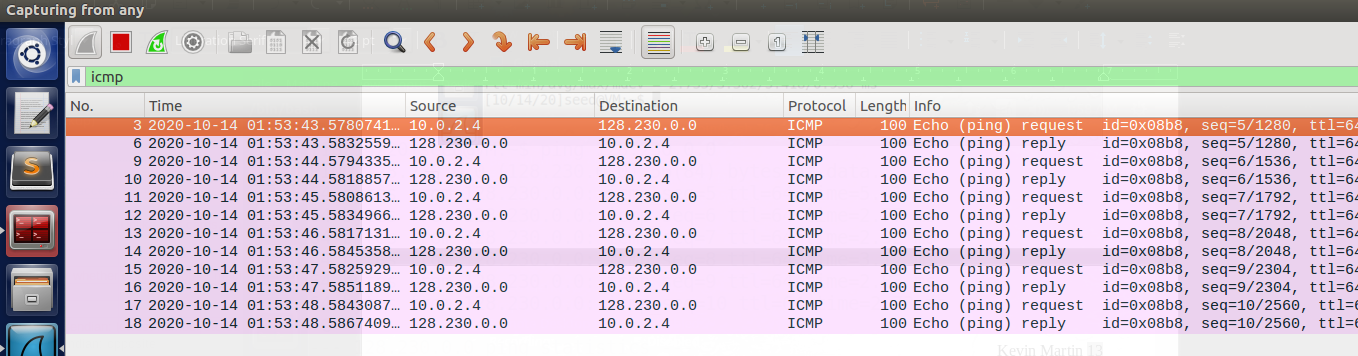
VM A pinging a random IP (128.230.0.0):

****

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**