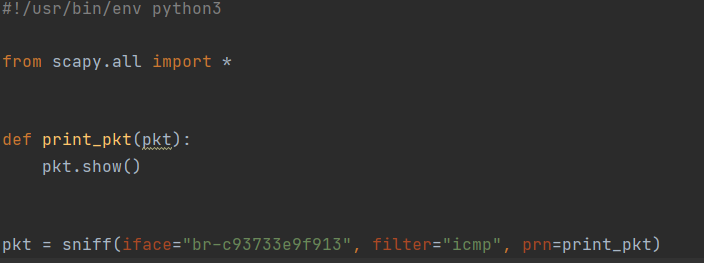
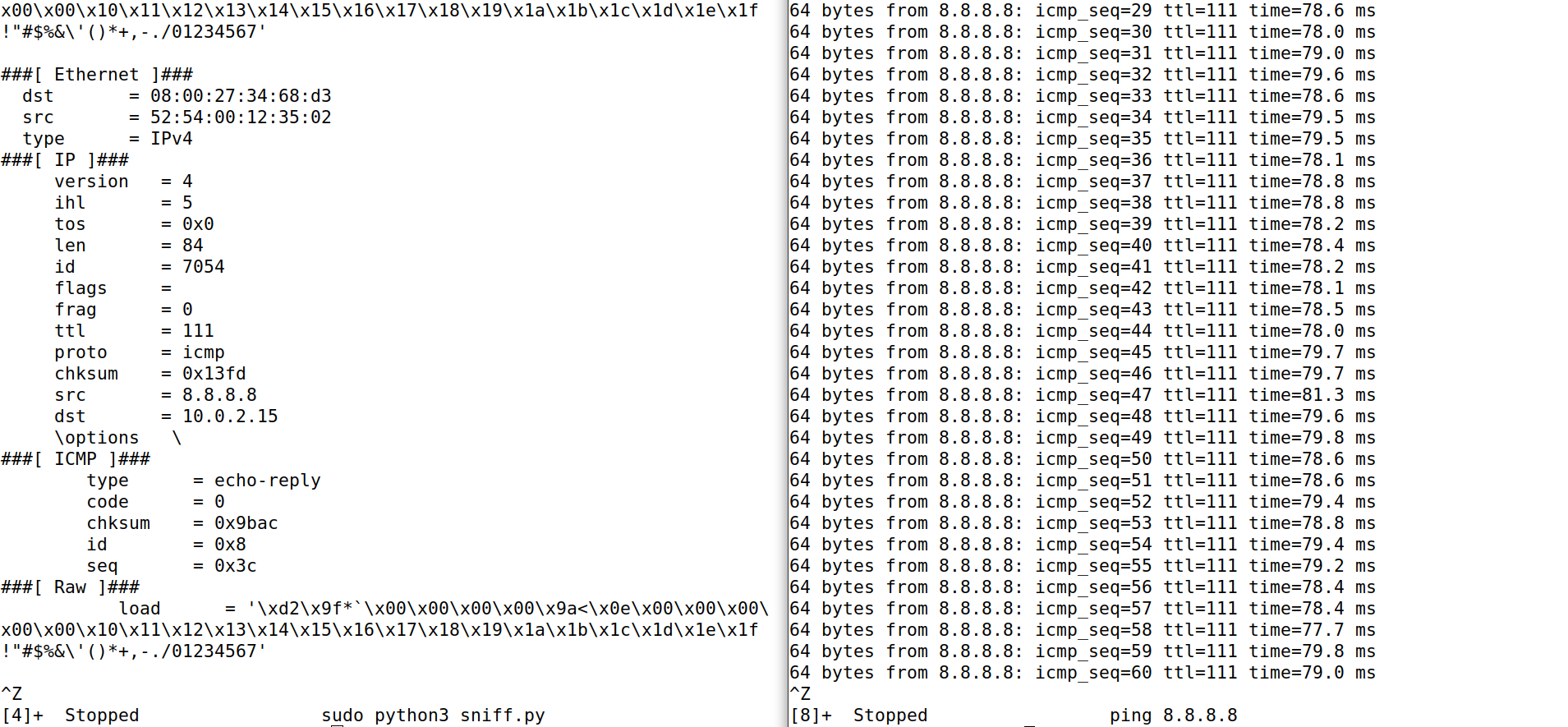
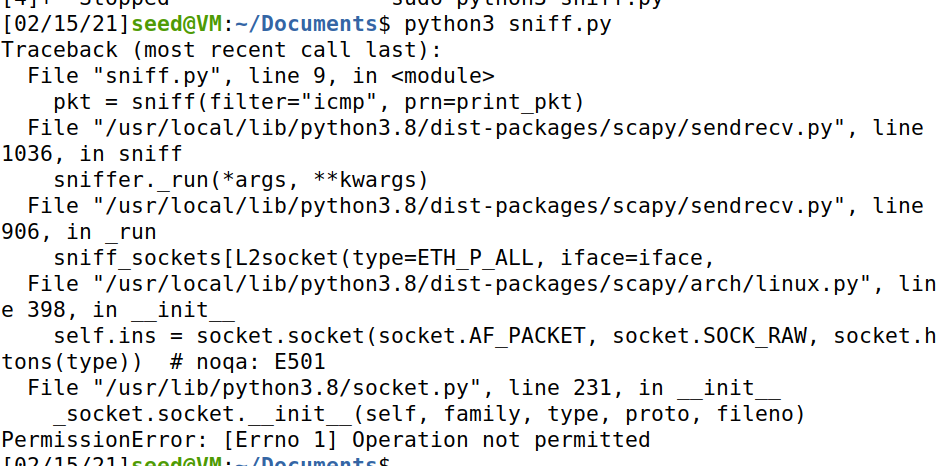
**Sniffing Spoofing Lab Report**

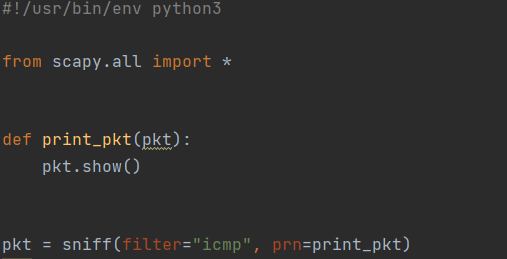
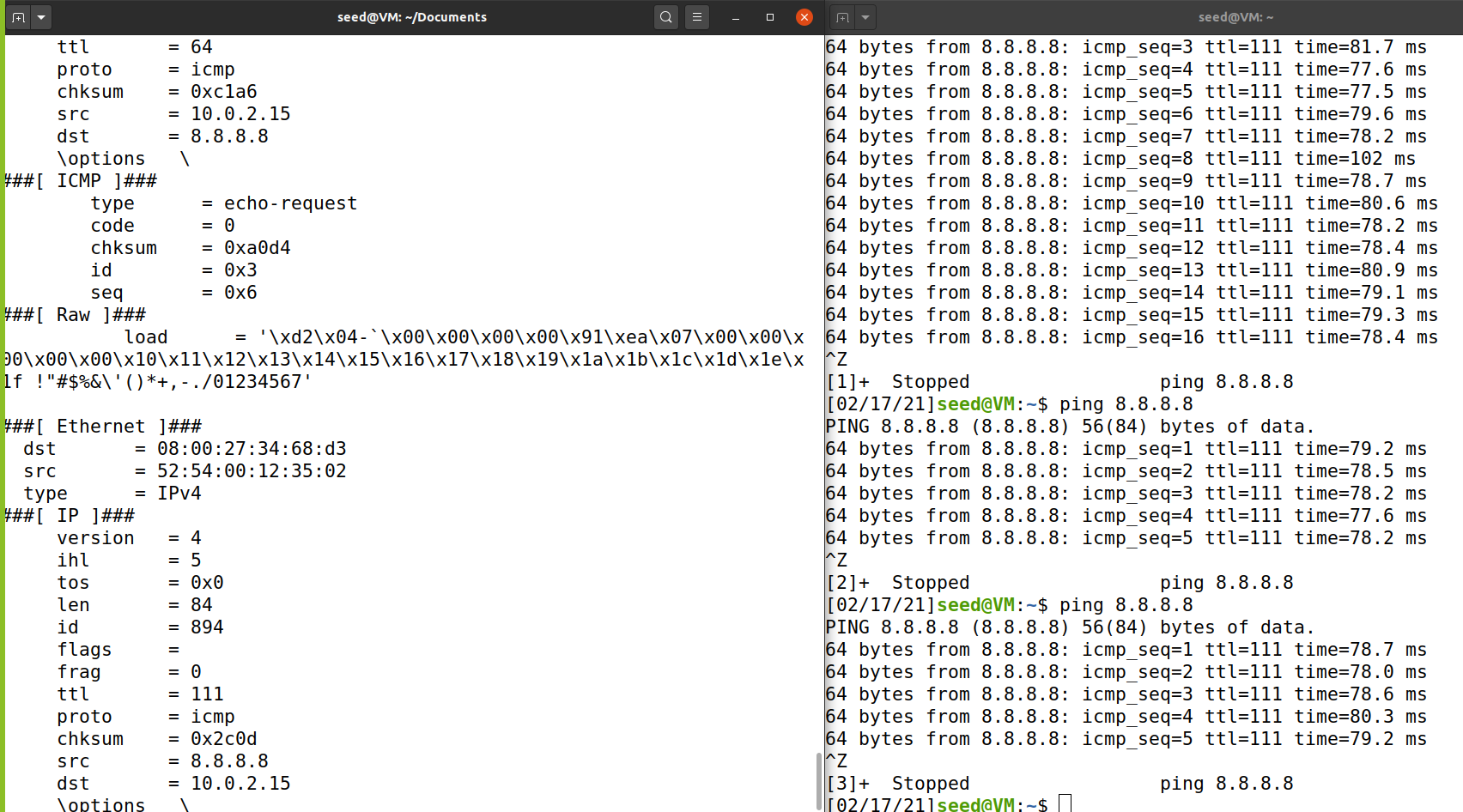
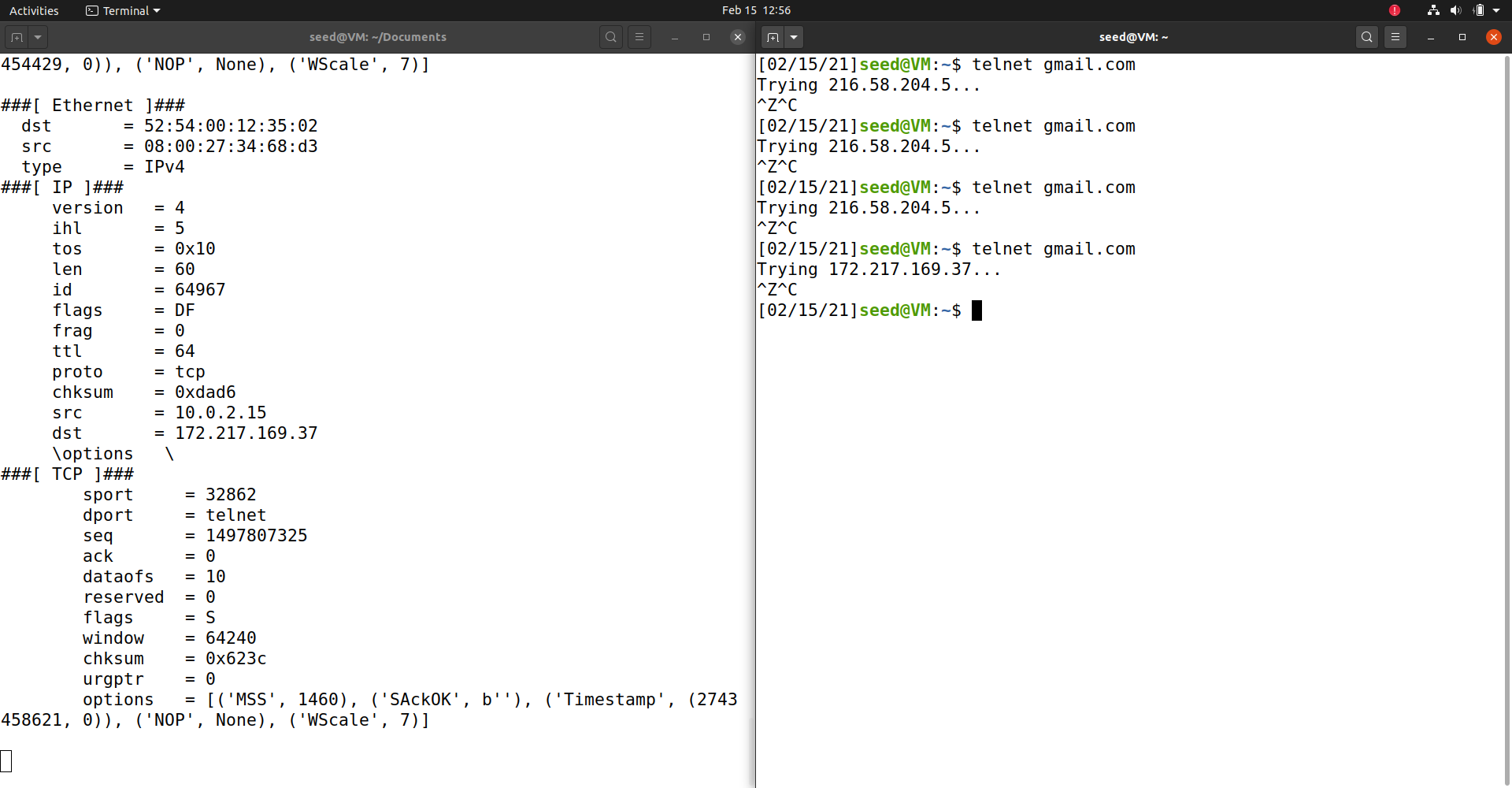
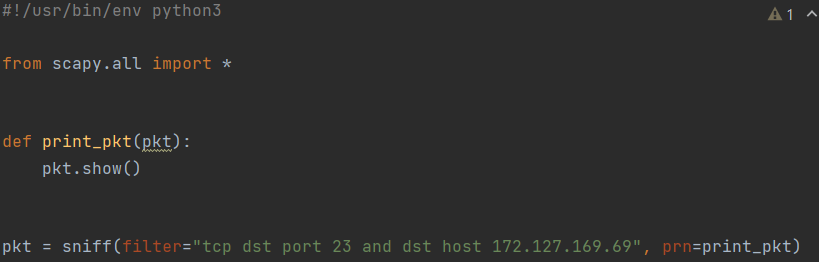
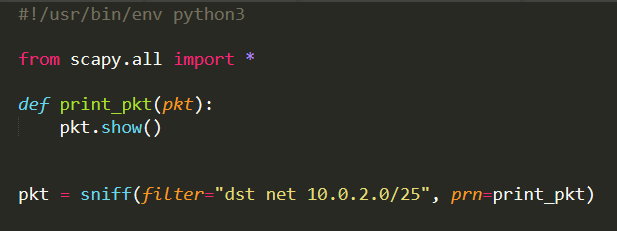
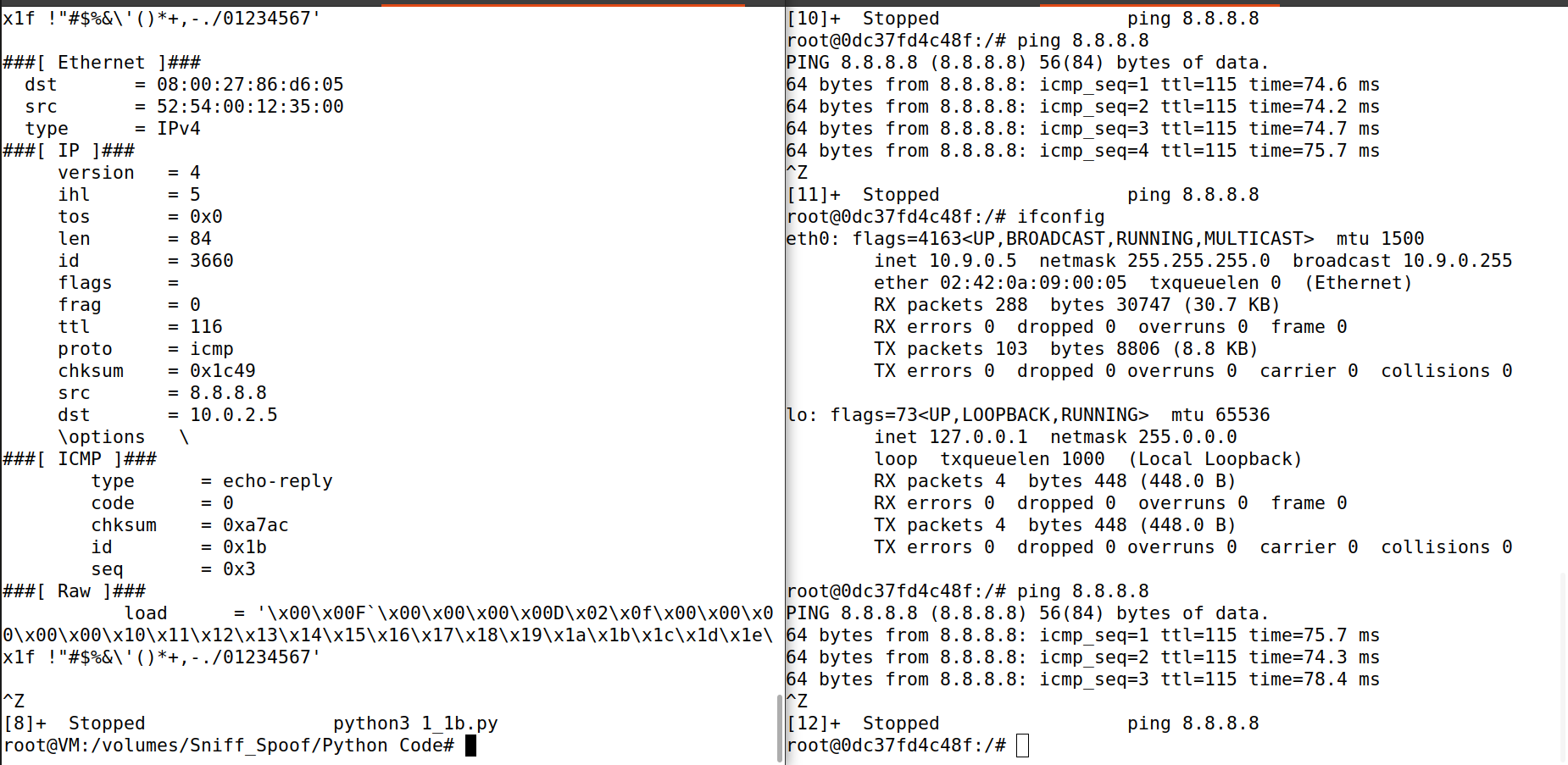
**Tasks 1.1A**:

In this following screenshot(1), it is evident that using the python script (2) with root privilege we were able to capture and document the packets and their content however, in the next screenshot (3) we got a permission error. Opening a raw socket requires higher privilege then given in this task.

1. 
2. 
3. 

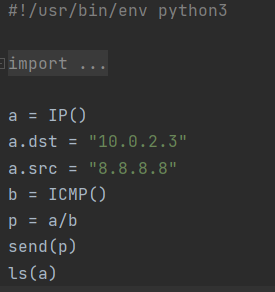
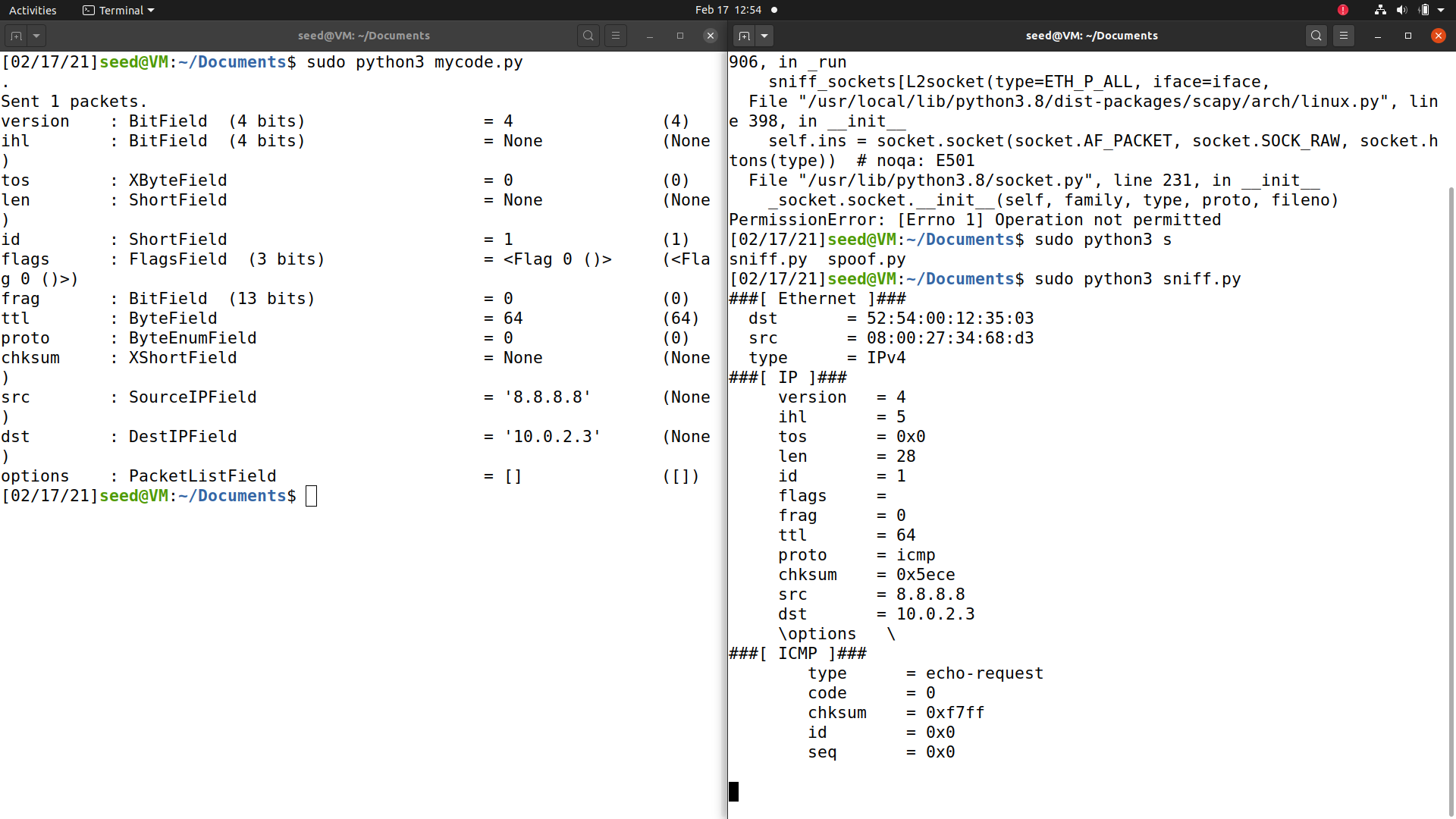
**Tasks 1.1B**:

Using BPF[[1]](#footnote-1) to capture:

1. Only ICMP packets- In this screenshot we used BPF for icmp and the result in the console of the packets that were captured.   
   
2. Any TCP packets from a particular IP with destination port 23- Because telnet is over TCP , we used it to sniff the TCP packets with the destination 23, for 172.217.169.37.
3. Capture packets comes from or to a particular subnet- We used the subnet adrress 10.0.2.0/25 to be sniffed.  
     
   

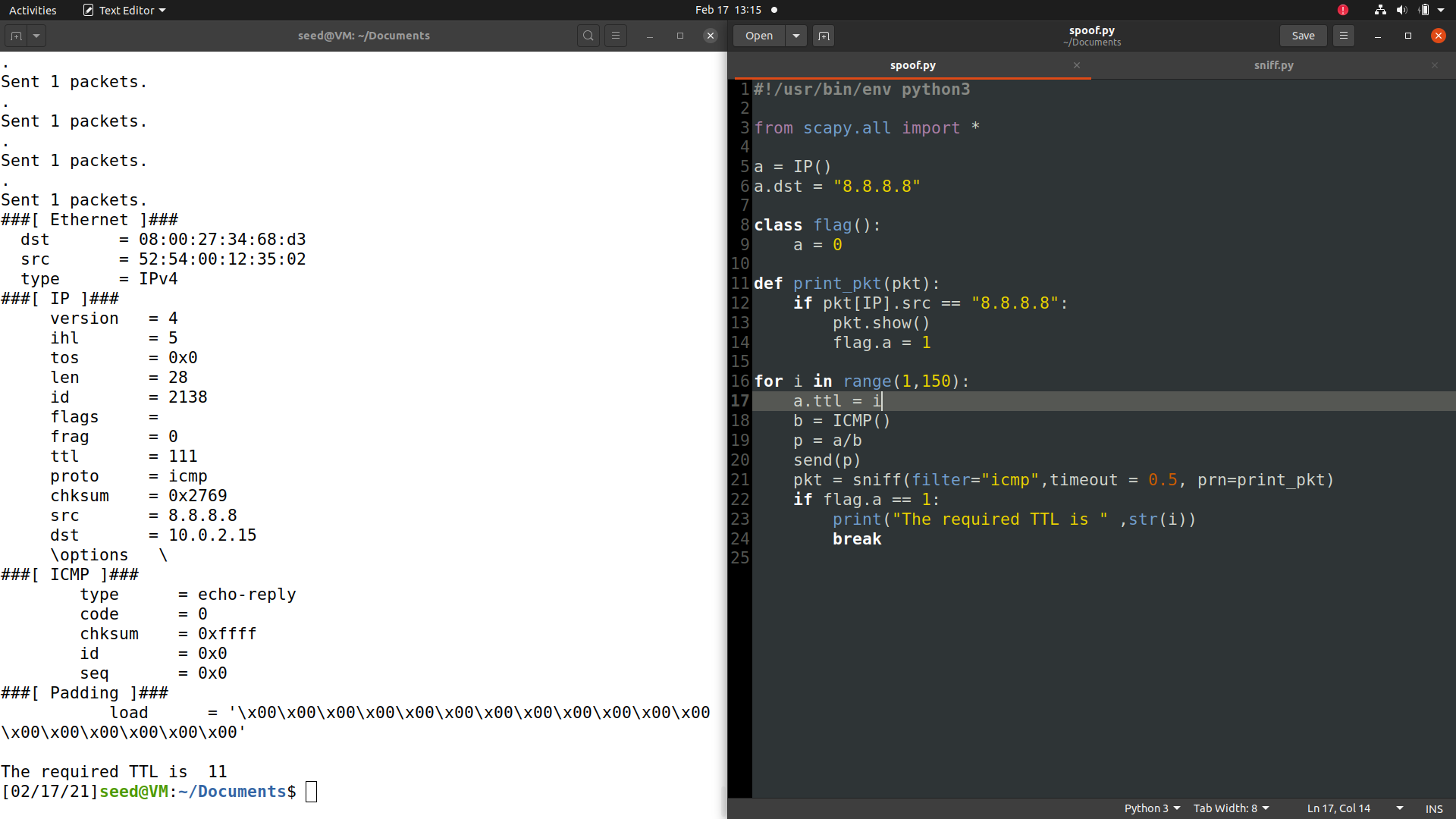
**Tasks 1.2**:

In the next screenshot (1) you can see we used the previous sniffer to sniff the ICMP packet we spoofed. We set the destination of the the packet as the VM we were working on and the source as google public DNS server. And as you can see the screenshot (2) the sniffer see’s the packet source as 8.8.8.8.

1. 
2. 

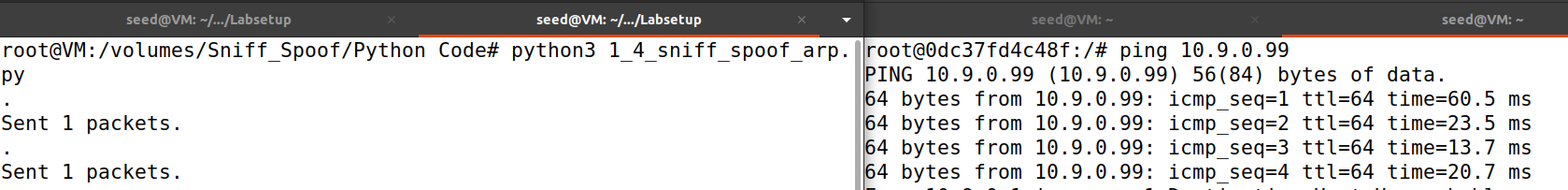
**Tasks 1.3**:

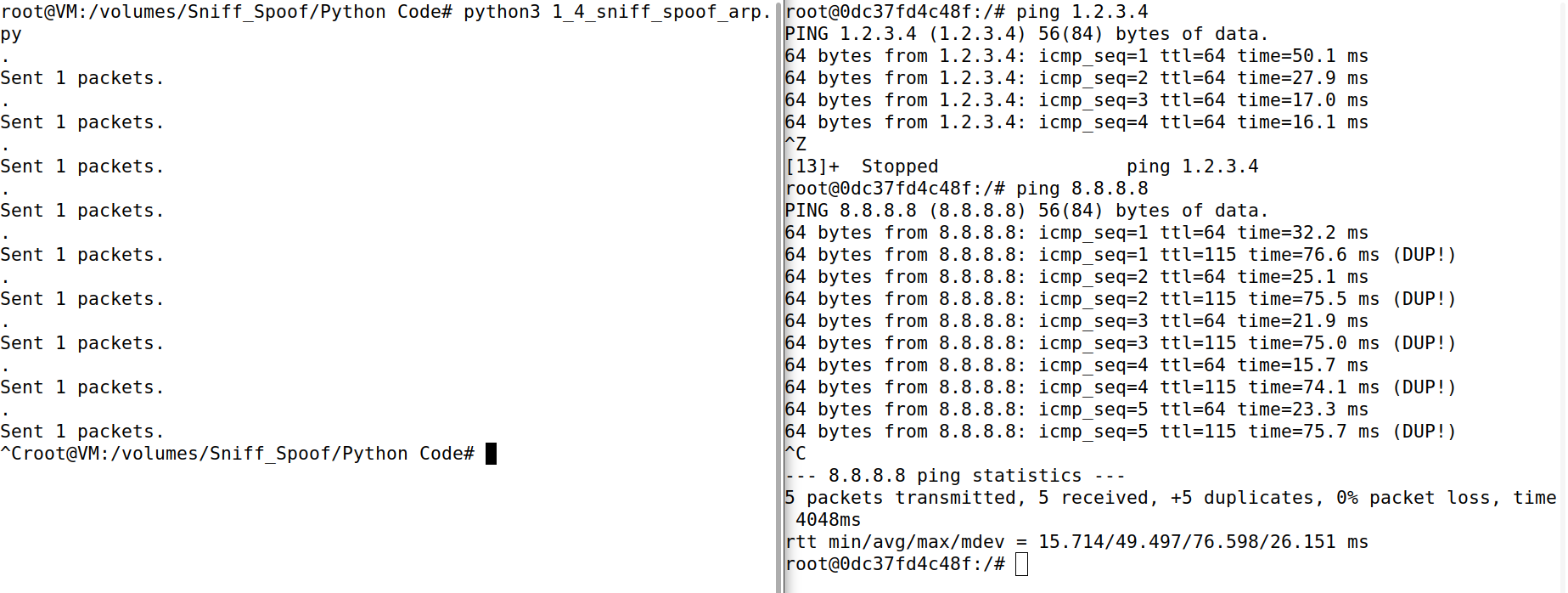
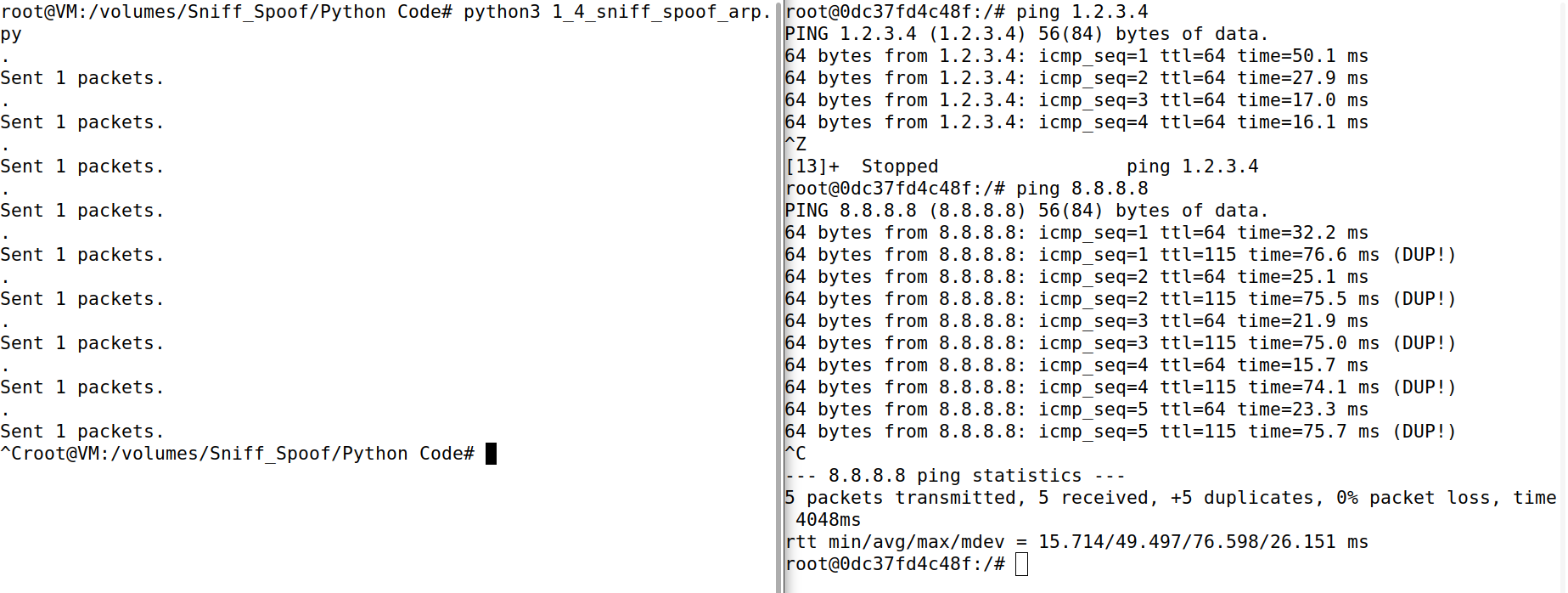
Traceroute - In the next python script we were asked to estimate the distance in terms of network nodes between our VM and a selected destination. We chose Google public DNS server. We made a Boolean flag to know when our packet got to the destination. We used a for loop (with an arbitrary amount of iterations that we knew would be sufficient) and pinged the server, once the response came from 8.8.8.8 we changed the flag to 1, and that’s how we made sure it came back from google. In the console screenshot, it is visible that the number of routers was 11.



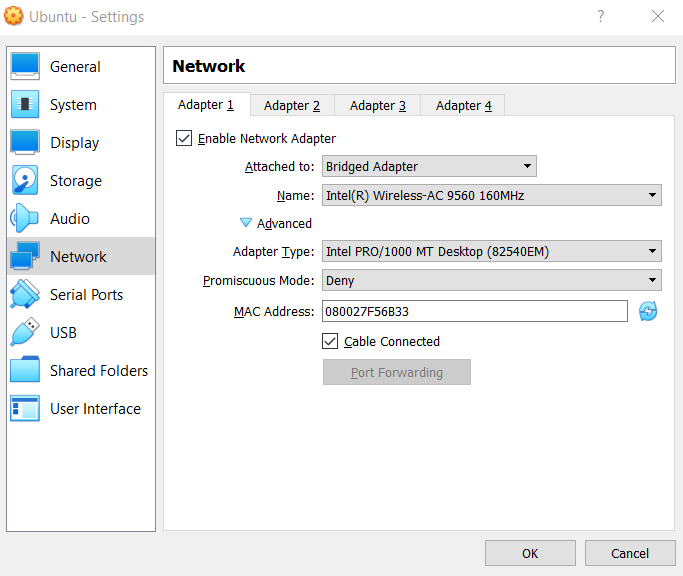
**Task 1.4:**

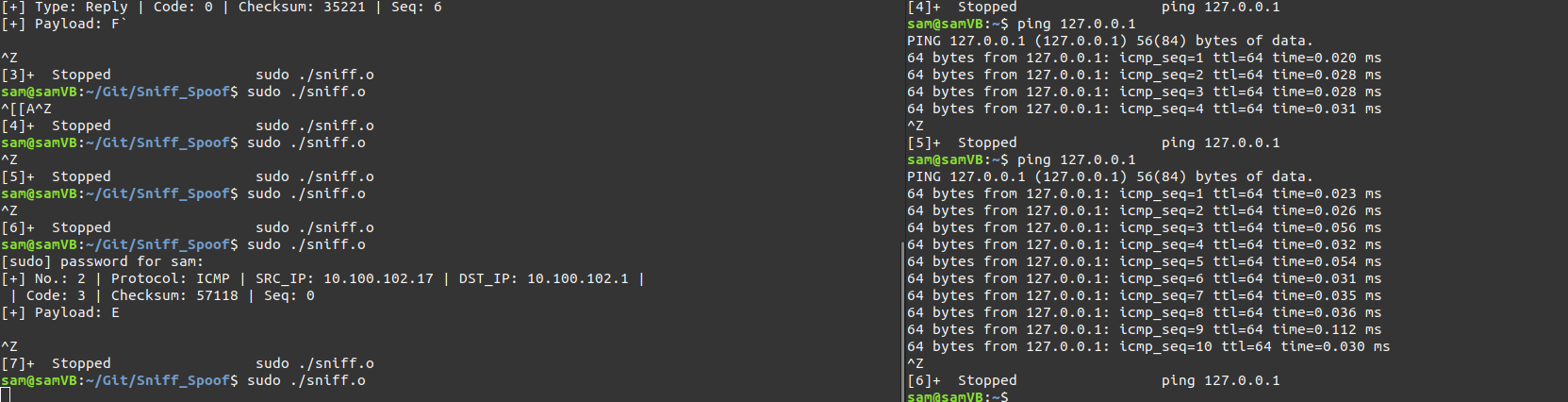
1. **10.9.0.99:**  
   Because this should be an IP in our own LAN our computer will try to reach it through it’s MAC address. Because the MAC will not e available in the OS’s Mac table it will result in us sending an ARP request to get the MAC. Those are broadcast requests sent to the whole LAN asking for the computer with the requested IP address to send over his MAC address. We sniffed the packets and after receiving suck a request we spoofed the ARP reply and than the ICMP reply.



1. **8.8.8.8:**Because we also receive a response from the real server and from our spoofing code we get a DUP (duplicated) response.  
   
2. **1.2.3.4:**Because this IP does not exists the only response we receive is from our spoofing code  
   

**Task 2.1A:**

1. Library calls:
   1. **pcap\_open\_live**: Starts sniffing on the chosen network interface. In here we define the buffer size, promisc mode on/off and delay in milliseconds. This needs root privileges.pcap\_datalink: Returns the kind of device we're capturing on.
   2. **pcap\_compile**: Compiles the filter expression stored in a regular string format to binary and sets in on the sniffing handle.
   3. **pcap\_setfilter:** Starts the above compiled filter,pcap\_freecode: Frees up allocated memory generated by pcap\_compile.
   4. **pcap\_loop:** Starts the actual sniffing session on the sniffing handle we opened previously with the open\_live function. For each receiving packet this loop will call the got\_packet function that will process each and every packet according to our settings.
   5. **pcap\_close:** Closes the sniffing session.
2. **Root access:**  
   We need root access because pcap library needs access to low level network interface functions like CAP\_NET\_RAW[[2]](#footnote-2). Any access like that needs root privilege in order to access low level and hardware drivers, that an OS security feature.
3. **Promiscuous mode:**  
   In order to turn Promiscuous the mode on/off we simply change the third int value of pcap\_open\_live to 1/0 accrodingly:  
   on -   
   off -   
   Promiscuous mode allows a network interface to access data and communication that it not intended to and directly access hardware interface and drivers. We will demonstrate by first changing our VM network setting to disable Promiscuous mode by default than we set networking to bridged.   
     
   We will send a ping to 127.0.0.1 which should be sniffed from loopback (“lo”) interface and will try to sniff them from the main “enp03” interface from the VM.   
   We can clearly see that when we try without the Promiscuous enabled in the code we cannot sniff those pakcets but as soon as we enable them we can sniff the loopback packets from a non-intended interface.

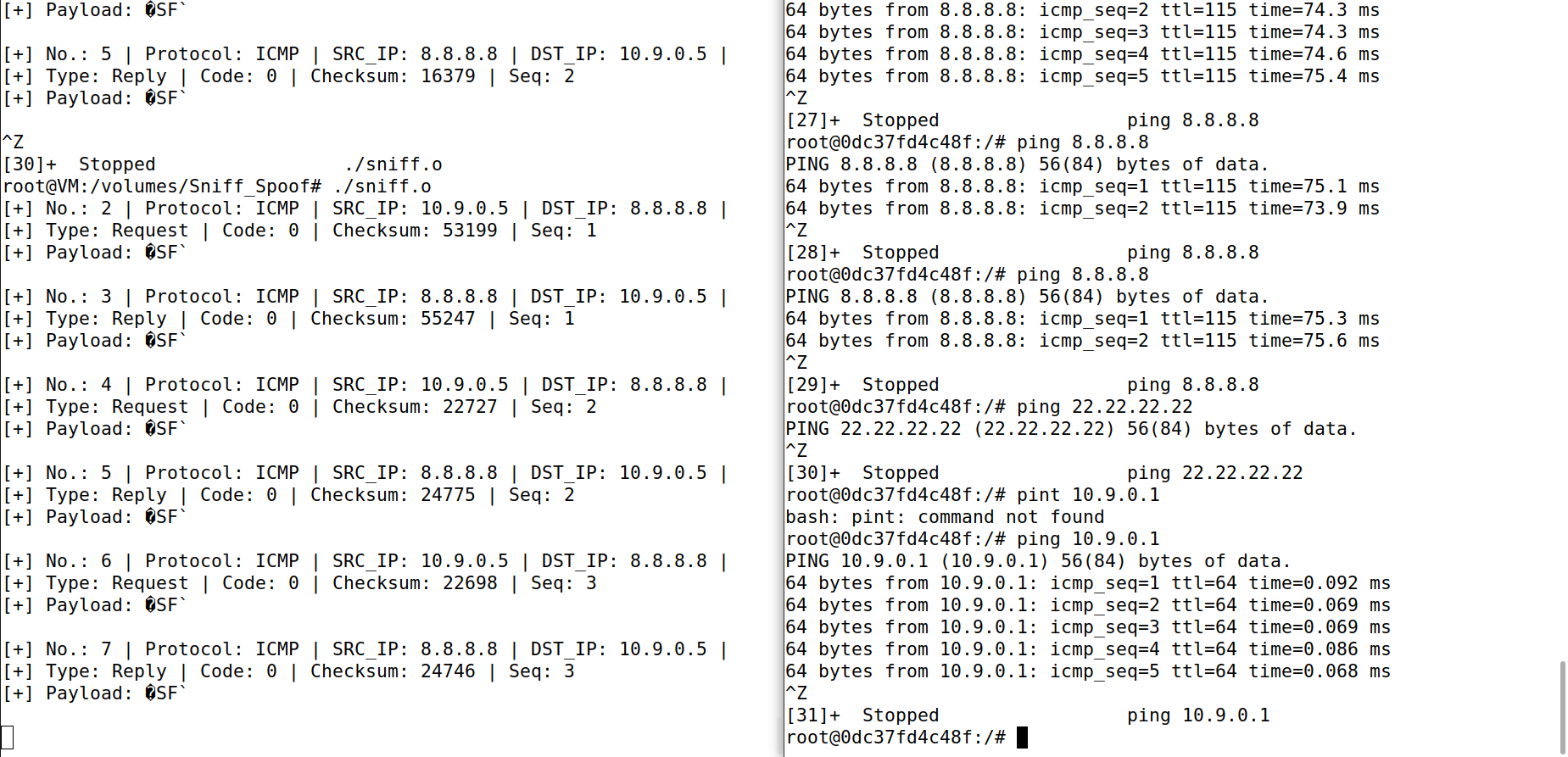


With promisc mode on

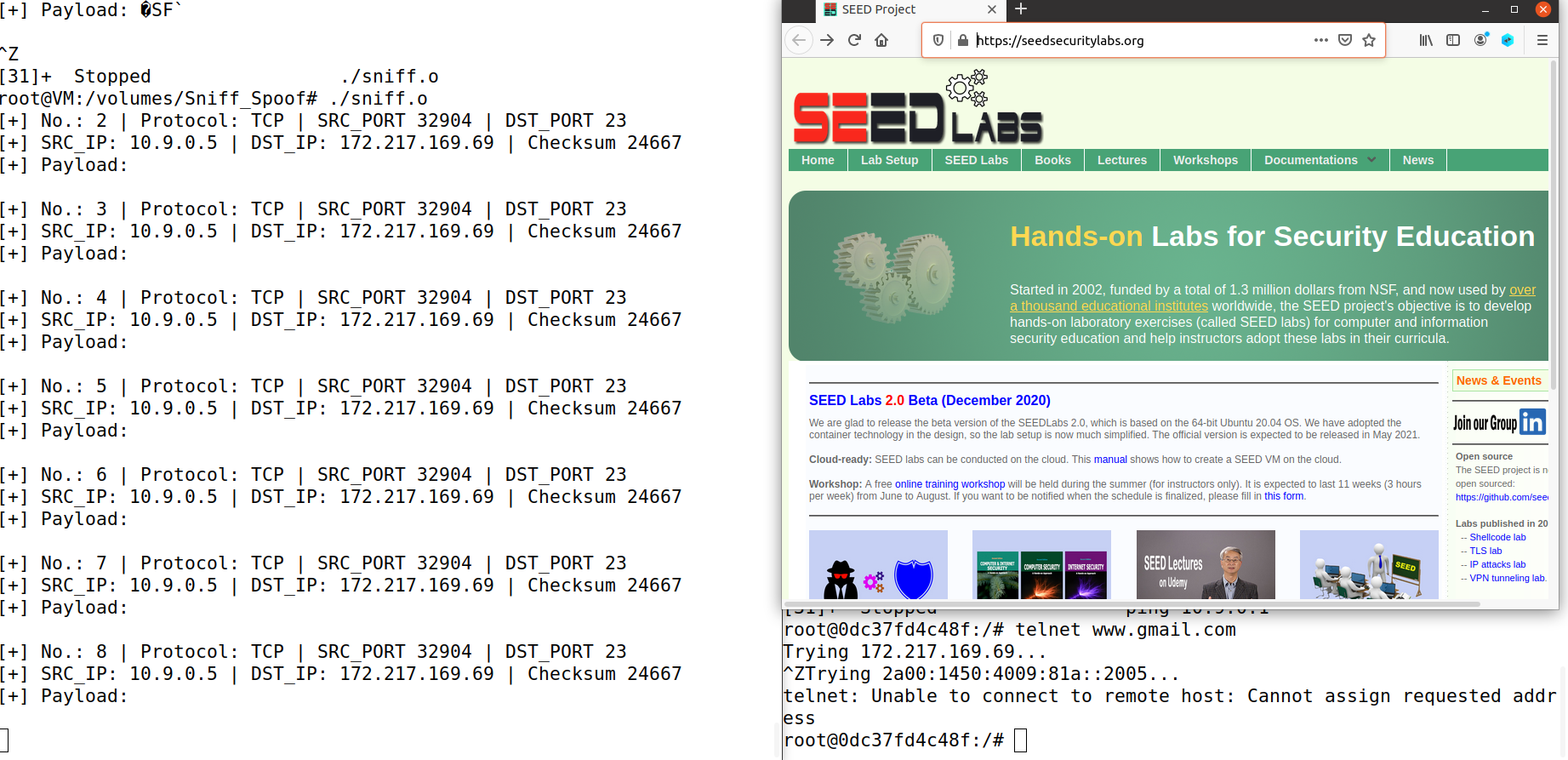
With promisc mode off

**Task 2.1B:  
  
ICMP Between to specific hosts:**

In the next screenshot we captured ICMP packets between 8.8.8.8 to 10.9.0.5  
  
as we can see in the next screenshot no packet that is not netween those IP’s is being sniffed no matter the command



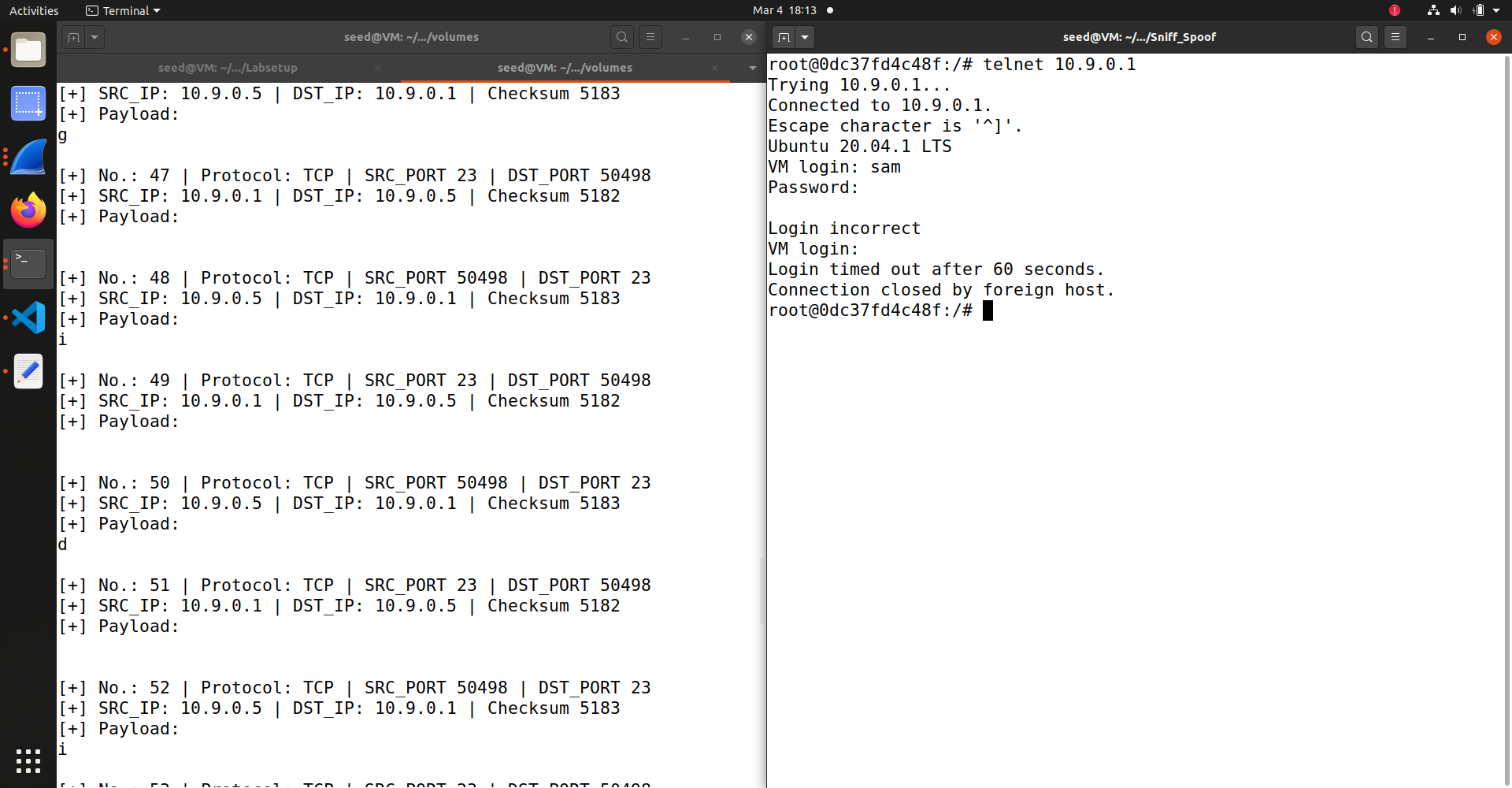
**TCP PORT Range:**   
  
In the next expression we sniffed tcp packets between ports 10 to 100  
****  
As we can see in the next screenshot when we send a telnet (port 23) packet it is being sniffed, but when we surf via HTTPS (port 443) there are no packets being sniffed:



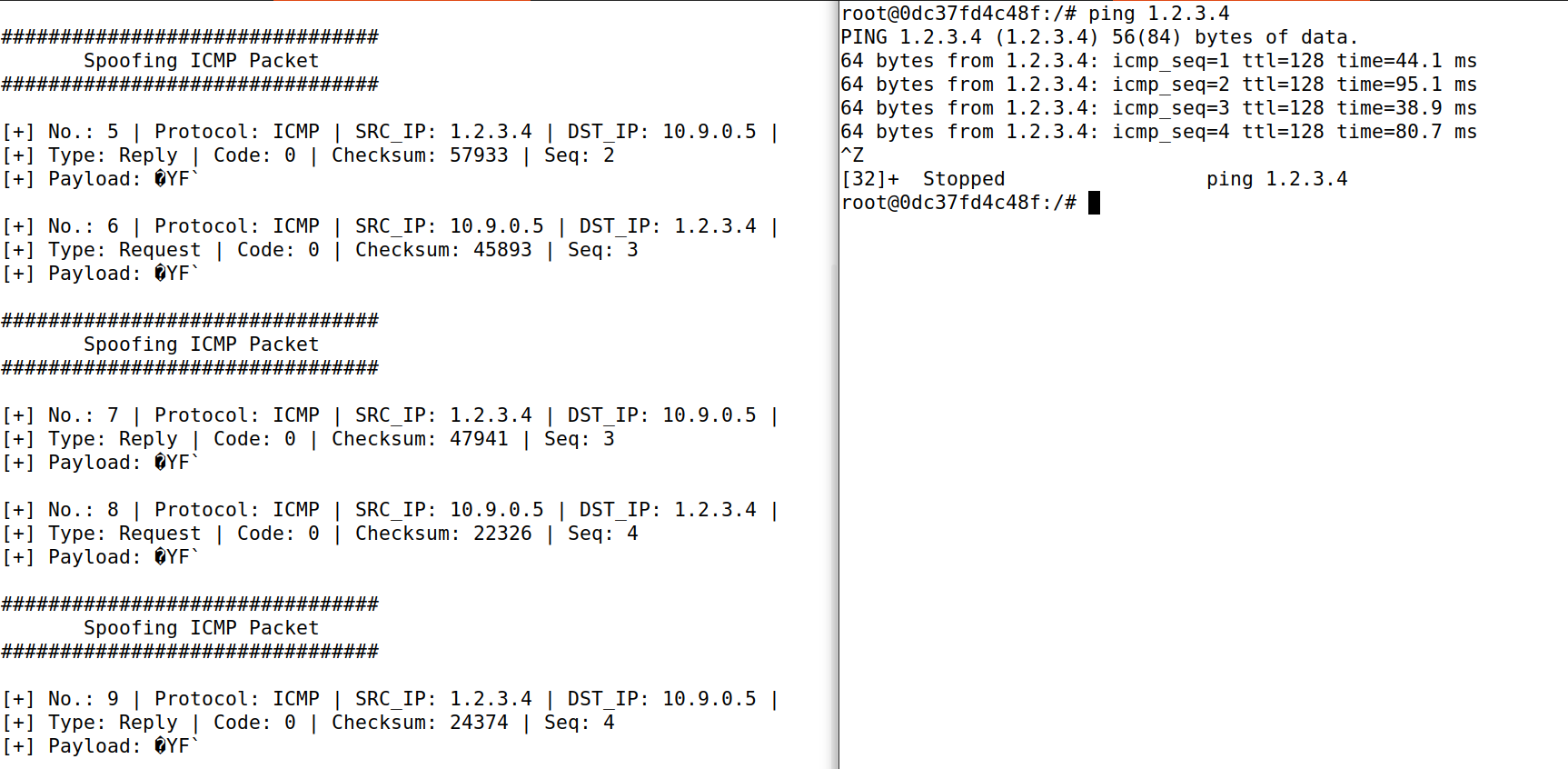
**Task 2.1C:**  
At the end of every Telnet packet there is a payload which contains the transferred data in plain text. Because Telnet is not encrypted, we can sniff this data and print it out with our sniffer program. We will try to connect to our own VM via telnet with these credentials:

Username: sam

Password: gidi

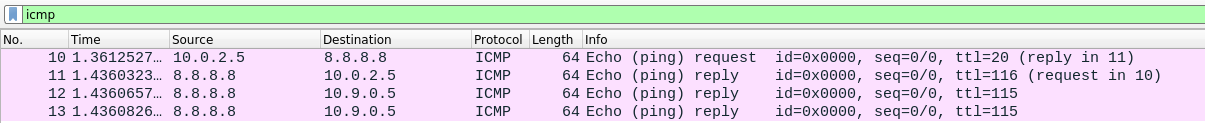
Telnet splits the passwords and transfers it in different packets, one for every char, so in our case, we will have four packets containing the word “gidi” as show in the next output:  
  


**Task 2.2A:**  
As we can see in the next screenshot we ping a non-existent IP and get a response from our spoofing program that sniffs and awaits the request.



**Task 2.2B:**

As we can see in our spoofed ICMP request we have sent google a spoofed request from one of our containers (the host) which is in IP 10.9.0.5.  
We see the packet being sent to 10.0.2.5 because it is the host VM that passes the packet down to the container which. The proof to that is that the TTL is being reduced accordingly an we can also see the detail in the Ethernet layer via the MAC addresses passing down those packets:



Bibliography:

1. struct ip source - https://unix.superglobalmegacorp.com/Net2/newsrc/netinet/ip.h.htmlsd

1. https://en.wikipedia.org/wiki/Berkeley\_Packet\_Filter [↑](#footnote-ref-1)
2. https://linux.die.net/man/7/capabilities [↑](#footnote-ref-2)