Dr. Chengyi Qu

Computer Security | CEN: 3078 | CRN: 15627

Abstract

This lab discusses and outlines the concepts of both “sniffing” and “spoofing” within network security to better understand the tools and technologies used in these processes. Python/Scapy programs, written by me, to highlight these techniques while Wireshark confirms our results.

Sniffing & Spoofing Lab: one

Nash William Morrison

1. **Lab Environment Setup:**

Before we can begin our sniffing and spoofing programs, we must first properly set up our environment. Once our Virtual Machine (VM) is set up with our “SEED-Ubunut20.04” and our “Labsetup.zip” files properly side-loaded onto our VM, we can start the process of starting up our docker and creating our containers to evaluate our programs.

A screenshot of a computer

Description automatically generated

Figure : Building Our Docker & Containers

To build and establish a network connection for our containers, we must open a terminal within our “/Labsetup” directory. From here, we must type two commands in our terminal:

&

Once typing these commands, our containers are up and operational and we can now begin our tasks associated with this lab.

1. **Setting Our Three Terminals:**

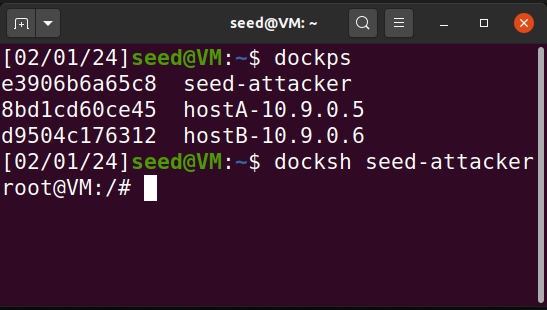
****

Figure : Attacker Terminal

For this lab, we need to set up three different terminals; one as our “Attacker” and two as our “User Machines.” The two following commands will establish a connection with our three separate terminals:

&

“dockps” prints to console our three different containers that are online. To get to our root privileges, we must use the “docksh <ID>” command within each terminal shown below:

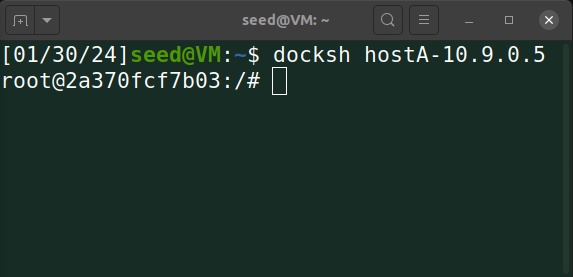


Figure : Host A Terminal

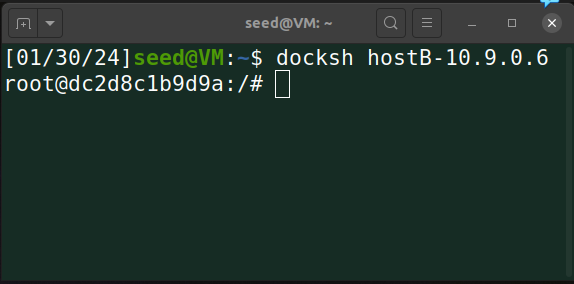


Figure : Host B Terminal

Our attacker shell throughout the rest of our lab will be “purple” while our two User Machines will be “green” respectively. The color coordination helps with keeping track of all these elements. Now we need to locate our special “Network Interface Name” so that we can attack our host computers.

1. **Finding Our Network Interface Name**

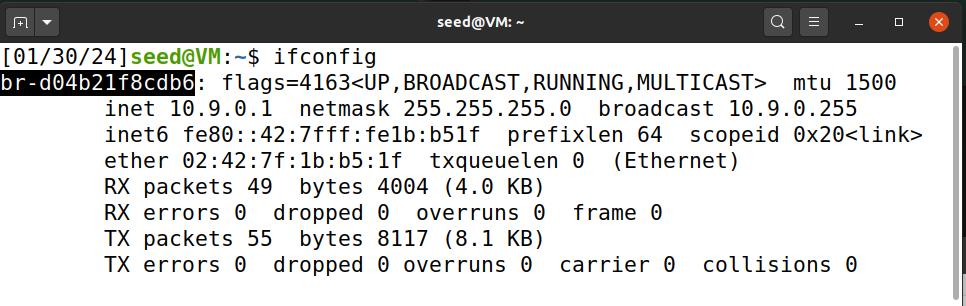


Figure : Locating Network Interface Name

To locate this, we use the following command:

**”**

This allows us to locate our specific “Network Interface Name.” Highlighted in “black,” ‘**br-d04b21f8cdb6’** , repeatedly used throughout our Python programs.

1. **Task 1A – Sniffing Packets**

For our tasks, we are going to be using a combination of “scapy” and “python.” Scapy is a packet manipulation tool while Python is a powerful programming language that when combined, allows us to sniff and spoof packets on a specific network. In this portion of tasks, we are going to be “**sniffing**” which is watching and capturing data packets passing through a network. The following code below is our python program which allows us to sniff packets.

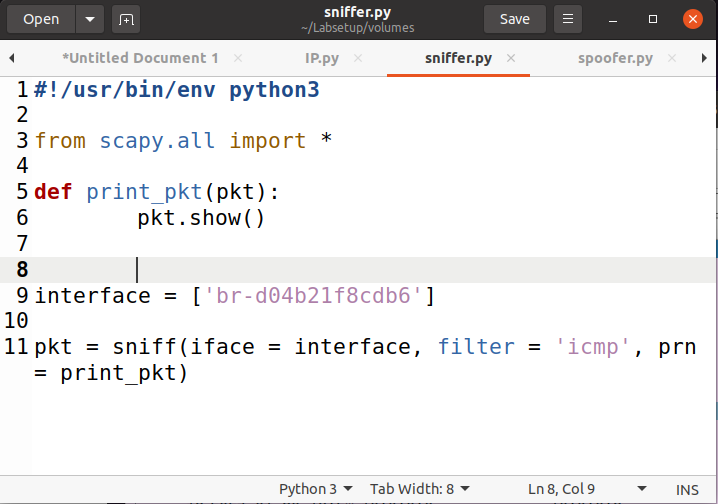


Figure : Sniffing Python Program

**From scapy.all import \* - Imports all the necessary tools to sniff and spoof packets.**

**Def print\_pkt(pkt): - Creates a function that will print our packets sniffed to console.**

**Pkt.show()**

**Interface = [**‘**br-d04b21f8cdb6’] - Creates our interface value to put into our sniff() function.**

**Pkt = sniff(iface = interface, filter = ‘icmp’, prn = print\_pkt) - Our sniff function sniffs our specific network.**

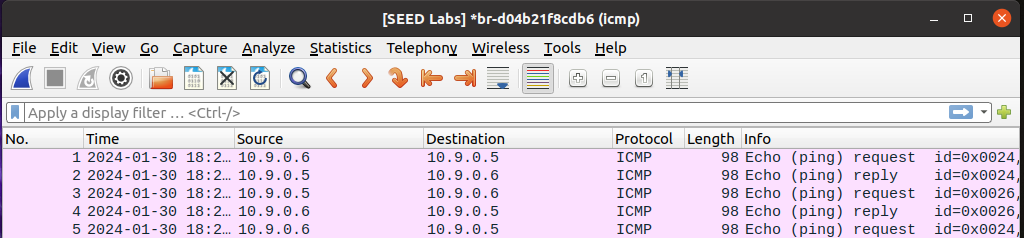
****

Figure : Wireshark Pinging Hosts IP's

The Wireshark picture above shows that we have a connection going back and forth between Host A and Host B. We achieved this by pinging each other’s respective IP addresses. Now that we have a connection, let us use our python program.



Figure : Sniffing Host A to B

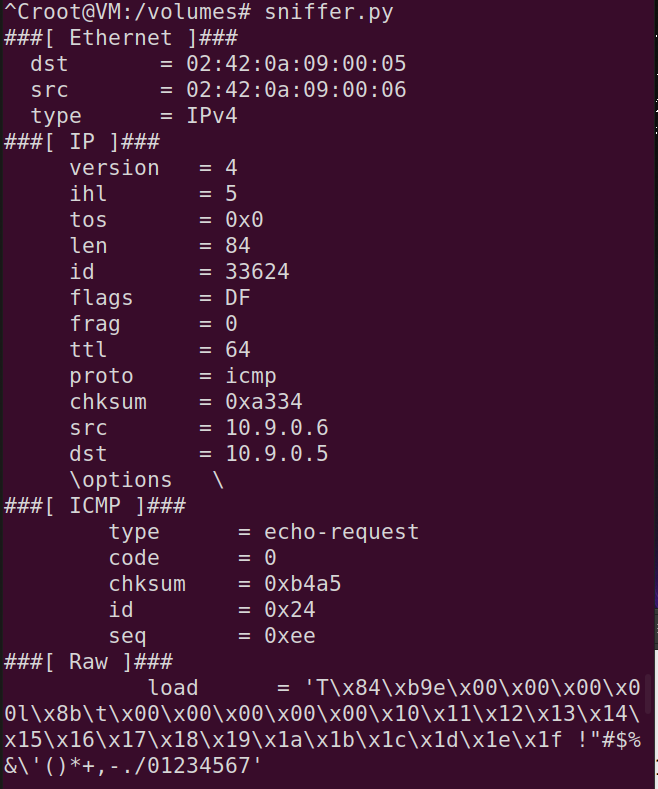


Figure : Sniffing Host B to A

The boxes show that we in fact did successfully sniff an echo-reply alongside an echo-request from both Host A and Host B. Our Id = 0x24 correlates with our Wireshark result, therefore proving our attempt. Keeping in mind that we must have root privilege to run scapy and these python programs. Let us try running our program without root privilege to see what our results are:

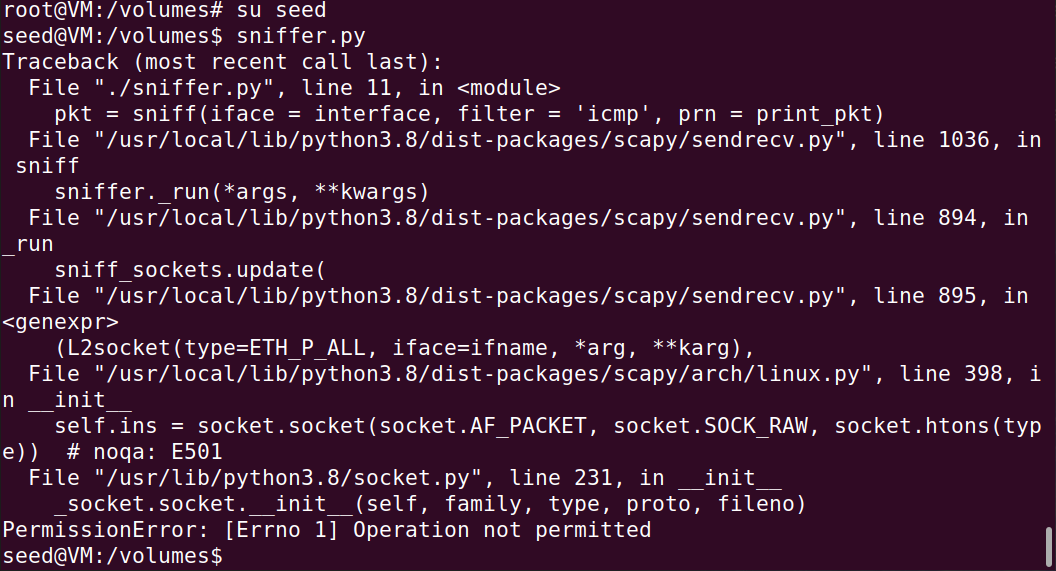


Figure : Running Sniffer Program Without Root Privilege

As seen above, we receive a permission error. This is because we need root privilege to run these programs with total access over our system. Now we are going to move onto the different filtering options scapy has to offer.

1. **Task 1.1B – Scapy’s Filtering (Extra Credit)**
2. **Capture Only ICMP Packets:**

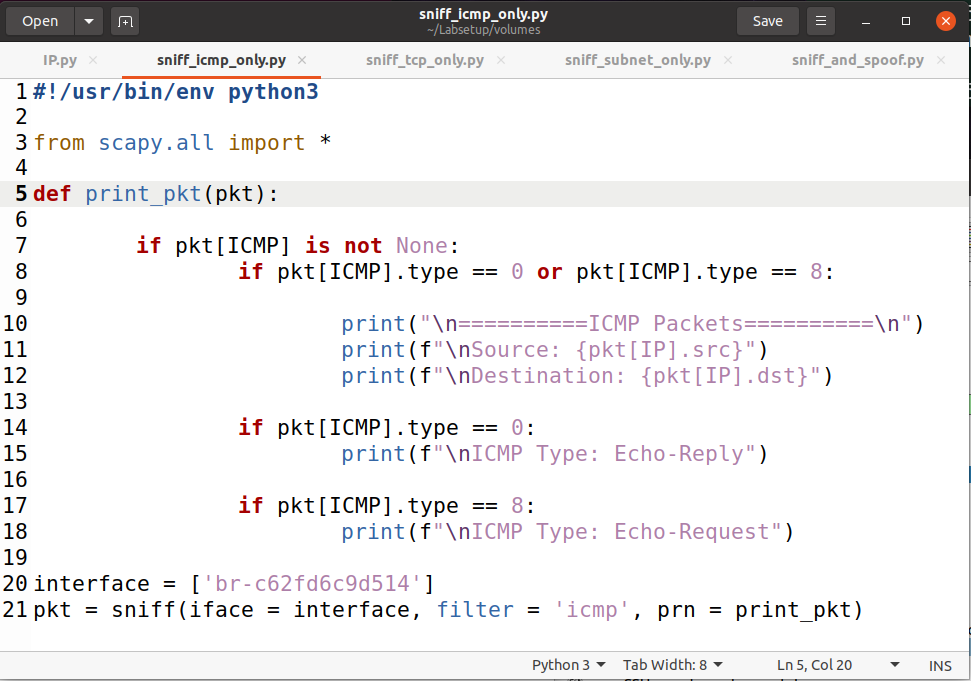
****

Figure : Sniff Only ICMP Packets Program

Key aspects of this program:

**If pkt[ICMP].type == 0 or pkt[ICMP].type == 8:** - We are funneling between echo replies and echo requests with a nested if statement so we can tell the difference between the ICMP packets.

**Filter = ‘icmp’** – We are filtering solely ICMP packets to sniff.

Everything else is the same as our normal sniffing program displayed above.

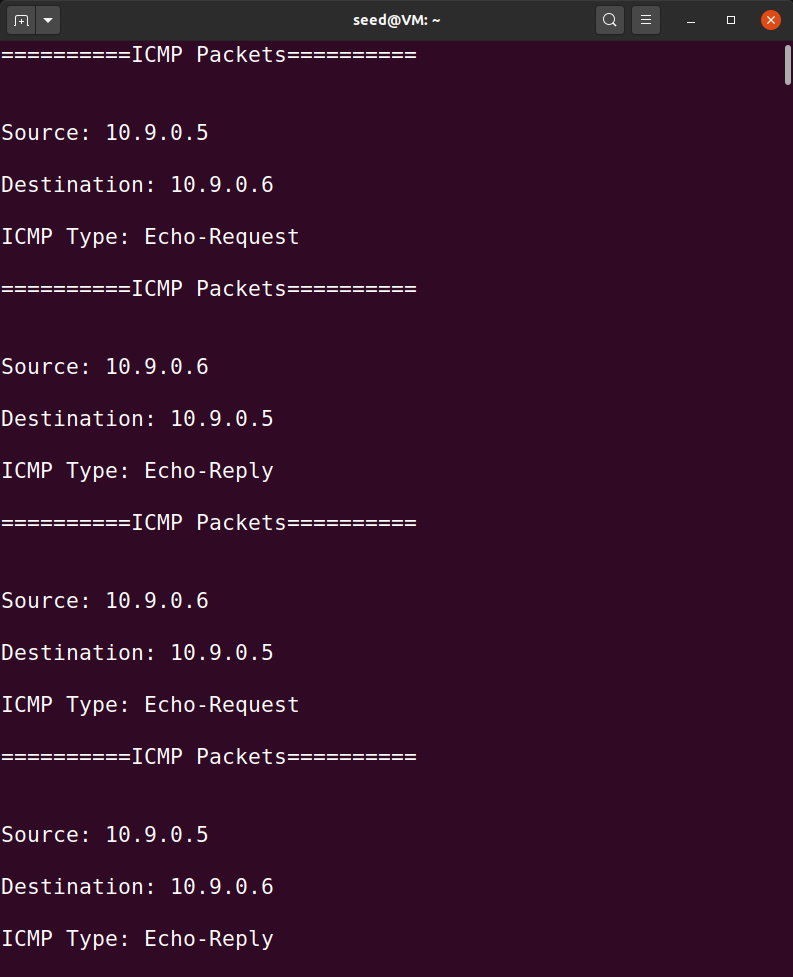
****

Figure : Sniffing ICMP Packets Terminal Output

My python code allows me to easily read between each ICMP packet. As we can see above, we have an echo reply and echo request going back and forth between both host A and B, and we are able to specifically pick up the ICMP packets without issues.

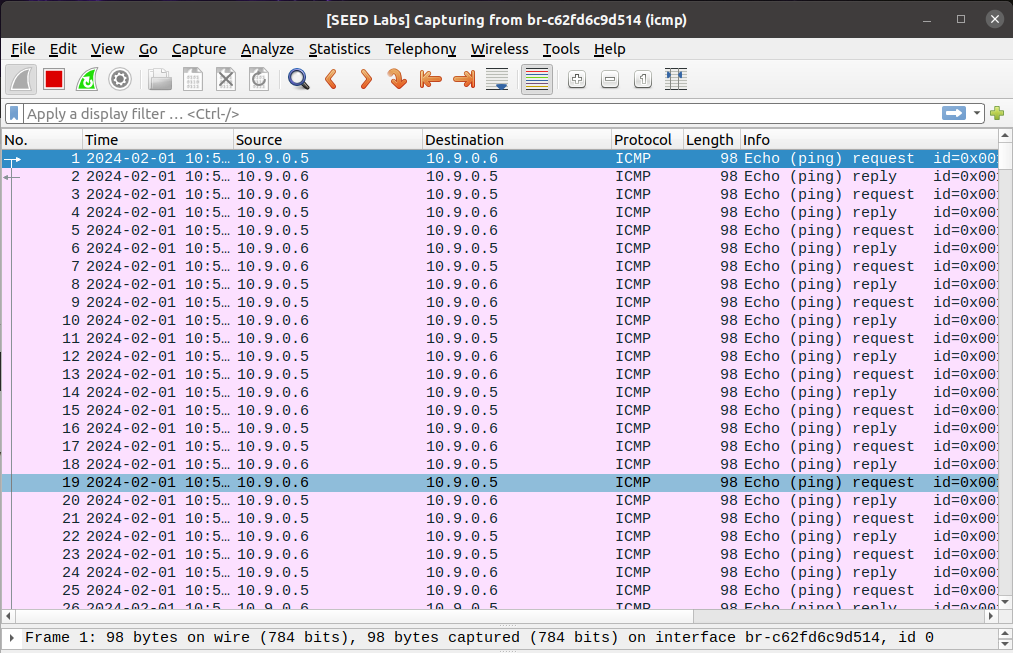


Figure : Wireshark ICMP Filter

Here our Wireshark result shows our filter of our ICMP packets going back and forth between both Host A and Host B, proving that we successfully completed this task.

1. **Capture any TCP packet that comes from a particular IP and with a  
   destination port number 23.**

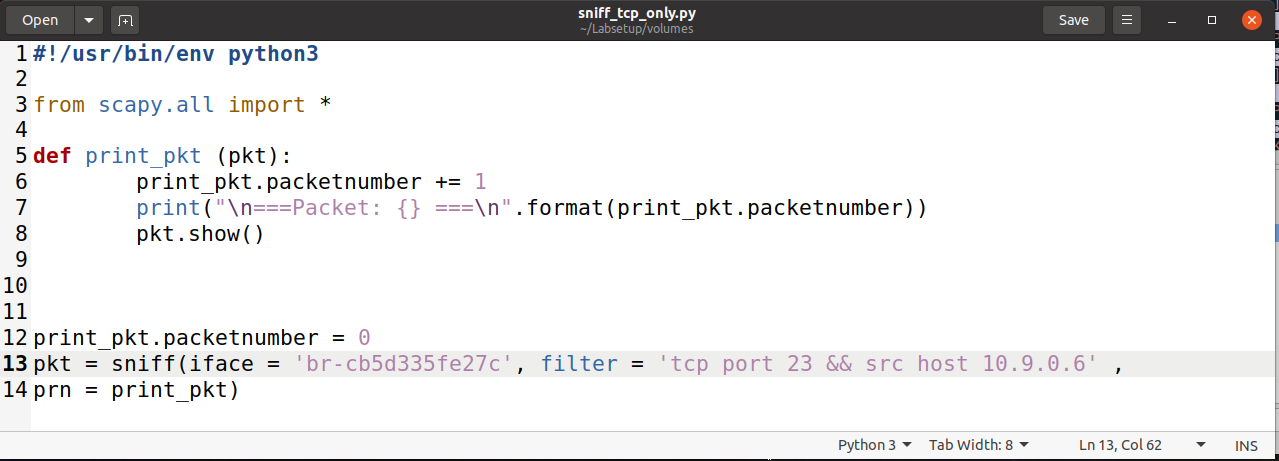
****

Figure : Sniffing Only TCP Packets Program

Key aspects of this program:

**Print\_pkt.packetnumber += 1** – Here we are incrementing our packet number by 1 for each packet received to print out to console. A poor man’s loop.

**Filter = ‘tcp port 23 && src host 10.9.0.6’** – Here we are setting our filter to capture any TCP packets that come from host 10.9.0.6 (Host B) and (&&) with a destination port of 23.

The rest is the same kind of setup as our other sniffer programs listed above.

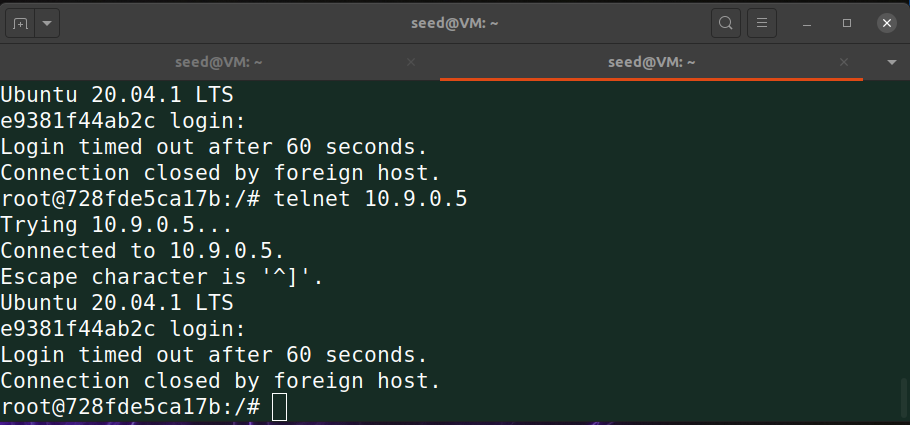


Figure : Host B Terminal

To achieve the start of our 3-way-handshake with TCP, we must first ping an IP address that starts the TCP connection. We ping telnet 10.9.0.5 from Host B (10.9.0.6) which waits for us to enter a login. Since we do not have the login credentials, telnet times us out after sixty seconds. We then receive our TCP packets that print to console.

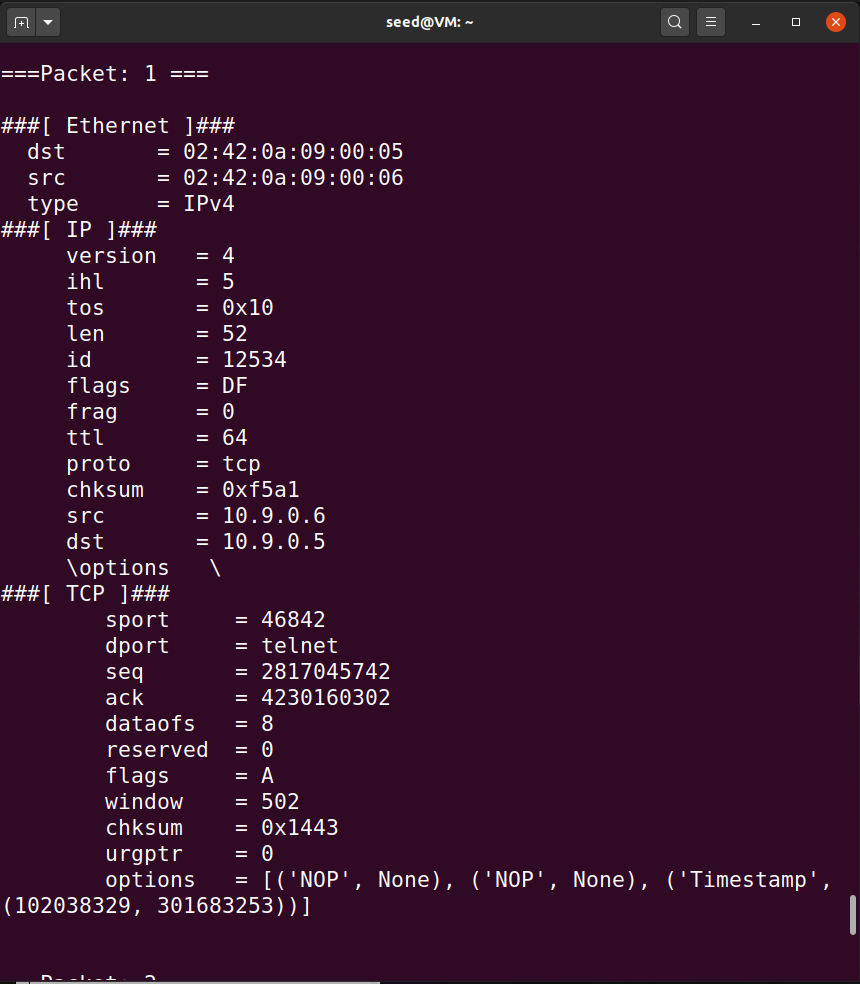


Figure : TCP Packet 1 Terminal Output

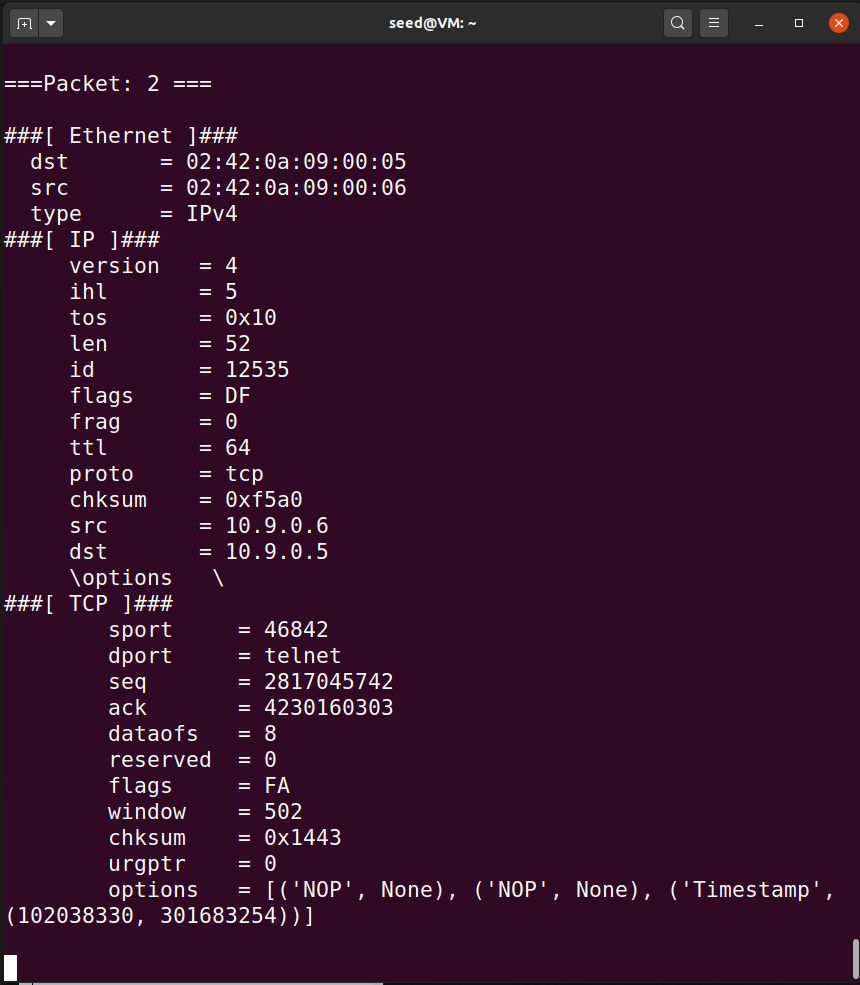


Figure : TCP Packet 2 Terminal Output

Our Terminal results show that we are in fact just sniffing only TCP packets from our source IP of 10.9.0.6 to our destination IP of 10.9.0.5.

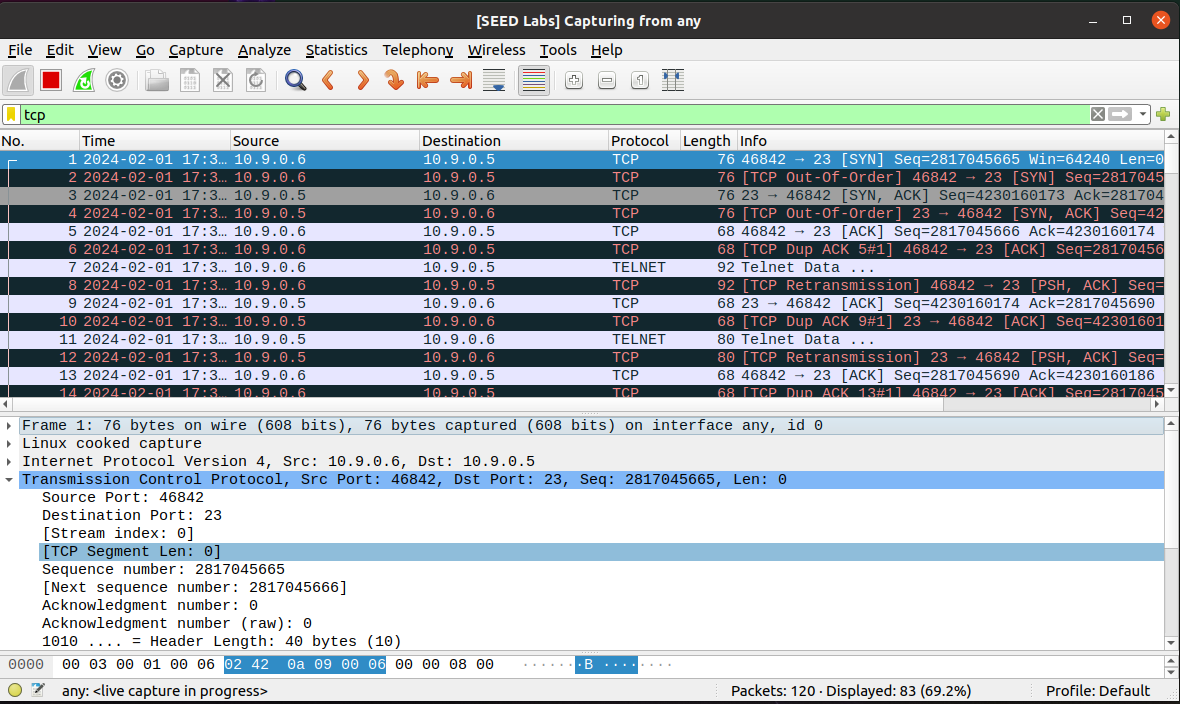


Figure : Wireshark TCP Only Filter

Here our Wireshark results filter our TCP packets and show that our source host (Host B) is trying to establish our 3-way-handshake. We also see that our destination port is also set to 23 which proves that we successfully completed our TCP filtering task.

1. **Capture packets coming from or going to a particular subnet. You can pick  
   any subnet, such as 128.230.0.0/16; you should not pick your VM subnet.**

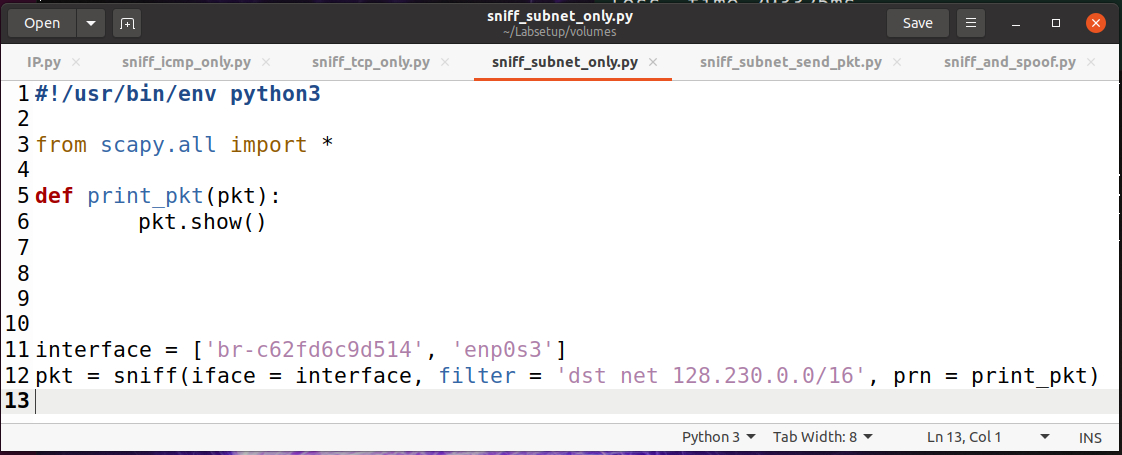


Figure : Sniffing Specific Subnet Python Program

Key Aspects of this code:

**Filter = ‘dst net 128.230.0.0/16’** – We are just using the filter given to us in our lab prompt to make things simple. We are filtering to a destination network IP of 128.230.0.0/16 only.

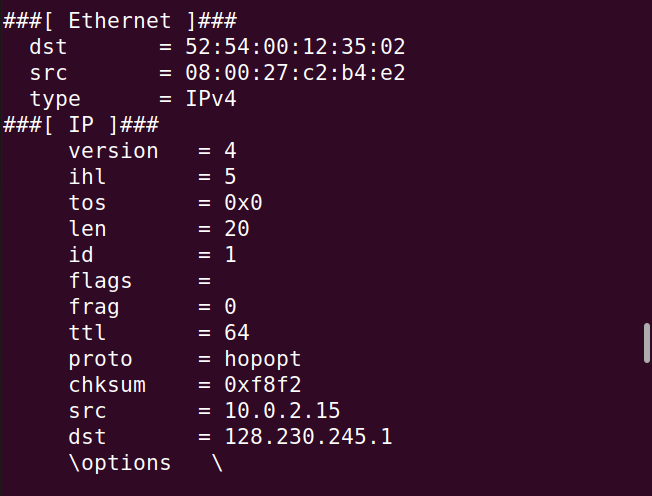


Figure : Sniff Subnet Terminal Output

Our terminal output above shows that we are specifically filtering the IP address of our destination IP address of **128.230**.245.1 which coincides with **128.230**.0.0 as our initial given filter value.

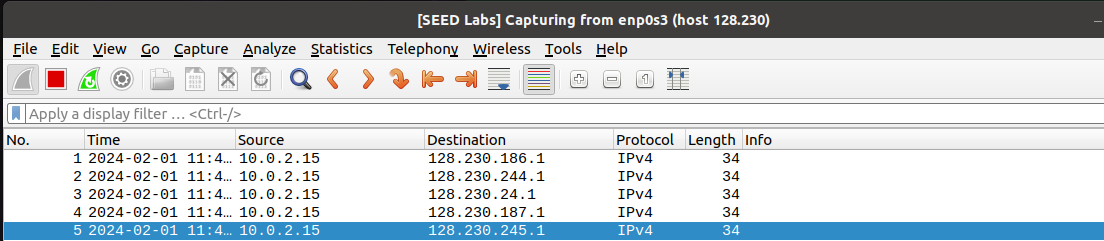


Figure : Sniffing Subnet Wireshark Output

For sending a packet to our subnet, this is the following code output:

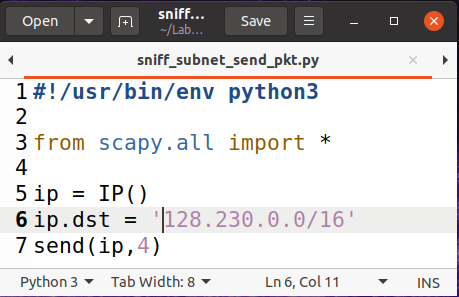


Figure : Send Packet to Subnet Program

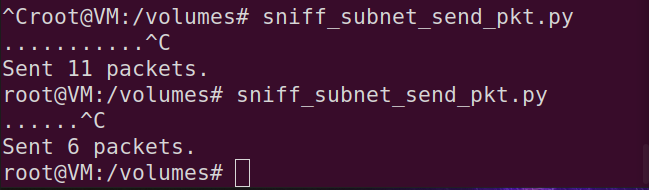


Figure : Send Packet to Subnet Terminal Output

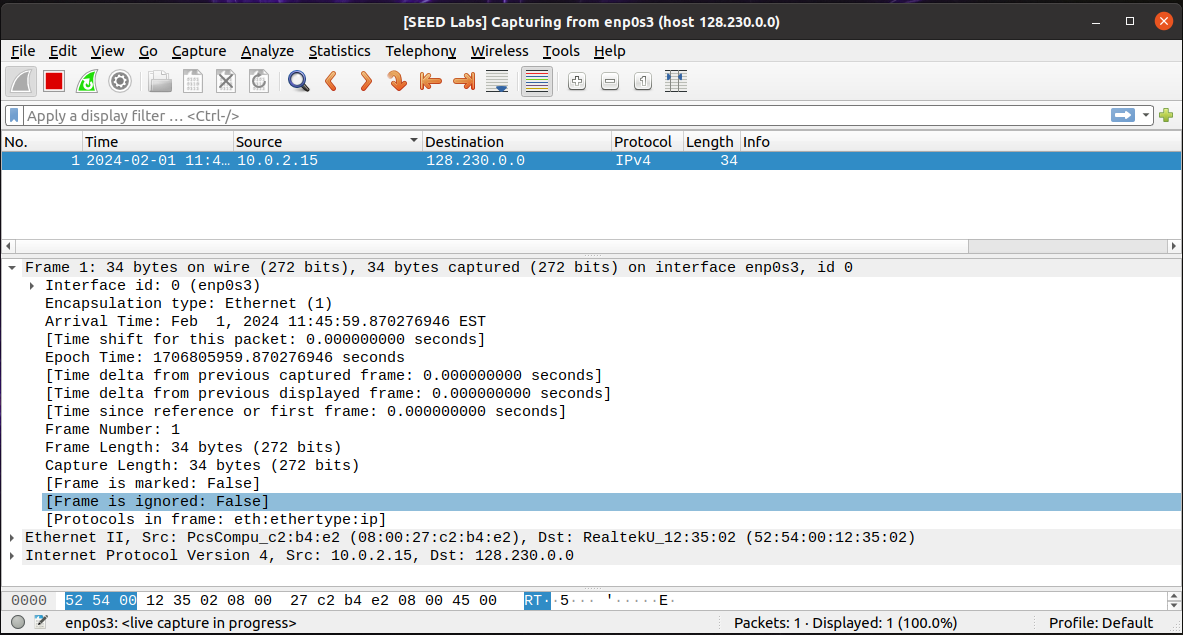


Figure : Send Packet to Subnet Wireshark Output

Our Wireshark outputs show that we sniffed and sent packets to and from our subnet destination of 128.230.0.0 which proves that we successfully completed this portion of our task.

1. **Task 1.2 – Spoofing ICMP Packets**

For this task, we are going to implement the concept of “**spoofing**,” which is where a user disguises themselves as a trusted source to gain valuable information from the victim or to cause harm.

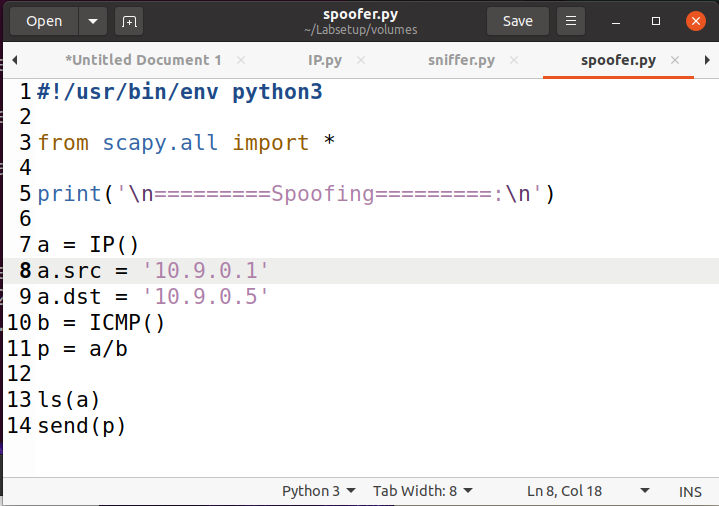


Figure : Spoofing ICMP Packets Program

**a = IP()** – Creates an IP object from the IP class found within scapy.

**a.src = ’10.9.0.1’** – Sets our source IP field.

**a.dst = ’10.9.0.5’** - Sets our destination IP field.

**b = ICMP()** – Creates an ICMP() object from the ICMP class found within scapy.

**p = a/b** - Add ‘b’ as the payload field of ‘a’ and modify the fields of ‘a’ which gives us an ICMP packet.

**ls(a)** – Lists all attribute names and values.

**Send(p)** – Sends our packet.

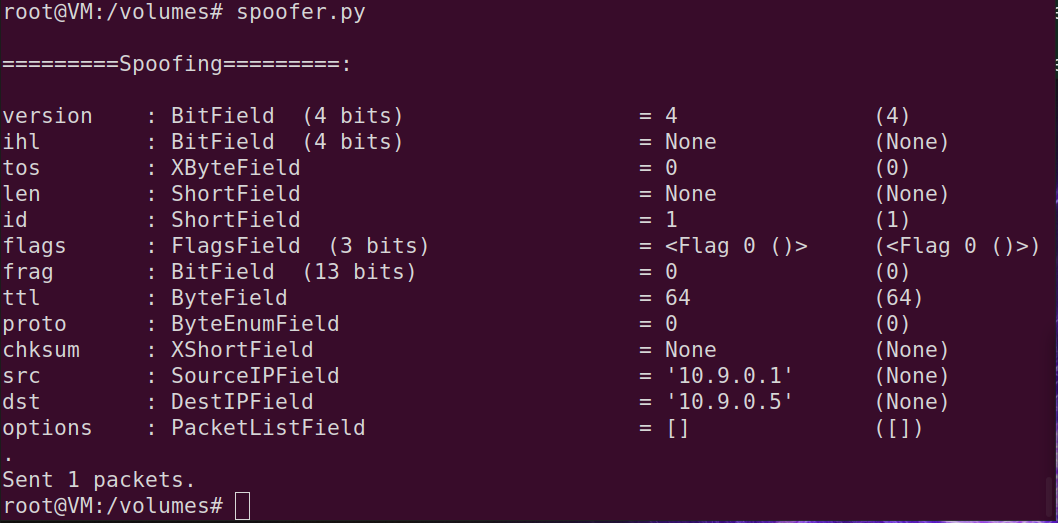


Figure : Spoofing ICMP Packets Terminal Output

We can see that our “SourceIPField” and our “DestIPField” correlate with our specified destinations within our Python program. It also provided us with a “sent 1 packet” to let us know that our program worked.

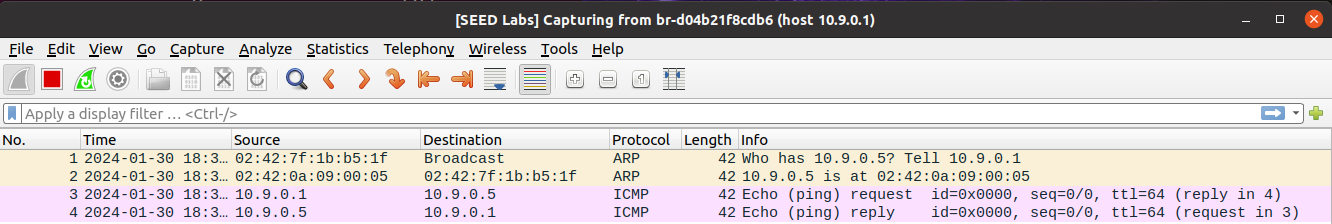


Figure : Spoofing ICMP Packets Wireshark Output

To further prove this, our Wireshark output shows that our echo request and reply went through and that we successfully spoofed an ICMP packet.

1. **Task 1.3 – Traceroute**

The idea behind **traceroute** is to gather an estimated distance between our Virtual Machine (VM) and our specified IP Destination. This tool estimates this by the number of routers it jumps between before it reaches our destination.

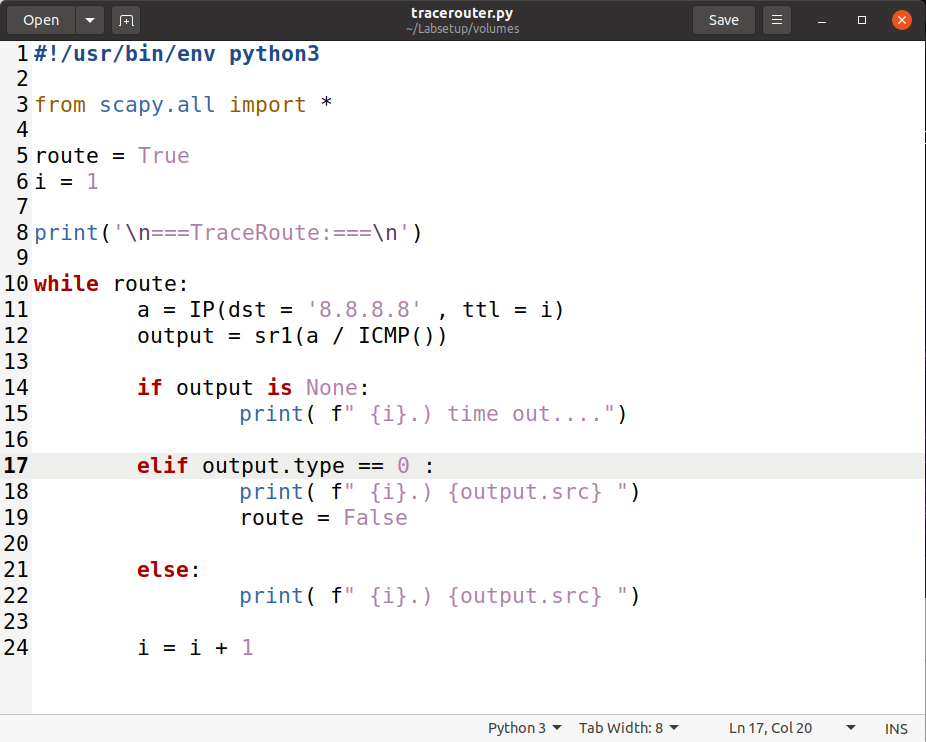


Figure : Traceroute Python Program

The idea behind this program is creating a Boolean value to keep our while-loop in limbo. If we do not receive any output, we print out a timeout message. If our output type is “0”, our Boolean value turns false and prints our final output which ends our while-loop. Our TTL (Time-To-Live) field increments by one as well to run the program to achieve a result. Our else block catches everything else to print to console.

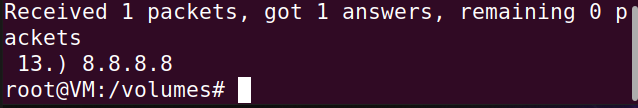


Figure : 13 "Hops"

Our python program took us “13” steps to get to our destination. We must keep in mind that this is nothing more than an estimate.

A screenshot of a computer

Description automatically generated

Figure : Traceroute Console Command

Our terminal traceroute command on our specified IP address shows that it took 30 Hopes max to get to its destination.

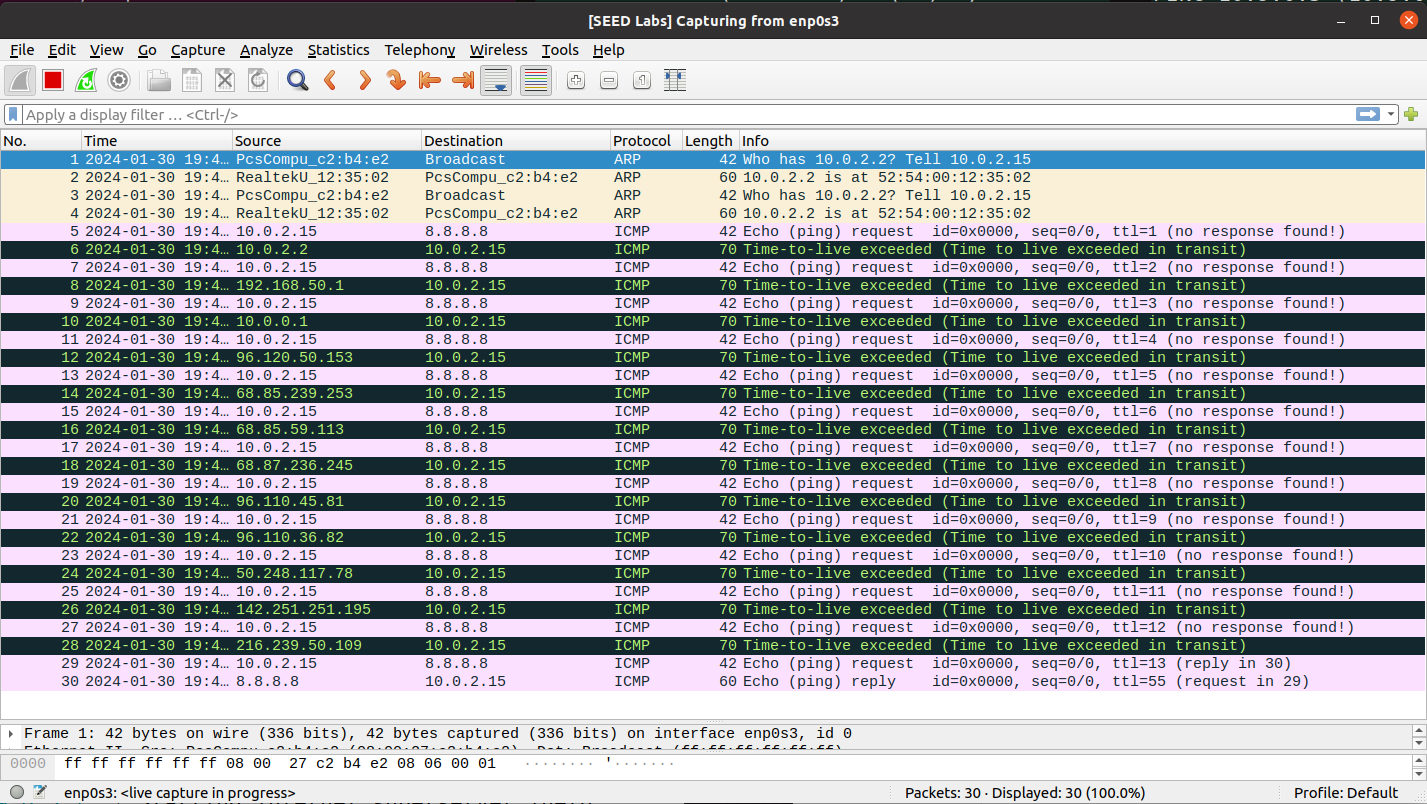


Figure : Traceroute Wireshark Output

From our Wireshark output, we can see that we did get to a hop of thirteen from our written python program and that our max hop went to thirty. We can also see that our TTL = 13, which coincides with the number of hops it took in our program, so there is a good chance that this proves that our traceroute program works.

1. **Task 1.4 – Sniffing & Spoofing (Extra Credit)**

For this task, we are going to ping three IP addresses from the user container. Report your observation and explain the results.

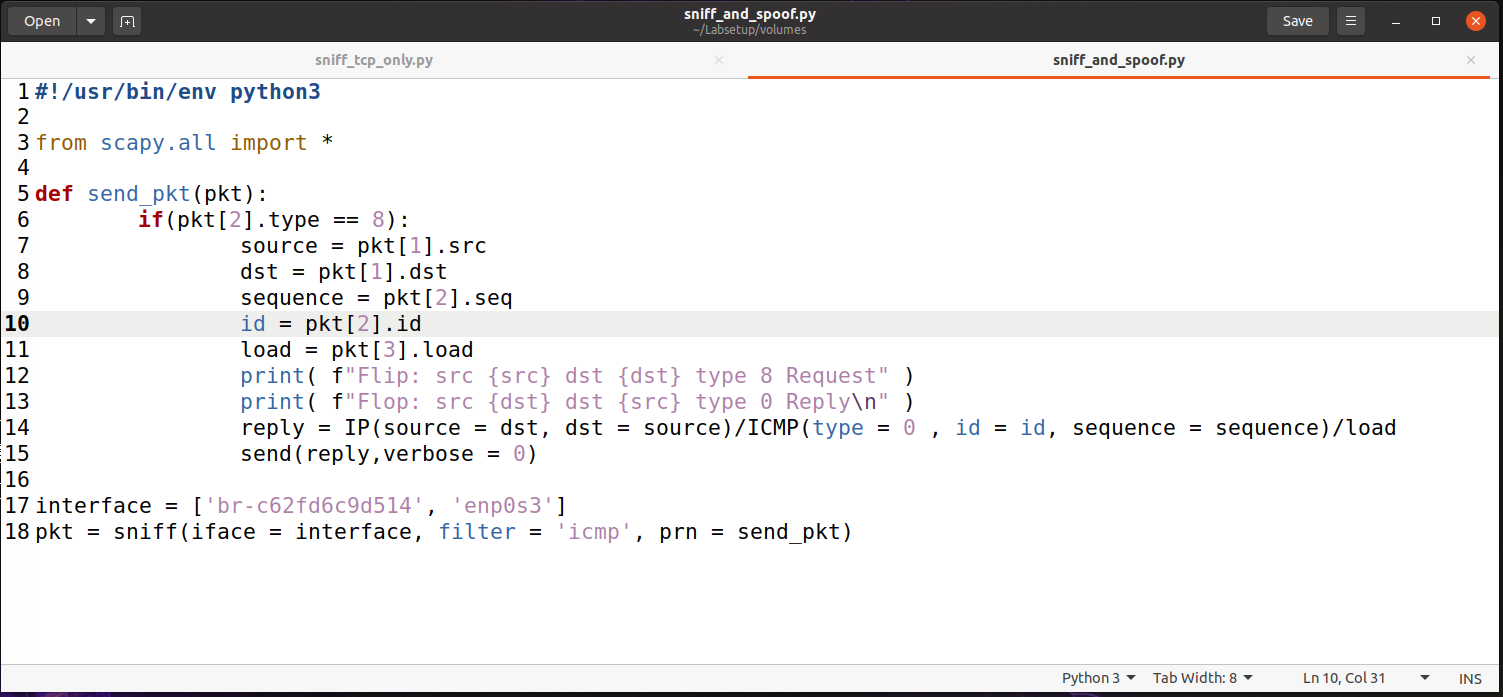
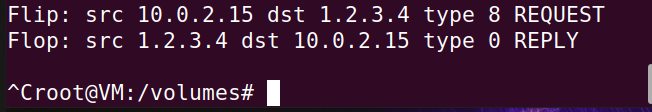
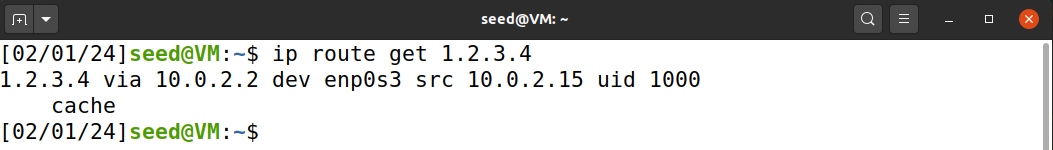


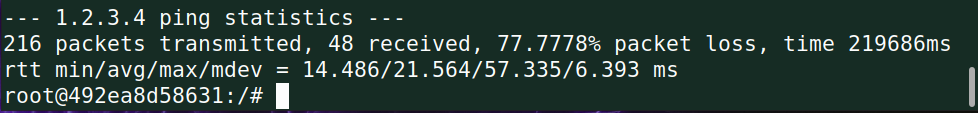
Figure : Sniff & Spoof Python Program

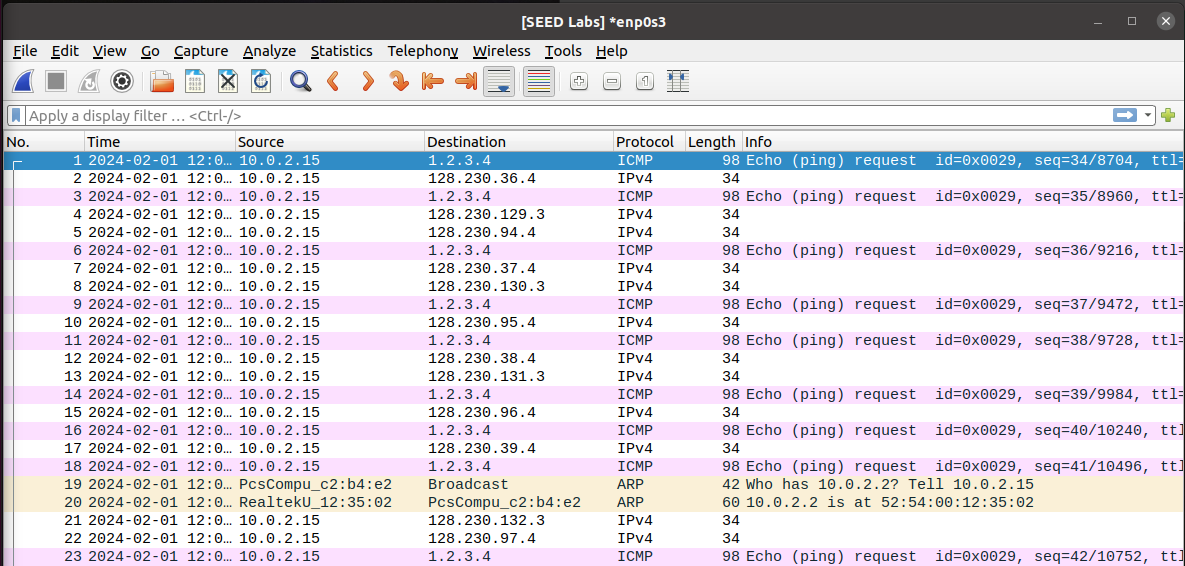
Our if statement looks to see if our ICMP is a request (== 8). If it turns out to be a request, our program should send an echo reply using the techniques of sniffing and spoofing learned throughout the duration of this lab. Pkt[3].load in our program stores our original data so that we have a proper message sent back to the sender.

1. **IP - 1.2.3.4 [non-existing host on the Internet]**



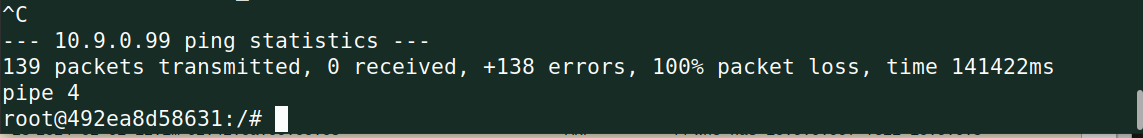


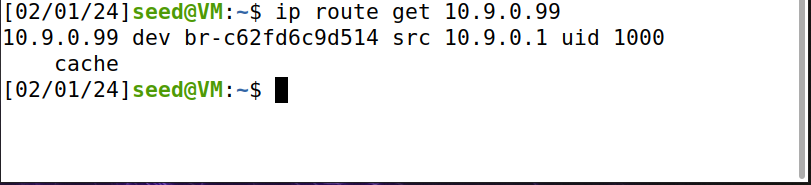


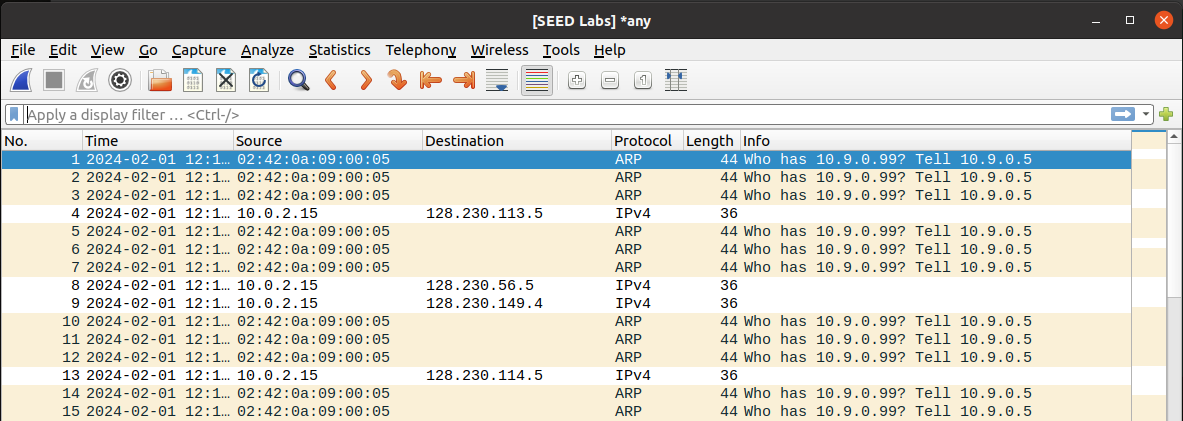


Without our python program, we would have a one-hundred percent packet loss because the IP address does not exist, which shows in our Wireshark output in our broadcast response.

1. **IP – 10.9.0.99 [non-existing host on the Lan]**

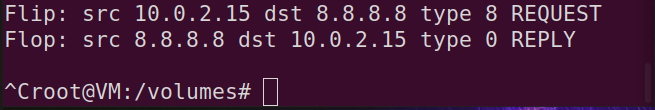


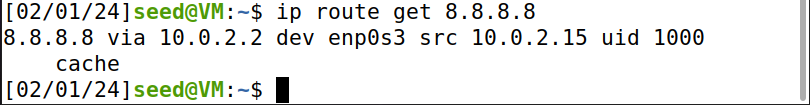


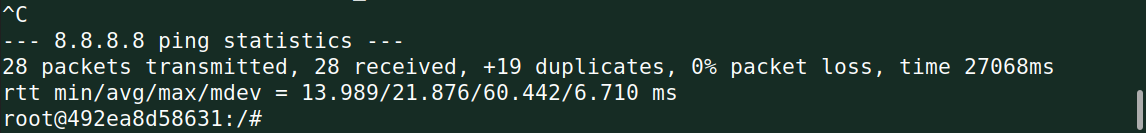


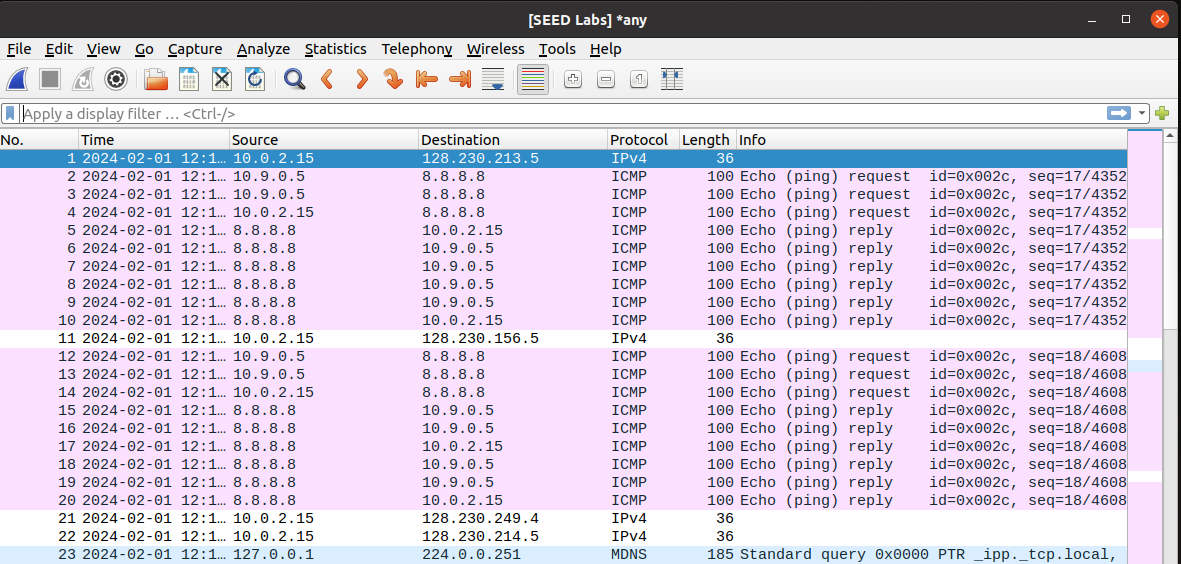
This situation is the same as the one above where our IP address does not exist, which from my testing gave me 100% packet loss even when running my python program. So maybe that is because we are dealing with a LAN connection and that may be different than pinging a non-existing host on the Internet.

1. **IP – 8.8.8.8 [existing host on the Internet]**









Because this IP address exists on the internet, we can notice from our Wireshark output that we are getting duplicate responses with both replies and requests. From our ping statistics, we had +19 duplicates with our testing.

1. **Conclusion**

This “Sniffing & Spoofing” lab taught me two important concepts. One in patience with trial and error and another in gaining more knowledge about the concepts of sniffing and spoofing. I have never done a project like this before where everything you are doing is all within the command line, so the feelings of joy that I felt figuring this out left me feeling inspired and wanting to learn more about these kinds of topics. These tools are valuable for anyone pursuing a degree in technology so that you can detect these types of attacks on your own home network in the future.