TCP/IP Attack: Lab 2

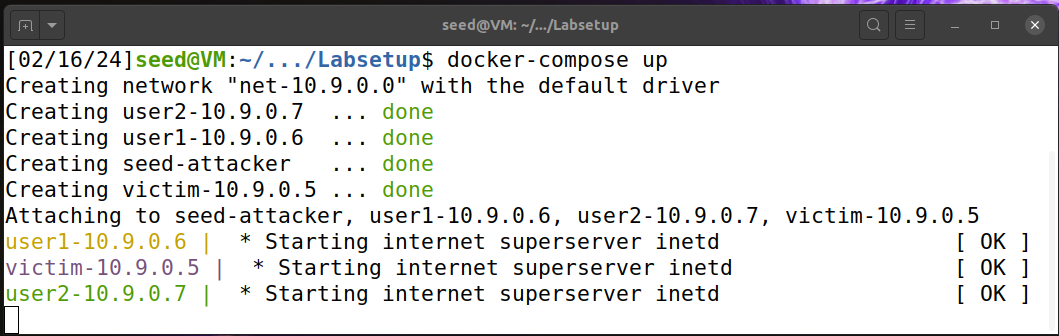
Abstract: This Lab focus primarily on TCP(Transmission Control Protocol) and the vulnerabilities that can be exploited through our Virtual Machine. Python and C Programs are used to achieve the following tasks: [TCP SYN Flood Attack, Reset Attack, Session Hijacking, and Reverse Shell]

Nash William Morrison

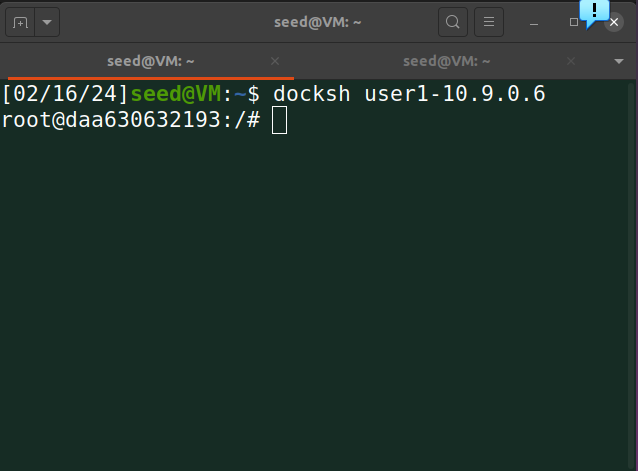
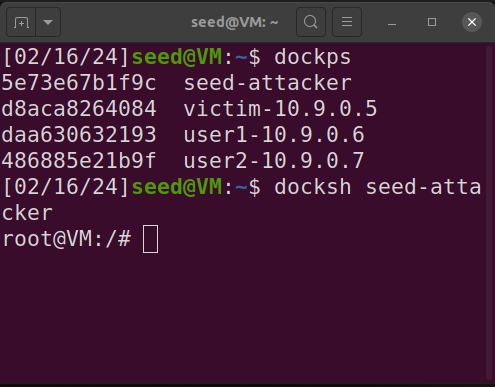
**Lab Environment**

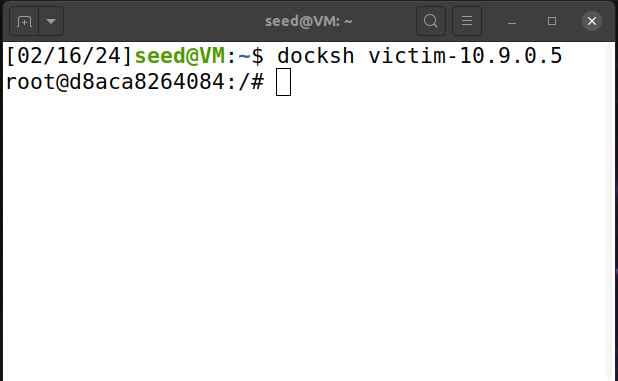
Our lab environment is going to follow the same guidelines that I set up for myself within our first lab. Our ATTACKER (10.9.0.1) terminal colored “Purple,” our USER/HOST (10.9.0.6/7) terminals will be “Green”, while the VICTIM terminal will be a normal “white” color.

Making sure our dockers are up and running with our **“docker-compose build”** command:



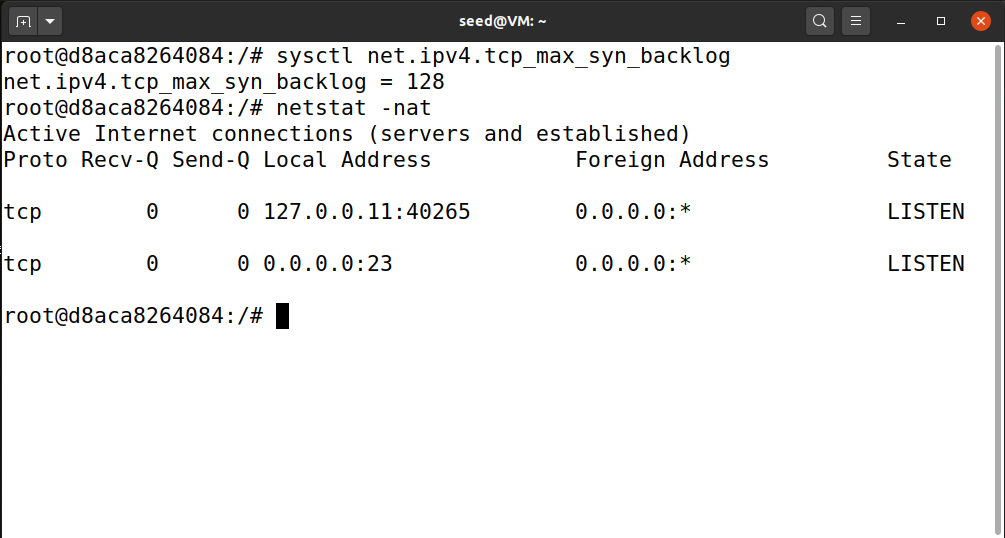
This is what our terminals look like once they are in their own roots using our **“docksh”** command:





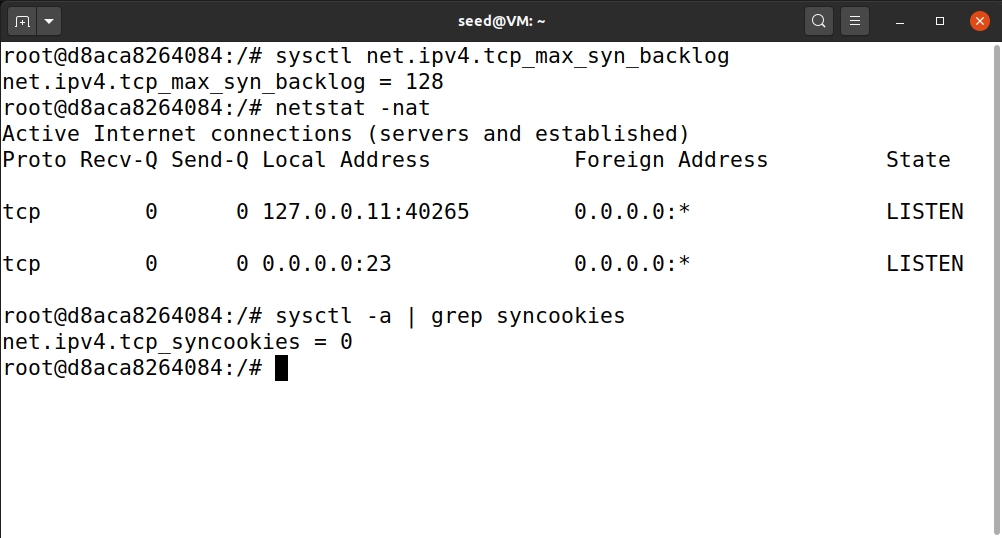
**Task 1 : Syn Flooding Attack**

The overall idea behind this attack is remarkably simple. A TCP connection requires the establishment of a 3-way-handshake which results in a SYN//SYN-ACK//SYN style of connection. What we are doing is flooding our VICTIM machine with a whole bunch of half-sent requests. What ends up happening is that if another computer tries to make a connection with the VICTIM, it will struggle to make that connection since it will flood with connections that cannot finish. To evaluate what our backlog number is for our VICTIM machine, we use the following command: **“sysctl net.ipv4.tcp\_max\_syn\_backlog”** which gives us the following result:



Our VICTIM queue allows for a backlog of 128 open slots that are available to receive TCP connections. We are going to change this number throughout this task to see if our Python code has a better chance of flooding the VICTIM connection. To also have a better chance of success with our code, let us first check to see if “SYN Cookie” protection is off with the command: **“sysctl -w net.ipv4.tcp\_syncookies=0”** which turns off the protection mechanism. Changing the value from a “0” to a “1” turns the cookie protection back on.

**Task 1 : SYN Cookie Countermeasure (Turned Off)**

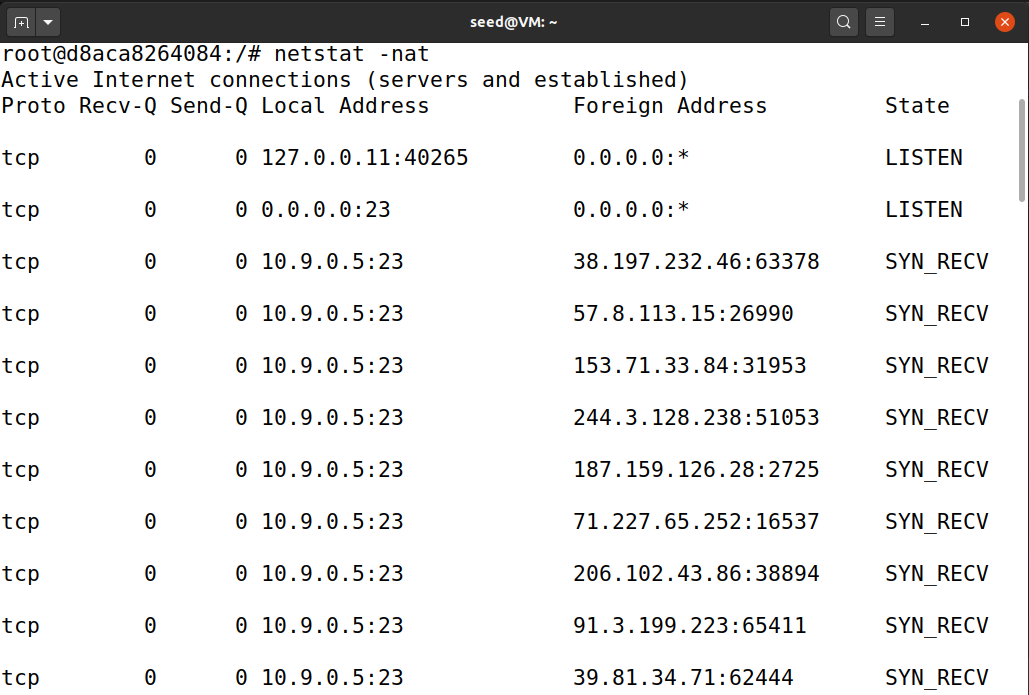
****

Here we are just highlighting that before we start our task, the cookie protection is off.

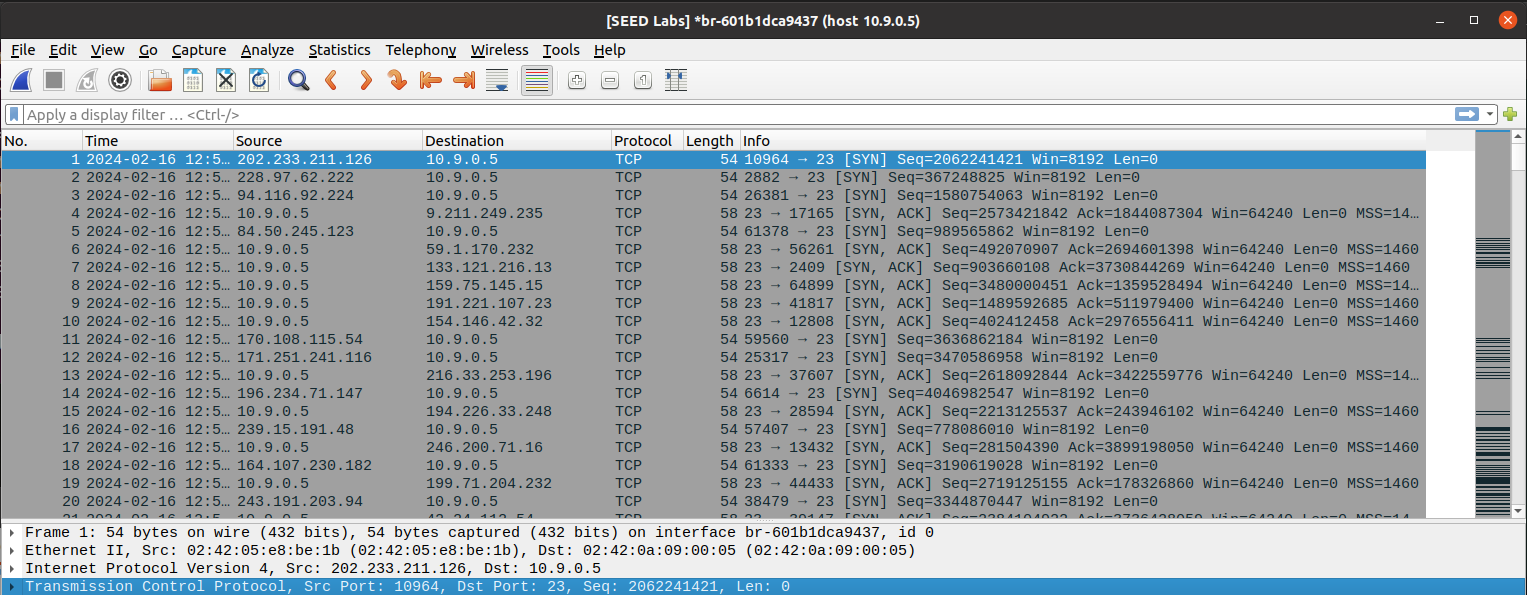
**Task 1.1: Launching Attack With “Python” Code:**

Here we are generating the ability to manipulate packets with our “Scapy” program within Python. We are setting our destination IP to our VICTIM, our destination port to “23” which I believe is the port for Telnet. While true establishes an endless loop to spam the VICTIM with half-open requests. Our Iface value, found with the **“Ifconfig”** command so that we can get really exact with our requests.

**Task 1.1: Launching Attack With “Python” Code (VICTIM Terminal):**

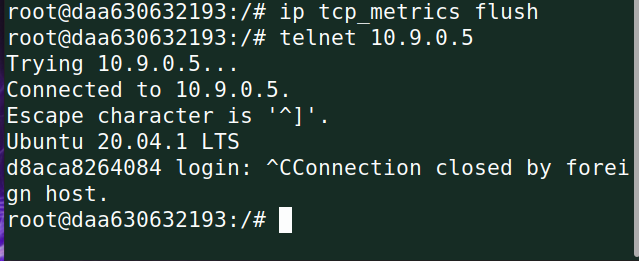
****

This is to show that our VICTIM is in fact flooded with SYN\_RECV requests from random IP addresses.

**Task 1.1: Launching Attack With “Python” Code (Wireshark Output):**

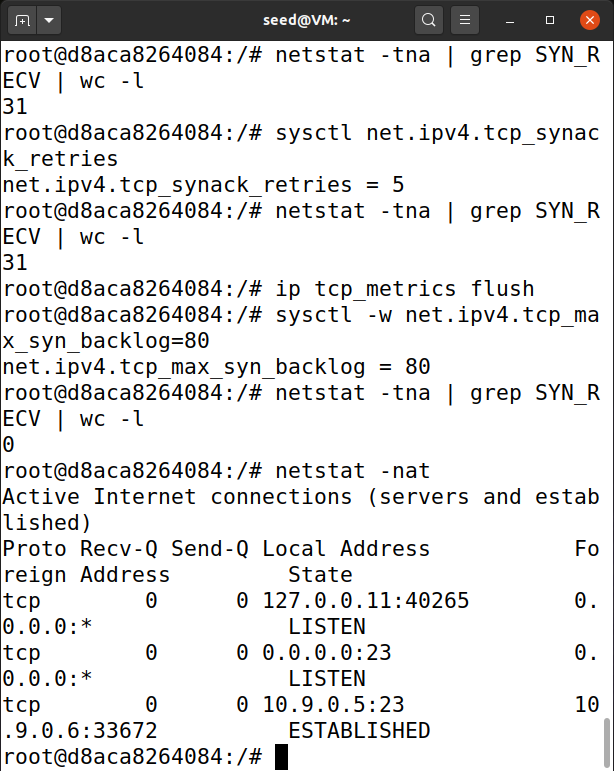
This is further confirmation from Wireshark that our python program is in fact flooding our VICTIM with a bunch of half-open requests.

**Task 1.1: Launching Attack With “Python” Code (HOST Telnet Connection):**

****

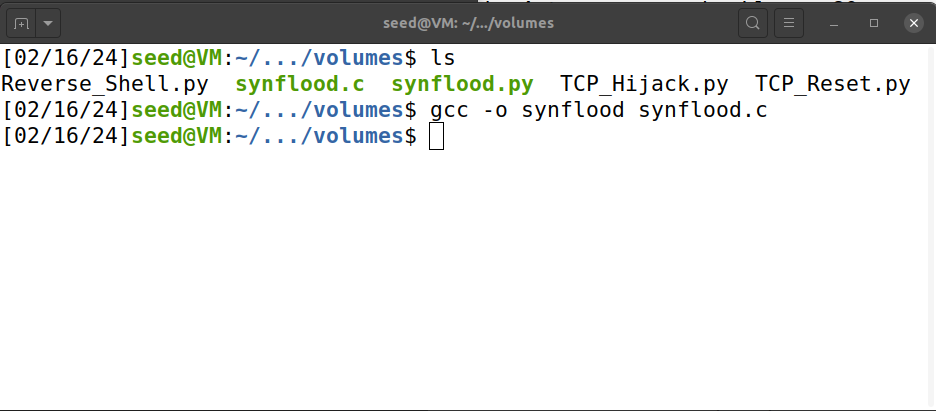
However, this python program is not fast enough, so we are able to still establish a connection with Telnet from our HOST to our VICTIM.

**Task 1.2: Launching Attack With “C” Code (Resetting VICTIM):**

****

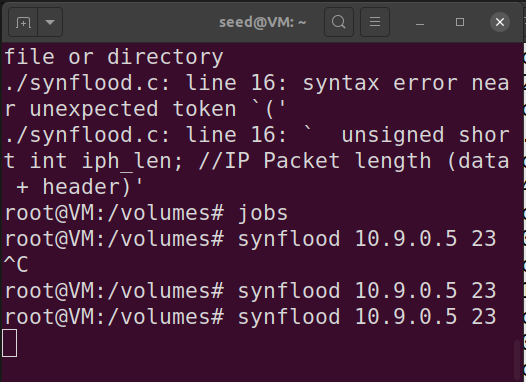
We use the following command: **“ip tcp\_metrics flush”** to flush the cache of memory found within our VICTIM terminal. This shows that we did in fact get rid of all those half-open requests. However, with enough time, the requests would have eventually timed out.

**Task 1.2: Launching Attack With “C” Code (Setting the Code):**

****

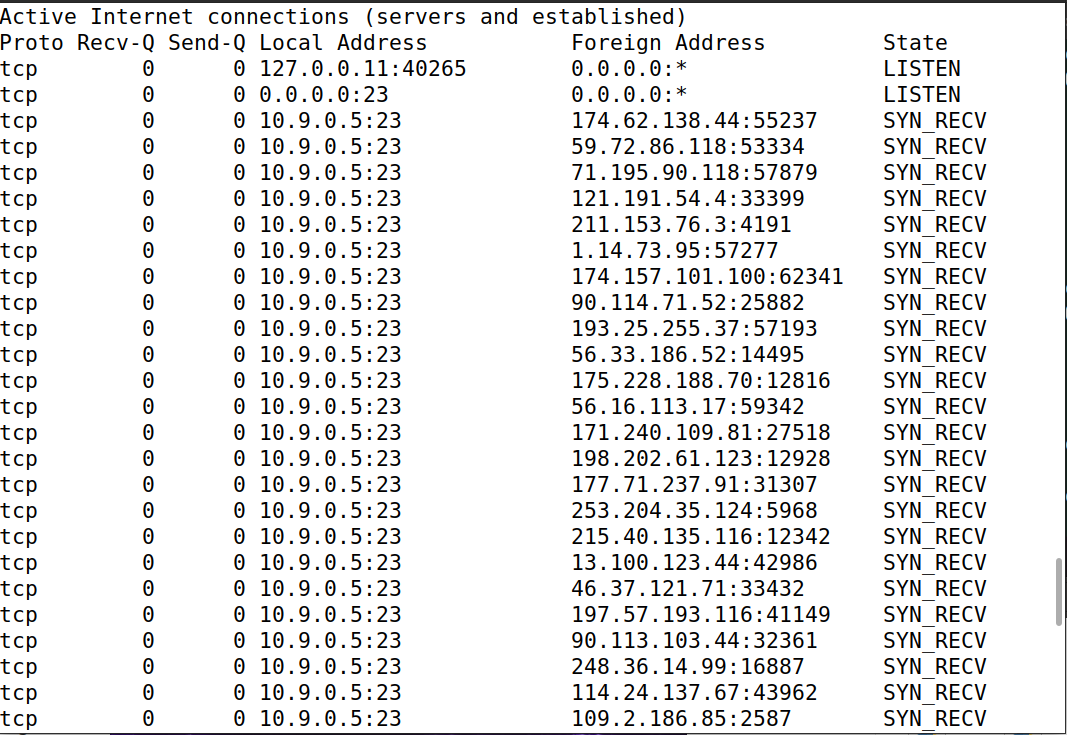
The C program, provided already to us in our lab, so I will not be providing the code snippet. However, what the code is doing is what the Python code did, but at a much faster rate to the point where it makes it incredibly hard to establish a connection through Telnet from our HOST to our VICTIM. The image above shows how to load the program to become executable.

**Task 1.2: Launching Attack With “C” Code (ATTACKER Terminal):**

****

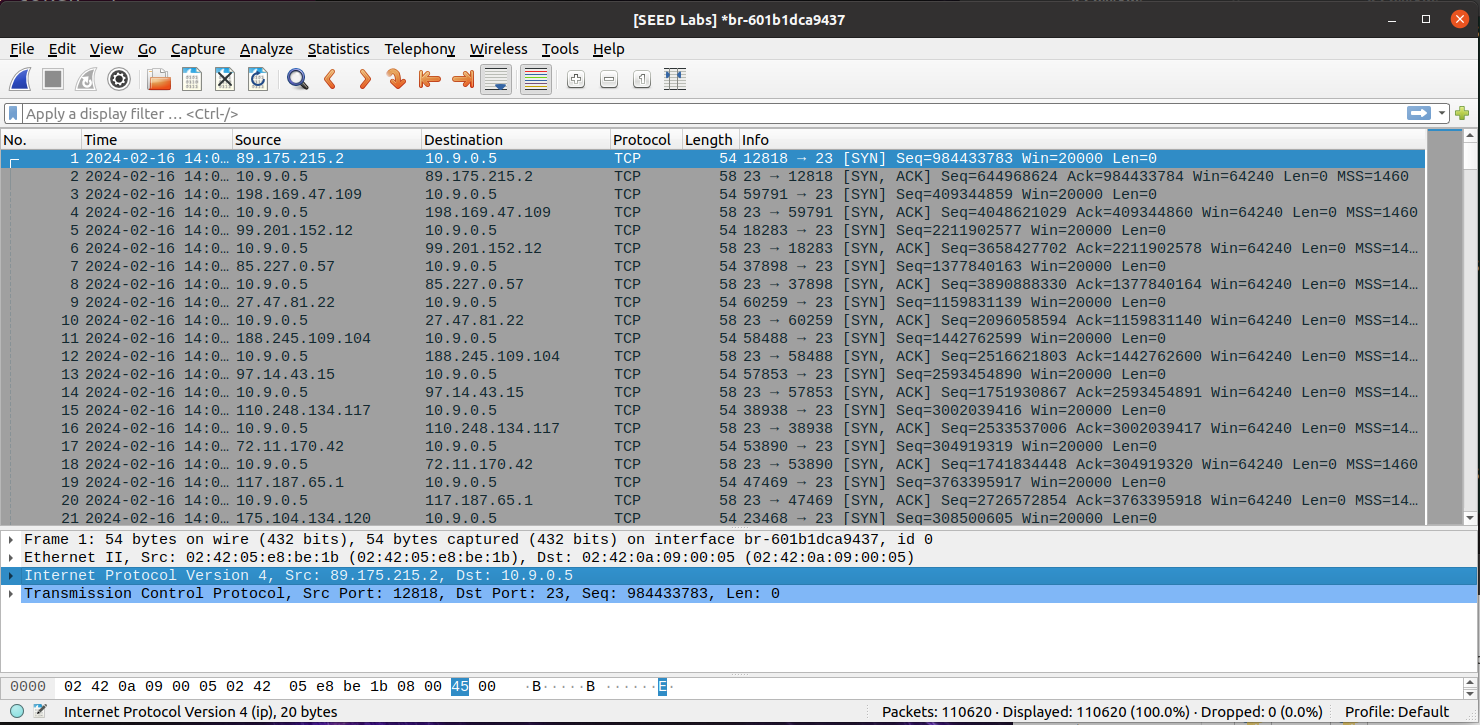
With our code executable, we type in the following: **“synflood 10.9.0.5 23”**, this starts up our synflood C program to the VICTIM IP address of 10.9.0.5 and with a destination Telnet port of twenty-three.

**Task 1.2: Launching Attack With “C” Code (VICTIM Terminal):**

****

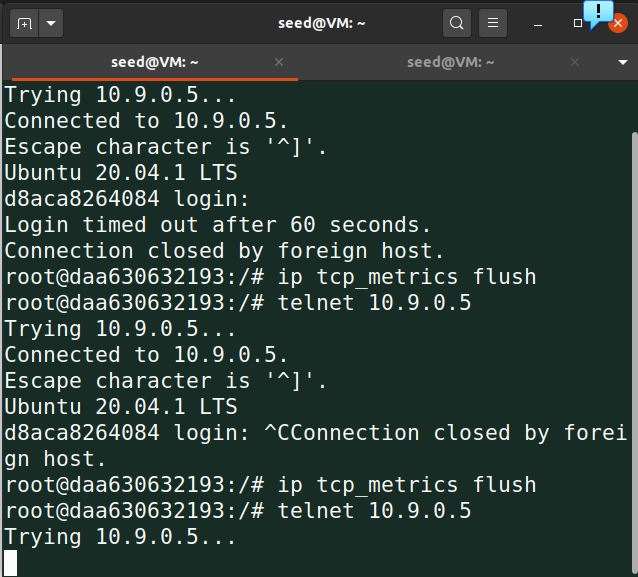
This small snippet does not do the program justice. This program flooded the VICTIM so much that it crashed my Virtual Machine. I had to initialize a backup save state and start back up the lab from this point forward, so my iface value will have changed as a result.

**Task 1.2: Launching Attack With “C” Code (Wireshark Output):**

****

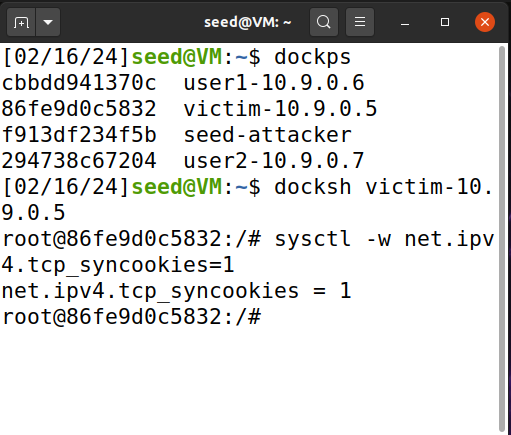
I did not want my VM to crash again, so I was quick with these images. I did scroll down to see that in under 10 seconds it had sent over 300,000 half-open requests which blew my mind. The fact that C is so much more optimal to conduct these attacks than Python is not something I thought I would have seen.

**Task 1.2: Launching Attack With “C” Code (HOST Telnet Connection Failed):**

****

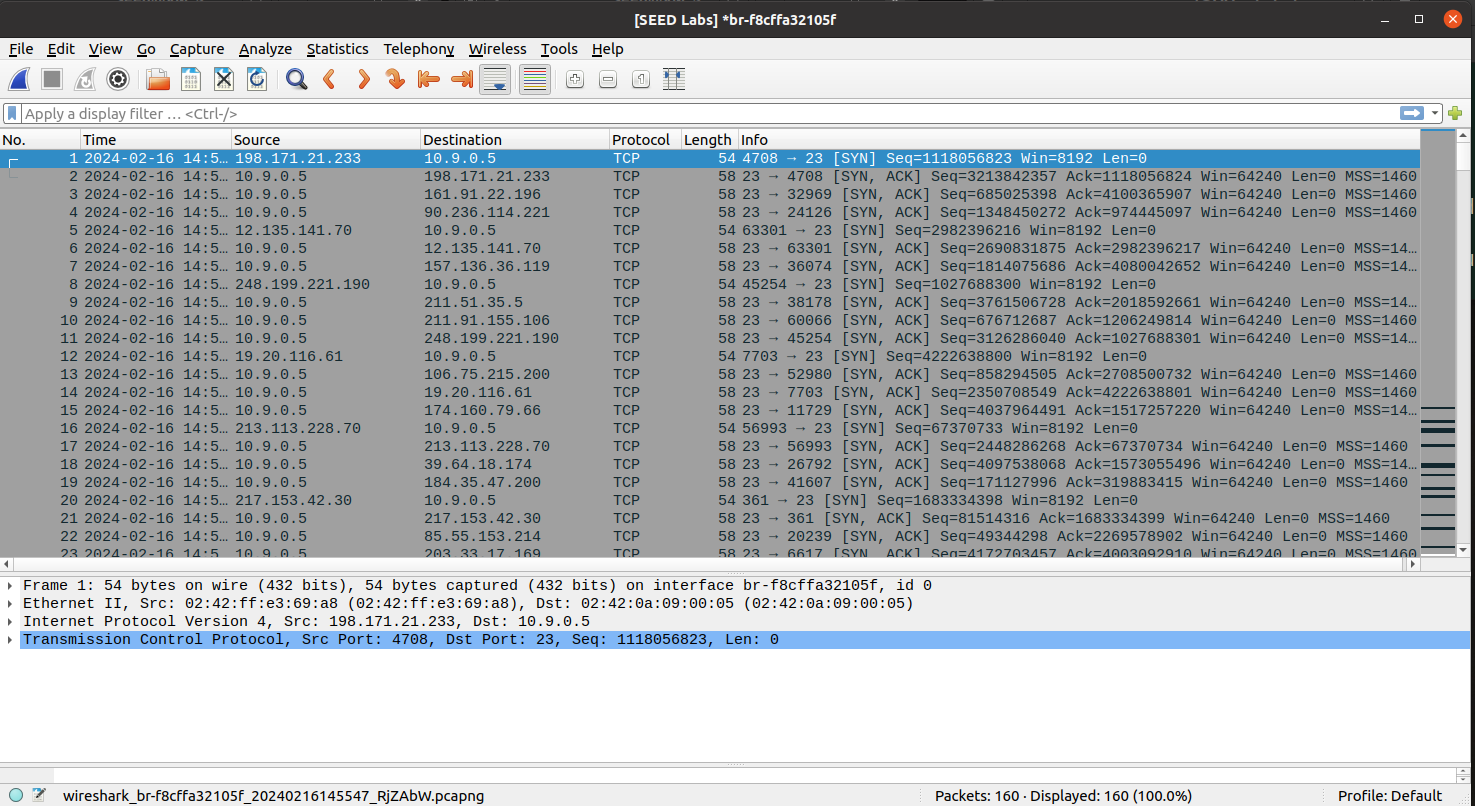
I kept in the initial attempts at establishing a connection with Telnet to show the comparison when it cannot establish a connection. It will continue to sit there in limbo, trust me I waited for a bit to see which crashed my system.

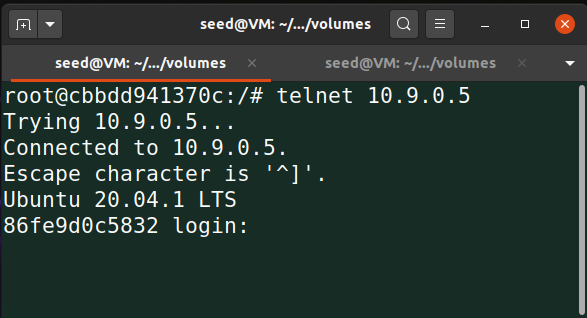
**Task 1.3: Syn Cookie Countermeasure (Turned On):**

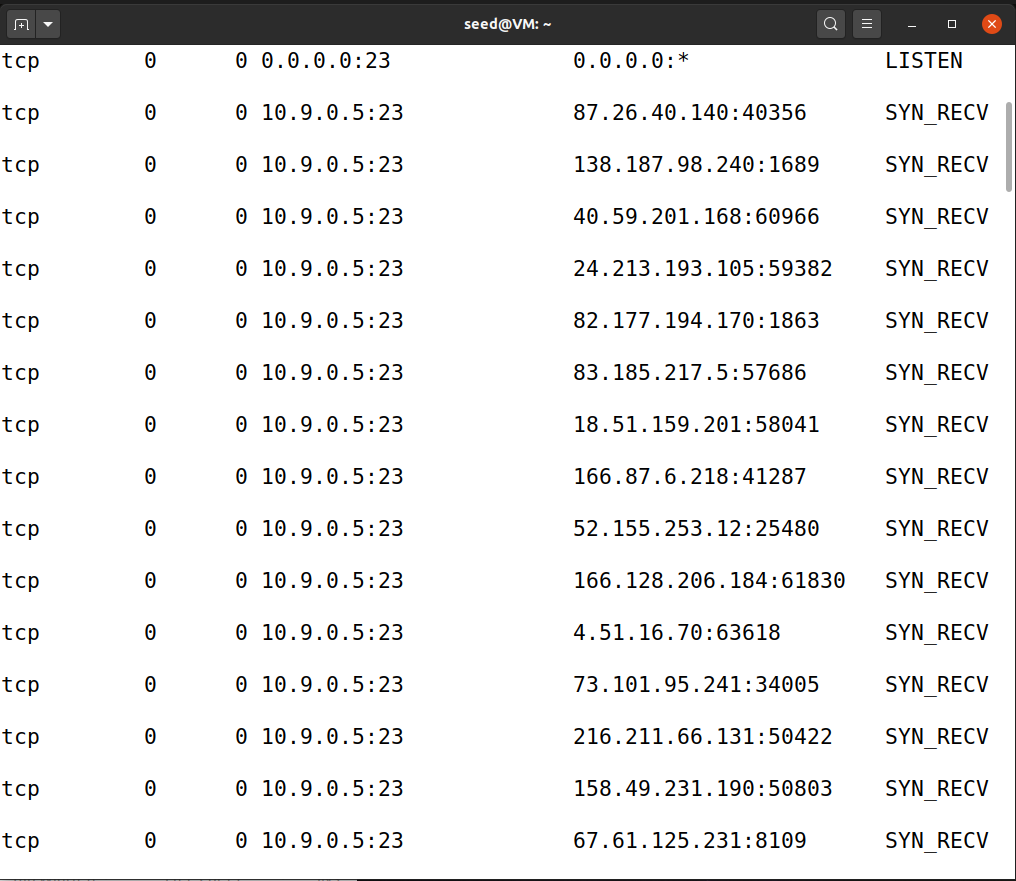
****

**Task 1.1: Launching Attack With “Python” Code:**

****

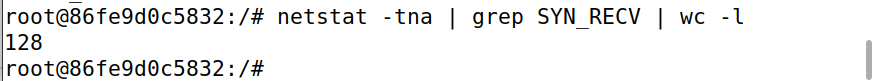
****

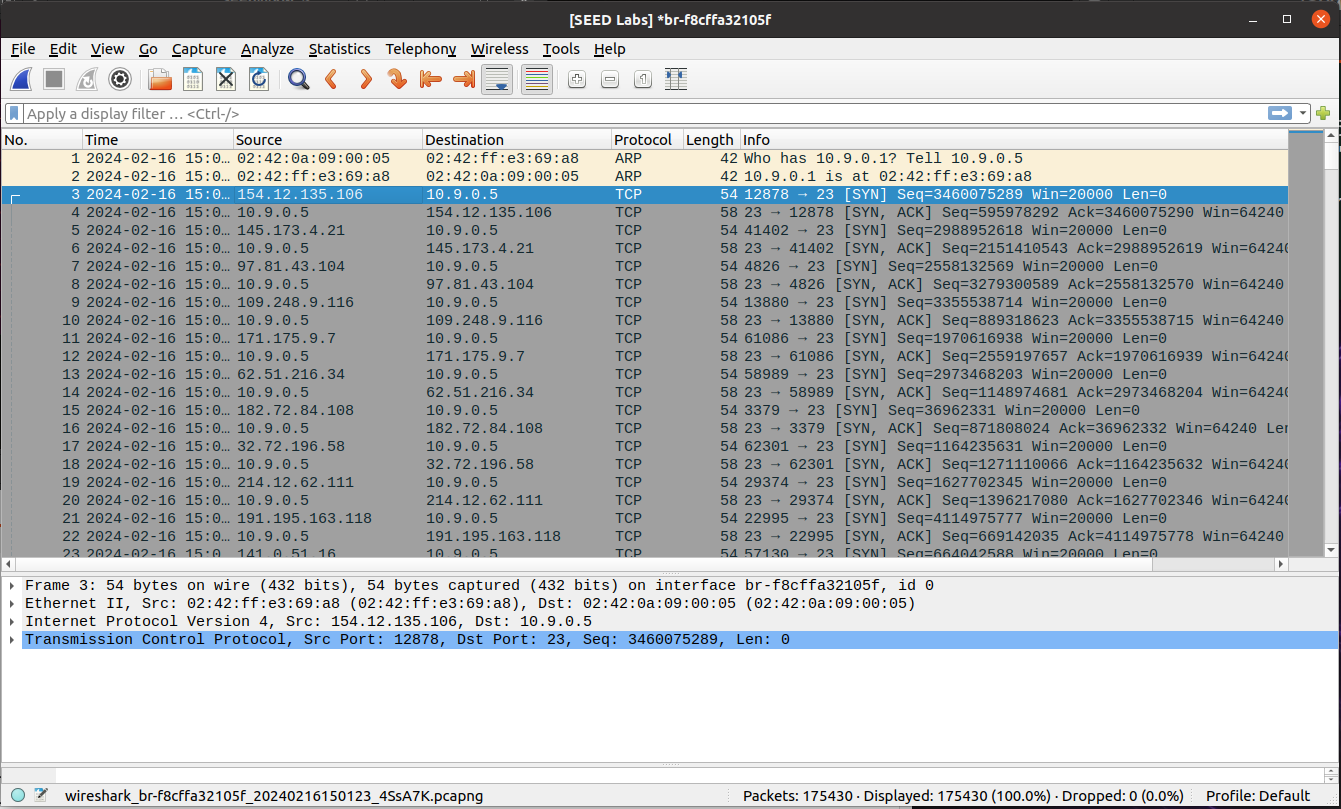
****

****

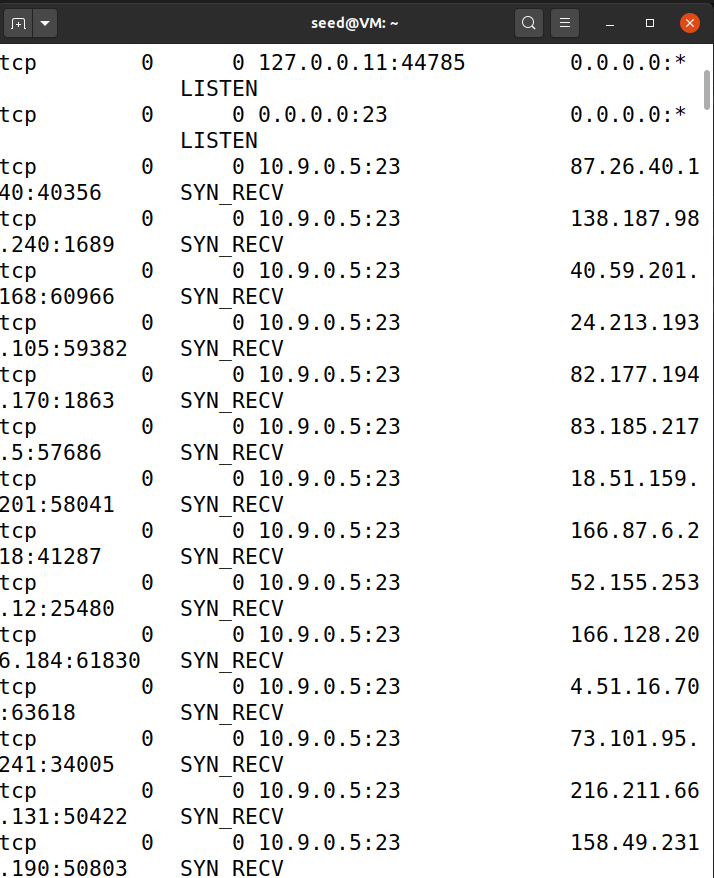
We are still able to conduct the attacks as usual, however it seems as though no matter what happens, we can still always establish a connection between HOST and VICTIM through Telnet.

**Task 1.2: Launching Attack With “C” Code:**

****

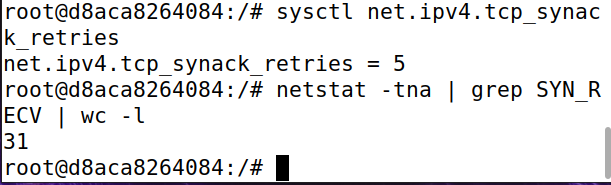
****

****

****

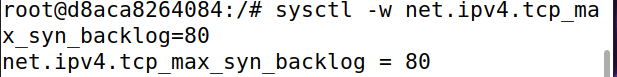
Even with our C program, with our Cookie countermeasure turned on it can still establish a connection between VICTIM and Host through Telnet.

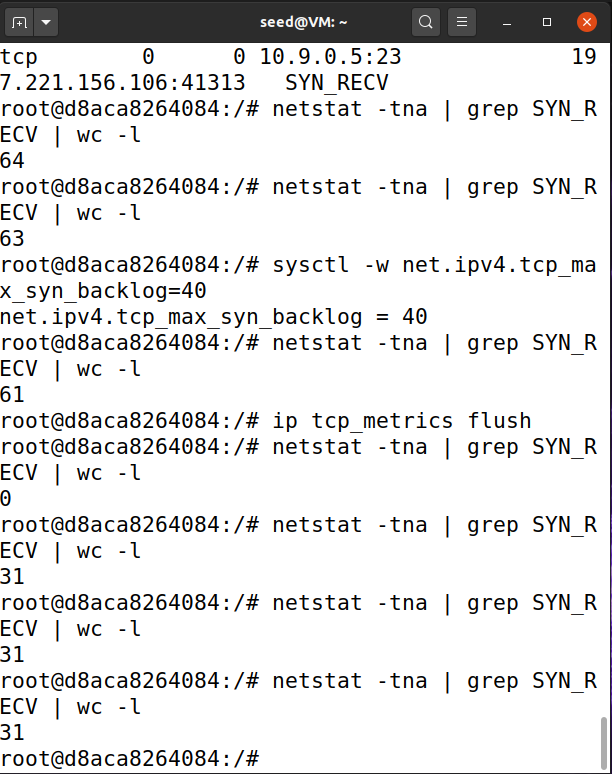
**Task 1: TCP Retransmission Issue:**

****

I wanted to highlight the issues surrounding this lab. I decided to give the retransmission issue a try by opening a bunch of ATTACKER shells to try and overload the VICTIM machine. I opened three extra terminals on top of my original ATTACKER, and it still had no effect on stopping connections.

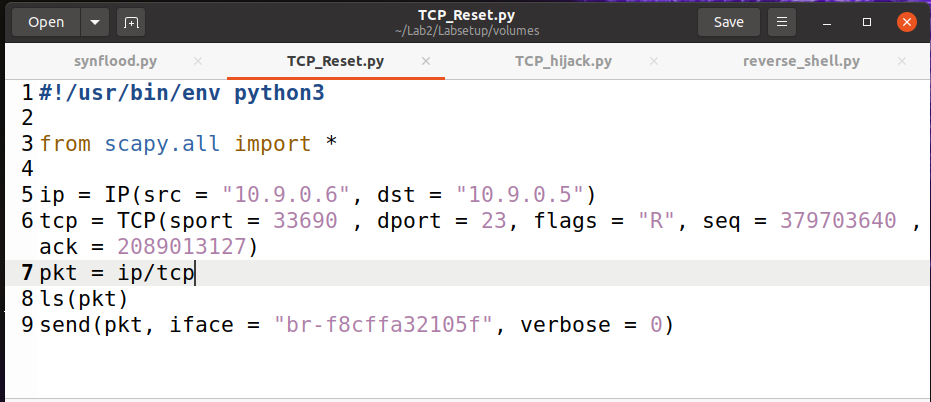
**Task 1: The Size of The Queue Issue:**

****

****

I tried to change the size of the queue two separate times, one at 80 and one at 40 shown above. It still did not have any impact on successfully attacking the TCP connection.

**Task 2: TCP Reset Attacks on Telnet Connection (Python Code):**

****

For this code we were able to find out our unknowns through our Wireshark app. We needed to locate the last packet sent between the HOST and VICTIM, specifically looking for the source port, sequence number, and ack number.

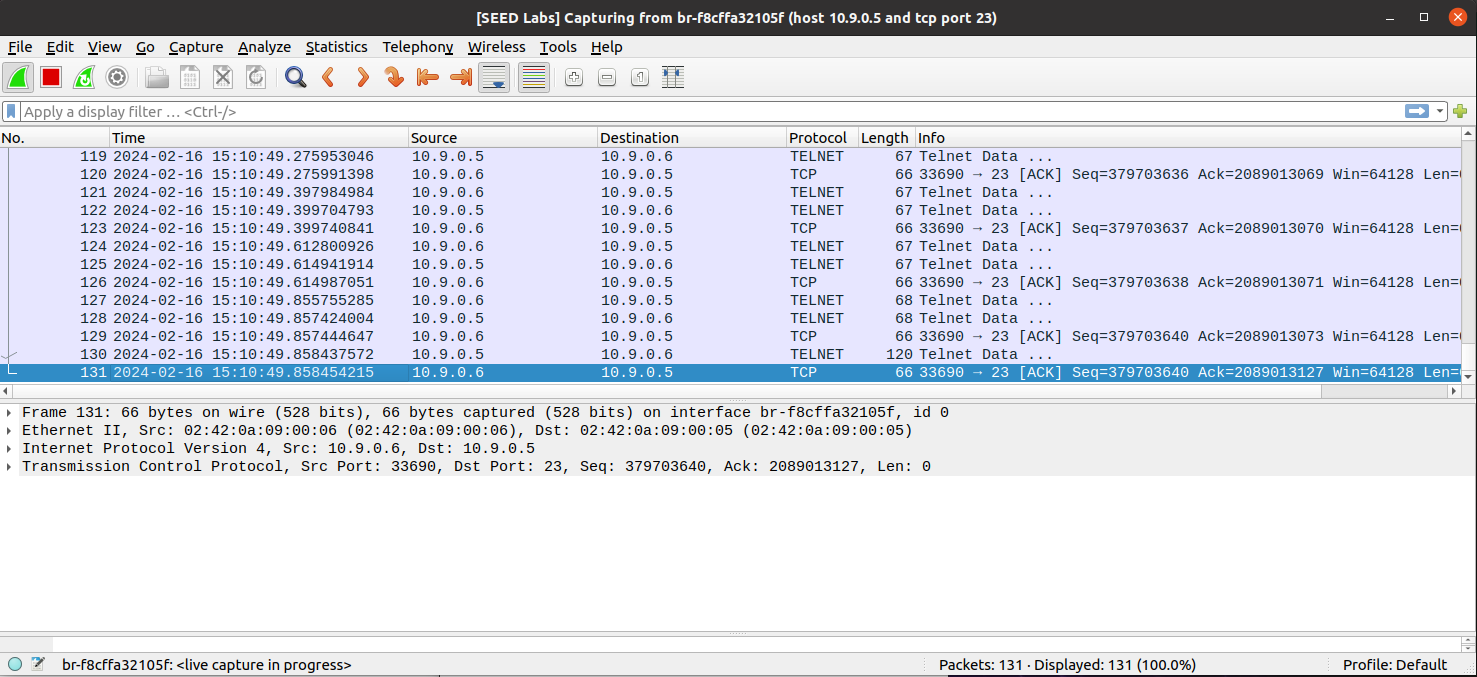
**Task 2: TCP Reset Attacks on Telnet Connection (Established Connection):**

**A screenshot of a computer

Description automatically generated**

This output shows that we have a healthy connection between VICTIM (10.9.0.5) and HOST (10.9.0.6) so that we can create the reset attack.

**Task 2: TCP Reset Attacks on Telnet Connection (Needed SEQ Packet - Wireshark):**

****

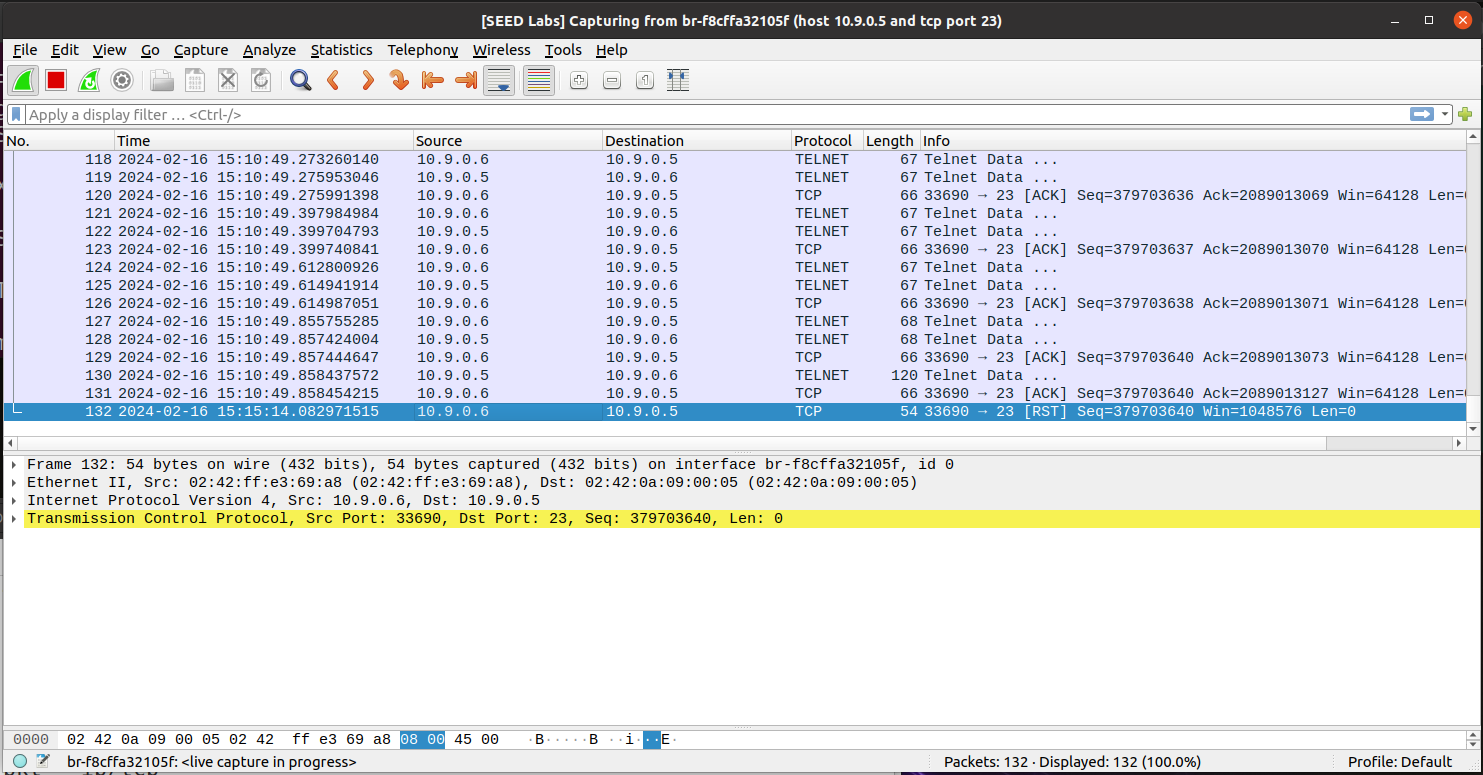
We send a random message from our HOST to our VICTIM so that packets send. We take this sent packet and gather our important data to input into our code.

**Task 2: TCP Reset Attacks on Telnet Connection (ATTACKER Terminal):**

**A screenshot of a computer program

Description automatically generated**

**Task 2: TCP Reset Attacks on Telnet Connection (Spoofed TCP Reset Packet):**

****

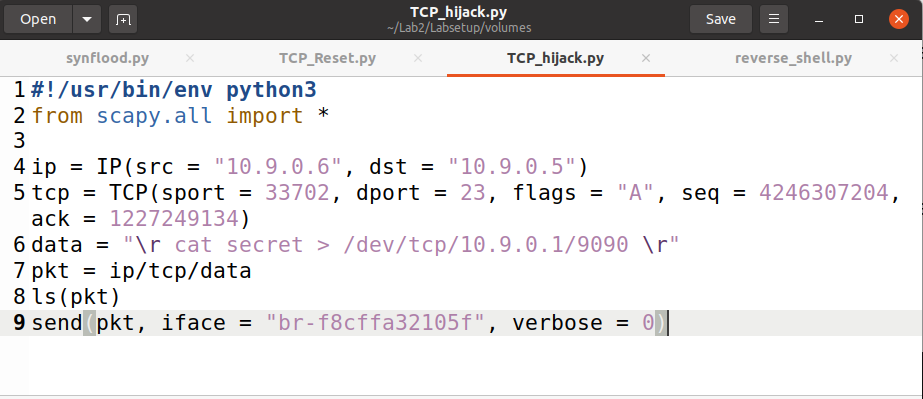
From our Wireshark output we can see that the [RST] packet was in fact sent.

**Task 2: TCP Reset Attacks on Telnet Connection (Connection After Reset Packet):**

****

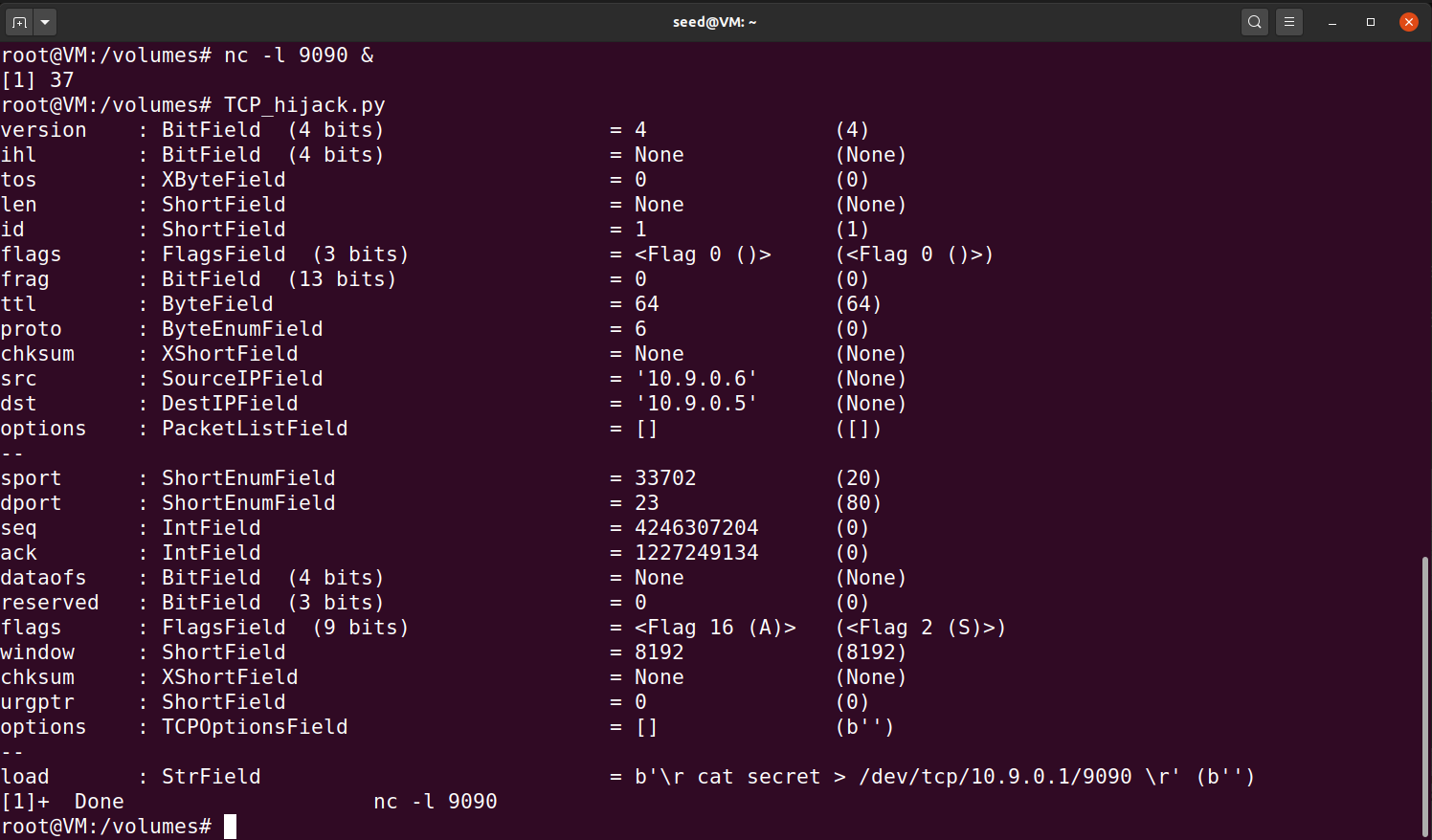
Looking at our VICTIM terminal, we can see that eventually the connection does end and that we successfully attacked using TCP reset with confirmation from Wireshark.

**Task 3: TCP Session Hijacking (Python Code):**

****

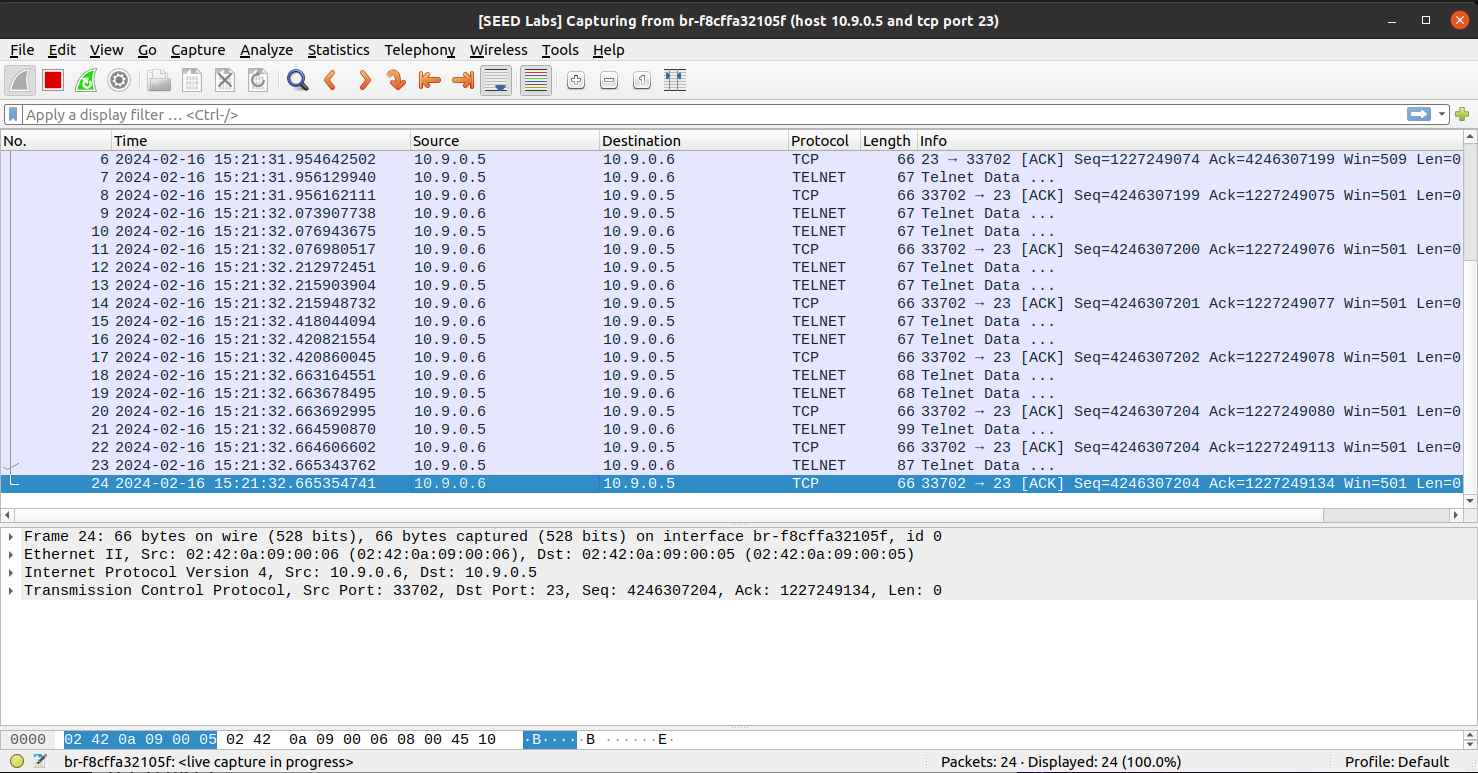
For hijacking, we are going to be establishing the same style of connection as we had above in our TCP Reset attack, however we are going to add a data line in our code that reads if the attack is successful. Again, we locate our tcp = () data from Wireshark to conduct our attack.

**Task 3: TCP Session Hijacking (ATTACKER Terminal):**

****

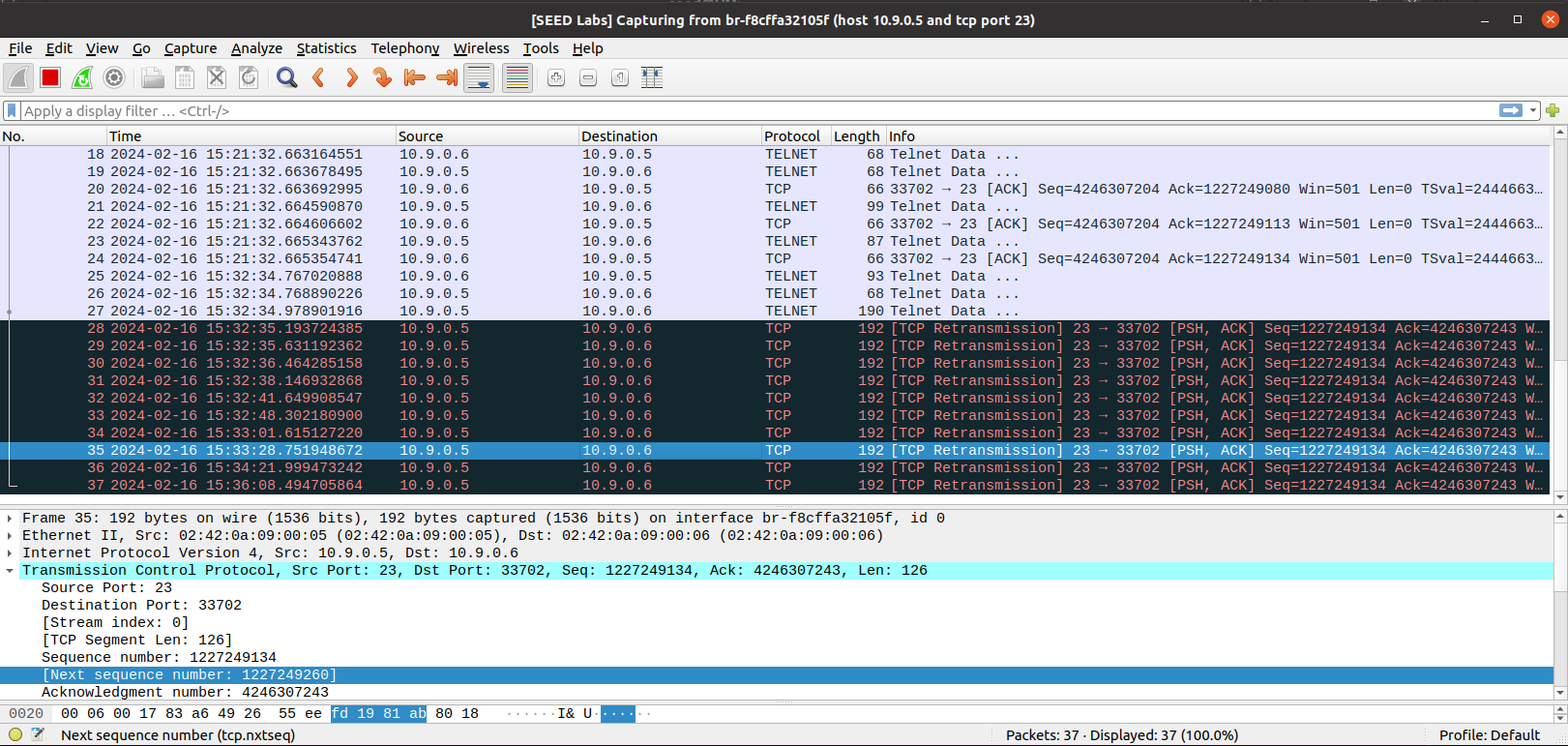
We run our python program which prints out information related to the data received and the data sent out. Specifically, we can see our “\r” command shown in our terminal.

**Task 3: TCP Session Hijacking (Before Hijack - Wireshark):**

****

This is our Wireshark output where we are locating our packet that we send between our HOST and VICTIM. We set it up the same way as our TCP Reset, by sending a random message and pulling specific data from that last packet sent.

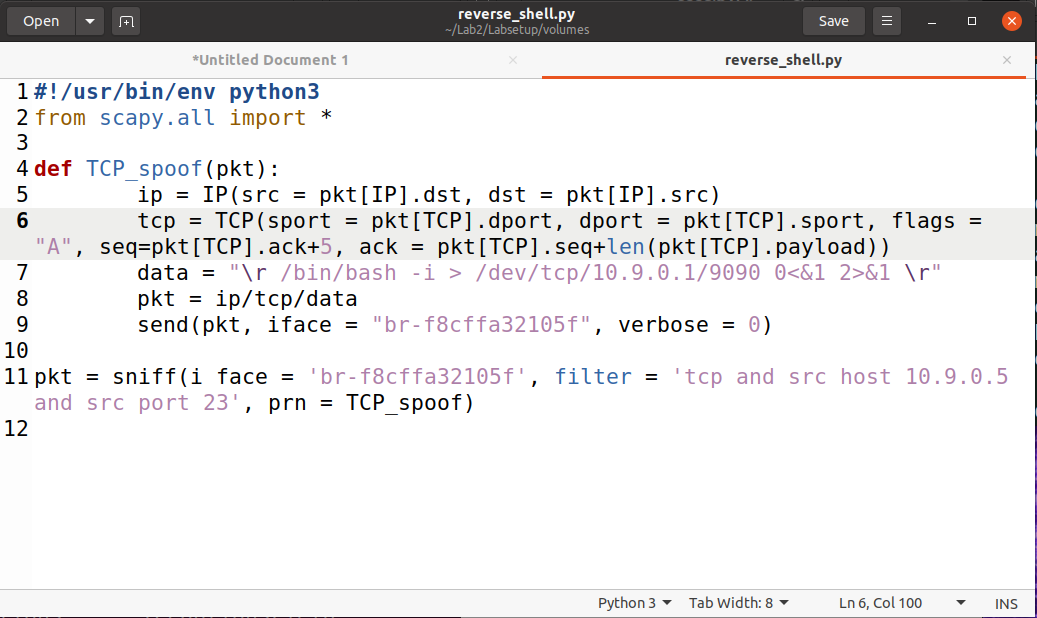
**Task 3: TCP Session Hijacking (After Hijack - Wireshark):**

****

After hitting run on our python program, these [TCP Retransmission]’s started popping up in a group of five, more followed after while idling and then they just stopped. We were able to successfully hijack and send over data to our VICTIM machine from our HOST that is currently hijacked.

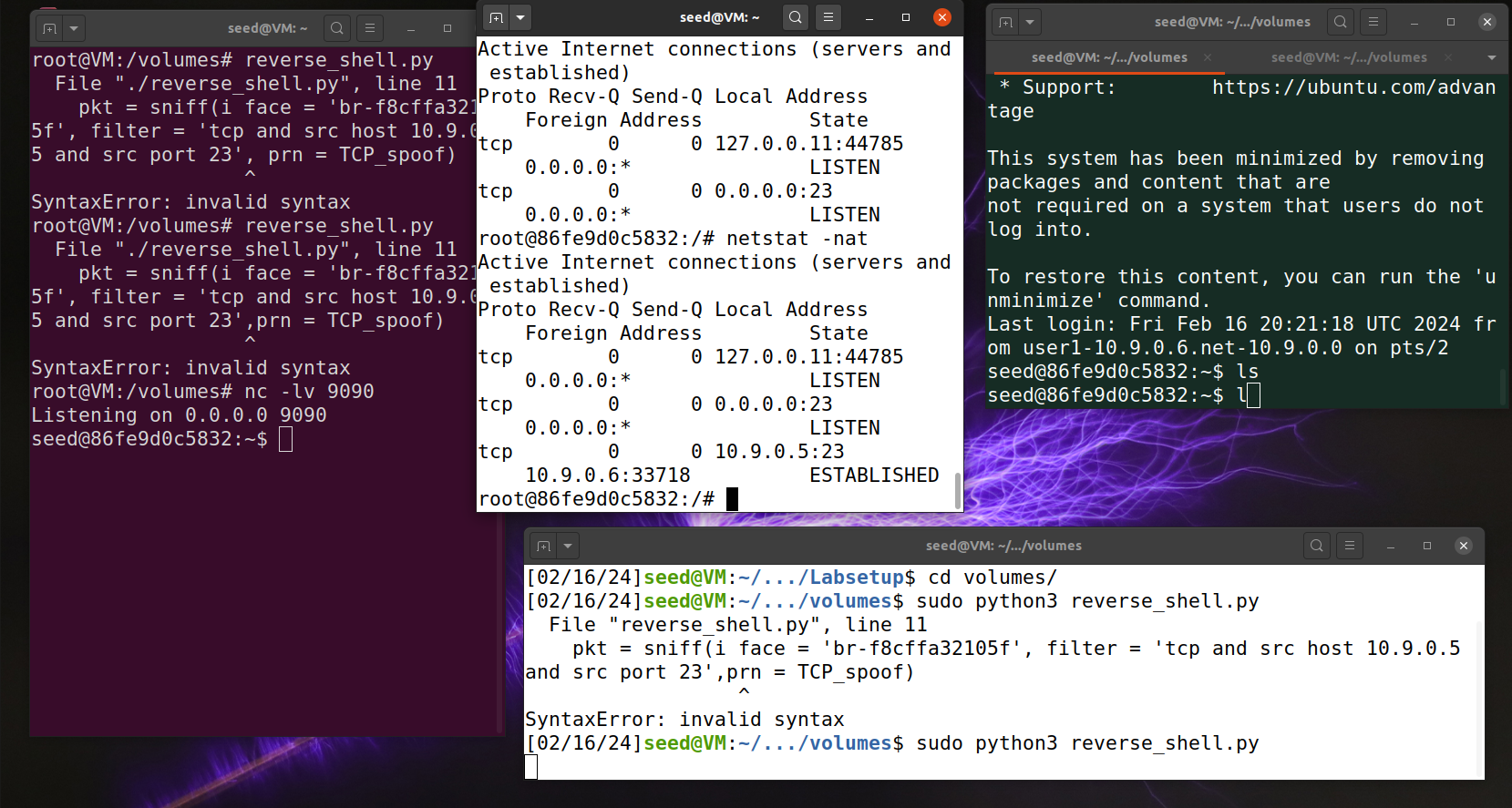
**Task 4 (BONUS): Creating Reverse Shell Using TCP Session Hijacking**

**(Python Code):**

****

The idea behind this attack is that we are establishing a connection between our VICTIM (10.9.0.5) and HOST (10.9.0.6), the same thing as we did with our TCP [Reset] & [Hijack]. We change our code around to focus specifically on TCP packets only through all our ports, sequence and ack numbers. The data line creates and starts a bash shell.

**(Reverse Shell Output):**

****

Initially, before we run our Python program, we can see in the red box that our ATTACKER terminal is still an ATTACKER. We listen with the command: **“nc -lv 9090”** which is listening on that specific port for any sort of traffic. When we go back to our HOST terminal and try to type commands, it will stop and now our ATTACKER terminal has the same seed shell as our HOST, highlighting that our reverse shell program worked and was successful.

**Conclusion**

My attacks were in fact successful across the board. Wireshark was helpful in confirming my results at each step for the specific tasks. I was expecting to see a flood attack, but when I ran my C program to flood the system, I was not expecting that kind of flooding (Over 300k in seconds). I tried everything in my power to save my system from crashing, but it was no luck. The most unique part of this lab for me was establishing a reverse shell connection. There was a moment where I even said to myself that the powers that come from knowing how to conduct these kinds of attacks are just extraordinary to carry-out and that one must be responsible when knowing these kinds of attacks, and I am curious to see what our other Labs have in store for our class.