# Instructions TCP SYN Flood

Vinci Nicolò

22 October 2021

## 1 Basic scripts

In this section, some basic scripts will be explained in order to wrap them in the following section 2. Firstly, the server has to be installed on the server machine and the flooder has to be installed on the attacker machine.

```
$server /share/education/TCPSYNFlood_USC_ISI/install-server
$attacker /share/education/TCPSYNFlood_USC_ISI/install-flooder
```

So, the two commands have been embedded in one script called *setup.sh* which can be found at *scripts/config/*. To run the script:

```
$ ./scripts/config/setup.sh
```

In the same folder there is another script called *syn_cookies.sh* that can be used to set to 1 or 0 the SYN cookies on the server machine. The parameter *enable* has to be passed to set the SYN cookies to 1 or the parameter *disable* to set them to 0.

```
$ ./scripts/config/syn_cookies.sh <enable, disable>
```

Then, in the folder *traffic* there is a script called *leg_traffic.sh* has been developed in order to generate a legitimate traffic from the client towards the server.

```
$ ./scripts/traffic/leg_traffic.sh
```

In the same folder there is also a script called *attacker.sh* used to launch the flooder in order to send a huge amount of *SYN* packet from the attacker to the server. The flooder is launched with a timeout of 120 seconds in order to follow the task. The flooder can be launched spoofing the attacker IP address or not, passing as script parameter *spoof* or *nospoof*.

```
$ ./scripts/traffic/attack.sh <spoof, nospoof>
```

The last basic script is in the sniffer folder and it is called *sniff.sh*. It is useful to run the *tcpdump* command on the client machine in order to sniff only TCP traffic on port 80. The sniffer is launched with a timeout of 180 seconds in order to follow the task. Two parameters need to be passed. One define the interface where the *tcpdump* has to sniff the traffic. The other defines the name of file where the sniffed traffic will be stored.

```
$ ./scripts/sniffer/sniff.sh <ETH> <FILE_NAME>
```

## 2 Data gathering

The script called *wrap_up.sh* is used to collect data. It calls scripts describe in the section 1. It has to be launched from the *otech2ah* machine, because command will be sent through SSH to server, attacker and client machine. It needs three parameters:

- FILE: to pass to *sniff.sh*.
- STATE: to pass to *syn_cookies.sh*
- SPOOF: to pass to *attack.sh*

```
$ ./scripts/wrap_up.sh <FILE> <STATE> <SPOOF>
```

First, it enables or disables the SYN cookies calling *syn_cookies.sh* on the server machine. Then, it launches *sniff.sh* and *leg_traffic.sh* with a timeout of 180 seconds on the client machine. After, it waits 30 seconds and runs *attack.sh* on the attacker machine. At the end, it waits for 150 seconds. To perform the task, the script has to be launched with SYN cookies disabled:

```
$ ./scripts/wrap_up.shno_cookies.pcap disabled spoof
```

Then, the script has to be launched with SYN cookies enabled:

```
$ ./scripts/wrap_up.sh yes_cookies.pcap enabled spoof
```

The two generated pcap files are analyzed by a python script called *main.py* The program defines every possible TCP connection with the respective packets, computes the duration for valid TCP connections and plot a graph. A TCP connection is uniquely represented by 4-tuple: IP source, IP destination, TCP source port, TCP destination port. The python program *uniqueConn.py* specifies a UniqueConn class and ovverides the method equal in order to compare two different UniqueConn objects.

```python
# Check if two connections are equals
def __eq__(self, other):
    if isinstance(other, UniqueConn):
        checkSrcIp = (self._srcIp == other.srcIp or self._srcIp == other.destIp)
        checkDstIp = (self._destIp == other.destIp or self._destIp == other.srcIp)
        checkSrcPort = (self._srcPort == other.srcPort or self._srcPort == other.destPort)
        checkDestPort = (self._destPort == other.destPort or self._destPort == other.srcPort)
        return checkSrcIp and checkDstIp and checkSrcPort and checkDestPort
    return False
```

Figure 1: Method to compare TCP connections

So, the UniqueConn object represents a unique TCP connection. Then, there is a list called *traces* that stores a dictionary formed by a UniqueConn object and packets associated to that TCP connection.

```
{
    "uniqueConn": UniqueConn
    "packets": [<TCP flag, time>]
}
```

For each packet a UniqueConn object is created and added to the list. If it already exists in the list, the packet will be added to *packets*. Otherwise, a new UniqueConn object will be added to the list and the packet will be added to its *packets*. Then, for each element in the list, the *packets* list is inspected. If the first packet has the flag S and the penultimate has the flag FA and the last has the flag A or R, the TCP connection is valid and the duration can be computed. Otherwise, the TCP connection is not valid and the duration is fixed to 200. Then, a graph is plotted with *matplotlib*. On x-axis there is when a TCP connection starts and on y-axis its duration. To launch the python program:

```
$ python3 main.py <PCAP File>
```

The results of the basic task are reported in memo.

# 3 Extra 1

The attacker has to start the *attack.sh* with *nospoof* parameter for the extra 1. So, the *wrap_up.sh* script will be launched in the following way:

```
$ ./scripts/sniffer/sniff.sh extra1_no_firewall.pcap disabled nospoof
```

The attacker will not be able to perform the SYN flood attack as shown in the memo. An iptables rule has been designed to allow the attacker to perform the SYN flood attack again. The rule drops any SYN ACK packet that comes in input to the attacker machine with source ip address of the server.

```
sudo iptables -I INPUT -s 5.6.7.8 -p tcp \
--tcp-flags URG,SYN,PSH,RST,FIN SYN -j DROP
```

The rule is set by a script called *rule.sh* located in *extra1_scripts/iptables/* folder. To launch it:

```
$ ./extra1_scripts/iptables/rule.sh
```

Now, the attacker should be able to perform the attack again. Data can be gathered again with the previous command changing only the file name.

```
$ ./scripts/sniffer/sniff.sh extra1_firewall.pcap disabled nospoof
```

Results are shown in memo.

# 4 Extra 2

The NSfile has been modified for the extra 2 in order to have point-to-point connections. The new NSfile defines two links: one for server-attacker and the other for attacker-server. The command to do so are:

```
set link1 [$ns duplex-link $server $client 1000Mb 0ms DropTail]
set link2 [$ns duplex-link $server $attacker 1000Mb 0ms DropTail]

tb-set-ip-link $server $link1 11.12.13.1
tb-set-ip-link $client $link1 11.12.13.2

tb-set-ip-link $server $link2 22.23.24.1
tb-set-ip-link $attacker $link2 22.23.24.2
```

Then, the scripts to gather data are defined in the *extra2_scripts*. Basically, they are the same used for the basic task. However, different parameters need to launch the *wrap_up2.sh*:

- STATE: to enable or disable SYN cookies.
- SOURCE: attacker source IP.
- DEST: destination IP to attack.
- SPOOF: spoof or not the attacker IP address.

```
$ ./scripts/wrap_up2.sh <STATE> <SOURCE> <DEST> <SPOOF>
```

The script generates four different pcap files:

- attacker.pcap: dump of attacker interface.
- client.pcap: dump of client interface.
- server_attacker.pcap: dump of server interface towards client.
- server_client.pcap: dump of server interface towards attacker.

Now, the exercise can be launched again without spoofing attacker IP address:

```
$ ./scripts/wrap_up2.sh disable 22.23.24.2 22.23.24.1 nospoof
```

And with attacker IP address spoofed:

```
$ ./scripts/wrap_up2.sh disable 22.23.24.2 22.23.24.5 spoof
```

Results are commented in memo.