Wenbin Li SUID: 687150735

Lab 11 Android Rooting Lab

Task 1: Build a simple OTA package

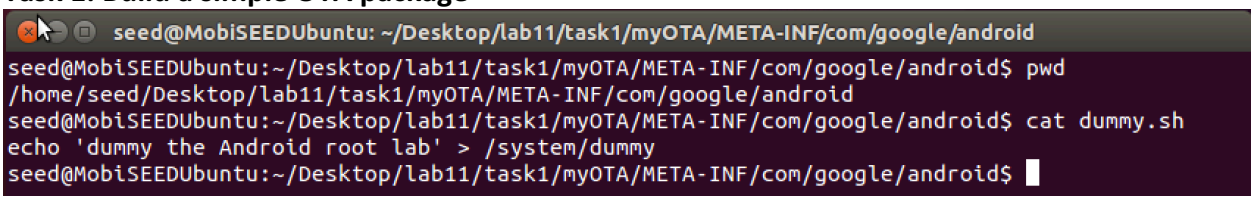

Fig 1.1 create the OTA structure folder and create a file named dummy.sh with some contents

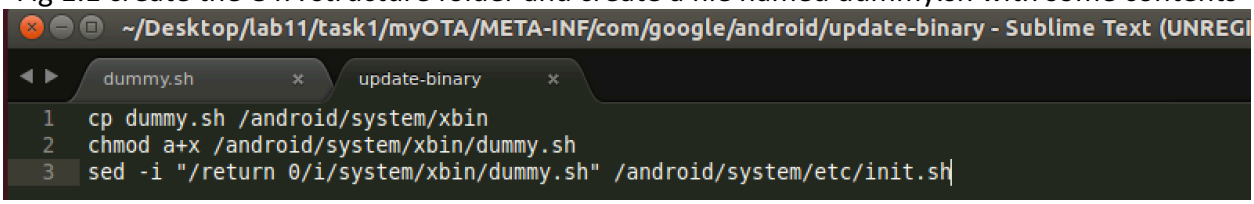

Fig 1.2 create update-binary file in OTA folder, and add some codes in it.

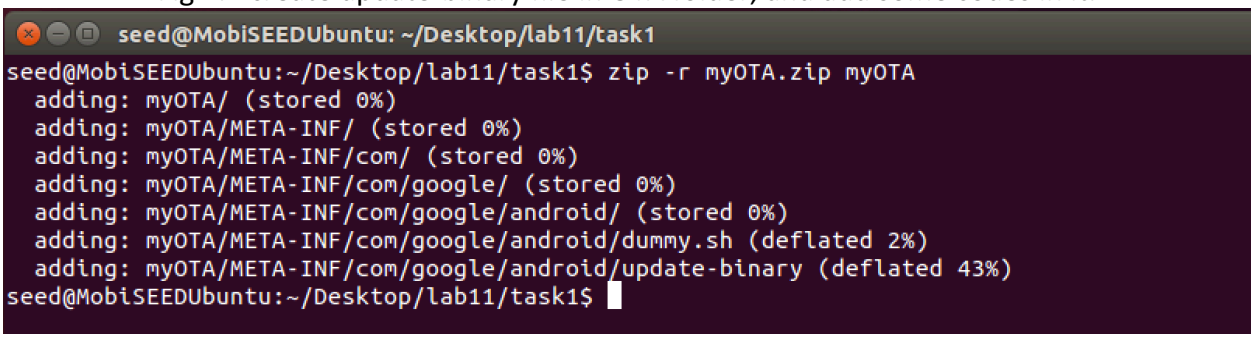

Fig 1.3 zip command to pack myOTA folder.

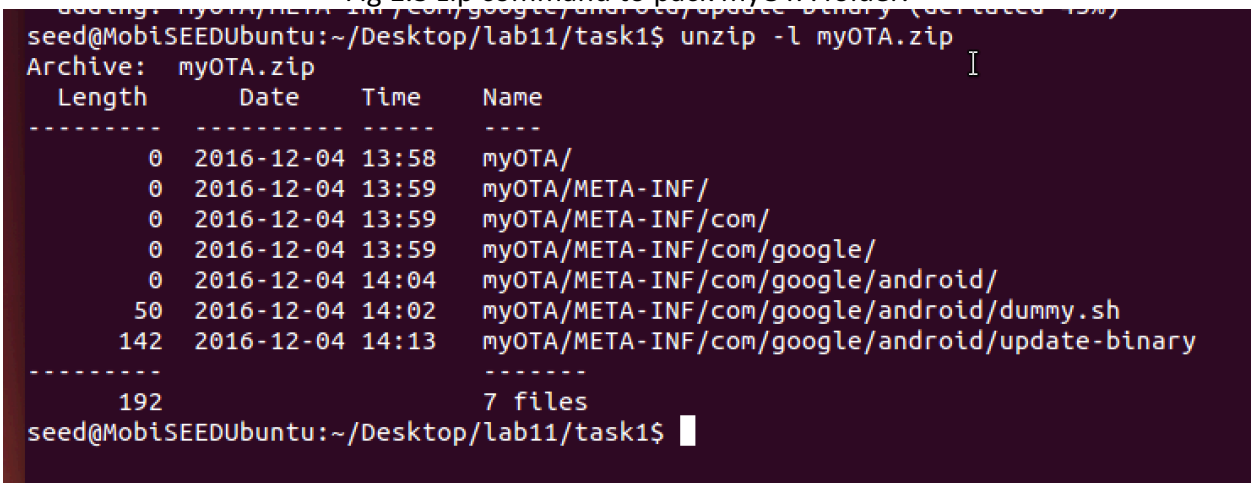

Fig 1.4 list the structure of myOTA files.

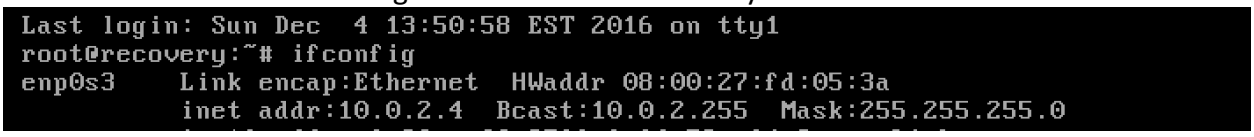

Fig 1.5 log into recovery os to check ip address

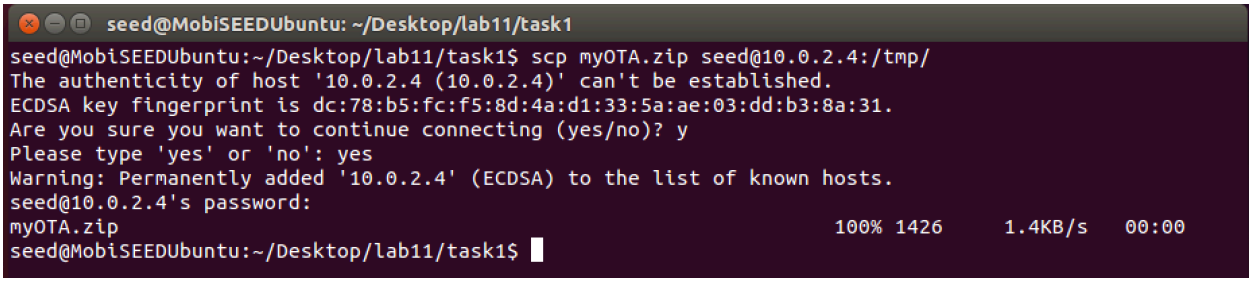

Fig 1.6 use scp command to transfer zip file to recovery os system.

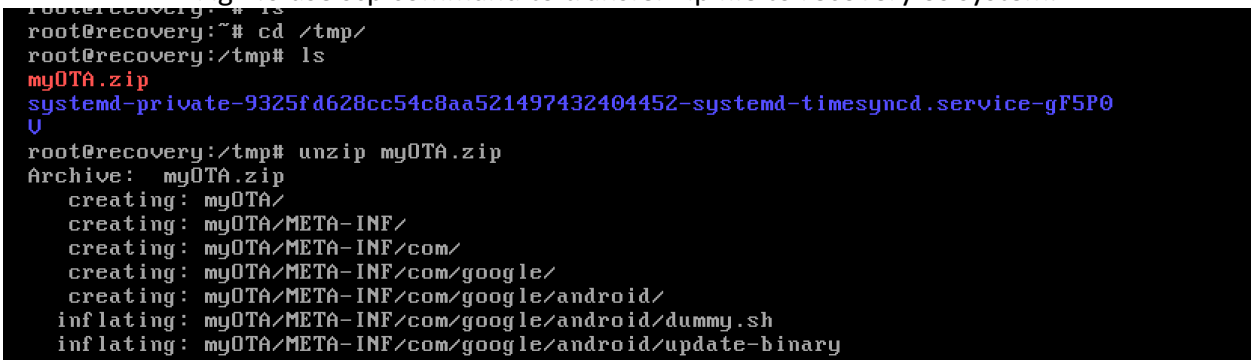

Fig 1.7 check recovery os got the file and unpack this zip file

Fig 1.8 run update-binary file and reboot

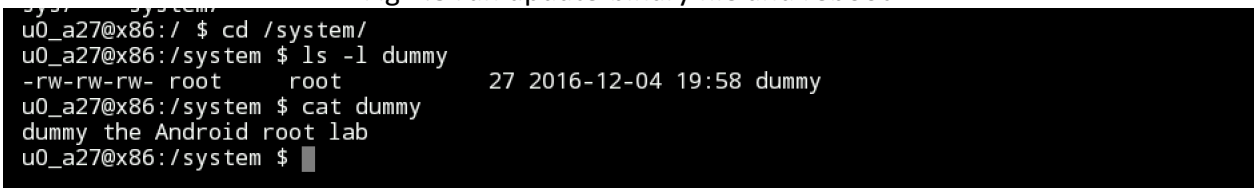

Fig 1.9 check dummy file in Android system and display its content

Explanation and Observation:

1. As fig 1.1, I created dummy.sh file with contents ‘dummy the Android root lab’. This dummy.sh file was placed in myOTA/META-INF/com/google/android/. This is the structure of OTA(over-the-air). In this myOTA/META-INF/com/google/android/ folder, another file named update-binary was placed, like fig 1.2. This update-binary code would be used in recovery os to modify android system.
2. Fig 1.3 showed that zip command could pack ota files into a zip file. The structure of this ota files was showed like fig 1.4.
3. Fig 1.5 showed that in recovery os, command ifconfig was used to check recovery o sip address. From fig 1.5, I got ip address ’10.0.2.4’.
4. Fig 1.6, scp command was used to trans myOTA.zip file to recovery os.
5. Fig 1.7 showed that in /tmp/ folder, recovery os got ota folder. And then unzip this file.

6. Like fig 1.8, I run update-binary file and then reboot the system.
7. As fig 1.9, a dummy file was created in /android/system/ with the contents 'dummy the Android rot lab'. This folder needs root privilege to create file. So it indicted that dummy.sh was run with root privilege. The reason was that update-binary modified init.sh in android OS. Dummy.sh would be run in init.sh. This init.sh was run with root privilege, so that dummy.sh was run with root privilege. init.sh was run when every time android system is started.

Task 2: Inject code via app process

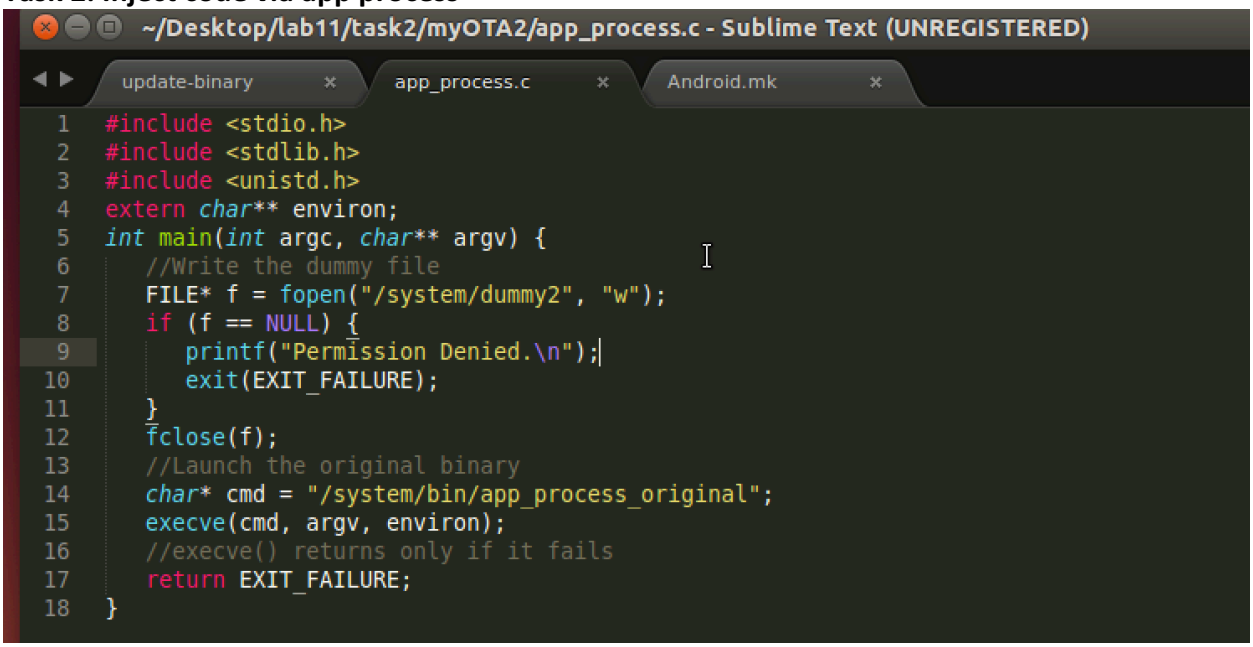

Fig 2.1 code of app_process.c to create dummy2 and run app_process_original

Fig 2.2 modified Android.mk file

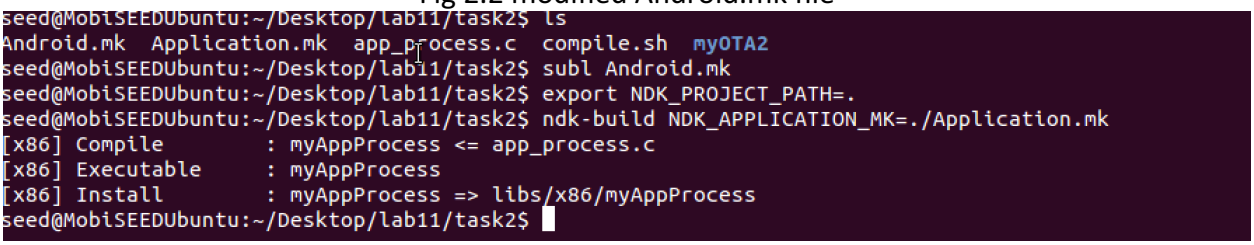

Fig 2.3 complie app_process.c file in MobiSeeedUbuntu.

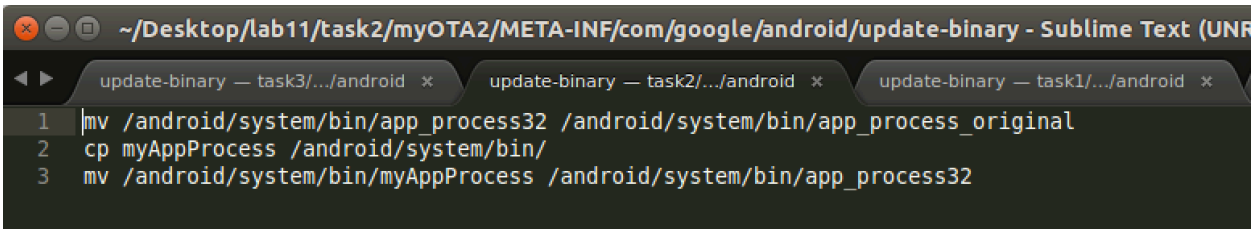

Fig 2.4 create update-binary file in OTA folder

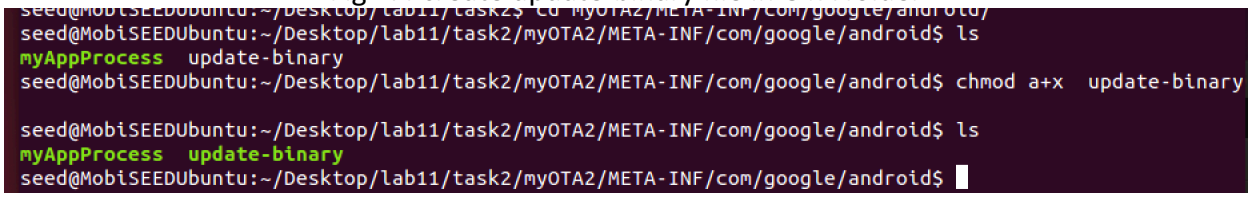

Fig 2.5 make update-binary executable

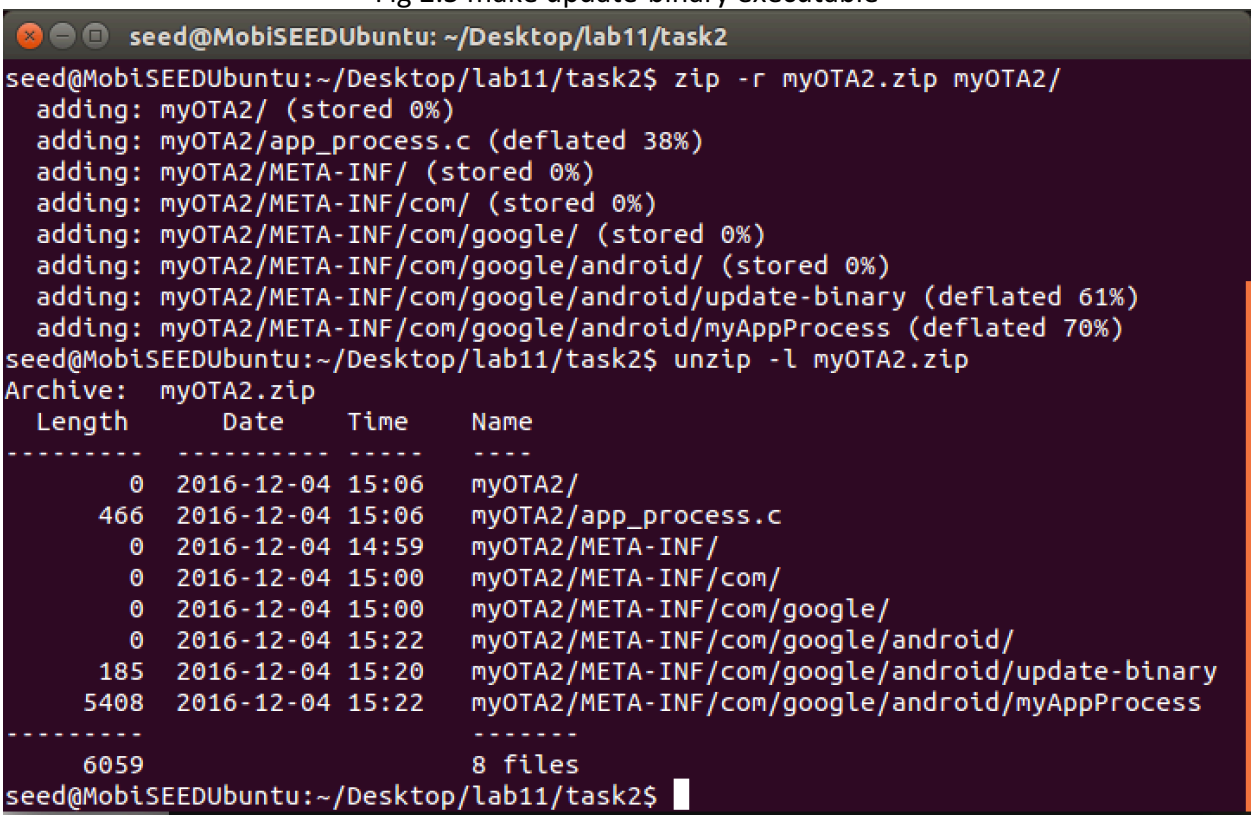

Fig 2.6 packed OTA file and showed its structure

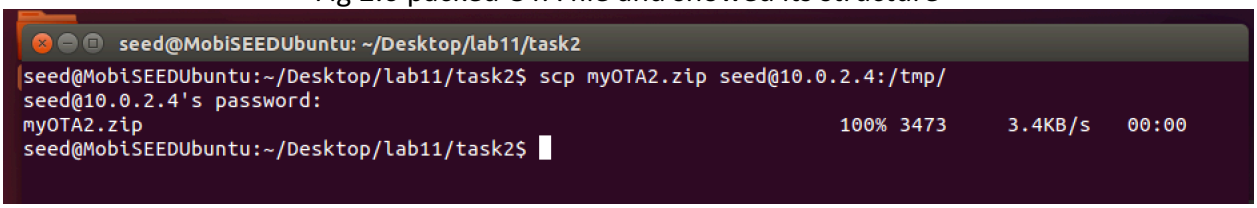

Fig 2.7 transferred myOTA2.zip to recovery os.

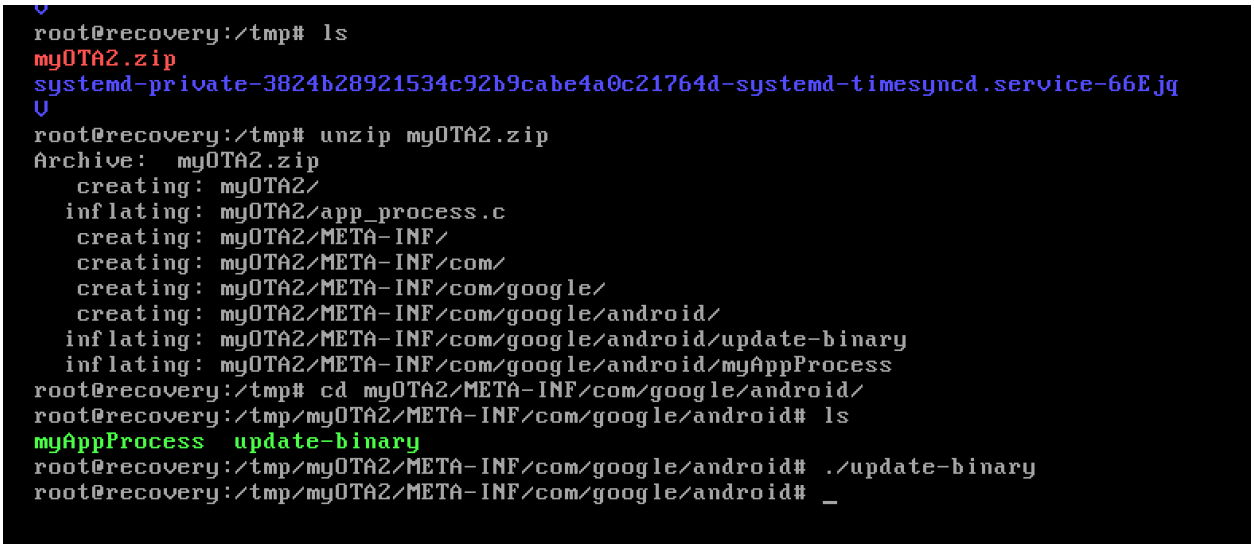

Fig 2.8 in recovery os, checked exist of myOTA2.zip and unpacked it.

Fig 2.9 run update-binary file and rebooted it

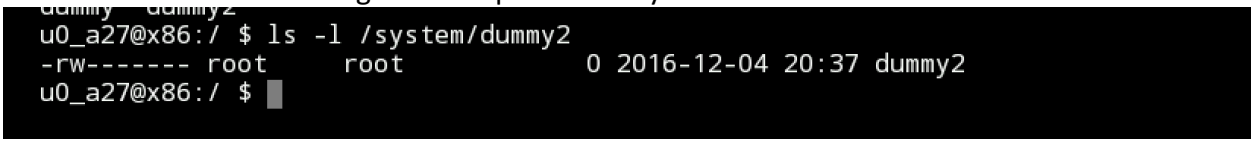

Fig 2.10 in Android system, found dummy2 file

Observation and Explanation:

1. As fig 2.1, app_process.c code was to create dummy2 in /system/, and run app_process_original. This code would be used by Zygote.
2. Like fig 2.2, I changed Android.mk, local_module is myAppProcess, and local_source_file is app_process.c, and its intention was to build app_process.c file to myAppProcess.
3. As fig 2.3, Android.mk, Application.mk and app_process.c placed in one folder. Like the command in this picture, out-put file named myAppProcess was generated.
4. As fig 2.4, update-binary file was created to rename app_process32 to app_process_original, and then copy myAppProcess file in OTA to /android/system/bin folder. And then renamed myAppProcess to app_process32.
5. As fig 2.5, make the update-binary file to an executable file, and placed myAppProcess in ota android folder.
6. Like fig 2.6 and 2.7 ota files were packed and transferred to recovery os.
7. Like fig 2.8, I unzipped ota, and as 2.9, I run update-binary file and reboot this system.
8. Like 2.10, when android system was logged in, file dummy2 in /system/ was found, and this file woned by root, so that this file was created with root privilege. It indicated that app_process32 was run with root privilege. The idea of this attack is that malicious code was added with name app_process32 and original app_process32 was called by new malicious code. So that some malicious code would be run before regular process.

Task 3: Implement SimpleSU for Getting Root Shell

```
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU$ cd mydaemon/
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU/mydaemon$ export NDK_PROJECT_PATH=.
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU/mydaemon$ ndk-build NDK_APPLICATION_MK=./Application.mk
[x86] Compile        : mydaemon <= mydaemonsu.c
[x86] Compile        : mydaemon <= socket_util.c
[x86] Executable     : mydaemon
[x86] Install        : mydaemon => libs/x86/mydaemon
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU/mydaemon$
```

Fig 3.1 compiled daemon

```
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU/mysu$ ndk-build NDK_APPLICATION_MK=./Application.mk
[x86] Compile        : mysu <= mysu.c
[x86] Compile        : mysu <= socket_util.c
[x86] Executable     : mysu
[x86] Install        : mysu => libs/x86/mysu
seed@MobiSEEDUbuntu:~/Desktop/lab11/SimpleSU/mysu$
```

Fig 3.2 complied mysu

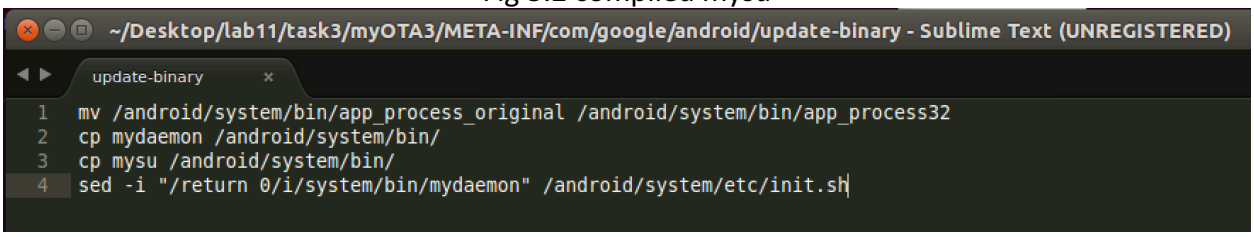

Fig 3.3 create update-binary file for task3

```
seed@MobiSEEDUbuntu:~/Desktop/lab11/task3/myOTA3/META-INF/com/google/android$ ls -l
total 36
-rwxr-xr-x 1 seed seed 5408 Dec  4 16:29 myAppProcess
-rwxr-xr-x 1 seed seed 9504 Dec  4 15:42 mydaemon
-rwxr-xr-x 1 seed seed 9504 Dec  4 15:40 mysu
-rwxrwxr-x 1 seed seed  332 Dec  4 16:34 update-binary
seed@MobiSEEDUbuntu:~/Desktop/lab11/task3/myOTA3/META-INF/com/google/android$
```

Fig 3.4 show the android folder file in OTA, please ignore myAppProcess.

```
seed@MobiSEEDUbuntu:~/Desktop/lab11/task3$ zip -r myOTA3.zip myOTA3
  adding: myOTA3/ (stored 0%)
  adding: myOTA3/META-INF/ (stored 0%)
  adding: myOTA3/META-INF/com/ (stored 0%)
  adding: myOTA3/META-INF/com/google/ (stored 0%)
  adding: myOTA3/META-INF/com/google/android/ (stored 0%)
  adding: myOTA3/META-INF/com/google/android/mysu (deflated 67%)
  adding: myOTA3/META-INF/com/google/android/update-binary (deflated 52%)
  adding: myOTA3/META-INF/com/google/android/mydaemon (deflated 61%)
  adding: myOTA3/META-INF/com/google/android/myAppProcess (deflated 69%)
seed@MobiSEEDUbuntu:~/Desktop/lab11/task3$ scp myOTA3.zip seed@10.0.2.4:/tmp/
seed@10.0.2.4's password:
myOTA3.zip                                    100%   10KB  10.2KB/s   00:00
seed@MobiSEEDUbuntu:~/Desktop/lab11/task3$
```

Fig 3.5 packed myOTA3 folder and transferred to recovery os

```
Last login: Sun Dec  4 15:30:05 EST 2016 on tty1
root@recovery:~# cd /tmp/
root@recovery:/tmp# ls
myOTA3.zip
systemd-private-d33995200b4244959b46d19d7909ad11-systemd-timesyncd.service-mx9ug
Q
root@recovery:/tmp# unzip myOTA3.zip
Archive:  myOTA3.zip
   creating: myOTA3/
   creating: myOTA3/META-INF/
   creating: myOTA3/META-INF/com/
   creating: myOTA3/META-INF/com/google/
   creating: myOTA3/META-INF/com/google/android/
  inflating: myOTA3/META-INF/com/google/android/mysu
  inflating: myOTA3/META-INF/com/google/android/update-binary
  inflating: myOTA3/META-INF/com/google/android/mydaemon
  inflating: myOTA3/META-INF/com/google/android/myAppProcess
root@recovery:/tmp# cd myOTA3/META-INF/com/google/android/
root@recovery:/tmp/myOTA3/META-INF/com/google/android# ls
myAppProcess  mydaemon  mysu  update-binary
```

Fig 3.6 check exist of myOTA3.zip and then unzipped it

```
root@recovery:/tmp/myOTA3/META-INF/com/google/android# ./update-binary _
```

Fig 3.7 run update binary file

```
root@recovery:/tmp/myOTA3/META-INF/com/google/android# reboot now_
```

Fig 3.8 reboot recovery os to log into android system

```
u0_a27@x86:/ $ ls -l /system/bin/mysu
-rwxr-xr-x root     root         9504 2016-12-05 00:23 mysu
u0_a27@x86:/ $ mysu
WARNING: linker: mysu: unused DT entry: type 0x6ffffffe arg 0x590
WARNING: linker: mysu: unused DT entry: type 0x6fffffff arg 0x1
/system/bin/sh: No controlling tty: open /dev/tty: No such device or address
/system/bin/sh: warning: won't have full job control
root@x86:/ # id
uid=0(root) gid=0(root)
root@x86:/ # whoami
root
root@x86:/ #
```

Fig 3.9 showed mysu existing, and run mysu to gained root privilege.

Observation and Explanation:

1. As fig 3.1 and 3.2, SimpleSU files were downloaded from seedlab website, and then were compiled to get mysu and daemon. Daemon was a process for background running, and mysu is a client to interactive with daemon.
2. As fig 3.3, the first line of update-binary was to rename app_process_original to app_process32. Reason was that in task2, app_process32 was changed name as app_original. So the system file should be renamed back. The second and third line of this code was to cp mydaemon and mysu to /android/system/bin/. The forth line of this code was to add command mydaemon in init.sh file, so that daemon would be executed when android os start.
3. Like fig 3.4, put mydaemon, mysu, and update-binary in ota android folder. These files were to modify android os to get root privilege.
4. Like fig 3.5, I packed myOTA3 file and transferred it to recoversy os.
5. In recovery os, like fig 3.6, unzip myOTA3.zip file and run update-binary file, like fig 3.7.
6. Reboot os like fig 3.8.

7. Like 3.9, when android os was logged into, mysu was found in /system/bin/. After I typed mysu in terminal, '$' became '#'. Id command showed that uid and gid was belong to root. 'whoami' command print the 'root'.

Where the Actions occur:(file name, function name, line number)

• Server launches the original *app_process* binary

in **mydaemonsu.c** file, **main** function, **line 253**

```
251     else {
252         argv[0] = APP_PROCESS;
253         execve(argv[0], argv, environ);
254     }
```

• Client sends its FDs

in **mysu.c, connect_daemon** function, line 101 ~ 102. **send_fd** is used to send clients' FD

```
101     send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
102     send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
103     send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2
```

send_fd is defined in socket_util.c line 109.

• Server forks to a child process

in **mydaemonsu.c**, function **run_daemon**, line **189**.

```
188     while ((client = accept(socket, NULL, NULL)) > 0) {
189         if (0 == fork()) {
```

• Child process receives client's FDs

in mydaemonsu.c, function child_process, line 147 ~ 149

```
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
```

this recv_fd() function is defined in socket_util.c, line 52

```
52  int recv_fd(int sockfd) {
```

• Child process redirects its standard I/O FDs

in mydaemonsu.c, function child_process, line 151 ~ 153

```
151     dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
152     dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
153     dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
```

dup2 is a function defined in unistd.h

```
SYNOPSIS
    #include <unistd.h>

    int
    dup(int fildes);

    int
    dup2(int fildes, int fildes2);
```

• Child process launches a root shell
in mydaemonsu.c, function execve(shell[0], shell, env), line 162.

```
158     //construct essentail environment variables
159     char* env[] = {SHELL_ENV, PATH_ENV};
160
161     char* shell[] = {DEFAULT_SHELL, NULL};
162     execve(shell[0], shell, env);
```