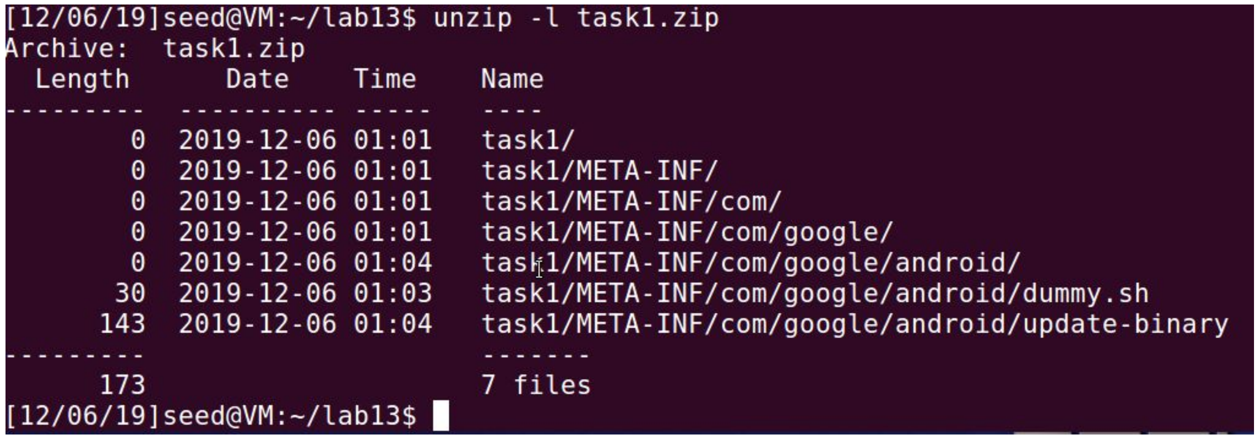
Lab13_AndroidRooting

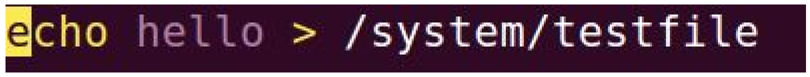
Task01

1. Create the OTA file structure. For easy to recognize the file, we call it ‘tesk1’.



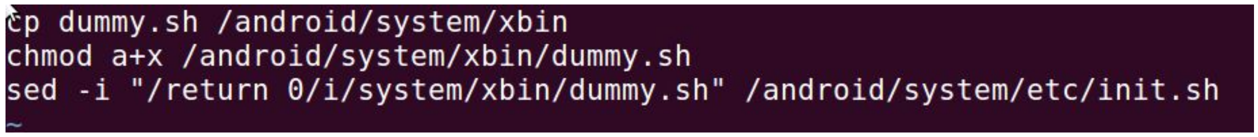
```
[12/06/19]seed@VM:~/lab13$ unzip -l task1.zip
Archive:  task1.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-12-06 01:01   task1/
        0  2019-12-06 01:01   task1/META-INF/
        0  2019-12-06 01:01   task1/META-INF/com/
        0  2019-12-06 01:01   task1/META-INF/com/google/
        0  2019-12-06 01:04   task1/META-INF/com/google/android/
       30  2019-12-06 01:03   task1/META-INF/com/google/android/dummy.sh
      143  2019-12-06 01:04   task1/META-INF/com/google/android/update-binary
---------                     -------
      173                     7 files
[12/06/19]seed@VM:~/lab13$
```

2. Code in the ‘dummy.sh’.



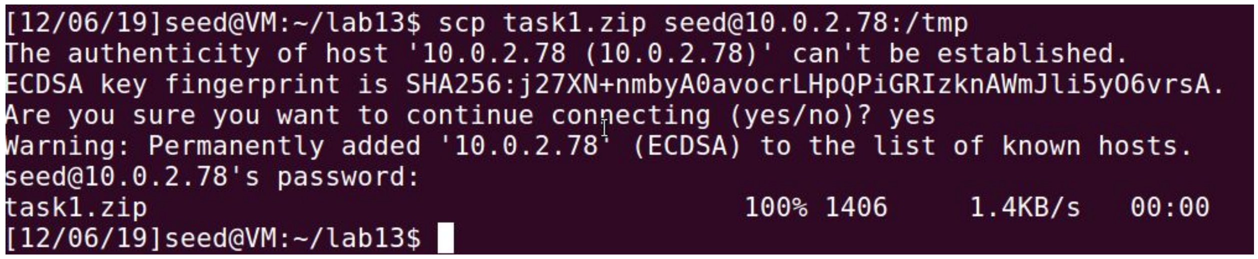
```
echo hello > /system/testfile
```

3. Code in the ‘update-binary’.

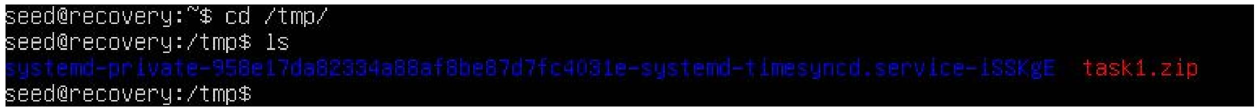


```
cp dummy.sh /android/system/xbin
chmod a+x /android/system/xbin/dummy.sh
sed -i "/return 0/i/system/xbin/dummy.sh" /android/system/etc/init.sh
~
```

4. Send the OTA to the recovery OS.

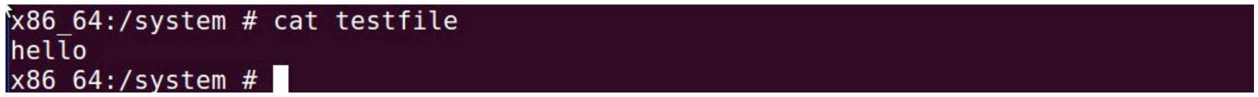


```
[12/06/19]seed@VM:~/lab13$ scp task1.zip seed@10.0.2.78:/tmp
The authenticity of host '10.0.2.78 (10.0.2.78)' can't be established.
ECDSA key fingerprint is SHA256:j27XN+nmbyA0avocrLHpQPiGRIzknAWmJli5yO6vrsA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.78' (ECDSA) to the list of known hosts.
seed@10.0.2.78's password:
task1.zip                                     100% 1406     1.4KB/s   00:00
[12/06/19]seed@VM:~/lab13$
```



```
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
systemd-private-958e17da82334a88af8be87d7fc4031e-systemd-timesyncd.service-iSSKgE  task1.zip
seed@recovery:/tmp$
```

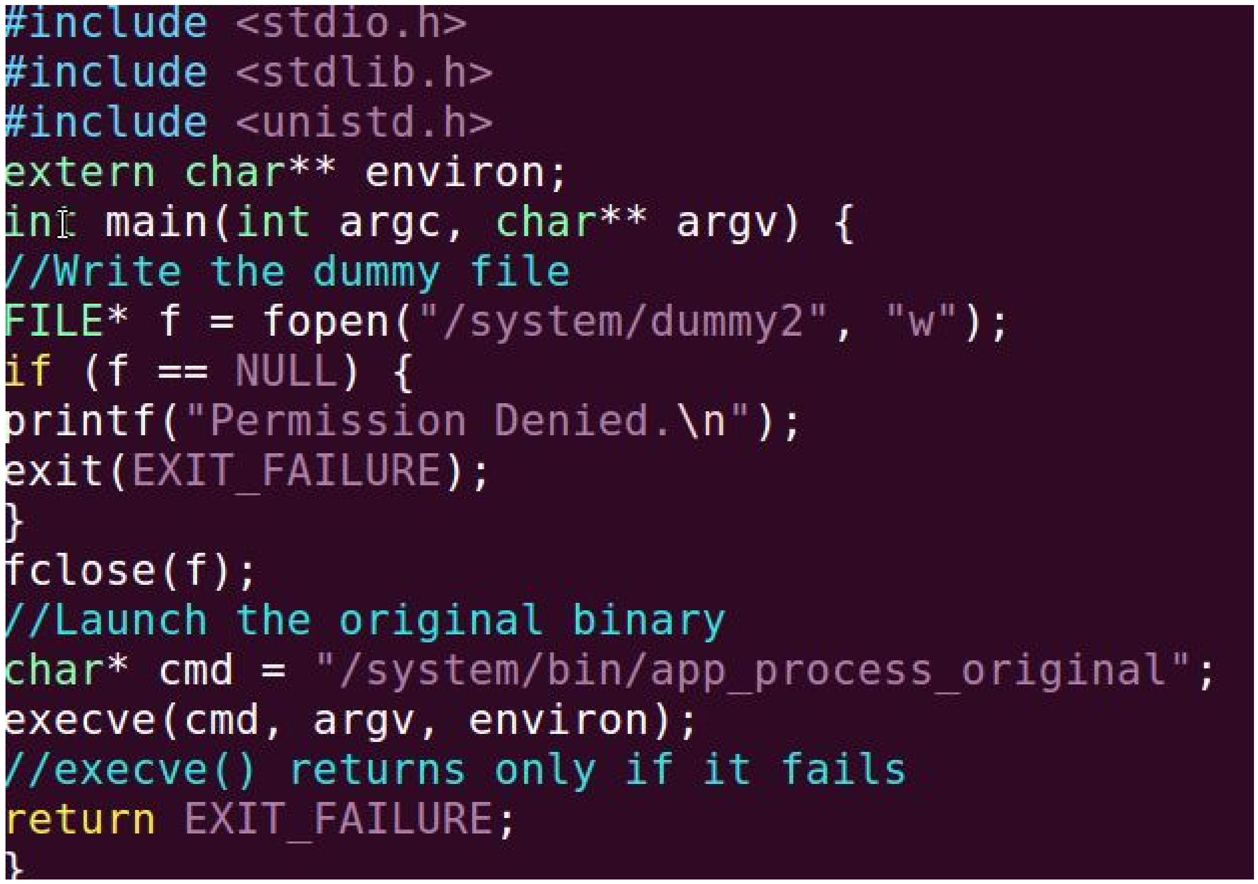
5. Run OTA and check the result. We successfully using the root privilege to modify a file.



```
x86_64:/system # cat testfile
hello
x86_64:/system #
```

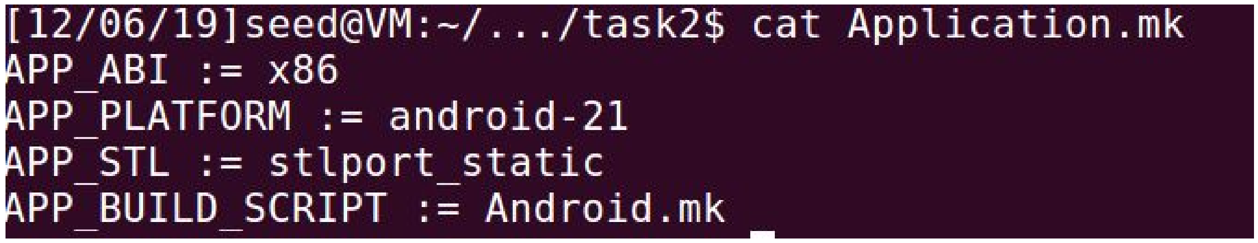
Task02

1. Create the file called: ‘my_process.c’.

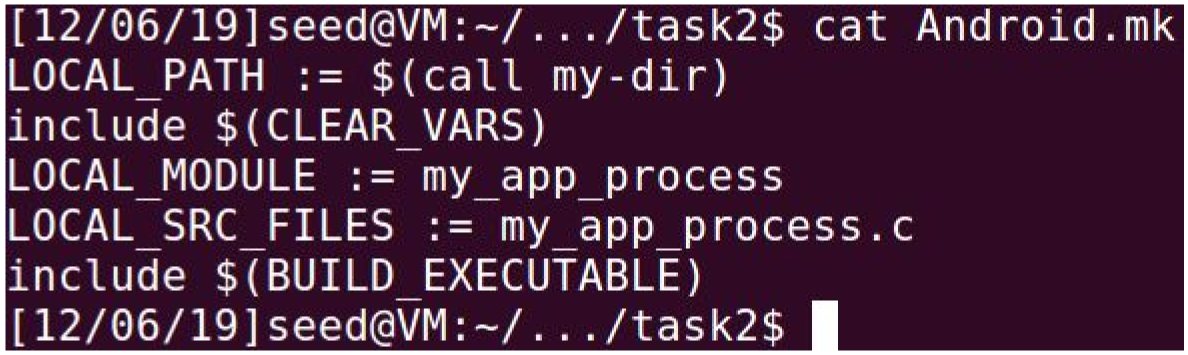


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
extern char** environ;
int main(int argc, char** argv) {
//Write the dummy file
FILE* f = fopen("/system/dummy2", "w");
if (f == NULL) {
printf("Permission Denied.\n");
exit(EXIT_FAILURE);
}
fclose(f);
//Launch the original binary
char* cmd = "/system/bin/app_process_original";
execve(cmd, argv, environ);
//execve() returns only if it fails
return EXIT_FAILURE;
}
```

2. Create ‘Application.mk’ and ‘Android.mk’.

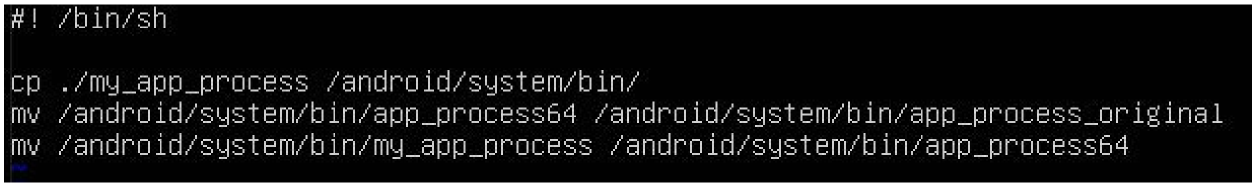


```
[12/06/19]seed@VM:~/.../task2$ cat Application.mk
APP_ABI := x86
APP_PLATFORM := android-21
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
```



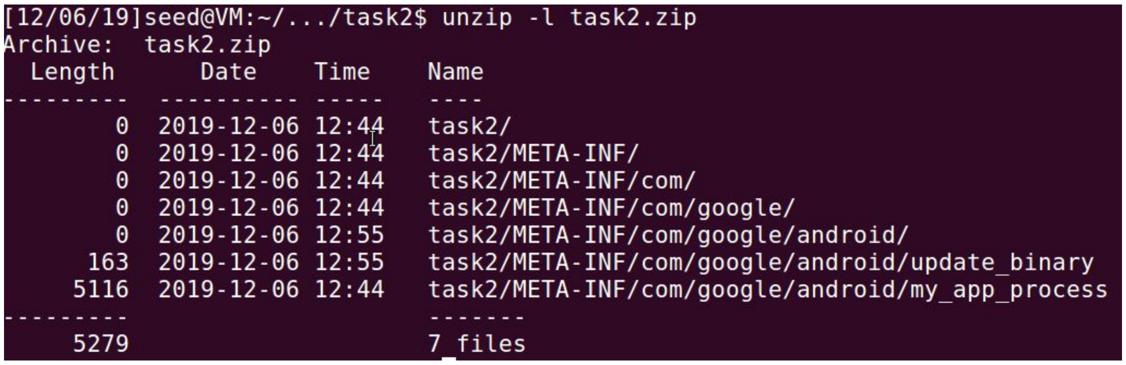
```
[12/06/19]seed@VM:~/.../task2$ cat Android.mk
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := my_app_process
LOCAL_SRC_FILES := my_app_process.c
include $(BUILD_EXECUTABLE)
[12/06/19]seed@VM:~/.../task2$
```

3. Compile my_app_process.c using NDK and copy it into the OTA package.
4. Write our own update_binary script.
   Our task is to cp ‘my_app_process’ to the ‘/system/bin/’, and modify its name to ‘app_process64’ and change the original one to ‘app_process_original’.



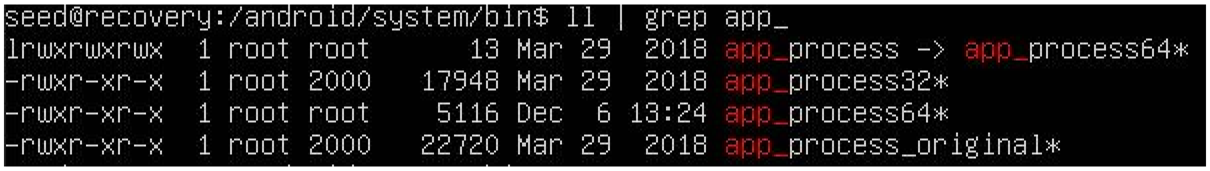
```
#! /bin/sh

cp ./my_app_process /android/system/bin/
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
mv /android/system/bin/my_app_process /android/system/bin/app_process64
```

5. The file structure of OTA package is like below.



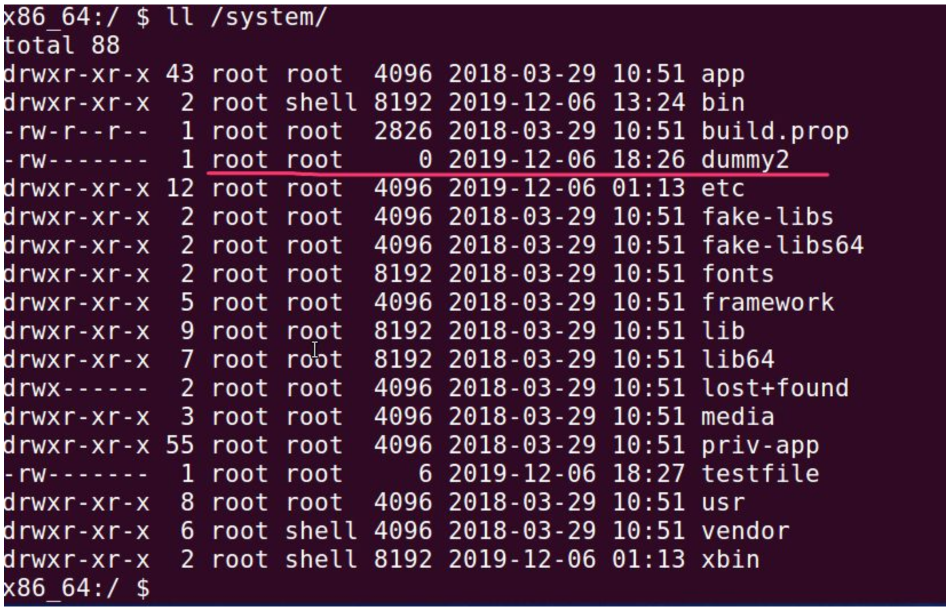
```
[12/06/19]seed@VM:~/.../task2$ unzip -l task2.zip
Archive:  task2.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-12-06 12:44   task2/
        0  2019-12-06 12:44   task2/META-INF/
        0  2019-12-06 12:44   task2/META-INF/com/
        0  2019-12-06 12:44   task2/META-INF/com/google/
        0  2019-12-06 12:55   task2/META-INF/com/google/android/
      163  2019-12-06 12:55   task2/META-INF/com/google/android/update_binary
     5116  2019-12-06 12:44   task2/META-INF/com/google/android/my_app_process
---------                     -------
     5279                     7 files
```

6. Zip the file and check whether we successfully create a ‘dummy2’ file in ‘/system/’.



```
seed@recovery:/android/system/bin$ ll | grep app_
lrwxrwxrwx  1 root root      13 Mar 29  2018 app_process -> app_process64*
-rwxr-xr-x  1 root 2000   17948 Mar 29  2018 app_process32*
-rwxr-xr-x  1 root root    5116 Dec  6 13:24 app_process64*
-rwxr-xr-x  1 root 2000   22720 Mar 29  2018 app_process_original*
```

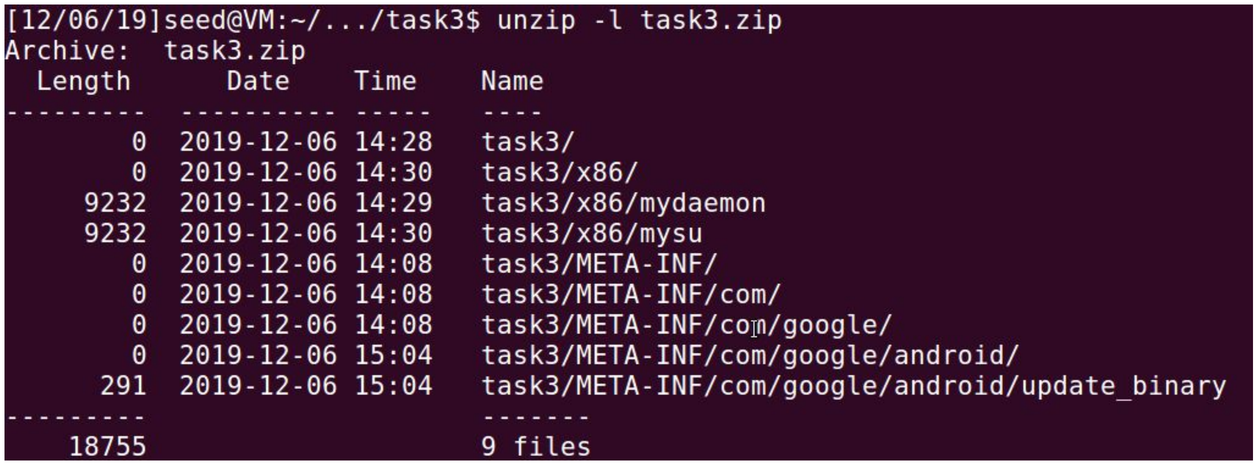
The app_process file is being modified. And ‘summy2’ file has been successfully created in the ‘/system’, which means our attack is success.



```
x86_64:/ $ ll /system/
total 88
drwxr-xr-x 43 root root  4096 2018-03-29 10:51 app
drwxr-xr-x  2 root shell 8192 2019-12-06 13:24 bin
-rw-r--r--  1 root root  2826 2018-03-29 10:51 build.prop
-rw-------  1 root root     0 2019-12-06 18:26 dummy2
drwxr-xr-x 12 root root  4096 2019-12-06 01:13 etc
drwxr-xr-x  2 root root  4096 2018-03-29 10:51 fake-libs
drwxr-xr-x  2 root root  4096 2018-03-29 10:51 fake-libs64
drwxr-xr-x  2 root root  8192 2018-03-29 10:51 fonts
drwxr-xr-x  5 root root  4096 2018-03-29 10:51 framework
drwxr-xr-x  9 root root  8192 2018-03-29 10:51 lib
drwxr-xr-x  7 root root  8192 2018-03-29 10:51 lib64
drwx------  2 root root  4096 2018-03-29 10:51 lost+found
drwxr-xr-x  3 root root  4096 2018-03-29 10:51 media
drwxr-xr-x 55 root root  4096 2018-03-29 10:51 priv-app
-rw-------  1 root root     6 2019-12-06 18:27 testfile
drwxr-xr-x  8 root root  4096 2018-03-29 10:51 usr
drwxr-xr-x  6 root shell 4096 2018-03-29 10:51 vendor
drwxr-xr-x  2 root shell 8192 2019-12-06 01:13 xbin
x86_64:/ $
```

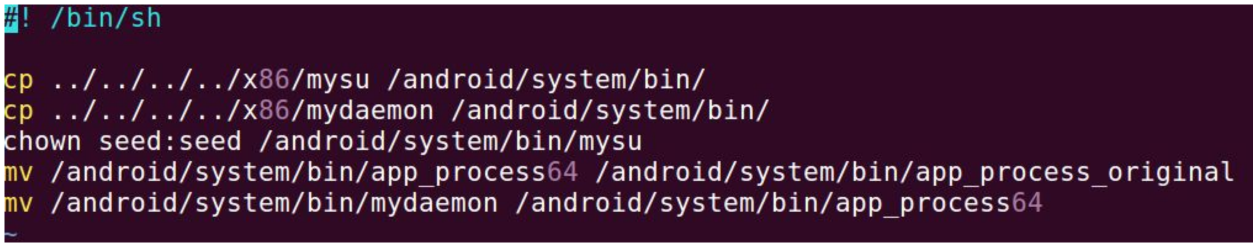
Task03

1. Create a file structure like task01 and task02.
2. The file structure is like below:



```
[12/06/19]seed@VM:~/.../task3$ unzip -l task3.zip
Archive:  task3.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-12-06 14:28   task3/
        0  2019-12-06 14:30   task3/x86/
     9232  2019-12-06 14:29   task3/x86/mydaemon
     9232  2019-12-06 14:30   task3/x86/mysu
        0  2019-12-06 14:08   task3/META-INF/
        0  2019-12-06 14:08   task3/META-INF/com/
        0  2019-12-06 14:08   task3/META-INF/com/google/
        0  2019-12-06 15:04   task3/META-INF/com/google/android/
      291  2019-12-06 15:04   task3/META-INF/com/google/android/update_binary
---------                     -------
    18755                     9 files
```

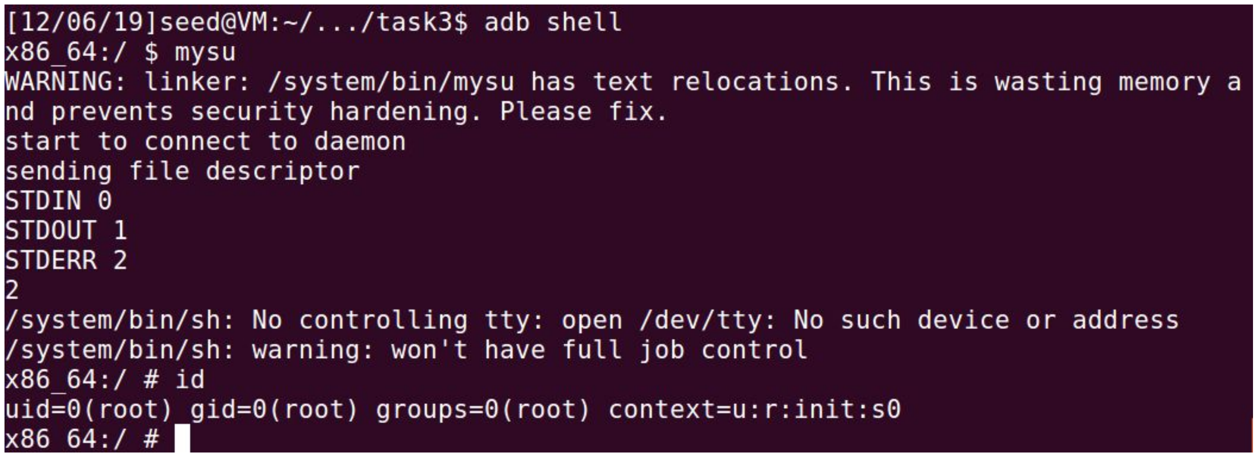
3. The code in update_binary is like below.



```
#! /bin/sh

cp ../../../../x86/mysu /android/system/bin/
cp ../../../../x86/mydaemon /android/system/bin/
chown seed:seed /android/system/bin/mysu
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
mv /android/system/bin/mydaemon /android/system/bin/app_process64
~
```

   Due to I overwrite the true app_process, the VM has to be restored. In this way, the code above will work.

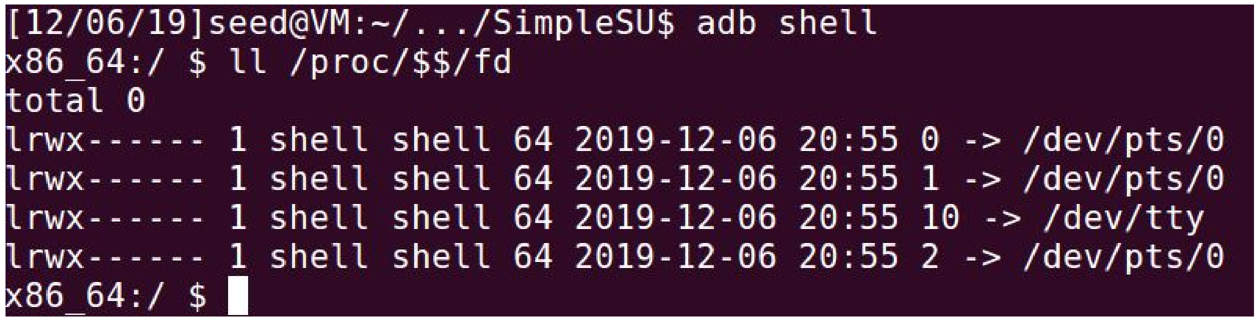
4. Send the zip package and reboot the VM, check whether we can get root privilege using 'mysu'.



```
[12/06/19]seed@VM:~/.../task3$ adb shell
x86_64:/ $ mysu
WARNING: linker: /system/bin/mysu has text relocations. This is wasting memory a
nd prevents 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:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:init:s0
x86_64:/ #
```

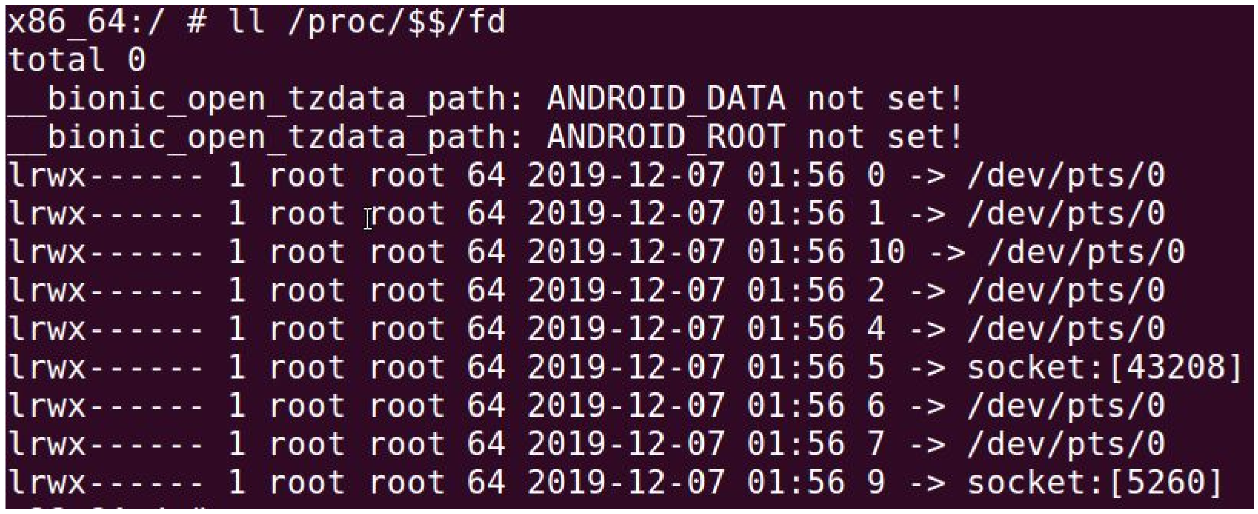
   We did it.

5. The Client process and the shell process do share the same standard file system.
   Fd of the normal user.



```
[12/06/19]seed@VM:~/.../SimpleSU$ adb shell
x86_64:/ $ ll /proc/$$/fd
total 0
lrwx------ 1 shell shell 64 2019-12-06 20:55 0 -> /dev/pts/0
lrwx------ 1 shell shell 64 2019-12-06 20:55 1 -> /dev/pts/0
lrwx------ 1 shell shell 64 2019-12-06 20:55 10 -> /dev/tty
lrwx------ 1 shell shell 64 2019-12-06 20:55 2 -> /dev/pts/0
x86_64:/ $
```

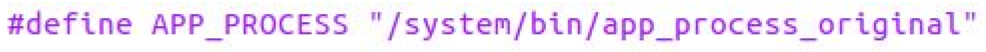
   Fd of the root which gets by ‘mysu’.



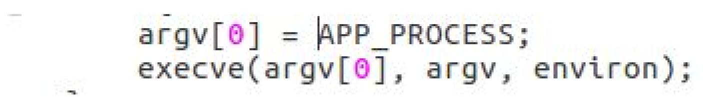
```
x86_64:/ # ll /proc/$$/fd
total 0
__bionic_open_tzdata_path: ANDROID_DATA not set!
__bionic_open_tzdata_path: ANDROID_ROOT not set!
lrwx------ 1 root root 64 2019-12-07 01:56 0 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 1 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 10 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 2 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 4 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 5 -> socket:[43208]
lrwx------ 1 root root 64 2019-12-07 01:56 6 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 7 -> /dev/pts/0
lrwx------ 1 root root 64 2019-12-07 01:56 9 -> socket:[5260]
```

Question & Answer about the code:

1. **The server launches the original app process binary.**
   This happened in the ‘mydaemon’. The daemon will be started in the Android boot procedure.

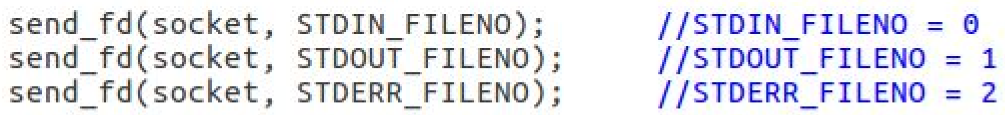


```
#define APP_PROCESS "/system/bin/app_process_original"
```



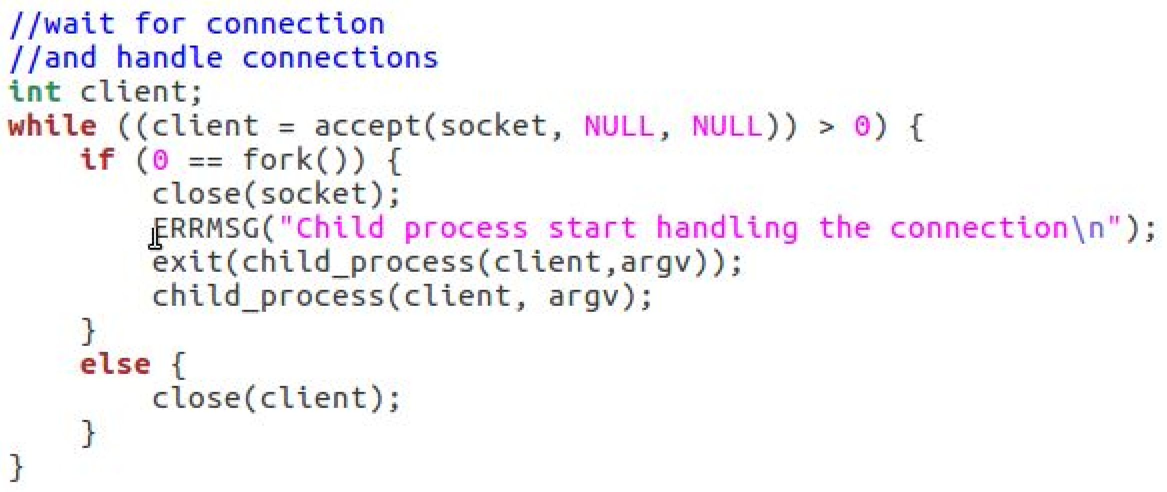
```
        argv[0] = APP_PROCESS;
        execve(argv[0], argv, environ);
```

2. **The client sends its FDs.**
   This happened in the ‘mysu.c’



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

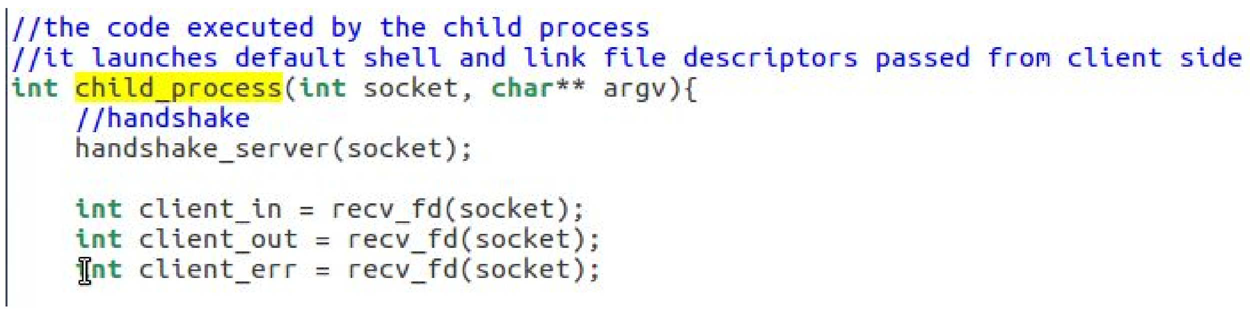
3. **Server forks to a child process.**
   This happened in the ‘mydaemon’.



```
//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);
    }
}
```

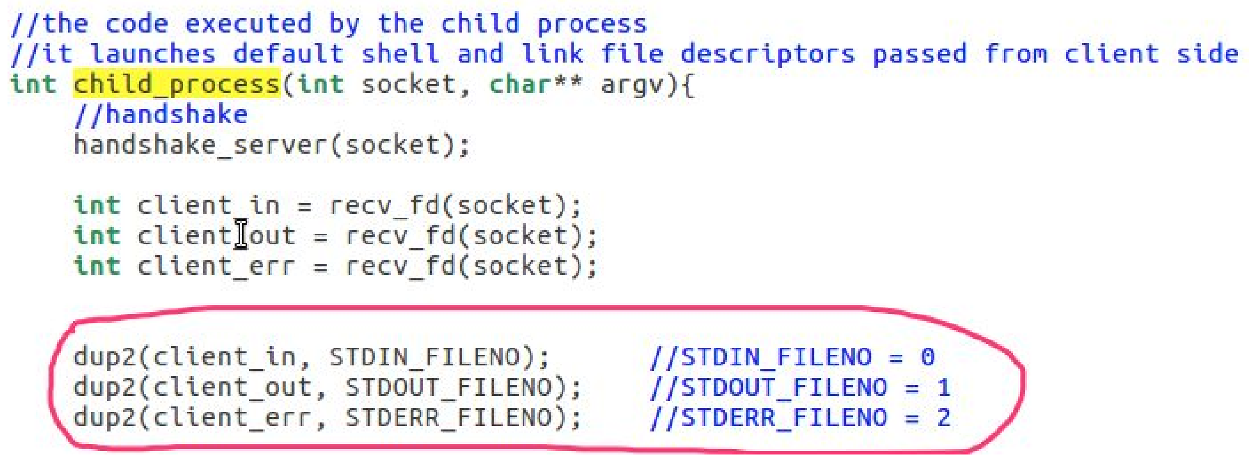
   When the server get a connection, it will forks a child process.

4. **The child process receives the client’s FDs.**
   In the ‘mydaemon’ file.



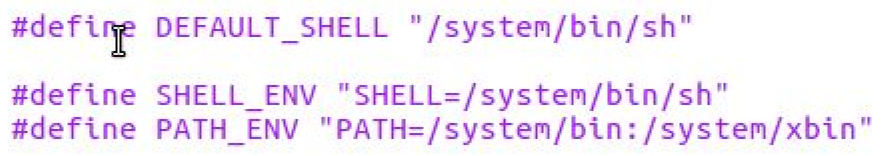
```
//the code executed by the child process
//it launches default shell and link file descriptors passed from client side
int child_process(int socket, char** argv){
    //handshake
    handshake_server(socket);

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

5. **The child process redirects its standard I/O FDs.**
   In the ‘mydaemon’ file, function ‘child_process’.

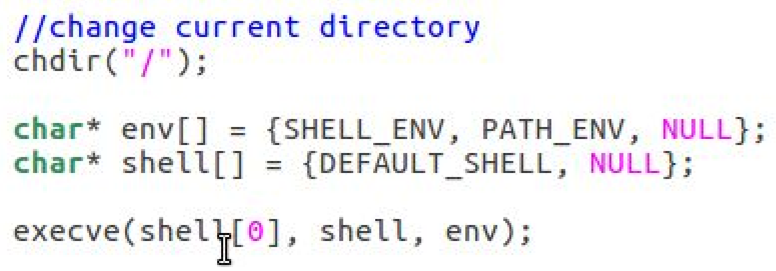


```
//the code executed by the child process
//it launches default shell and link file descriptors passed from client side
int child_process(int socket, char** argv){
    //handshake
    handshake_server(socket);

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

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

6. **The child process launches a root shell.**
   In the ‘mydaemon’ file, function ‘child_process’.



```
#define DEFAULT_SHELL "/system/bin/sh"

#define SHELL_ENV "SHELL=/system/bin/sh"
#define PATH_ENV "PATH=/system/bin:/system/xbin"
```



```
//change current directory
chdir("/");

char* env[] = {SHELL_ENV, PATH_ENV, NULL};
char* shell[] = {DEFAULT_SHELL, NULL};

execve(shell[0], shell, env);
```