

CSE643 Lab13
Yishi Lu
11/30/2018

Task1: Build a simple OTA package

```
Ubuntu 16.04.4 LTS recovery tty1

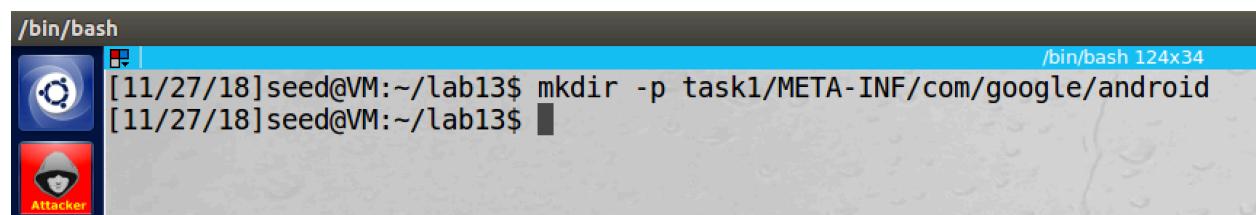
recovery login: seed
Password:
Last login: Fri May 18 15:17:56 EDT 2018 on tty1
Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.4.0-116-generic x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage
seed@recovery:~$ ifconfig
enp0s3    Link encap:Ethernet HWaddr 08:00:27:93:4a:6f
          inet addr:10.0.2.6 Bcast:10.0.2.255 Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe93:4a6f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:12 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:6437 (6.4 KB) TX bytes:2210 (2.2 KB)

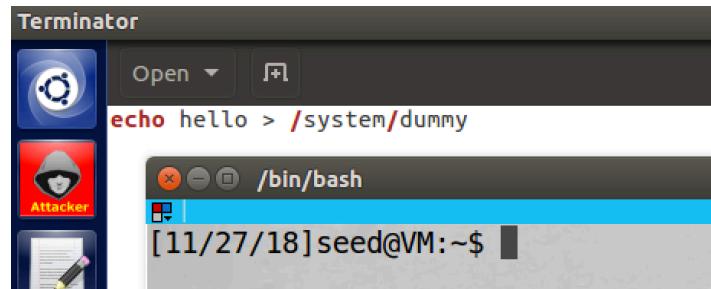
lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:160 errors:0 dropped:0 overruns:0 frame:0
          TX packets:160 errors:0 dropped:0 overruns:0 carrier:0
 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

seed@recovery:~$
```

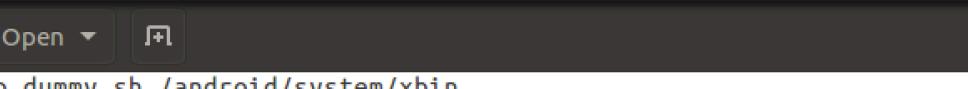
screenshot1, we login to recovery OS of the Android VM



screenshot2, we create the structure of the OTA package



screenshot3, we create dummy.sh file, and we add above content to it



The screenshot shows a terminal window titled 'bin/bash' with the command line '/bin/bash 80x23' at the top. The main text area displays the following exploit code:

```
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
```

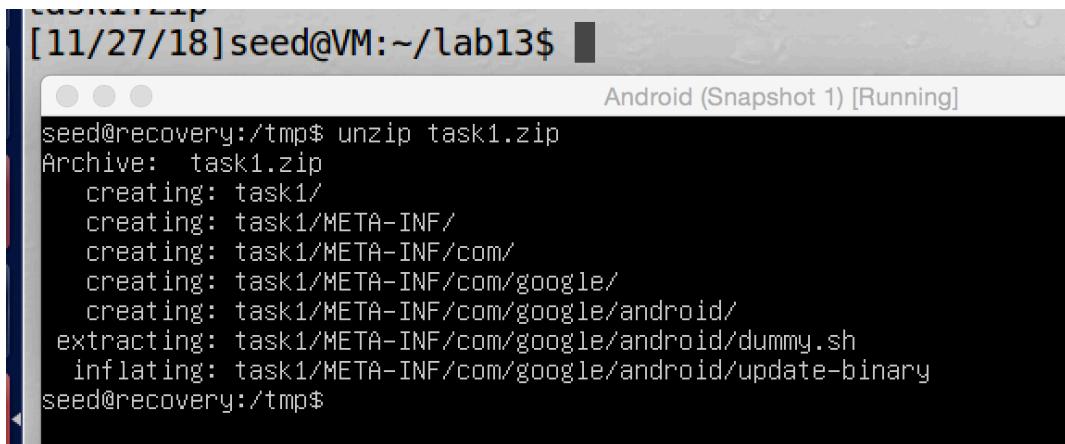
screenshot4, we create update-binary file, and we add above content to it. And we also run chmod a+x to make it executable (for update-binary files in later tasks, we also run this command, so we do not repeat again later)

```
[11/27/18]seed@VM:~/lab13$ zip -r task1.zip task1
adding: task1/ (stored 0%)
adding: task1/META-INF/ (stored 0%)
adding: task1/META-INF/com/ (stored 0%)
adding: task1/META-INF/com/google/ (stored 0%)
adding: task1/META-INF/com/google/android/ (stored 0%)
adding: task1/META-INF/com/google/android/dummy.sh (stored 0%)
adding: task1/META-INF/com/google/android/update-binary (deflated 44%)
[11/27/18]seed@VM:~/lab13$
```

screenshot5, we compress the OTA package into zip file

```
[11/27/18]seed@VM:~/lab13$ scp task1.zip seed@10.0.2.6:/tmp
The authenticity of host '10.0.2.6 (10.0.2.6)' can't be established.
ECDSA key fingerprint is SHA256:j27XN+nmbyA0avocrLHpQPiGRIZknAwMJI5y06vrsA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.2.6' (ECDSA) to the list of known hosts.
seed@10.0.2.6's password:
task1.zip
[11/27/18]seed@VM:~/lab13$
```

screenshot6, we send the OTA package to the Android recovery OS



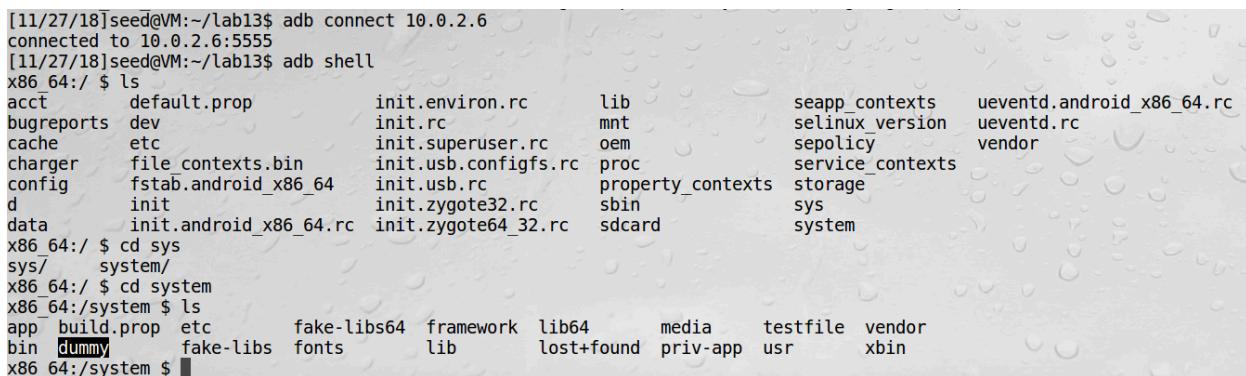
```
seed@recovery:/tmp$ unzip task1.zip
Archive: task1.zip
  creating: task1/
  creating: task1/META-INF/
  creating: task1/META-INF/com/
  creating: task1/META-INF/com/google/
  creating: task1/META-INF/com/google/android/
  extracting: task1/META-INF/com/google/android/dummy.sh
  inflating: task1/META-INF/com/google/android/update-binary
seed@recovery:/tmp$
```

screenshot7, on the recovery OS, we unzip the OTA package



```
seed@recovery:/tmp$ cd /tmp/task1/META-INF/com/google/android/
seed@recovery:/tmp/task1/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/task1/META-INF/com/google/android$ _
```

screenshot8, we run the update-binary file, then we reboot the Android VM



```
[11/27/18]seed@VM:~/lab13$ adb connect 10.0.2.6:5555
connected to 10.0.2.6:5555
[11/27/18]seed@VM:~/lab13$ adb shell
x86_64:/ $ ls
acct  default.prop  init.environ.rc  lib  seapp_contexts  ueventd.android_x86_64.rc
bugreports  dev  init.rc  mnt  selinux_version  ueventd.rc
cache  etc  init.superuser.rc  oem  sepolicy  vendor
charger  file_contexts.bin  init.usb.configfs.rc  proc  service_contexts
config  fstab.android_x86_64  init.usb.rc  property_contexts  storage
d  init  init.zygote32.rc  sbin  sys
data  init.android_x86_64.rc  init.zygote64_32.rc  sdcard  system
x86_64:/ $ cd sys
sys/  system/
x86_64:/ $ cd system
x86_64:/system $ ls
app  build.prop  etc  fake-lib64  framework  lib64  media  testfile  vendor
bin  dummy  fake-libs  fonts  lib  lost+found  priv-app  usr  xbin
x86_64:/system $
```

screenshot9, we connect Android VM from the Ubuntu VM by adb command. In the system folder, we see dummy file is created. So our attack is successful

Observation and Explanation:

In this lab, we will build a OTA package, and this package contains our code. Once we install the package in the recovery OS, a dummy file will be added to the /system folder of the Android OS. We divide this task into three steps.

The first step is to write the update script. We create two files; one is called dummy.sh. This file contains our code, we want it to be run when Android boots up with root privilege, and then it will create a dummy file under system folder. Another file is the update-binary file; in this file, we add shell command of copying dummy.sh to /system/xbin folder on Android OS, and then we make dummy.sh executable. Finally, we insert a command in file init.sh. This file

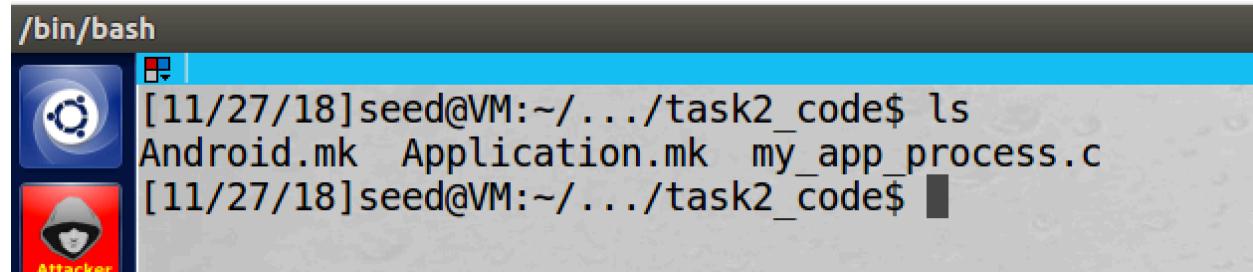
will be run with root privilege when Android system booting. So if we place command to run dummy.sh in this file before it finish, dummy.sh can be run with root privilege.

The second step is to build the OTA package. The structure of the OTA package is shown in screenshot2, and we place dummy.sh and update-binary files all in folder android.

The last step is to run the OTA package on the recovery OS, then after we rebooting to Android system, the dummy file should be added in folder /system.

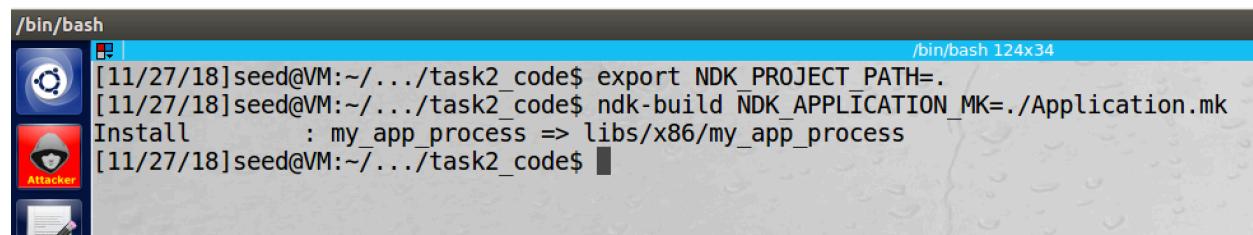
As screenshot1 shows, we first login to recover OS. Then we create the OTA package (screenshot2). As screenshot 3 and 4 show, we build dummy.sh and update-binary files, and we put them under /android. Then we zip the OTA package and send it to the recovery OS (screenshot 5 and 6). On the recovery OS, we unzip the OTA package, and we run the update-binary file (screenshot7 and 8). Then we reboot the Android OS. To check if our attack is successful or not, we go back to the Ubuntu VM, and we connect to the Android OS by adb command; and then we run a shell program on Android by command adb shell. As screenshot9 shows, the dummy file is added under /system. So our attack is successful.

Task2: Inject Code via app-process



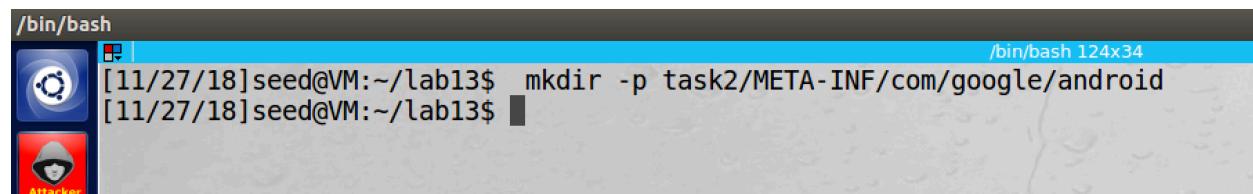
```
/bin/bash
[11/27/18]seed@VM:~/.../task2_code$ ls
Android.mk  Application.mk  my_app_process.c
[11/27/18]seed@VM:~/.../task2_code$
```

screenshot1, we copy code Android.k, Application.mk, and my_app_process.c from lab description



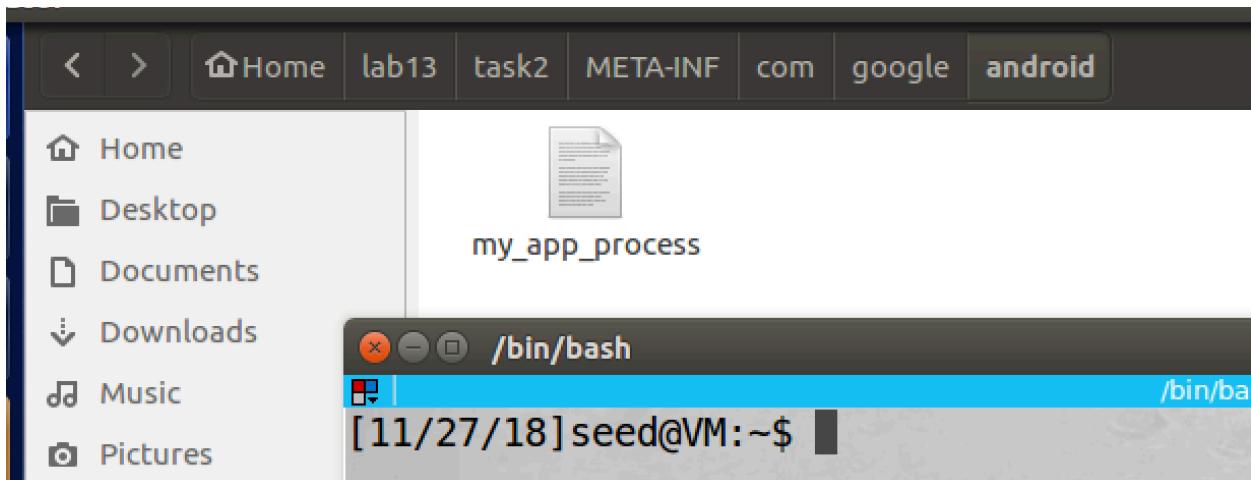
```
/bin/bash
[11/27/18]seed@VM:~/.../task2_code$ export NDK_PROJECT_PATH=.
[11/27/18]seed@VM:~/.../task2_code$ ndk-build NDK_APPLICATION_MK=./Application.mk
Install : my_app_process => libs/x86/my_app_process
[11/27/18]seed@VM:~/.../task2_code$
```

screenshot2, we compile our program by NDK successfully

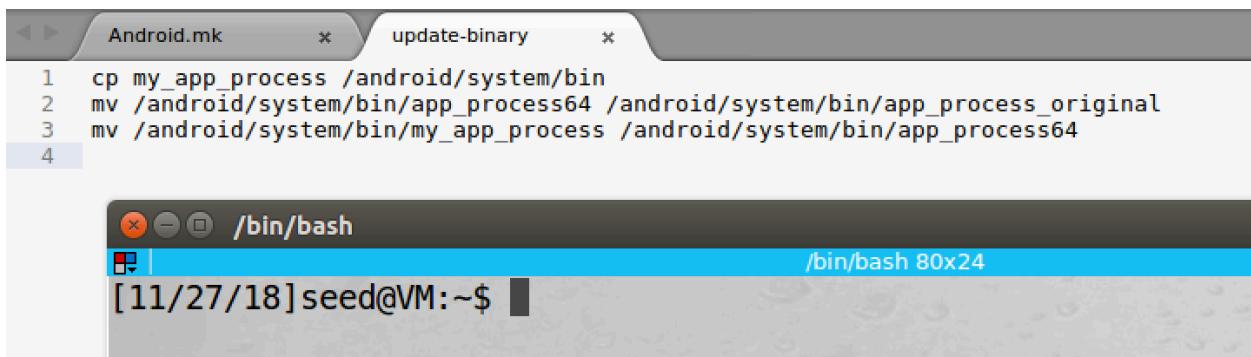


```
/bin/bash
[11/27/18]seed@VM:~/lab13$ mkdir -p task2/META-INF/com/google/android
[11/27/18]seed@VM:~/lab13$
```

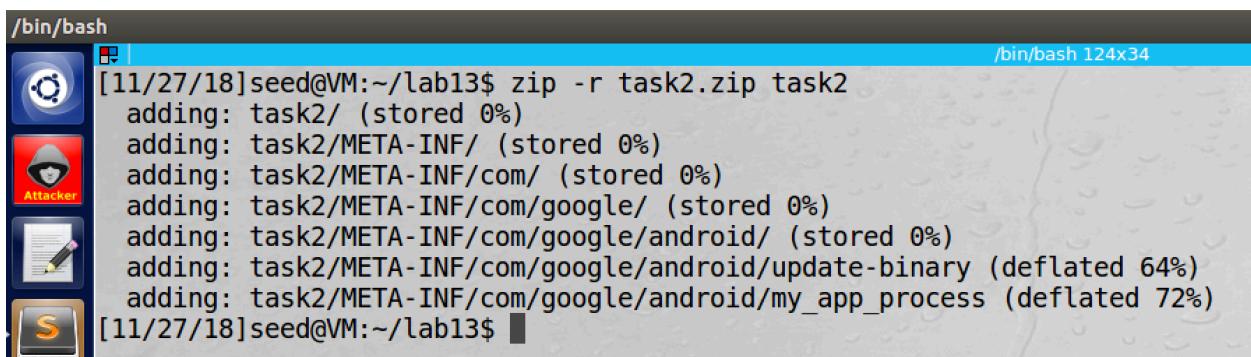
screenshot3, we create OTA package structure, it is similar as the one in task1



screenshot4, we put the compiled file in folder /android of OTA package



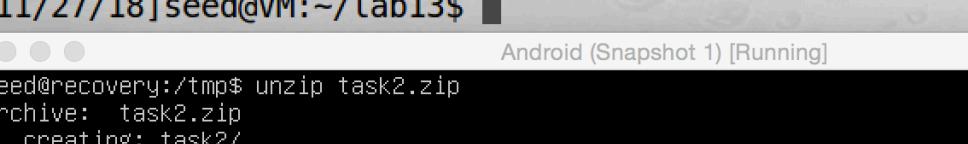
screenshot5, we create the update-binary file in the /android of OTA package



screenshot6, we zip the task2 OTA package



screenshot7, we send it to the recovery OS



[11/27/18] seed@VM:~/lab13\$

Android (Snapshot 1) [Running]

```
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/my_app_process
seed@recovery:/tmp$ cd /tmp/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$ sudo reboot
```

screenshot8, on the recovery OS, we unzip the OTA package, and we run the update-binary file, then we reboot the VM to enter android OS

```
[11/27/18]seed@VM:~/lab13$ adb devices
List of devices attached

[11/27/18]seed@VM:~/lab13$ adb connect 10.0.2.6
connected to 10.0.2.6:5555
[11/27/18]seed@VM:~/lab13$ adb shell
x86_64:/ $ ls
acct      default.prop      init.environ.rc      lib      seapp_contexts  ueventd.android_x86_64.rc
bugreports  dev      init.rc      mnt      selinux_version ueventd.rc
cache      etc      init.superuser.rc      oem      sepolicy
charger    file_contexts.bin  init.usb.configfs.rc  proc      service_contexts
config      fstab.android_x86_64  init.usb.rc      property_contexts  storage
data      init.android_x86_64.rc  init.zygote32.rc  sbin      sys
                     init.zygote64_32.rc  sdcard      system

x86_64:/ $ cd sys
sys/
x86_64:/ $ cd system
x86_64:/system $ ls
app  build.prop  etc      fake-lib64  framework  lib64      media      usr      xbin
bin  dummy2      fake-libs  fonts      lib      lost+found  priv-app  vendor
x86_64:/system $
```

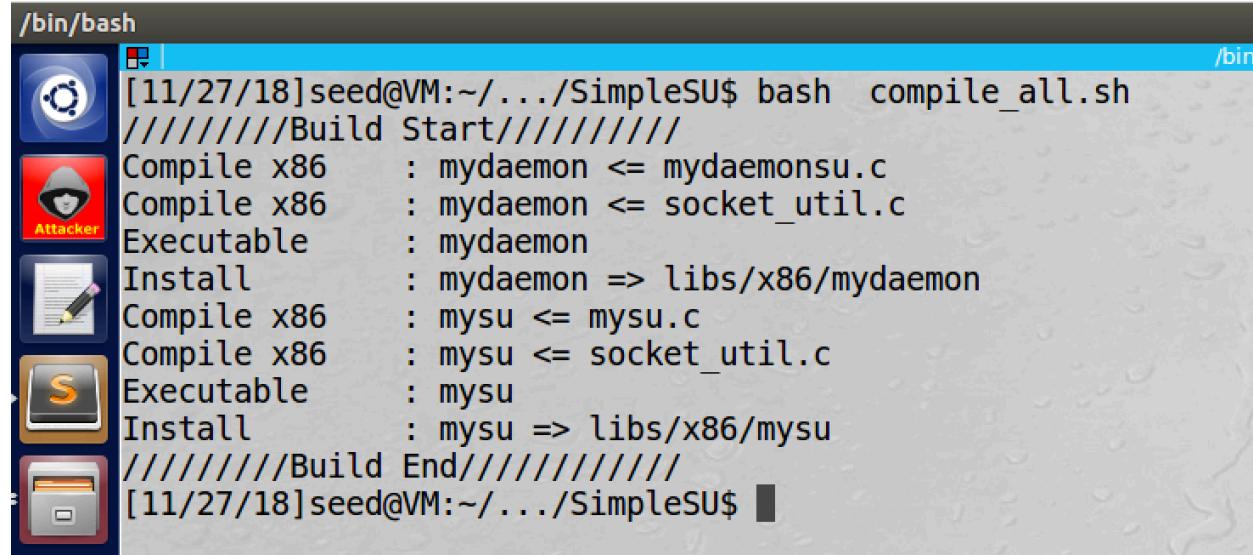
screenshot9, we still use adb command to access the system file of the android OS, and we see dummy2 is added in the /system folder, so our attack is successful

Observation and Explanation:

In this task, we perform same attack as task1, but this time we use app_process approach. When android OS is booting, a sequence of procedures will be run. There are three main phases. First, the kernel phase, in this phase, Android kernel will be loaded to initialize the system. The next phase is the Init process. It is the first user-space process, and it running with root privilege. The third phase is the Zygote process. In this phase, a daemon called Zygote will be run, and it will execute app_process binary, and it also run with root privilege. This is our attack point in this task. Our goal is to modify app_process, so our code can be run with root privilege. So we first need to create the app_process binary, we copy the code from lab description. To compile the app_process program, we need to use NDK. To use NDK, we need to create two files, Application.mk and Android.mk, we also get them from lab description (screenshot1). After we compile the app_process program, we can find the compiled binary file in the folder /lib/x86 (screenshot2). Then we need to build the OTA package and update-binary

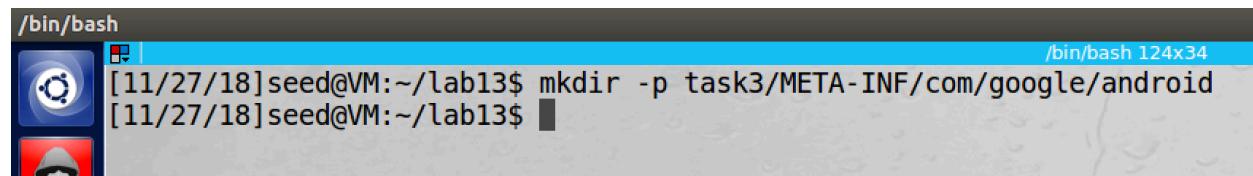
file. OTA package has same structure as the one task1, so we do not repeat here (screenshot3). The update-binary file is different, in this task, we need to do two things; first we need to move our app_process file into /android/system/bin folder. Then we need to change the name of the original app_process file to other name (app_process_original), then we also change the name of our app_process file to app_process64 (screenshot5). After we contracture the update-binary file, we still put it in the folder /android of OTA package; moreover, we put our app_process file in the same folder as well (screenshot4). Now, we have everything we need. Then we follow step in task1 to install the OTA package in the recover OS (screenshot6 and 7). After we run the update-binary file on the recovery OS, we rebooting the Android OS (screenshot8). And we also use adb command on Ubuntu VM to check the /system folder of the Android OS, and we see dummy2 is added in the /system; thus our attack is successful (screenshot9).

Task3: Implement SimpleSU for Getting Root Shell



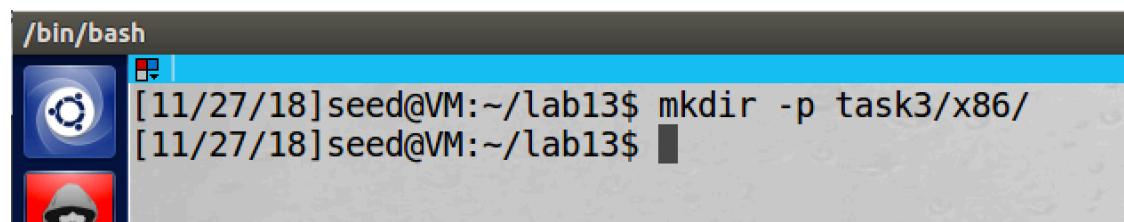
```
[11/27/18]seed@VM:~/.../SimpleSU$ bash compile_all.sh
/////////Build Start/////////
Compile x86      : mydaemon <= mydaemonsu.c
Compile x86      : mydaemon <= socket_util.c
Executable       : mydaemon
Install          : mydaemon => libs/x86/mydaemon
Compile x86      : mysu <= mysu.c
Compile x86      : mysu <= socket_util.c
Executable       : mysu
Install          : mysu => libs/x86/mysu
/////////Build End/////////
[11/27/18]seed@VM:~/.../SimpleSU$
```

screenshot1, we get the SimpleSU file form lab website, then we run bash compile_all.sh to compile the program, and then we get two compiled binary files mydaemon and mysu.



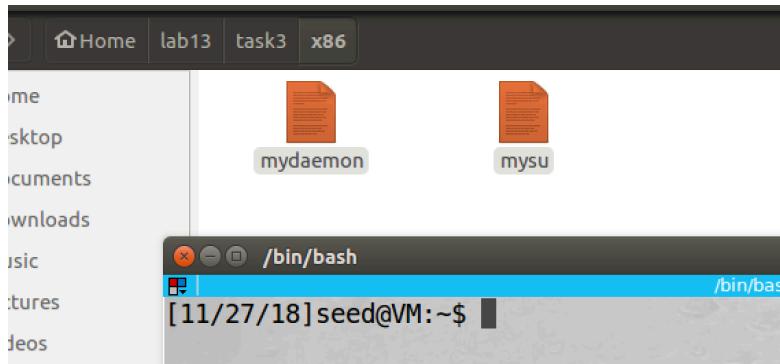
```
[11/27/18]seed@VM:~/lab13$ mkdir -p task3/META-INF/com/google/android
[11/27/18]seed@VM:~/lab13$
```

screenshot2, we create OTA structure for task3

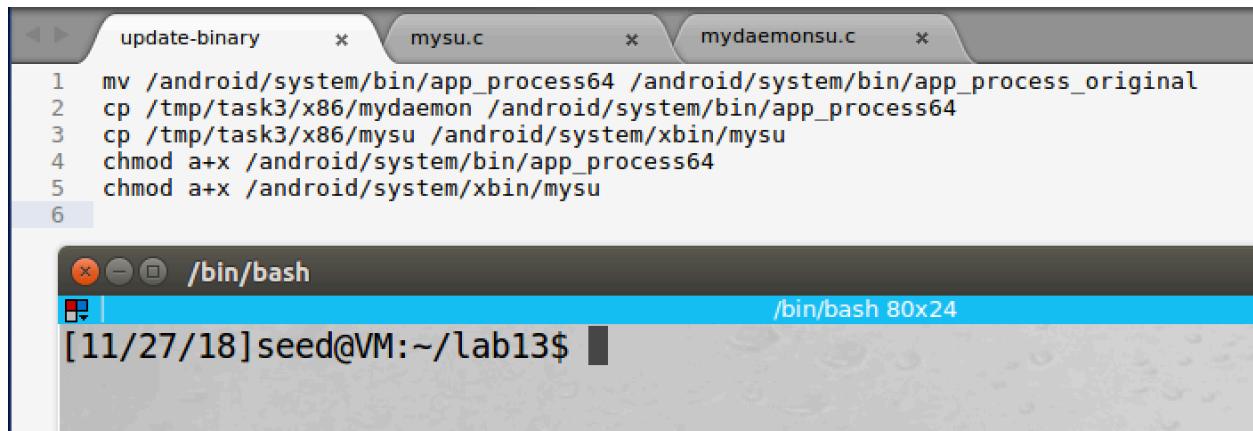


```
[11/27/18]seed@VM:~/lab13$ mkdir -p task3/x86/
[11/27/18]seed@VM:~/lab13$
```

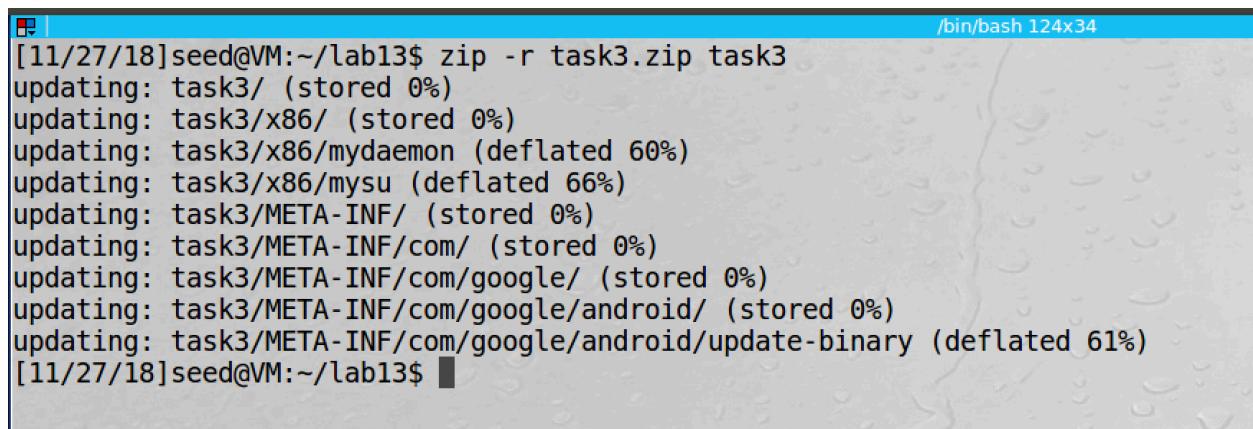
screenshot3, we create /x86 folder under the OTA structure



screenshot4, we move mydaemon and mysu files into /x86 folder



screenshot5, we construct the update-binary file, and we put it in the /android of OTA package



screenshot6, we compress the OTA package



screenshot7, we send it to recovery OS

VM1 (Snapshot 1) [Running]

/bin/bash

```
[11/27/18]seed@VM:~/lab13$ scp task3.zip seed@10.0.2.6:/tmp
seed@10.0.2.6's password:
task3.zip
[11/27/18]seed@VM:~/lab13$
```

Android (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:~$ cd /tmp
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
  inflating: task3/META-INF/com/google/android/mydaemon
  inflating: task3/META-INF/com/google/android/mysu
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
```

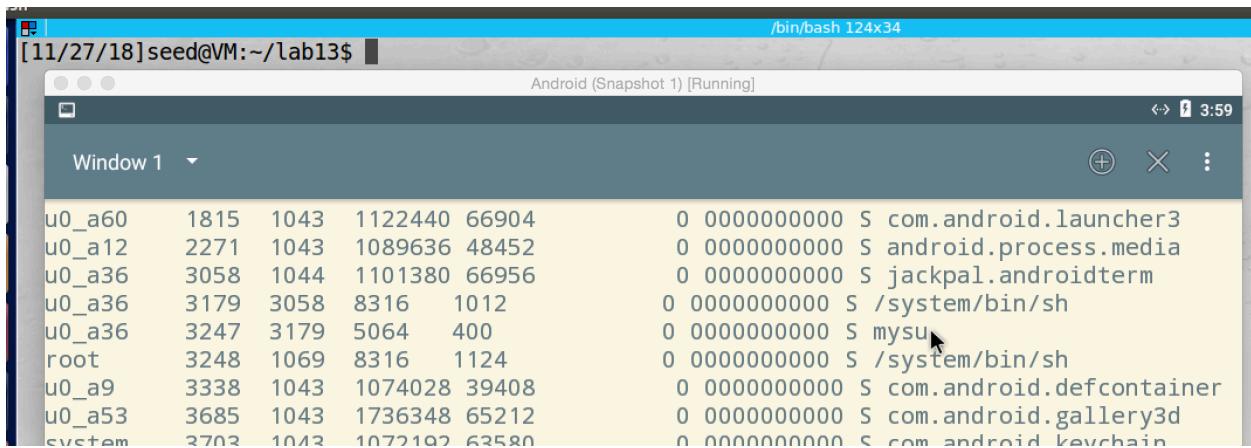
screenshot8, we run the update-binary file on recovery OS, then we reboot the system

[11/27/18]seed@VM:~/lab13\$

Android (Snapshot 1) [Running]

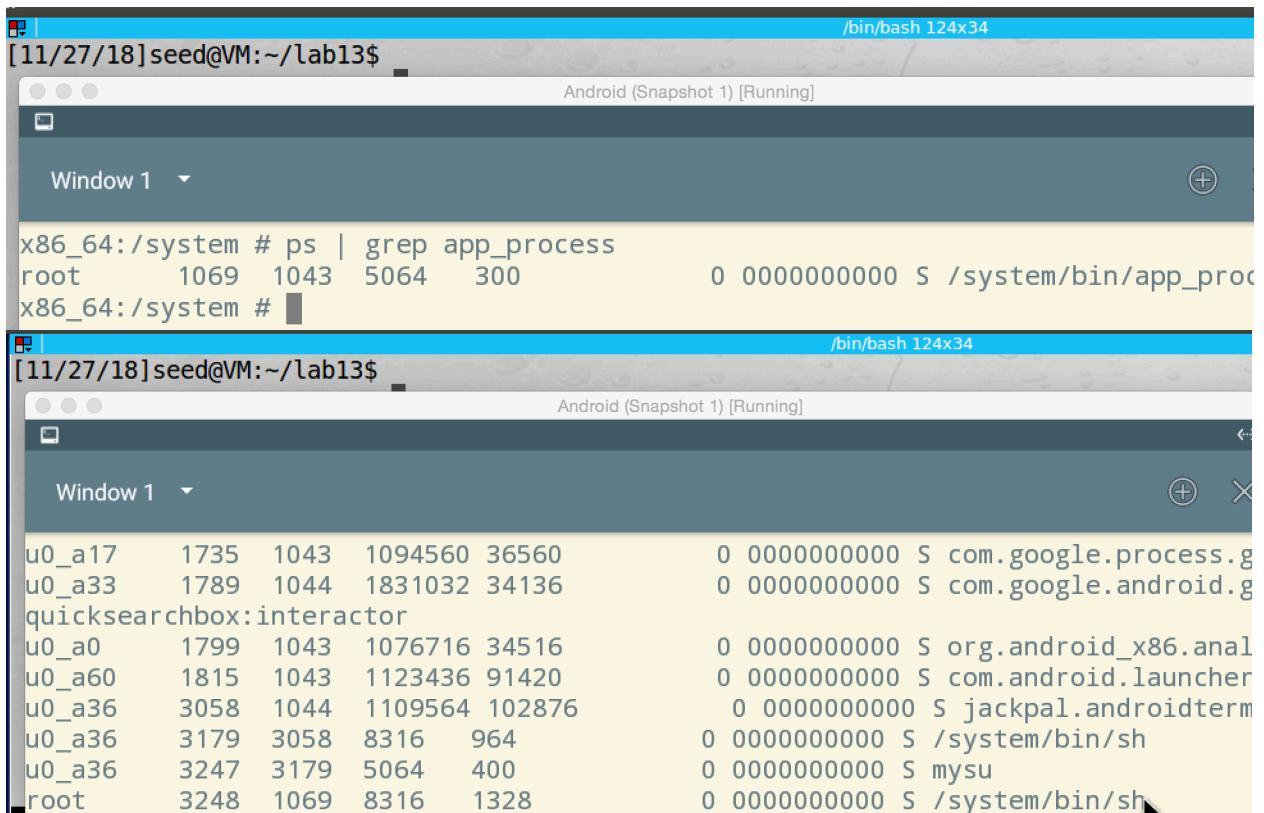
```
Window 1  ▾
x86_64:/ $ id
uid=10036(u0_a36) gid=10036(u0_a36) groups=10036(u0_a36),3003(inet),9997(everybody),5
0036(all_a36) context=u:r:untrusted_app:s0:c512,c768
x86_64:/ $ mysu
WARNING: linker: /system/xbin/mysu has text relocations. This is wasting memory and p
revents security hardening. Please 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:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:init:s0
x86_64:/ #
```

screenshot9, we run mysu on the terminal, and we get a root shell



```
[11/27/18]seed@VM:~/lab13$ ps
Android (Snapshot 1) [Running]
Window 1
u0_a60 1815 1043 1122440 66904 0 00000000000 S com.android.launcher3
u0_a12 2271 1043 1089636 48452 0 00000000000 S android.process.media
u0_a36 3058 1044 1101380 66956 0 00000000000 S jackpal.androidterm
u0_a36 3179 3058 8316 1012 0 00000000000 S /system/bin/sh
u0_a36 3247 3179 5064 400 0 00000000000 S mysu
root 3248 1069 8316 1124 0 00000000000 S /system/bin/sh
u0_a9 3338 1043 1074028 39408 0 00000000000 S com.android.defcontainer
u0_a53 3685 1043 1736348 65212 0 00000000000 S com.android.gallery3d
system 3702 1043 1072102 63580 0 00000000000 S com.android.keychain
```

screenshot10, we get PID of mysu which is 3247

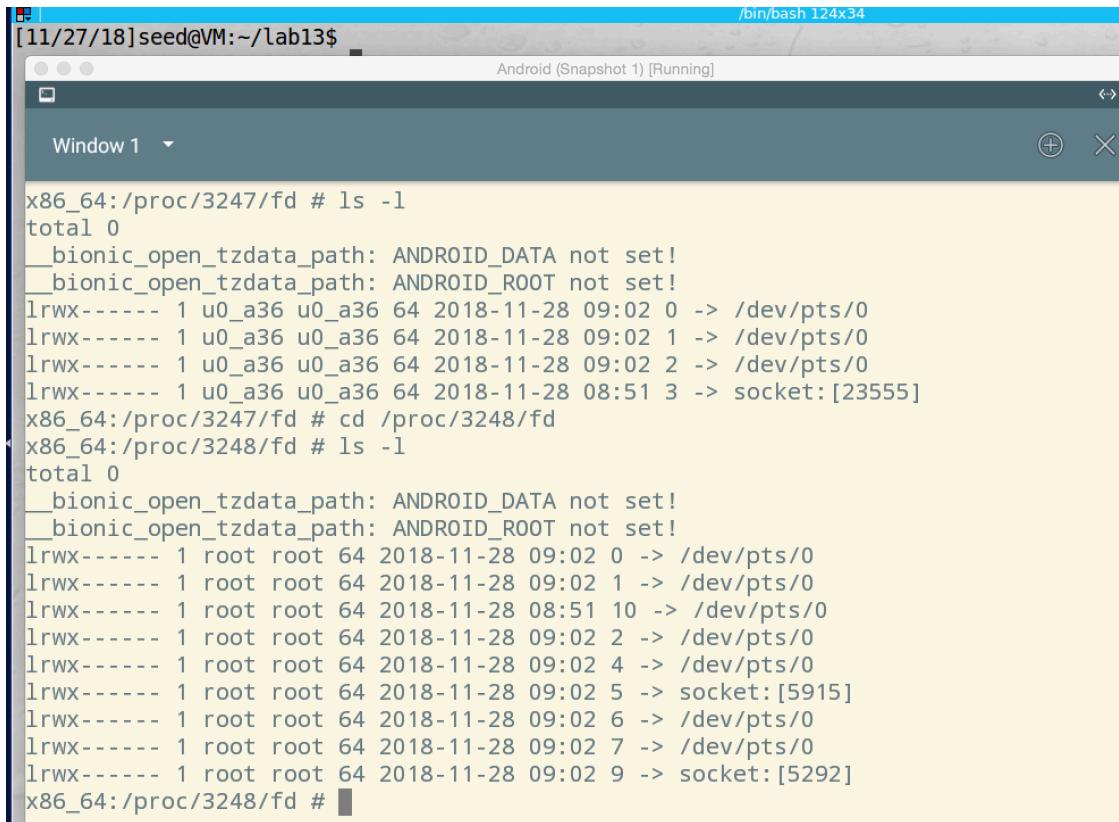


```
[11/27/18]seed@VM:~/lab13$ ps | grep app_process
root 1069 1043 5064 300 0 00000000000 S /system/bin/app_process
x86_64:/system #
```




```
[11/27/18]seed@VM:~/lab13$ ps
Android (Snapshot 1) [Running]
Window 1
u0_a17 1735 1043 1094560 36560 0 00000000000 S com.google.process.g
u0_a33 1789 1044 1831032 34136 0 00000000000 S com.google.android.g
quicksearchbox:interactor
u0_a0 1799 1043 1076716 34516 0 00000000000 S org.android_x86.anal
u0_a60 1815 1043 1123436 91420 0 00000000000 S com.android.launcher
u0_a36 3058 1044 1109564 102876 0 00000000000 S jackpal.androidterm
u0_a36 3179 3058 8316 964 0 00000000000 S /system/bin/sh
u0_a36 3247 3179 5064 400 0 00000000000 S mysu
root 3248 1069 8316 1328 0 00000000000 S /system/bin/sh
```

screenshot11, to find the child process PID, we first find server process PID which is 1069. Then we run command ps to find process with PPID 1069, which is 3248, so this is the child process



The screenshot shows a terminal window titled 'Window 1' with the command '/bin/bash 124x34' at the top. The terminal is running on an Android device, as indicated by the title 'Android (Snapshot 1) [Running]'. The terminal output is as follows:

```
x86_64:/proc/3247/fd # ls -l
total 0
_bionic_open_tzdata_path: ANDROID_DATA not set!
_bionic_open_tzdata_path: ANDROID_ROOT not set!
lwx----- 1 u0_a36 u0_a36 64 2018-11-28 09:02 0 -> /dev/pts/0
lwx----- 1 u0_a36 u0_a36 64 2018-11-28 09:02 1 -> /dev/pts/0
lwx----- 1 u0_a36 u0_a36 64 2018-11-28 09:02 2 -> /dev/pts/0
lwx----- 1 u0_a36 u0_a36 64 2018-11-28 08:51 3 -> socket:[23555]
x86_64:/proc/3247/fd # cd /proc/3248/fd
x86_64:/proc/3248/fd # ls -l
total 0
_bionic_open_tzdata_path: ANDROID_DATA not set!
_bionic_open_tzdata_path: ANDROID_ROOT not set!
lwx----- 1 root root 64 2018-11-28 09:02 0 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 1 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 08:51 10 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 2 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 4 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 5 -> socket:[5915]
lwx----- 1 root root 64 2018-11-28 09:02 6 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 7 -> /dev/pts/0
lwx----- 1 root root 64 2018-11-28 09:02 9 -> socket:[5292]
x86_64:/proc/3248/fd #
```

screenshot12, we get the file descriptor of mysu and the child process

Observation and Explanation:

In this task, we show how to gain a root shell by SimpleSU on Android OS. Because on Android OS, there is no Set-UID program, we cannot gain a root shell by it. So we use another approach, we run a daemon program when the OS is booting, so the daemon program run with root privilege. Then we run a client program to make a request to the daemon program, then the daemon program create a shell process and give the control to us. Then we get a root shell. However, there is a problem which we need to solve. The shell process generated by the daemon program, and it inherits output device, input device, and error device from its parent, so we cannot control these devices. Our solution is to give the input, output, and error device of the client program to the shell process and make these devices also become devices of the shell process. Then user can use these devices to input to the shell process and receive result (print) from shell process. Moreover, in Linux, each process use file descriptors (FDs) 0, 1, 2 to represent input, output, error devices; and it also allow process send its FDs to other process by Unix Domain Socket. So the whole process is following: when the Android OS is booting, a daemon program is running (server). After client program connect to it, the server will fork a shell process, and the shell process inherits input, output, and error devices from the server. Then the client process sends its FDs to the shell process by Unix Domain Socket, and then the shell process uses these FDs as its stdin, stdout, and stderr. Now the client process can have the full control of the shell process. Because the shell process is running with root privilege, the client process actually run with root privilege as well.

As screenshot1 shows, we get the server and client program from lab website, and we compile them to get two binary file. Then we construct the OTA package, and we also add a new folder /x86, we put these two compiled files into this folder (screenshot 2, 3, 4). Then we construct the update-binary file (screenshot5). when this file run, the original app_process file will be renamed to app_process_original. Then we also put the mydaemon file into the /system/bin with name app_process64. We also put mysu file into /system/xbin. And we also set them to be runnable by command chmod a+x. Now we constructed the OTA package, so we compress it, and send it to the recovery OS (screenshot 6, 7). On the recovery OS, we unzip the OTA package, and then we run the update-binary file, then we also reboot the system (screenshot8). Then we enter the Android OS, and we run terminal app. We first run command id to check the user id, which is a normal user. Then we run the client program mysu to connect to the server, and we get a root shell. To prove this, we run command id again, and it shows root (screenshot9). To find the PID of process mysu, we run command ps, and we see its PID is 3247 (screenshot10). We also need to find the child process ID. We know that the server process has name app_process64, so we get its PID; then we run command ps again to find process with PPID 1069. We find it, the child process has PID 3248 (screenshot11). Then we run command ls -l to get their FDs. As screenshot12 shows, we get all FDs of child process and client process. Their FD 0, 1, 2 point to same place, which is terminal device /dev/pts/0; moreover, FD 4, 6, and 7 of child process are also point to /dev/pts/0. So they share the same stand in, stand out, and stand error devices.

Server launches the original app_process binary:

File: mydaemonsu.c, Function: main(), Line#: 255

Client sends its FDs:

File: mysu.c, Function: connect_daemon(), Line#: 112-114

Server forks to a child process:

File: mydaemonsu.c, Function: run_daemon(), Line#: 189-193

Child process receives client's FDs

File: mydaemonsu.c, Function: child_process(), Line#: 147-149

Child process redirects its standard I/O FDs

File: mydaemonsu.c, Function: child_process(), Line#: 152-154

Child process launches a root shell

File: mydaemonsu.c, Function: child_process (), Line#: 162