운영체제보안 과제 #3

Android Rooting

32151648 32151288 32154022

박동학 김현수 장현웅

목차

Android Rooting 실습 개요

안드로이드 OS는 사용자가 기기의 주인이더라도 기본적으로 Root 권한이 아닌 일반 사용자의 권한으로 사용하도록 설정되어 있다. 이는 일반 사용자는 신경을 쓰지 않는 부분이다. 하지만 기기를 Customize하고 싶다면 Root 권한을 얻어야 한다. 이 때 기기의 Root 권한을 얻는 것을 Rooting이라고 한다.

이번 실습의 목표는 이러한 Android 기기에서의 Root 권한을 얻는 Rooting이다. Android기기의 Root 권한을 가지게 되며 일반 사용자의 권한보다 많은 권한을 가지게 되고 이를 통해서 많은 행위를 수행 할 수 있다. 안드로이드 기기는 기본적으로 사전에 시스템 App들이 설치되어 오는데, 이 App들을 bloatware라고 한다. 이 App들은 보호된 위치에 설치되어 있어서, Root 권한을 통해서 삭제 하여야 한다. 또한 Rooting을 통해 기기를 Customizing 할 수 있고, 새로운 특징 추가 등 시스템에 변화를 줄 수 있다.

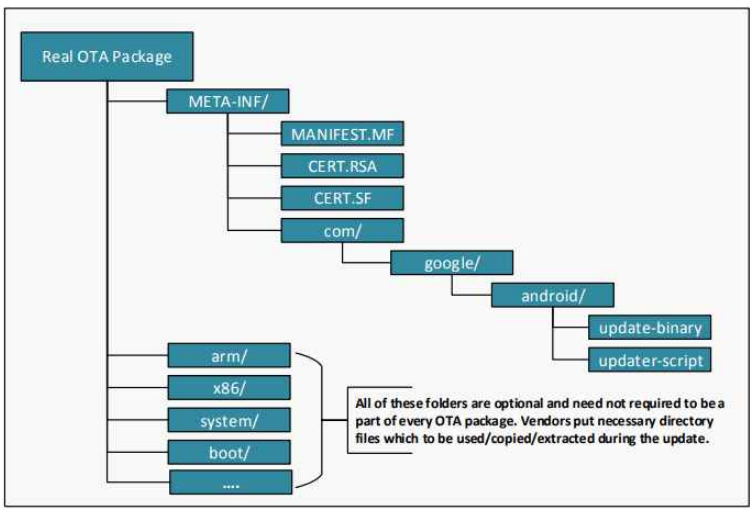
이러한 목적들을 위해서 안드로이드 Rooting 하는 방법은 다양하지만, 이번 실습에서는 Recovery OS를 교체하는 방식을 이용한다.

Recovery OS 교체하기

안드로이드 기기는 Dual Boot로 구성되어 있는데, 기본적으로 평소에 사용하는 OS와 업데이트할 때 사용하는 Recovery OS(Second OS)로 이루어져 있다. 기존에 제조사가 만들어서 최초로 존재하는 Recovery OS를 Stock Recovery OS라고 하는데, 이번 실습에 우리는 이 Recovery OS를 통째로 바꿔서 Rooting을 한다. 우리가 새로 교체할 Recovery OS는 Custom Recovery OS라 하고, 기존에 Stock Recovery OS에 존재하는 인증 부분이 없기 때문에, OTA 패키지를 삽입 할 수 있다. 이를 통해 안드로이드의 파티션을 변경할 수 있다.

OTA (Over The Air) 란?

여기서 OTA는 안드로이드 OS를 업데이트할 때 사용하는 표준 기술이다. OTA 패키지는 단순한 zip 파일인데, 구조는 아래와 같다.



우리가 실습 때 중요한 폴더는 META-INF 이다, 실습에 있어서 중요한 파일인 update-binary와 updater-script가 위치한다. update-binary 파일은 Recovery OS가 OTA 업데이트를 적용할 때 실행되는데 이 파일은 updater-script를 로드하고 실행 한다. updater-script 파일은 update-binary에 의해 실행되는 스크립트이며 업데이트를 하기 위해 Edify라는 스크립트 언어로 작성되었다. OTA 패키지는 인증을 한 후에, Recovery OS에서 /tmp 디렉토리로 OTA 패키지를 unzip하여 update-binary를 실행하여 업데이트를 진행한다. update-binary 실행 후 Recovery OS는 실행 로그를 /cache/recovery 디렉토리에 복사하고, 안드로이드 OS를 재부팅 한다. 재부팅 후에 안드로이드 OS는 실행로그에 접근하여 업데이트를 한다.

실습(Rooting) 목표

이번 실습의 목표는 안드로이드의 Rooting이며 목표를 이루는데 있어 단계를 나누어 풀어나갈 것이다. 다음과 같이 3단계의 Task로 나누어 실습을 진행한다.

1. Recovery OS에서 Android OS로 프로그램을 주입할 것인가?
2. Root 권한으로 주입한 프로그램을 자동으로 실행시킬 것인가?
3. Root Shell을 부여할 수 있는 프로그램을 어떻게 작성할 것인가?

Task 1 : Build a simple OTA package

1.1 실습(Task1) 개요

실습을 진행 하기 앞서 실제 안드로이드 기기는 Stock Recovery OS를 바꾸는 것을 예방하기 위해 다른 접근 제어가 있는데 이를 Bootloader라고 한다. Bootloader는 시스템이 켜질 때 OS 및 시스템의 소프트웨어를 Load하는 Low Level 코드이다. Bootloader가 잠겨있을 때(Locked)는 설치되어 있는 OS를 Control 하지 못하고, OS를 바로 Load한다. 하지만 Bootloader가 unlocked 상태에서는 기존의 OS를 바꿀 수 있고, 새로운 OS를 추가할 수도 있다. 실습함에 있어서 기기의 Bootloader가 unlock될 수 있고, Stock Recovery OS가 변경될 수 있다고 가정하고 진행한다.

1.2 실습 목표

Task1의 목적은 Recovery OS를 이용하여 Android OS에 프로그램을 주입하는 것이다. 실습 목적을 이루기 위해서는 다음과 같은 단계가 필요하다.

1. update-binary 스크립트 파일을 작성한다.
2. OTA 패키지를 생성한다.
3. OTA 패키지를 실행한다.

먼저 update-binary 스크립트 파일을 작성해야 한다. OTA 패키지에서 Recovery OS는 최초로 update-binary 파일을 실행시킨다. 우리가 만들 update-binary 파일의 목적은 다음과 같다.

1. 안드로이드 OS에 dummy.sh 파일을 주입한다.

2. 안드로이드 OS의 구성 파일을 변경한다. 변경함으로써 우리가 주입한 dummy.sh 파일이 자동으로 실행되게 만든다.

그러므로 먼저, dummy.sh 파일의 위치와, Permission 설정을 어떻게 할지 생각해야 한다. Recovery OS에서 /android 디렉토리에 이미 mount된 안드로이드 파티션에 파일을 위치시킨다. 다음으로, 안드로이드가 부팅될 때 dummy.sh가 자동으로 수행되어야 하는데, 이 때 root 권한을 가지고 실행이 되어야한다. Linux를 이용하여 접근을 하는데, 안드로이드는 Linux OS 위에 만들어져 있다. 부팅될 때 필요한 Daemon 프로세스들을 시작하고, 시스템 초기화를 수행하는 Linux를 먼저 켜서 기반으로 하여 부팅한다. 부팅 과

정에 Root 권한을 이용하여 초기화 과정의 일부인 /system/etc/init.sh 파일을 실행시킨다. 따라서 우리는 init.sh 파일에 명령어를 삽입하여 Root 권한을 가지고 dummy.sh 파일을 실행시키고자 한다. Init.sh 파일에 명령어를 삽입하는 다음의 sed 명령어를 이용하여 목적을 달성한다.

```
sed -i "/return 0/i/ system/xbin/dummy.sh" /android/system/etc/init.sh
```

다음 과정으로 OTA 패키지를 만들면 되는데, 위의 그림과 같이 디렉토리 구조를 만들고 우리가 사용하고자 하는 파일들을 넣으면 된다. 이 task를 수행하기 위해서는 update-binary의 위치와 update-binary의 명령어와 dummy.sh 파일의 경로를 맞춰 OTA 패키지를 만들면 된다. Zip 명령어를 통해 zip 파일을 만든다.

```
zip -r my_ota.zip ./
```

마지막 과정으로 OTA 패키지를 자동으로 실행시킬 Recovery OS에 제공한다. 우리의 실습 환경은 Ubuntu를 Recovery OS로 사용하고 있기 때문에, 제대로 된 Recovery의 기능이 있지 않으므로, 모방해서 진행해야 한다. 즉 우리는 정석대로 OTA 패키지를 unzip 명령어를 통해 추출하고, MEAT-INF/com/google/android 폴더에 있는 update-binary 파일을 찾아서 실행시킨다. 위의 과정대로 실습을 진행하면 Android가 업데이트가 됐을 것이며, 실습 목표인 /system 폴더 안에 dummy 파일이 생성된다.

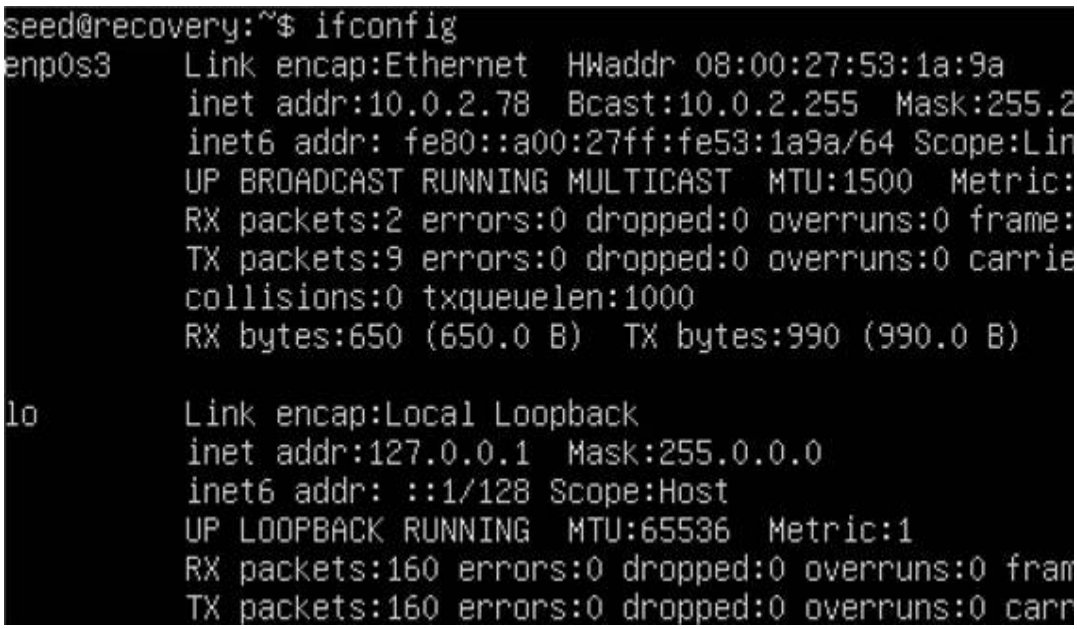
1.3 실습 수행 결과

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

* Root 권한만 작성할 수 있는 /system 폴더에 dummy 파일이 생성됨을 확인 가능

1.4 동작 분석

1) Seed Ip 찾기



우리가 Recovery OS로 사용하는 Ubuntu와 안드로이드 Recovery OS와의 scp 명령어를 통한 파일 전송을 위해 seed의 IP를 검색한다.

2) dummy.sh 및 update-binary 파일 내용

```
echo hello > /system/dummy
```

Task1의 목표인 /system 폴더에 hello 내용의 dummy 파일을 생성하는 명령어가 있는 dummy.sh 파일이다.

```
cp ./dummy.sh /android/system/xbin/dummy.sh
chmod 777 /android/system/xbin/dummy.sh
sed -i "/return 0/i /system/xbin/dummy.sh" /android/system/etc/init.sh
```

먼저 /android에 디렉토리에 이미 mount된 안드로이드 파티션에 파일을 위치시킨다.

cp ./dummy.sh /android/system/xbin/dummy.sh 를 통해 우리가 만든 OTA 패키지의 dummy.sh 파일을 /android/system/xbin 폴더에 파일을 이동시킨다. cp 명령어는 src

파일을 똑같이 복사하여 des 파일에 생성하는 명령어이다. 다음으로 우리가 달성하고 하는 것이 안드로이드가 부팅되면서 주입한 프로그램이 자동으로 실행되는 것이다.

chmod 777 /android/system/xbin/dummy.sh 를 통해 위에서 안드로이드 파티션으로 이동시킨 dummy.sh 파일에 실행권한을 부여한다. chmod는 해당 파일의 권한을 변경하는 명령어이다. 실행권한을 부여함으로써 init.sh 프로세스가 실행될 때 자동으로 우리 주입 프로그램이 실행될 수 있도록 설정해준다. 마지막으로 init.sh 프로세스가 우리 프로그램을 실행시킬 수 있게 만들어야 한다.

sed –i "/return 0/i/ system/xbin/dummy.sh" /android/system/etc/init.sh 를 통해 주입한 프로그램을 실행시키는 명령어를 넣는다. /android/system/etc/init.sh 파일의 내용에 return 0을 만나기 전에 우리가 주입한 프로그램을 실행시키는 /system/xbin/dummy.sh 명령어를 주입한다.

3) OTA 패키지 생성 및 안드로이드 Recovery OS 전송

```
[11/30/19]seed@VM:~/.../CSOS$ ls
HW1  HW2  HW-BONUS  otaP1
[11/30/19]seed@VM:~/.../CSOS$ zip -r otaP1.zip otaP1/
  adding: otaP1/ (stored 0%)
  adding: otaP1/META-INF/ (stored 0%)
  adding: otaP1/META-INF/com/ (stored 0%)
  adding: otaP1/META-INF/com/google/ (stored 0%)
  adding: otaP1/META-INF/com/google/android/ (stored 0%)
  adding: otaP1/META-INF/com/google/android/dummy.sh (stored 0%)
  adding: otaP1/META-INF/com/google/android/update-binary (deflated 46%)
[11/30/19]seed@VM:~/.../CSOS$ ls
HW1  HW2  HW-BONUS  otaP1  otaP1.zip
[11/30/19]seed@VM:~/.../CSOS$ scp ./otaP1.zip seed@10.0.2.78:/tmp/
seed@10.0.2.78's password:
otaP1.zip                                     100% 1406     1.4KB/s   00:00
[11/30/19]seed@VM:~/.../CSOS$
```

update-binary와 주입시킬 프로그램인 dummy.sh를 zip 명령어를 통해 OTA 패키지를 생성한다. OTA 패키지는 안드로이드 Recovery OS의 /tmp 디렉토리에서 실행되어야 한다. 따라서 맨 처음 찾아낸 seed ip를 이용하여 scp 명령어로 생성한 OTA 패키지를 Recovery OS의 /tmp 디렉토리로 전송한다.

4) 안드로이드 Recovery OS에서 unzip 및 update-binary 실행

```
seed@recovery:~$ cd /tmp
seed@recovery:/tmp$ ls
otaP1.zip  systemd-private-07e95abc9763443bbf4dda72454aa20e-systemd-timesyncd.
seed@recovery:/tmp$ unzip -l otaP1.zip
Archive:  otaP1.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-11-29 09:13   otaP1/
        0  2019-11-29 09:53   otaP1/META-INF/
        0  2019-11-29 09:13   otaP1/META-INF/com/
        0  2019-11-29 09:13   otaP1/META-INF/com/google/
        0  2019-11-30 12:21   otaP1/META-INF/com/google/android/
       27  2019-11-30 12:21   otaP1/META-INF/com/google/android/dummy.sh
      155  2019-11-30 12:20   otaP1/META-INF/com/google/android/update-binary
---------                     -------
      182                     7 files
```

```
seed@recovery:/tmp/otaP1/META-INF/com/google/android$ ls
dummy.sh  update-binary
seed@recovery:/tmp/otaP1/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/otaP1/META-INF/com/google/android$
```

안드로이드 가상머신을 켜서 부팅과정에 shift 키를 눌러 Recovery OS로 부팅을 한다. Recovery OS가 부팅되면 /tmp 디렉토리로 이동하여 우리가 전송한 OTA 패키지가 있는지 확인을 할 수 있다. Unzip 명령어를 통해 zip 상태인 OTA 패키지를 풀어주고, update-binary를 찾아서 실행한다. 프로그램의 동작이 제대로 되었으면 안드로이드는 업데이트 됐을 것이며, /system 디렉토리에 dummy 파일이 생성된다.

Task2. Inject Code via App_process

2.1 실습 개요

Bootloader란?

스마트폰에서 가장 낮은 레벨의 소프트웨어이다. 운영체제를 시작하기 위한 모든 코드를 실행한다. Bootloader는 부팅하기 전에 시스템 이미지의 신호를 확인한다. 확인 후 일을 수행한다.

안드로이드 부팅 순서

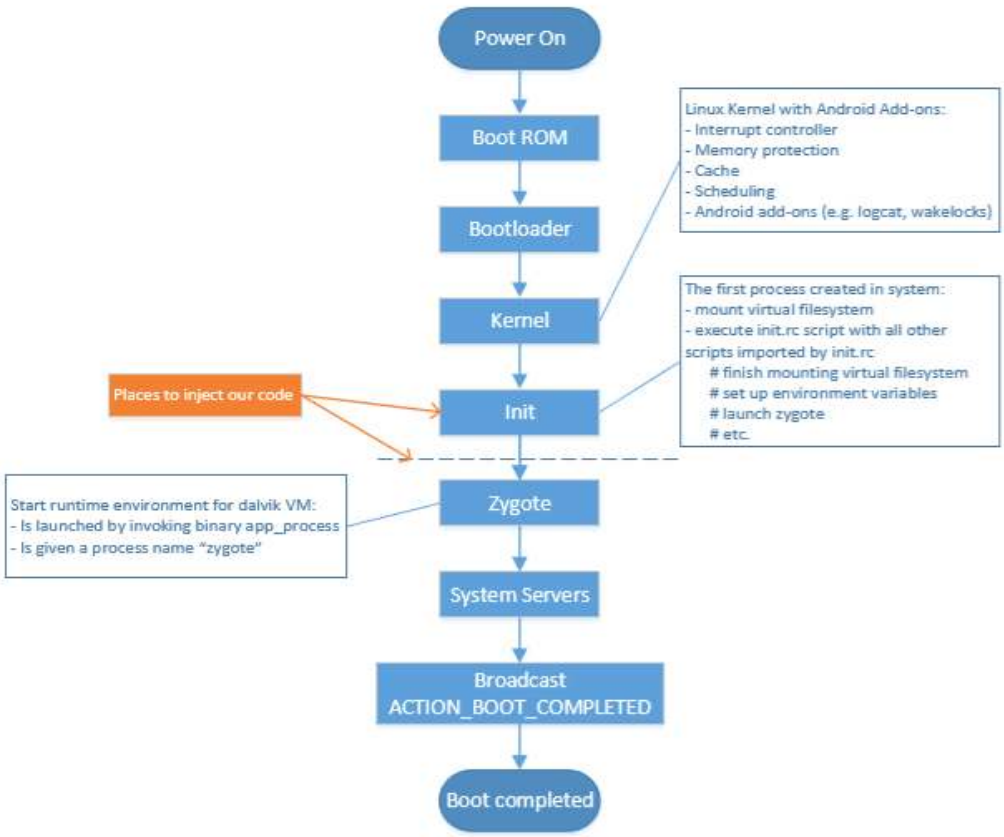


Figure 5: Detailed Booting Process

1단계. Boot ROM

RAM은 휘발성 메모리이기 때문에 부팅전에는 아무것도 없는 상태이다. 때문에 가장 먼저 비휘발성 메모리인 ROM에 접근한다.

2단계. Bootloader

ROM에 위치한 Bootloader를 실행한다. Bootloader는 메모리에 kernel을 올린다.

3단계. The Kernel

Bootloader가 안드로이드 시스템을 선택한다. 안드로이드 커널을 메모리에 load하고 시스템 초기화를 시작한다. 안드로이드 커널은 사실상 리눅스 커널로, 인터럽트, 메모리 보호, 스케줄링 등 시스템의 필수적인 부분을 처리한다.

4단계. The Init Process

커널이 로드되고 첫 번째 프로세스로 Init이 생성된다. Init은 모든 프로세스의 시작점이며 루트권한에서 실행된다. Init은 가상 파일시스템을 초기화하고, 하드웨어를 찾고, 스크립트 파일 init.rc를 실행하여 시스템을 구성한다. Init.rc는 가상 파일시스템 내부에 파일을 mount하고, 시스템 daemon을 초기화한다. 또한 환경 변수 설정, 아키텍처의 특정 명령어 실행, zygote 실행 등 다양한 목표를 위해 다른 rc 스크립트 파일을 불러온다.

```
init.${ro.zygote}.rc
```

이 파일은 매우 중요한 daemon인 zygote를 실행한다. 변수 ${ro.zygote}는 Init 바이너리에 상속된다, Android-x86 VM에서는 init.zygote.32.rc 이다.

5단계. The Zygote Process

Init.${ro.zygote}.rc 파일에서 “service /system/bin/app_process….” 명령어를 통해 Init 프로세스에서 zygote가 실행된다. Zygote는 app_process를 실행시킨다. Zygote는 안드로이드 실행의 시작점이며, 자바 프로그램 가상머신 환경을 실행의 시작이다. 안드로이드에서, 시스템 서버와 대부분의 어플리케이션은 자바로 쓰여진다. 때문에 Zygote는 필수적인 daemon이며 루트 권한으로 실행된다.

App_process

우리는 app_procsee에 rooting 코드를 삽입한다. App_process는 실제 이진 파일이 아

니다. 시스템 에 따라 App_process32나 app_process64를 가리키는 Symbolic link이다. 그러므로 우리는 우리의 코드를 가리키도록 Symbolic link만 바꿔주면 된다. 우리는 두개의 프로세스가 필요하다. 하나는 우리의 rooting 코드를 실행하는 것이고 또 다른 하나는 원래 app_process를 실행하는 것이다.

2.2 실습 목표

Zygote는 root권한을 가지며, app_process를 실행한다. 즉 zygote는 모든 app_process의 부모이다. Zygote가 실행될 때, app_process를 수정하는 것을 목표로 한다. app_process 대신 my_app_process가 수행되게 하여 task1과는 다르게 코드를 주입하여 안드로이드가 실행됨에 따라 자동으로 실행되게 하여 System 폴더에 dummy2 파일을 생성하여 root 권한을 얻었음을 확인한다.

2.3 실습 수행 결과

Step 1. Compile the code. : CSOS_Uduntu 환경에서 작업.

Application.mk 및 Android.mk 생성 및 complie the code

Application.mk

```
APP_ABI := x86
APP_PLATFORM := android-25
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
```

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)
```

Compile the code

```
[12/04/19]seed@VM:~/.../my_app_process$ ls
Android.mk  Application.mk  compile.sh  libs  my_app_process.c  obj  otaP2.pdf
[12/04/19]seed@VM:~/.../my_app_process$ chmod 777 compile.sh
[12/04/19]seed@VM:~/.../my_app_process$ ls
Android.mk  Application.mk  compile.sh  libs  my_app_process.c  obj  otaP2.pdf
[12/04/19]seed@VM:~/.../my_app_process$ ./compile.sh
Compile x86    : my_app_process <= my_app_process.c
Executable     : my_app_process
Install        : my_app_process => libs/x86/my_app_process
[12/04/19]seed@VM:~/.../my_app_process$ cat compile.sh
export NDK_PROJECT_PATH=.
ndk-build NDK_APPLICATION_MK=./Application.mk
```

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

OTA package

```
[12/04/19]seed@VM:~/android_test$ cd otaP2/META-INF/com/google/android/
[12/04/19]seed@VM:~/.../android$ ls
my_app_process  update-binary
```

Update-binary

```
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
cp ./my app process /android/system/bin/app process64
```

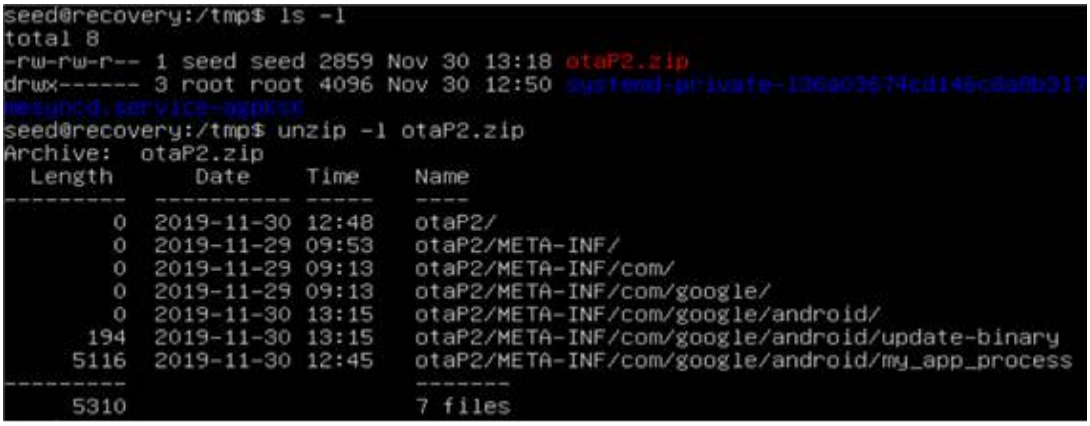
Zip

```
[12/04/19]seed@VM:~/android_test$ zip -r otaP2.zip otaP2/
  adding: otaP2/ (stored 0%)
  adding: otaP2/META-INF/ (stored 0%)
  adding: otaP2/META-INF/com/ (stored 0%)
  adding: otaP2/META-INF/com/google/ (stored 0%)
  adding: otaP2/META-INF/com/google/android/ (stored 0%)
  adding: otaP2/META-INF/com/google/android/update-binary (deflated 52%)
  adding: otaP2/META-INF/com/google/android/my_app_process (deflated 72%)
```

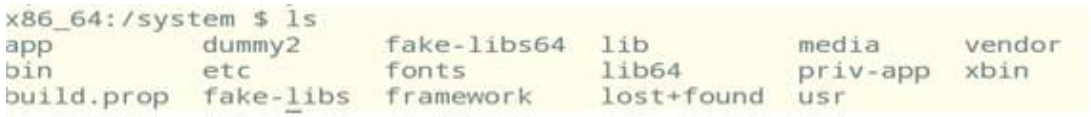
recoveryOS로 전송

```
[12/04/19]seed@VM:~/android_test$ scp ./otaP2.zip seed@10.0.2.78:/tmp/
seed@10.0.2.78's password:
otaP2.zip                                     100% 2820     2.8KB/s   00:00
```

recovatyOS에서 update-binary 파일 실행



/system 에 dummy2 파일 확인



2.4 동작 분석

1. Compile the code

다음은 compile.sh이다.

```
export NDK_PROJECT_PATH=.
ndk-build NDK_APPLICATION_MK=./Application.mk
```

Compile.sh는 Application.mk를 실행한다. 다음은 Application.mk이다.

```
APP_ABI := x86
APP_PLATFORM := android-25
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
```

Application.mk는 Android.mk를 실행한다. 다음은 Android.mk이다.

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := my_app_process
LOCAL_SRC_FILES := my_app_process.c
include $(BUILD_EXECUTABLE)
```

Android.mk를 통해 my_app_process.c가 compile된다. My_app_process.c는 다음과 같다.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main(int argc, char ** argv) {
        FILE * f = fopen("/system/dummy2", "w");
        if (f == NULL) {
                printf("Permission Denied.\n");
                exit(EXIT_FAILURE);
        }
        fclose(f);

        char * cmd = "/system/bin/app_process_original";
        execve(cmd, argv, environ);

        return EXIT_FAILURE;
}
```

My_app_process.c의 역할은 다음과 같다. 먼저 /system에 dummy2를 쓰기 권한으로 열거나 생성한다. 다음으로 app_process_original을 실행한다. 여기서 app_process_original은 본래의 app_process 역할을 하는 app_process64 파일에 대해서 이름만 바꾼 것이다. 우리의 목표는 /system에 dummy2를 생성하는 것이기 때문에 프로그램의 정상 작동을 위해 app_process_original을 실행 해줘야 한다.

다음은 compile.sh를 실행했을 때 상황이다.

```
[12/05/19]seed@VM:~/.../my_app_process$ ./compile.sh
Compile x86    : my_app_process <= my_app_process.c
Executable     : my_app_process
Install        : my_app_process => libs/x86/my_app_process
```

이를 통해 /libs/x86 폴더에 my_app_process 실행 파일이 생성되었음을 알 수 있다.

2. update-binary

Update-binary는 OTA package에 들어가는 실행파일이다. Recovery OS에서 실행되며 이 파일을 통해 기존 app_process64를 app_process_original로 이름 변경하고, my_app_process를 app_process64로 복사하여, 안드로이드 부팅 시 실행되도록 해야 한다. 다음은 직접 작성한 update-binary이다.

```
mv /android/system/bin/app_process64 /android/system/bin/app_process_original
cp ./my app process /android/system/bin/app process64
```

명령어 mv를 통해 app_process64의 이름을 app_process_original로 변경하였다. 그리고 명령어cp를 통해 my_app_process 파일을 app_process64로 이름 변경하여 /android/system/bin 폴더에 복사하였다. 명령어 cp에서 my_app_process에 쉽게 접근하기 위해 /META-INF/com/google/android폴더를 다음과 같이 구성한다.

```
[12/05/19]seed@VM:~/.../android$ ls
my_app_process  update-binary
```

3. recovertOS에서 update-binary 실행

임의로 만든 otaP2 폴더를 압축한 후 명령어 scp를 통해 recoveryOS에 전송한다. recoveryOS에서 압출 파일은 푼 뒤 update-binary를 실행한다.

```
seed@recovery:/tmp$ ls -l
total 8
-rw-rw-r-- 1 seed seed 2859 Nov 30 13:18 otaP2.zip
drwx------ 3 root root 4096 Nov 30 12:50 systemd-private-[redacted]
seed@recovery:/tmp$ unzip -l otaP2.zip
Archive:  otaP2.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-11-30 12:48   otaP2/
        0  2019-11-29 09:53   otaP2/META-INF/
        0  2019-11-29 09:13   otaP2/META-INF/com/
        0  2019-11-29 09:13   otaP2/META-INF/com/google/
        0  2019-11-30 13:15   otaP2/META-INF/com/google/android/
      194  2019-11-30 13:15   otaP2/META-INF/com/google/android/update-binary
     5116  2019-11-30 12:45   otaP2/META-INF/com/google/android/my_app_process
---------                     -------
     5310                     7 files
```

```
seed@recovery:/tmp/otaP2/META-INF/com/google/android$ ls
my_app_process  update-binary
seed@recovery:/tmp/otaP2/META-INF/com/google/android$ sudo ./update-binary
[sudo] password for seed:
seed@recovery:/tmp/otaP2/META-INF/com/google/android$ _
```

Update-binary가 실행되면, app_process64는 app_process_original로 바뀌고, my_app_process는 app_process64로 저장된다.

4. 안드로이드 부팅

안드로이드 부팅을 하게 되면, 부팅 순서에 따라 Zygote를 실행한다. Zygote는 app_process64를 실행한다. 하지만 이때 실행되는 app_process64는 우리가 미리 바꿔 놓은 my_app_process이다. My_app_process는 /system에 dummy2를 만들고 app_process_original, 즉 본래 app_process64를 실행한다. 우리는 /system에 dummy2이 생성되는 것을 통해서 root권한을 얻었음을 알 수 있다.

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

Task 3 : Implement SimpleSU for Getting Root Shell

3.1 실습 개요

루트 권한을 얻어 원하는 모든 동작을 수행하기 위해서는 root shell을 동작 시켜야 한다. 이전 task들과 같은 방법으로는 interactive한 shell program을 직접 exit 명령을 통해서 종료 시키지 않는 이상 booting sequence가 끝나지 않아 정상적인 rooting이 힘들다. 따라서 우리는 non-interactive하고 부팅이 가능하지만 root shell을 실행 시킬 수 있는 방법이 필요하다.

리눅스 프로그램에서는 SetUid/SetGid Program을 통하여 프로그램을 부팅이 완료된 후 실행시켜 root shell을 얻을 수 있지만 리눅스와는 다르게 안드로이드는 보안상의 이유로 SetUid/SetGid Program을 허용하고 있지 않다.

따라서 우리는 부팅 과정 동안의 root daemon을 실행 시키고 이를 통해서 root shell을 실핼 시킬 필요가 있다. root daemon은 Booting 시에 수행되어 background에서 작동하고 있고 유저가 root shell을 수행시킬 때 client program이 root daemon에게 요청을 보내면 요청을 받으면 daemon은 Shell을 수행한다.

이러한 과정을 수행하기 위해서는 shell 장치의 표준 입력과 출력 장치를 제어 할 수 있어야한다. shell 프로그램은 생성시의 부모 shell의 표준 입출력을 상속 받는다. 하지만 상속 받은 표준 입,출력,에러는 root의 소유이기 때문에 user cilent program은 이를 제어 할 수 없다.

따라서 우리는 이러한 부분은 우회하기 위해서 사용자 client의 입출력을 shell에게 주어야 한다. 이러한 부분을 성공시키기 위해서 2가지를 알아야 한다.

(1) 어떻게 다른 프로세스에게 표준 입,출력을 전달 할 수 있는가?

(2) 프로세스가 file descriptors 받고나서 어떻게 이것을 입,출력 장치로서 사용할 수 있는가 ?

file descriptors 란?

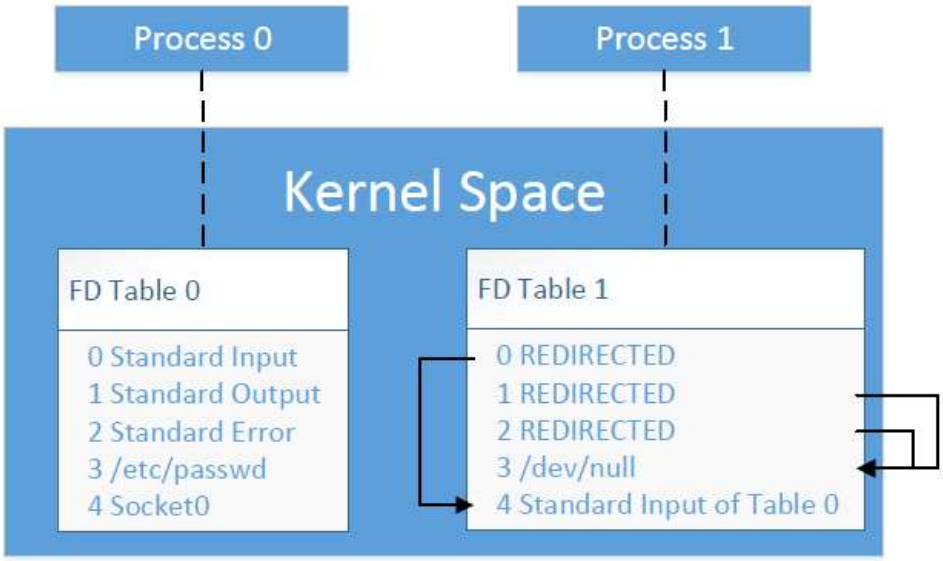
리눅스 시스템은 전형적으로 3가지 I/O 장치와 연관되어 있다.

1. Standard input (STDIN) : file descriptor 0
2. Standard output (STDOUT) : file descriptor 1
3. Standard Error (STDERR) : file descriptor 2

프로세스는 이러한 장치에 Standard POSIX Application Programming Interface을 통해서 파일 디스크립터를 사용할 수 있다. 이러한 descsriptor들은 마치 file처럼 취급된다.

이번 과제에서는 한 프로세스에서 다른 프로세스로 이 디스크립터를 전달해야한다.
파일 디스크립터는 다른 프로세스로 상속 또는 explicit sending을 통해서 전달 될 수 있다. 부모가 fork()를 통해서 자식을 생성하면 이러한 descriptor들은 자식에게 상속된다. 하지만 이러한 과정에서 부모가 다른 디스크립터를 자식에게 전달하고 싶으면 Unix Domain Socket을 이용해서 전달 할 수 있다. 이번 실습에서는 데몬에 의해서 생성도니 디스크립터를 shell에 전당한다.

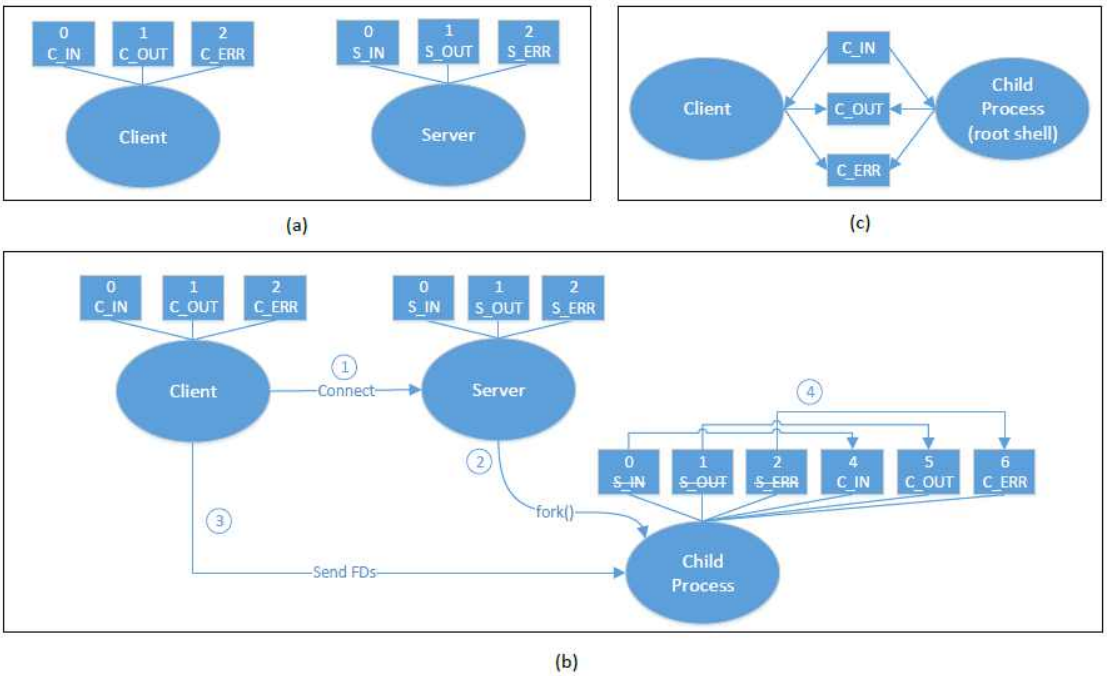
파일 디스크립터는 system call dup2(int dest, int src)를 통해서 redirected 될 수 있다.



새로운 프로세스 생성하기

Unix 시스템에서는 fork()를 통해서 새로운 프로세스를 생성할 수 있다. 부모 프로세스는 자식 프로세스의 pid를 return 하고 자식 프로세스는 0을 return 한다.

파일 디스크립터 전달하기 (Passing the File Descriptors)



위의 그림과 같이 클라이언트와 서버는 처음에는 표준 입,출력,에러 (0,1,2) 3개의 파일 디스크립터를 가진다.

3.2 실습 목표

Task3의 목표는 SimpleSU를 실행 시키는 것이다. SimpleSu를 실행 시킨다는 것은 사용자가 안드로이드환경에서 simplesu 명령을 통해서 루트 권한을 얻는 것을 말한다. 즉 루트 권한으로 실행되는 Shell을 얻는 것 (root Shell) 이를 통해서 사용자는 원하는 작업을 수행할 수 있는 권한을 얻을 수 있다.

3.3 실습 수행 결과

3.3.1 bash ./compile_all.sh를 실행

```
[12/01/19]seed@VM:~/.../task 3$ ls
compile_all.sh  mydaemon  mysu  otaP3.pdf  server_loc.h  socket_util
[12/01/19]seed@VM:~/.../task 3$ bash ./compile_all.sh
/////////Build Start/////////
Install        : mydaemon => libs/x86/mydaemon
Install        : mysu => libs/x86/mysu
/////////Build End////////////
[12/01/19]seed@VM:~/.../task 3$
```

3.3.2 update-binary 수정

```
[12/01/19]seed@VM:~/.../android$ cat update-binary
cp ./mydaemon /android/system/xbin/mydaemon
cp ./mysu /android/system/xbin/mysu

sed -i "/return 0/i /system/xbin/mydaemon" /android/system/etc/init.sh
sed -i "/return 0/i /system/xbin/mysu" /android/system/etc/init.sh
```

3.3.3 otaP3 생성

```
[12/01/19]seed@VM:~/.../CSOS$ zip -r otaP3.zip otaP3/
  adding: otaP3/ (stored 0%)
  adding: otaP3/META-INF/ (stored 0%)
  adding: otaP3/META-INF/com/ (stored 0%)
  adding: otaP3/META-INF/com/google/ (stored 0%)
  adding: otaP3/META-INF/com/google/android/ (stored 0%)
  adding: otaP3/META-INF/com/google/android/update-binary (deflated 60%)
  adding: otaP3/META-INF/com/google/android/mydaemon (deflated 60%)
  adding: otaP3/META-INF/com/google/android/mysu (deflated 66%)
```

3.3.4 android로 전송

```
[12/01/19]seed@VM:~/.../CSOS$ scp ./otaP3.zip seed@10.0.2.78:/tmp/
seed@10.0.2.78's password:
otaP3.zip                                     100% 8477     8.3KB/s   00:00
```

3.3.5 unzip

```
seed@recovery:/$ cd /tmp
seed@recovery:/tmp$ ls
otaP3.zip  systemd-private-dd6706003814453b820aa60fdbaca0db-systemd-timesyncd.
seed@recovery:/tmp$ unzip -l otaP3.zip
Archive:  otaP3.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2019-12-01 09:25   otaP3/
        0  2019-11-29 09:53   otaP3/META-INF/
        0  2019-11-29 09:13   otaP3/META-INF/com/
        0  2019-11-29 09:13   otaP3/META-INF/com/google/
        0  2019-12-01 09:45   otaP3/META-INF/com/google/android/
      219  2019-12-01 09:45   otaP3/META-INF/com/google/android/update-binary
     9232  2019-12-01 09:24   otaP3/META-INF/com/google/android/mydaemon
     9232  2019-12-01 09:24   otaP3/META-INF/com/google/android/mysu
---------                     -------
    18683                     8 files
```

3.3.6 update-binary 실행

```
seed@recovery:/tmp/otaP3/META-INF/com/google/android$ ls
mydaemon  mysu  update-binary
seed@recovery:/tmp/otaP3/META-INF/com/google/android$ sudo ./update-binary
seed@recovery:/tmp/otaP3/META-INF/com/google/android$ _
```

3.3.7 android에서 mysu 실행 시 모습

```
x86_64:/ # ps|grep my
root      1016  1     5064   296             0 0000000000 S /system/xbin/mydaemon
root      1030  1016  0      0               0 0000000000 Z mydaemon
root      1535  1016  0      0               0 0000000000 Z mydaemon
root      1818  1016  0      0               0 0000000000 Z mydaemon
u0_a36    3081  3050  5064   1820            0 0000000000 S mysu
x86_64:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:init:s0
x86_64:/ #
```

3.4 동작 분석

1. Server launches the original app process binary
: 서버가 root권한의 원래 process를 수행시킨다. 이 process의 소유는 root이기 때문에 높은 권한을 가진다.

2. Client sends its FDs
: Client가 자신의 파일 디스크립터를 전달한다.

3. Server forks to a child process
: 디스크립터를 전달 받은 서버는 자식 프로세스를 fork()를 통해서 생성한다.

4. Child process receives client’s FDs
: 자식 프로세스는 Client의 파일디스크립터를 전달한다.

5. Child process redirects its standard I/O FDs
: 자식 프로세스에서 dup2() 명령어등을 통해서 표준 입출력을 재지정 해준다.

6. Child process launches a root shell
: 자식 프로세스는 root권한의 shell을 실행한다.

mysu.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>       //socket() bind() listen() accept() AF_UNIX
#include <fcntl.h>            //fcntl()
#include <string.h>           //strerror()
#include <errno.h>            //errno
#include <sys/un.h>           //struct sockaddr_un
#include "../socket_util/socket_util.h"
#include "../server_loc.h"
//필요한 라이브러리 include
#define ERRMSG(msg) fprintf(stderr, "%s", msg)
#define DEFAULT_SHELL "/system/bin/sh"
#define SHELL_ENV "SHELL=" DEFAULT_SHELL
#define PATH_ENV "PATH=/system/bin:/system/xbin"

int config_socket() {

    struct sockaddr_un sun;
    int socket_fd = socket(AF_UNIX, SOCK_STREAM, 0);
    if (socket_fd < 0) { //소켓 생성 실패 시 에러 처리
        ERRMSG("failed to create socket fd\n");
        exit (EXIT_FAILURE);
```

```
    }

    if (fcntl(socket_fd, F_SETFD, FD_CLOEXEC)) {  //에러 처리
        ERRMSG("failed on fcntl\n");
        exit (EXIT_FAILURE);
    }

    memset(&sun, 0, sizeof(sun));
    sun.sun_family = AF_UNIX;
    strncpy(sun.sun_path, SERVER_LOC, sizeof(sun.sun_path));

    //서버로 연결
    if (0 != connect(socket_fd, (struct sockaddr*)&sun, sizeof(sun))) {
        ERRMSG("failed to connect server\n");
        exit (EXIT_FAILURE);
    }

    return socket_fd;
}
// server-client 통신

int connect_daemon() {

     //소켓 얻기
    int socket = config_socket();

    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
     //소켓_FD송신
    char dummy[2];
    ERRMSG("2 \n");
    int flag = 0;
    do {
        flag = read(socket, &dummy, 1);
    } while (flag > 0);

    ERRMSG("3 \n");
```

```
    close(socket);

    //print out error message if has
    if (flag < 0) {
        ERRMSG("Socket failed on client: ");
        ERRMSG(strerror(errno));
        ERRMSG("\n");
        return (EXIT_FAILURE);
    }
    return (EXIT_SUCCESS);
}

int main(int argc, char** argv) {

    if (getuid() != 0 && getgid() != 0) {
        ERRMSG("start to connect to daemon \n");
        return connect_daemon();
    } //권한 검사 ( root가 아닐시 )

    char* shell[] = {"/system/bin/sh", NULL};
    execve(shell[0], shell, NULL);  //루트 권한일 시 shell을 띄운다.
    return (EXIT_SUCCESS);
}
```