Android Device Rooting Lab

Copyright © 2018 Wenliang Du, Syracuse University.
The development of this document was partially funded by the National Science Foundation under Award No. 1303306 and 1718086. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.

1 Overview

Android devices do not allow their owners to have the root privilege on the device. This is fine for normal customers, but for users who want to make deep customizations on their devices, this is too restrictive. There are ways to overcome these restrictions, most of which require the root privilege on the device. The process of gaining the root device on Android devices is called *rooting*. Being able to root Android devices is a very useful skill for security experts.

The objective of this lab is two-fold. First, through this lab, students will get familiar with the process of device rooting and understand why certain steps are needed. Many people can root Android devices, but not many people fully understand why things have to be done in a particular way. Second, the entire rooting mechanism involves many pieces of knowledge about the Android system and operating system in general, so it serves as a great vehicle for students to gain such in-depth system knowledge. In this lab, we will ask students to develop a complete rooting package from scratch, and demonstrate how to use the package to root the Android VM provided by us.

2 Background Knowledge

2.1 Background Knowledge about Rooting

The purpose of rooting an Android device is to gain the root privilege inside the Android OS. There are many reasons why people want to do that. For example, Android devices often come with many pre-installed system apps that are useless most of the time, but they take storage space, RAM, system resources, and drain device battery. These useless apps are generally known as bloatware; they are installed in protected places, and only the root user (or other privileged user) can remove them. Although the device is owned by a user, the user is not allowed to run code using the root privilege. Users can only get the root privilege if the device is rooted. In general, rooting allows users to customize their phones, removing restrictions, adding new features, and making changes to the system. There are several approaches to root an Android device, and we will discuss them in more details in this lab. Figure 1 provides a summary of these approaches.

Modifying Android from inside. The first rooting approach is doing it from inside Android. For an unrooted device, everything that the user runs is running with a normal user ID (not root). Therefore, rooting from inside Android means that the user has to gain the root privilege as a normal user. Obviously, this is impossible for a perfectly implemented operating system. However, the chances are that Android may not be perfectly implemented, so there might exist vulnerabilities in the system. If these vulnerabilities are inside the kernel or daemons running with the root privilege, users can exploit these vulnerabilities to gain the root privilege. This is one of the techniques to root Android devices.

In the past, there were many rooting exploits, such as `RageAgainstTheCage`, which took advantage of `RLIMIT_NPROC` max, the value specifying how many processes a given UID can run. This exploit first uses "`adb shell`" to run a shell on the device via the `adb` daemon. The UID of the shell will be 2000. The exploit then forks new processes until the number of processes have reached the limit and the fork call will fail. At this point, the exploit kills the `adbd` daemon, and then reconnect to it using "`adb shell`". This causes the system to restart the `adb` daemon. When `adb` is started, it always has the root privilege, but it will drop the privilege to UID 2000 using `setuid(2000)`. Unfortunately, the UID 2000 has already used up its process quota, so the call will fail. Unfortunately, the `adb` daemon fails to handle the failure correctly: instead of exiting, it keeps running, while retaining the root privilege. As results, the "`adb shell`" command will give users a root shell. This vulnerability has already been fixed after Android 2.2.

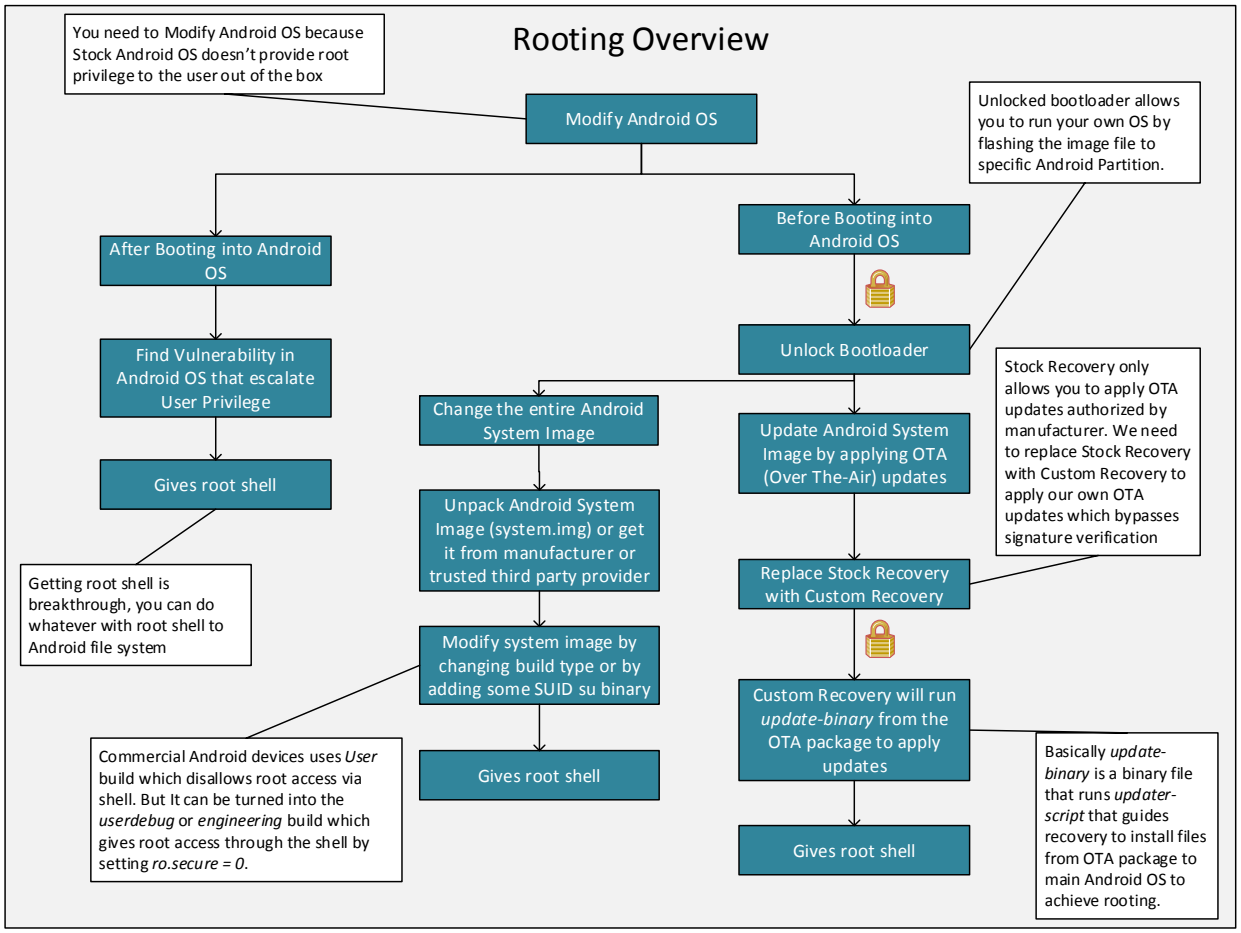


Figure 1: Rooting Android Devices

Modifying Android from outside. Since Android's access control prevents a normal user from modifying the part of the system that is only modifiable by the root, doing it inside Android is going to subject to the access control. However, if we can do it from outside, we are not subject to Android's access control any more. This leads to the second approach, which is to directly modify Android's system files from outside.

Assume that another operating system is installed on your Android device. This means that the device

has a dual-boot configuration, allowing you to boot into any of these two operating systems when the device powers on. If we boot into this second operating system, and become the root for this OS, we can mount the partition used by the Android system. Once this partition is mounted, we can access all the files stored inside the partition. Because Android is not running, its access control has no use. Therefore, we can make arbitrary changes to its files. For example, we can place some programs in the system, and modify Android's initialization scripts, and ask Android to run our programs as a root during its initialization process.

The good news is that most Android devices do have the second operating system installed, and it is called *recovery OS*. As the name indicates, it is meant for recovery purposes, but it is mostly used for updating operating system. The bad news is that this operating system also has access controls, preventing users from running arbitrary programs or commands. Recovery OS is typically placed on devices by vendors, who use the OS to update Android, but not wanting users to make arbitrary updates. To enforce that, recovery OSes do not give users a shell prompt, preventing users from running arbitrary commands. Instead, they take a package provided from outside (either from users or downloaded from the Internet); the package contains the commands and files needed for updating the Android OS. This mechanism is widely used for OS update, and it is called Over-The-Air (OTA) update. The package is called OTA package, which has a standard file structure that we will talk about later.

Most recovery OSes only accept the packages made by the vendors to ensure that any update to the Android OS is approved by the vendors. This is enforced using digital signatures, i.e., the package needs to be signed by vendors, or it will not be accepted by the recovery OS. This protection becomes a roadblock for rooting, because the package that we use for rooting purposes will not come from the vendor of the device. We need to find ways to bypass this access control.

Reinstall recovery OS. Instead of bypassing the access controls of the recovery OS, the easiest way is to replace the entire stock recovery OS with another recovery OS that does not have such access controls. This new recovery OS, called custom recovery OS, will not include the signature verification part, so we can provide any OTA packages to the recovery OS. This will allow us to make arbitrary changes to the Android partition.

Here is another bad news. There is another access control that prevents us from replacing the stock recovery OS, and this time it is the bootloader. Bootloader is a low level code that loads an operating system or some other system software for the computer after the computer is powered on. When a bootloader is "locked", it will simply load one of the OSes that is already installed on the device, leaving no chances for users to modify any of the pre-installed OSes. If a bootloader can be unlocked, it will add another option, which allows users to install custom operating systems on the device, as well as replacing the existing ones. The process is often referred to as flashing custom OS.

Manufacturers usually make their bootloaders locked out of the box because they want to be in control of what software is running the device. However, most manufacturers do provide ways for users to unlock the bootloader on their devices, provided that by doing so users will lose all of their data, as well as the warranties.

2.2 Background Knowledge about OTA

OTA is a standard technique for devices to update Android operating systems. Since rooting also needs to update Android OS, OTA becomes a popular choice. In this section, we will describe the structure of the OTA package. Students need to build their own OTA package from the scratch in this lab.

OTA package is just a zip file and its structure is depicted in Figure 2. Of particular interest to this lab is the `META-INF` folder, which includes signature and certificates of the package along with two very important files named `update-binary` and `updater-script`.

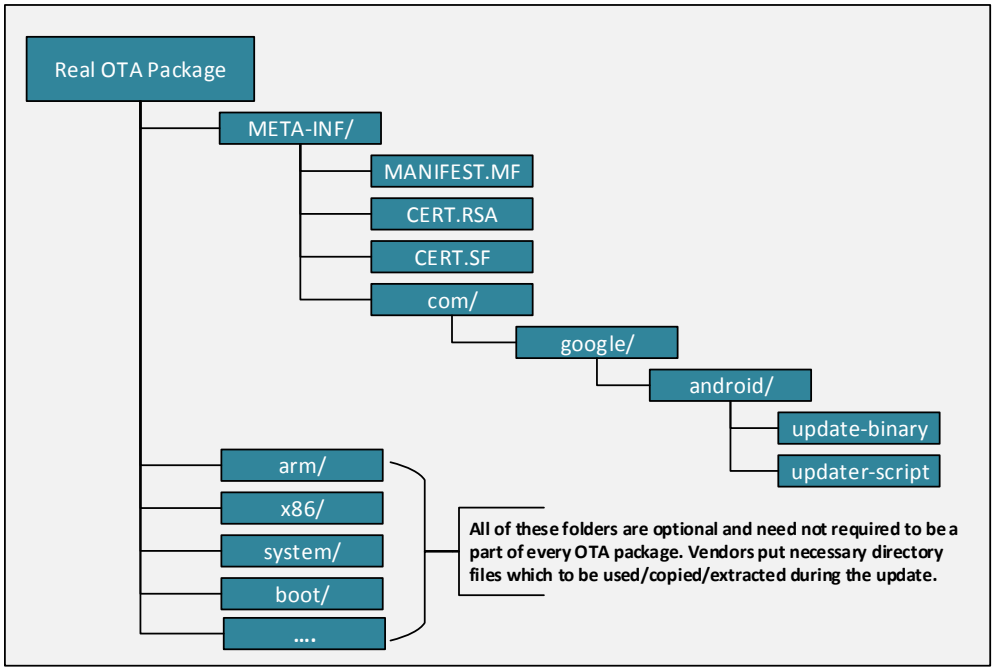


Figure 2: OTA Structure

- `META-INF/com/google/android/update-binary`: This binary is executed by the recovery OS to apply OTA updates; it loads and execute `updater-script`.
- `META-INF/com/google/android/updater-script`: This is an installation script which is interpreted by `update-binary`. It is written using a script language called `Edify`, which describes the required action to be performed to apply updates.

After signature verification on the OTA package, the recovery OS extracts the `update-binary` executable from the OTA package to the `/tmp` directory and runs it by passing three arguments as follows:

```
update-binary version output package
where, version is the version of recovery API,
       output is the command pipe that update-binary uses to  communicate
                        with recovery,
       package is the path of the OTA package,


An Example would be:  update-binary 3 stdout /sdcard/ota.zip
```

On a successful execution of `updater-script`, the recovery OS copies the execution logs to the `/cache/recovery/` directory and reboots into the Android OS. Execution logs can be accessible from the Android OS after rebooting. This is how android system is updated by using OTA package.

3 Lab Environment

In this lab, we assume that the bootloader on the device can be unlocked, and the stock recovery OS can be replaced. Replacing the recovery OS in the VM environment is quite difficult, and it is not within the scope of this lab. The Android VM that you download from our web site already has a custom recovery OS installed. Actually, we simply use Ubuntu 16.04 as the "recovery OS". Technically, this is not a recovery OS; it is just another OS installed on the device. We use this general-purpose OS to emulate what users can do using a custom recovery OS. Once users boot into this OS, they can run any arbitrary commands (as root), and be able to make arbitrary changes to the Android partition.

Students who have a physical Android device and want to know how to root a real device, they can follow our guidelines in Section 7. However, other than the step to replace the recovery OS, everything else is the same as what we do in this lab.

Another important thing to know is that our Android VM is already rooted. That was done when we built the VM from the Android source code. Our VM build is actually a *userdebug* build, and several doors were built into the VM to allow root access. We choose this build because several of our other labs require the root access. Real Android devices use *user* build, which do not have these doors for root access. Details about these doors can be found in our VM manual. In this lab, students are not allowed to use those doors to gain the root access; they have to use the OTA mechanism to achieve that.

4 Lab Task 1: Build a simple OTA package

In this lab, students will build a simple OTA package from scratch, and use it to root Android OS. We divide this complicated task into several tasks, each focusing on a specific goal. Here are the goals that we would like to achieve:

- How to inject a program into the Android OS from the recovery OS?
- How to get our injected program to run automatically, and with the root privilege?
- How to write a program that can give us the root shell?

In this Task 1, we focus on how to inject a program into the Android OS from the recovery OS, and then get Android to run our injected program using the root privilege. At this point, we are not focusing on the actual thing that we can do in this program, as long as it is something that requires the root privilege. For the sake of simplicity, let us simply create a dummy file in the `/system` folder of Android, which requires the root privilege (the `/system` folder is not writable by normal users). The following command puts a word "hello" in `/system/dummy` (we put this command in a shell script file called `dummy.sh`):

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

Step 1: Write the update script. The `update-binary` file in the OTA package is the executable file that will be first executed by the recovery OS. This is where we get the OS update started. This file can be a binary executable, or just a simple script file. For the latter case, the recovery OS should already have the binary executable (e.g. `bash`) to execute the script file. In this task, we will simply use a shell script, as our recover OS (`Ubuntu`) does have `bash` installed.

Our purpose for `update-binary` is two-fold: (1) inject our `dummy.sh` program into the Android OS, and (2) change the Android OS configuration file, so our `dummy.sh` can be automatically executed with the root privilege when Android boots up. For the first task, students need to figure out where to place

`dummy.sh`, and how to set up its permissions. It should be noted that the file has to be placed into the Android partition, which is already mounted to the `/android` directory in the recovery OS.

For the second purpose, we need to get `dummy.sh` to run automatically when Android boots up, and we need to make sure that it runs with the root privilege. There are many ways to achieve this. In this task, we use one approach related to Linux, and in the next task, we will use a different approach related to Android framework.

Android is built on top of the Linux operating system. When it boots up, its underlying Linux boots up first, which conducts system initialization, including starting essential daemon processes. The booting procedure, using the root privilege, runs a file called `/system/etc/init.sh` [1] for part of the initialization. Therefore, if we can insert a command into `init.sh` file, we can run our `dummy.sh` file with the root privilege.

If we can do it manually, we can simply edit the `init.sh`, and add a new command in it, but we are writing an OTA package, so the actions to modify the file need to be coded in our `update-binary` file. There are many ways to code that, and we will use the `sed` command, which is stream editor for filtering and transforming text. Our idea is to find where the statement `"return 0"` is inside `init.sh`, and insert a command before that, essentially placing the command before the program in `init.sh` finishes.

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

Explanation:
  -  "-i": edit files in place.
  -  "/return 0/": match the line that has the content return 0.
  -  "i": insert before the matching line.
  -  "/system/xbin/dummy.sh": the content to be inserted. We need to copy
        the dummy.sh file to the corresponding folder first.
  -  "/android/system/etc/init.sh": the target file modified by "sed".
```

Step 2: Build the OTA Package. Constructing an OTA package is quite straightforward. All we need to do is to put our files in their corresponding folders according to Figure 2. We need to maintain the same structure as what is shown in the figure, but you do not need to create all the files that are not needed for our task (such as signature and optional files). You can put the `dummy.sh` file in any place of your OTA package, as long as the location matches with your command in the `update-binary`. After creating the file structure, we can use the `zip` command to create a zip file:

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

You should include the file structure of your OTA package in your report. You can run the `"unzip -l"` command to do that.

Step 3: Run the OTA Package After building the OTA package, we can provide it to the recovery OS, which will run it automatically. However, that is how it works with a real recovery OS. In our lab environment, we are using `Ubuntu` as our recovery OS, but it does not have the needed recovery functionality. Therefore, we have to emulate the recovery functionality. This means, we have to manually unpack the OTA package (using the `unzip` command), go to the folder `META-INF/com/google/android` folder to find the `update-binary` file, and run it. If you have written everything correctly, your Android is now updated. Now, boot up your Android OS, and see whether the dummy file is created inside `/system`. In your report, you should include screenshots to provide evidences.

[1] This is for Android-x86 build; for the ARM build, the file name is different.

5 Task 2: Inject code via `app_process`

In the previous task, we modify the `init.sh` file to get our injected program to run automatically, and with the root privilege. This initialization script file is used by the underlying Linux operating system. Once the Linux part is initialized, Android OS will bootstrap its runtime that is built on top of Linux. We would like to execute our injected program during this bootstrapping process. The objective of this task is not only to find a different way to do what we have done in the previous task, but also to learn how Android gets bootstrapped.

Before conducting this task, please read the guideline in Section 7.1 about the Android booting sequence. From the guideline, we can see that when the Android runtime bootstraps, it always run a program called `app_process`, using the root privilege. This starts the `Zygote` daemon, whose mission is to launch applications. This means that `Zygote` is the parent of all app processes. Our goal is to modify `app_process`, so in addition to launch the `Zygote` daemon, it also runs something of our choice. Similar to the previous task, we want to put a dummy file (`dummy2`) in the `/system` folder to demonstrate that we can run our program with the root privilege.

The following sample code is a wrapper for the original `app_process`. We will rename the original `app_process` binary to `app_process_original`, and call our wrapper program `app_process`. In our wrapper, we first write something to the dummy file, and then invoke the original `app_process` program.

```
#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;
}
```

It should be noted that when launching the original `app_process` binary using `execve()`, we should pass all the original arguments (the `argv` array) and environment variables (`environ`) to it.

Step 1. Compile the code. We need to compile the above code in our `SEEDUbuntu` VM, not inside the recovery OS or Android OS, as neither of them has the native code development environment installed; we have installed the Native Development Kit (NDK) in our SEEDUbuntu VM. NDK is a set of tools that allow us to compile C and C++ code for Android OS. This type of code, called native code, can either be a

stand-alone native program, or invoked by Java code in Android apps via JNI (Java Native Interface). Our wrapper `app_process` program is a standalone native program, which needs to be compiled using NDK. For more detailed instructions about NDK, please refer to the instructional manual linked in the web page.

To use NDK, we need to create two files, `Application.mk` and `Android.mk`, and place them in the same folder as your source code. The contents of these two files are described in the following:

```
The Application.mk file

APP_ABI := x86
APP_PLATFORM := android-21
APP_STL := stlport_static
APP_BUILD_SCRIPT := Android.mk
```

```
The Android.mk file

LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := <compiled binary name>
LOCAL_SRC_FILES := <all source files>
include $(BUILD_EXECUTABLE)
```

We run the following commands inside the source folder to compile our code. If the compilation succeeds, we can find the binary file in the `./libs/x86` folder.

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

Step 2. Write the update script and build OTA package. Just like the previous task, we need to write `update-binary` to tell the recovery OS what to do. Students need to write the shell script code in this task. Here are some guidelines:

- We need to copy our compiled binary code to the corresponding location inside Android.
- We need to rename the original `app_process` binary to something else, and then use our code as `app_process`. The actual name of `app_process` can be either `app_process32` or `app_process64`, depending on the architecture of the device. Our Android VM is a 64-bit device, so the name should be `app_process64`.

Students should then repeat Steps 2 and 3 of the previous task, and describe their observations and explanation.

6 Task 3: Implement `SimpleSU` for Getting Root Shell

Now we know how to inject our code into the Android system and gain the root privilege, but we have not completely achieved our ultimate goal yet. An important reason for users to root their Android devices is to execute any command of their choice using the root privilege. When building the OTA package, the commands are already decided; if users want to run other commands after the programs in the OTA package is executed, they will not be able to do that, unless they can get a shell that runs with the root privilege. Such a shell is called *root shell*.

We can launch the root shell using the methods from the previous tasks, but that is problematic: shell programs are interactive, meaning they will not terminate unless users type an exit command; this will stop the system booting process, so the OS will never be able to complete its booting sequence. The interesting question is how to run something non-interactive during the booting process that enables us to get an interactive root shell later on.

If we were on a typical Linux system, we can easily solve the above problem by using the `chmod` command to turn on the Set-UID bit of any shell program (e.g. `bash`) that is owned by the root. Later on, when any user runs this shell program, the shell will run with the owner's (i.e. root) privilege. Unfortunately, for security reasons, Android has removed the Set-UID mechanism from its underlying Linux OS since version 4.3 (API Level 18). The official document of security updates on Android 4.3 says the following:"No setuid/setgid programs. Added support for filesystem capabilities to Android system files and removed all setuid/setguid programs. This reduces root attack surface and the likelihood of potential security vulnerabilities."

Another approach is to start a root daemon during the booting process, and then use this daemon to help users get a root shell. This is the approach used by some of the popular rooting OTA packages, such as `SuperSU` developed by Chainfire. In this task, students will write such a daemon and use it to understand how it helps users to get a root shell. The main idea of this approach is quite simple. When users want to get a root shell, they run a client program, which sends a request to the root daemon. Upon receiving the request, the daemon starts a shell process, and "give" it to the client, i.e., allowing users to control the shell process. The tricky part is how to let the user control the shell process that is created by the daemon.

For users to control the daemon-generated shell process, they need to be able to control the standard input and output devices of the shell process. Unfortunately, when the shell process is created, it inherits its standard input and output devices from its parent process, which is owned by root, so they are not controllable by the user's client program. We can find a way to let the client program control these devices, or we can do it in a different way by giving the client program's input and output devices to the shell process, so they also become the input/output devices for the shell process. This way, the user has a complete control of the shell process: whatever the user types in the input device of the client program will also be fed into the shell process; whatever the shell process prints to its output device will be showing to client program.

Writing the code to implement the above idea is not easy, as we need to have two essential pieces of knowledge: (1) how to send the standard input/output devices (file descriptors) to another process, and (2) once a process receives the file descriptors, how it can use them as its input/output devices. We provide some background knowledge regarding these.

6.1 Background

File descriptors. Each process in Linux systems typically has three associated I/O devices: standard input device (`STDIN`), standard output device (`STDOUT`), and standard error device (`STDERR`). These devices are where the process gets its user input and prints out results and error messages. Processes access these devices through the standard POSIX application programming interface that uses file descriptors. Basically, I/O devices are treated just like they are files. The file descriptors for `STDIN`, `STDOUT`, and `STDERR` are 0, 1, and 2, respectively. In this task, we need to pass file descriptors from one process to another.

File descriptors can be passed to another process either via inheritance or explicit sending. When a parent creates a child process using `fork()`, all the parent's file descriptors are automatically inherited by the child process. Beyond this stage, if the parent wants to share a new file descriptor with its children, or if two unrelated processes want to share the same file descriptor, they have to explicitly send the file descriptor, which can be achieved using the Unix Domain Socket. In our code, our client program sends its file descriptors to the root shell process created by the daemon.

File descriptors can be redirected. The system call `dup2(int dest, int src)` can redirect the `src` file descriptor to the `dest` one, so the file descriptor entry at index `src` actually points to the entry at `dest`. Therefore, whenever the process uses the `src` file descriptor, it actually uses the entry stored in the `dest` entry. For example, assume that we open a file, and get a file descriptor 5. If we call `dup2(5, 1)`, we basically let the file descriptor 1 points to 5, causing anything printed out by `printf()` to be saved to the file that was just opened. This is because `printf()` by default prints out everything to the standard output device, which is represented by the file descriptor 1.

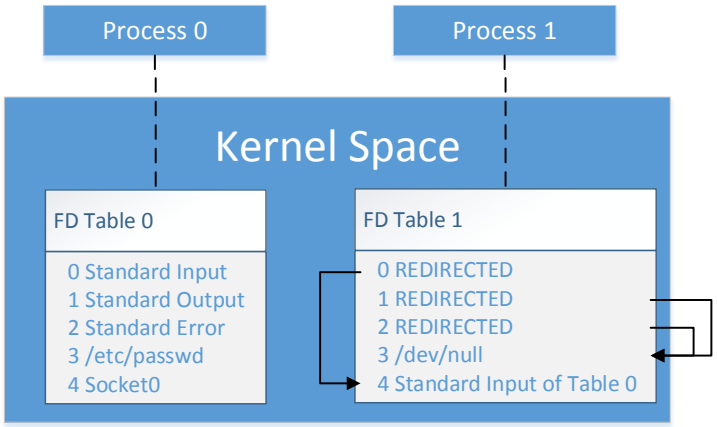


Figure 3: File Descriptor Tables

Figure 3 shows the file descriptor tables of two processes. The table of Process 0 is quite simple. It has three standard I/O FDs (indices 0, 1, 2), a file descriptor (index 3) for an opened file, and another file descriptor (index 4) for a socket. Table 1 is a little bit more complicated. It opened a device named `/dev/null`, and received a file descriptor from Table 0, and store the descriptor at index 4. Moreover, Process 1's standard output and error are redirected to `/dev/null`, while its standard input is redirected to the one (index 4) received from Process 0. The consequence of such redirections is that Process 1 will take exactly the same input as Process 0, but all outputs are abandoned (`/dev/null` is a standard device that functions like a black hole: nothing written to it gets out).

Creating new process. In Unix systems, we use the `fork()` system call to create a new process. The `fork()` call returns an integer: for the child process, the return value is 0, while for the parent process, the return value is the actual process id (which is non-zero) of the newly created child process. The child process inherits the parents data and execution status, as well as the file descriptors. A sample code is provided in the following:

```
pid_t pid = fork();
if (pid == 0) {
    // This branch will only be executed by the child process.
    // Child process code is placed here ...
}
else {
    // This branch will only be executed by the parent process.
    // Parent process code is placed here ...
}
```

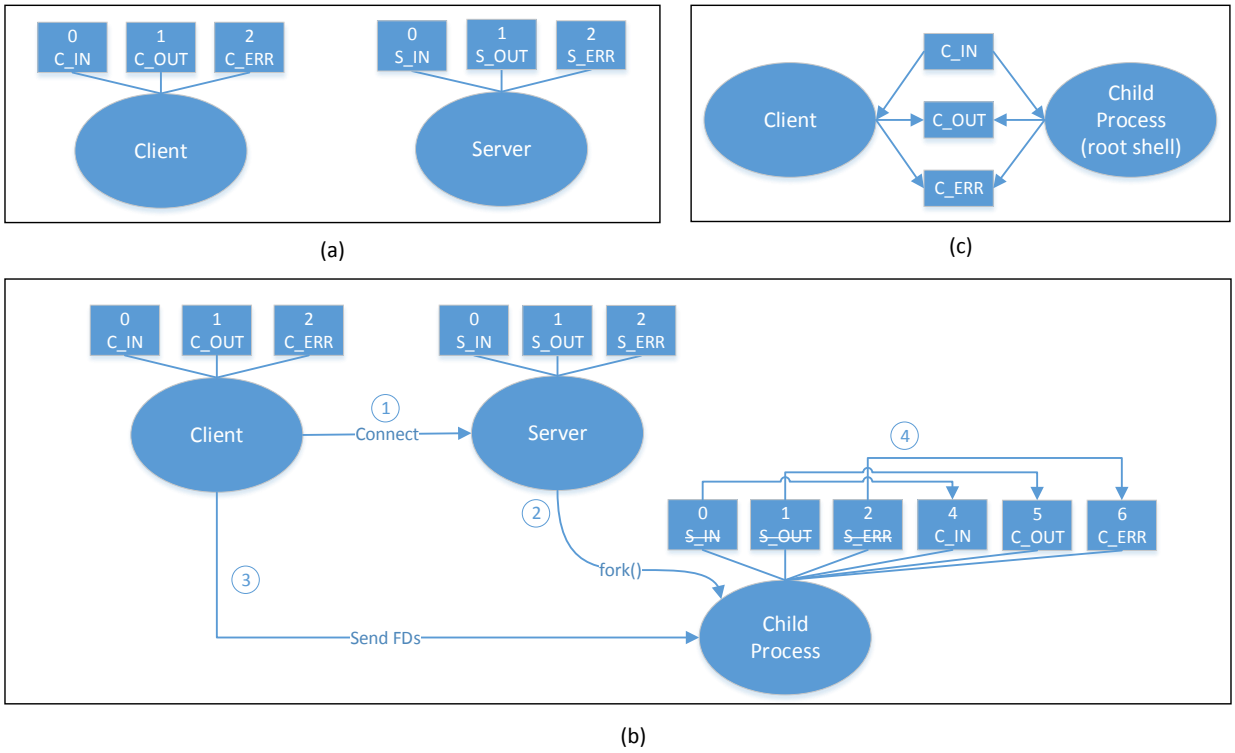


Figure 4: How one process (Client) gains the control of the input/output devices of another process

Passing the File Descriptors. Figure 4 shows how to use the three standard I/O file descriptors to help the client gain the full control of the root shell created by the server. Initially, as shown in Figure 4(a), the client and server are running in different processes, while the client has a normal privilege and the server has the root privilege. Each of them has its own standard I/O FDs 0, 1, 2. In the figure, the client FDs are denoted as `C_IN`, `C_OUT`, and `C_ERR`, and the servers FDs are denoted as `S_IN`, `S_OUT`, and `S_ERR`. In Figure 4(b), we depict how the client and the server work together to help the client get the root privilege.

1. The client connects to the server using the Unix Domain Socket.
2. Upon receiving the request, the server forks a child process and runs a root shell. The child process inherits all the standard I/O FDs from the parent.
3. The client sends its FDs 0, 1, and 2 to the server's child process using the Unix Domain Socket. These FDs will be saved in the table indices 4, 5, and 6, respectively.
4. The child process redirects its FDs 0, 1, 2 to the FDs received from the client, resulting in FDs 4, 5, and 6 being used the as the standard input, output, and error devices. Since these three devices are the same as those in the client, essentially, the client process and the server's child process now share the same I/O devices (Figure 4(c)). Although the client process is still running with a normal user privilege, it has the full control of the server's child process, which is running with the root privilege.

6.2 The Task

Due to the complexity of the client and server program, we provide all the source code to students from our web site. Please compile the code using NDK, and use the method described in the previous task to build an OTA package. Students need to demonstrate that they can successfully get the root shell after they have rooted the Android OS using their OTA package.

Moreover, they need to show that their client process and the shell process do share the same standard input/output devices. In Unix-like systems, file descriptors of a process can be found in the `/proc` virtual file system, inside the `/proc/<PID>/fd/` folder, where `<PID>` is the process id. We can use the `ps` command to find out the id of a process.

After completing the task, students need to look at the source code, and indicate where the following actions occur. Filename, function name, and line number need to be provided in the answer.

- Server launches the original `app_process` binary
- Client sends its FDs
- Server forks to a child process
- Child process receives client's FDs
- Child process redirects its standard I/O FDs
- Child process launches a root shell

7 Guidelines

7.1 Android Booting Sequence and `app_process`

Figure 5 shows a detailed booting process. In the figure, we assume that the bootloader choses to boot the Android OS, not the recovery OS.

Phase I: The Kernel. After the bootloader switches the control to Android system, Android kernel will be loaded and starts initializing the system. Android kernel is in fact a Linux kernel, which handles some essential parts of the system, such as interruptions, memory protections, scheduling etc. Some Android-specific functionalities are added to the kernel, including logcat logger and wakelocks.

Phase II: The `Init` Process. After the kernel is loaded, `Init` is created as the first user-space process. It is the starting point for all other processes, and it is running under the root privilege. `Init` initializes the virtual file system, detects hardware, then executes script file `init.rc` to configure the system. The script `init.rc` itself mainly focuses on mounting files inside virtual file system and initializes system daemons. However, it imports some other `rc` script files for various purposes, such as setting up environment variables, executing architecture specific commands, and launching `zygote`. Here are the files imported to `init.rc`:

- `init.environ.rc`: Environment variables are set by `init.environ.rc`, which provides some important path-related environment variables. These paths are very important for launching further processes because many of them will try to access these paths using the corresponding environment variable names. In one of the lab tasks, we need to pass environment variables to the child process; now we see why that is needed.

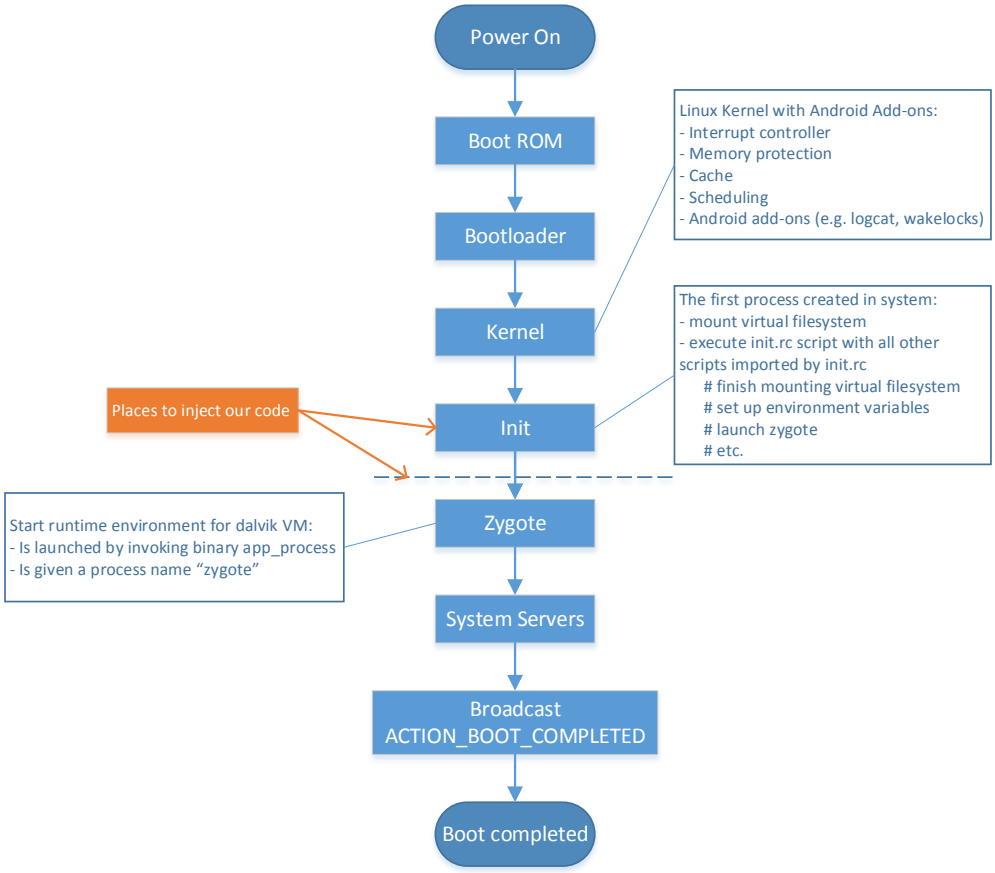


Figure 5: Detailed Booting Process

- `init.${ro.hardware}.rc`: Some commands or codes are architecture specific. The variable `${ro.hardware}` is inherited from the `Init` process and passed to the `init.rc` script. On our Android-x86 VM, it will be "android_x86" and the script file `init.android_x86.rc` file will be invoked. Inside this file, `init.sh` is called. This `init.sh` file is the one used in In Task 1, where we inject the invocation of our code.

- `init.${ro.zygote}.rc`: This file starts a very important daemon called `Zygote`. The variable `${ro.zygote}` is inherited from the `Init` binary. It can be `zygote32` or `zygote64`, for 32-bit and 64-bit architecture, respectively; it can also be `zygote32_64` or `zygote64_32` for hybrid modes (mainMode_secondaryMode). In our Android-x86 VM, it is `init.zygote32.rc`.

All these `rc` script files are stored in an image file, named `ramdisk.img` on our Android-x86 VM; on real devices, these script files are inside `boot.img`, which contains `ramdisk.img` and the kernel. The `ramdisk.img` file will be extracted into the memory while booting up. Making modifications directly on an image file is not very easy. That is the main reason why we only change the `init.sh` file in our first task, because the `init.sh` file is inside the `/system` folder, not inside those image files.

Phase III: The `Zygote` Process. In the `init.${ro.zygote}.rc` file, a special daemon is launched by the `Init` process via the `"service zygote /system/bin/app_process ..."` command

(options are omitted). The command starts a daemon called `zygote`, which executes the `app_process` binary. Zygote is the starting point of the Android runtime. It starts the runtime environment of Dalvik or ART, which are virtual machines that run Java programs. In Android, system servers and most applications are written in Java, so `Zygote` is an essential daemon, and it runs with the root privilege.

That makes `app_process` another place to insert our rooting code. The `app_process` file is not a real binary file; it is a symbolic link, pointing to either `app_process32` or `app_process64`, depending on the architecture of the system. Therefore, we just need to change the symbolic link, and let it point to our code. In our code, we will have two processes, one running our rooting code, and the other running the original `app_process` code. This approach is commonly used by many existing rooting OTA packages.

7.2 How to unlock the boot loader on a real device

As what is shown in Figure 5, when the power button is pressed, the device firsts goes to a fixed location in its ROM, and run instructions from there. These instructions will then go to a pre-defined location on the disk or flash drive to load the bootloader, and pass the control to it. Bootloader then loads the operating system, and eventually gives the control to the loaded OS.

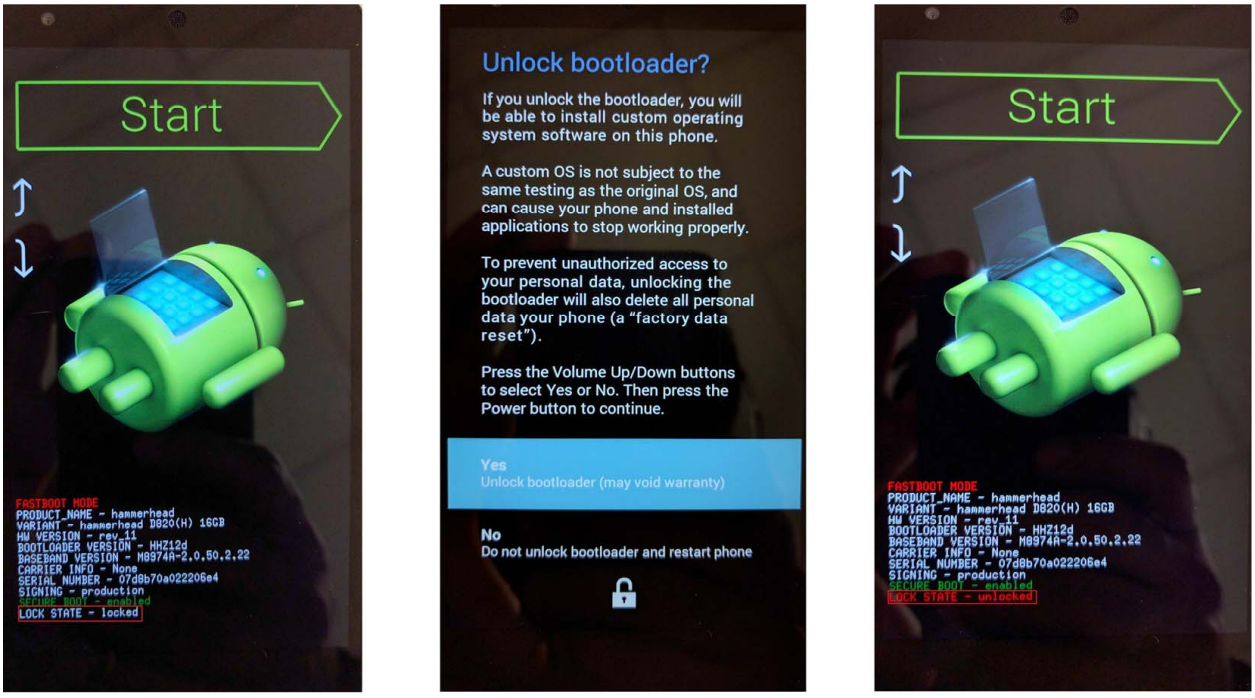
Most of the Android devices come with two operating systems, an Android OS and a recovery OS. By default, the bootloader will choose the Android OS to boot; however, if some special key combination is pressed during the booting, the bootloader will boot the recover OS instead. On Nexus devices, this is achieved by pressing the "Volume Down" and "Power" buttons together.

Bootloader usually have another functionality that is often disabled by default. This functionality allows users to replace (often called *flash*) the OS images on any of the partitions, so users can install a different recovery OS or Android OS. Most manufacturers do not want users to make such kind of modifications to their devices, so before shipping devices to customers, they disable the functionality, and hence we say that "the bootloader is locked". With a locked bootloader, any attempt to flash the installed OS will be denied by the bootloader. The following command tries to flash the recovery OS with a locked bootloader; as we can see, we get an error message:

```
# fastboot flash recovery CustomRecoveryOS.img
sending 'recovery' (11600 KB) ...
OKAY [ 0.483s]
writing 'recovery' ...
FAILED (remote: not supported in locked device)
finished. total time: 0.585s
```

Some vendors choose to permanently lock the bootloader; in this case, it will be very hard to flash the OSes on the devices. However, many vendors choose not to do so, and instead, they provide instructions to unlock the bootloader, so users who really want to flash their devices can still do that. We will show how to unlock the bootloader. For the demonstration purpose we are using a Nexus device. To unlock it, first we need to load bootloader of the Nexus device by passing the "`adb reboot bootloader`" command or by interrupting the normal boot process using the "volume down" and "power" button combination. Figure 6(a) shows the bootloader screen of a Nexus 5 device, which indicates that the bootloader is locked.

Bootloader of Nexus devices can be unlocked by using the "`fastboot oem unlock`" command. Be very careful while unlocking the bootloader because it will void the manufacturer's warranty and completely wipe out personal data on the device. We advise you to backup your personal data before unlocking the bootloader. You can also backup your installed apps and app data by using the "`adb backup -apk -all -f backup.ab`" command, which creates a file called `backup.ab`. After unlocking the bootloader, you can restore the data by running "`adb restore backup.ab`". Figure 6(b) shows the



(a) Locked (b) Unlocking (c) Unlocked

Figure 6: Unlock Bootloader

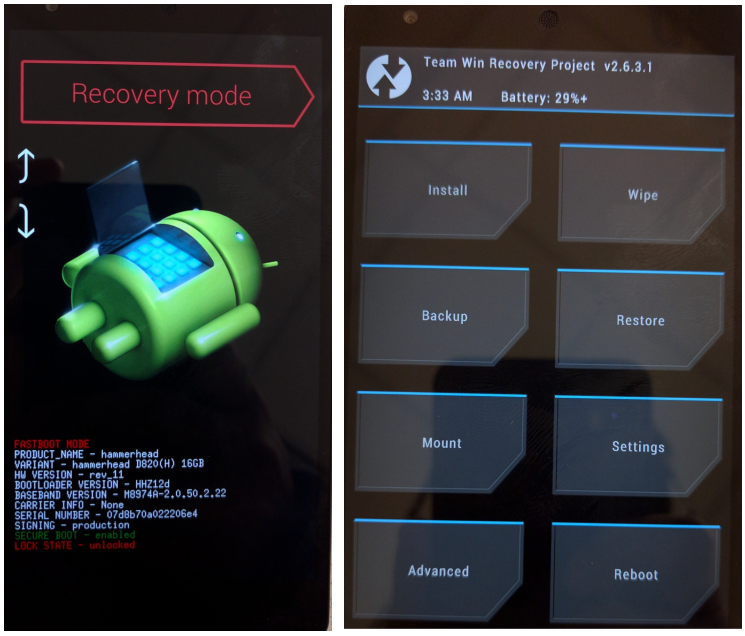
confirmation screen, and Figure 6(c) shows that the bootloader is now unlocked.

7.3 How to replace the existing recovery OS on a real device

On real devices, to remove the restrictions imposed by the stock recovery OS, such as signature verification, we have to replace it with a custom recovery OS that does not impose such restrictions. A number of custom recovery OSes are available on the market. TWRP and ClockworkMod are two of the best choices. We are going to use TWRP in our description. To flash custom recovery we need to unlock the bootloader of the device. We assume that you have already unlocked the bootloader. We can boot the custom Recovery OS by using the "`fastboot boot CustomRecoveryOS.img`" command, or we can permanently replace the device's stock recovery OS with TWRP. The following command flashing the custom recovery OS onto the recovery partition:

```
# fastboot flash recovery CustomRecoveryOS.img
sending 'recovery' (11600 KB) ...
OKAY [ 0.483s]
writing 'recovery' ...
OKAY [ 0.948s]
finished. total time: 1.435s
```

After that, we can boot into the recovery OS by pressing the "volume down" and "power" button combination during the boot-up process. Figure 7(a) shows how to boot into the recovery OS, and Figure 7(b) shows the user interface of the TWRP recovery OS. As you can see, it has several useful features.



(a) Boot into recovery OS (b) TWRP Custom Recovery OS

Figure 7: Custom Recovery OS

8 Submission and Demonstration

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets (if needed). You also need to provide explanation to the observations that are interesting or surprising. You are encouraged to pursue further investigation, beyond what is required by the lab description.