Android Security (and Not) Internals

Yury Zhauniarovich

Version 1.01

Android Security (and Not) Internals

ABOUT THE AUTHOR

Yury Zhauniarovich (http://zhauniarovich.com) is a postdoctoral researcher at the University of Trento. In Security Research Group he works under the guidance of Prof. Bruno Crispo.

He received his Ph.D. degree in Information and Communication Technology from the University of Trento in April, 2014. He earned M.Sc. degree in Computer Science from the Belarusian State University in 2007. From 2007 till 2009, he worked at Itransition.

His research interests include design, implementation and evaluation of security enhancements of mobile operating systems, runtime security, smartphone applications security and mobile malware.

Contents

Preface

In 2011 during my Ph.D. study I changed my research topic and started to work in the area of mobile operating system security. At that time a promising open-source mobile operating system called Android has been conquering the world. Due to its openness it has quickly become a reference testbed for researchers exploring mobile operating systems. During the last several years hundreds of research papers has been appearing only in the area of the security of this operating system. Thus, not surprisingly that we also selected the Android OS as a reference platform for our research.

Unfortunately, at the time when I started to work with this operating system and even three years after the information about this operating system is sparse and scattered around different resources. This does not concern Android application programming – during the last several years lots of books and web resources appeared describing the process and best practices how to develop Android apps. Moreover, the official documentation about app programming is quite complete and a credible source of information on that topic. On the contrary, the official documentation about system programming is pure and gives you good insights only how to download the Android sources and build them. Additional information covers only part of the topics and do not provide you the whole picture. The situation in case of security is even more dismal. Luckily, recently a few sources of valuable information on this topic appeared that flash light on the shadows. The book "Embedded Android" [19] by Karim Yaghmour, the presentations about Android internals by Aleksandar and Marko Gargenta and other resources (often referenced in this work) begin to stick together the puzzles in my mind how the Android system operates.

Despite the presence of these information sources, I still feel the lack of deep information on how the security of Android works. To me, it is interesting not the high-level words about system's operating but deep-dive consideration of particular security features with code examples forming

big picture of the security procurement in the Android operating system.

During the time of Ph.D. thesis writing I decided to collect all bits and pieces on the topic and put them together as a part of my thesis [20]. Basically, I consider this book as the continuation of the work, which first version is appeared in my Ph.D. dissertation. So as mostly I did the exploration by myself, this work may contain inconsistencies and even mistakes. I summon you to not judge strictly and notify me about them. That is why I decided to publish this work as open-source. I do not know if it is going to be successful or not. However, I want to share my knowledge with the Android friendly community that willingly helps if you do not understand something.

Currently, code examples in this book are provided for Android 4.2.2_r1.2 version and for Androlized Linux kernel 3.4 version.

Chapter 1

Android

The comprehension of the Android security architecture helped me not only understand how Android works but also open my eyes how mobile operating systems and Linux are constructed. This chapter considers the basics of the Android architecture from the security perspective. In Section 1.1 we consider the main layers of Android, while Section 1.2 gives high-level overview of the security mechanisms implemented in this operating system.

1.1 Android Stack

Android is a software stack for a wide range of mobile devices and a corresponding open-source project led by Google [9]. Android consists of four layers: *Linux Kernel*, *Native Userspace*, *Application Framework* and *Applications*. Sometimes *Native Userspace* and *Application Framework* layers are combined into the one called *Android Middleware*. Figure 1.1 represents the layers of the Android software stack. Roughly saying, in this figure the green blocks correspond to the components developed in C/C++, while the blue cohere with the ones implemented in Java. Google distributes the most part of the Android code under Apache version 2.0 licence. The most notable exception to this rule is the changes in the *Linux Kernel*, which are under GNU GPL version 2 licence.

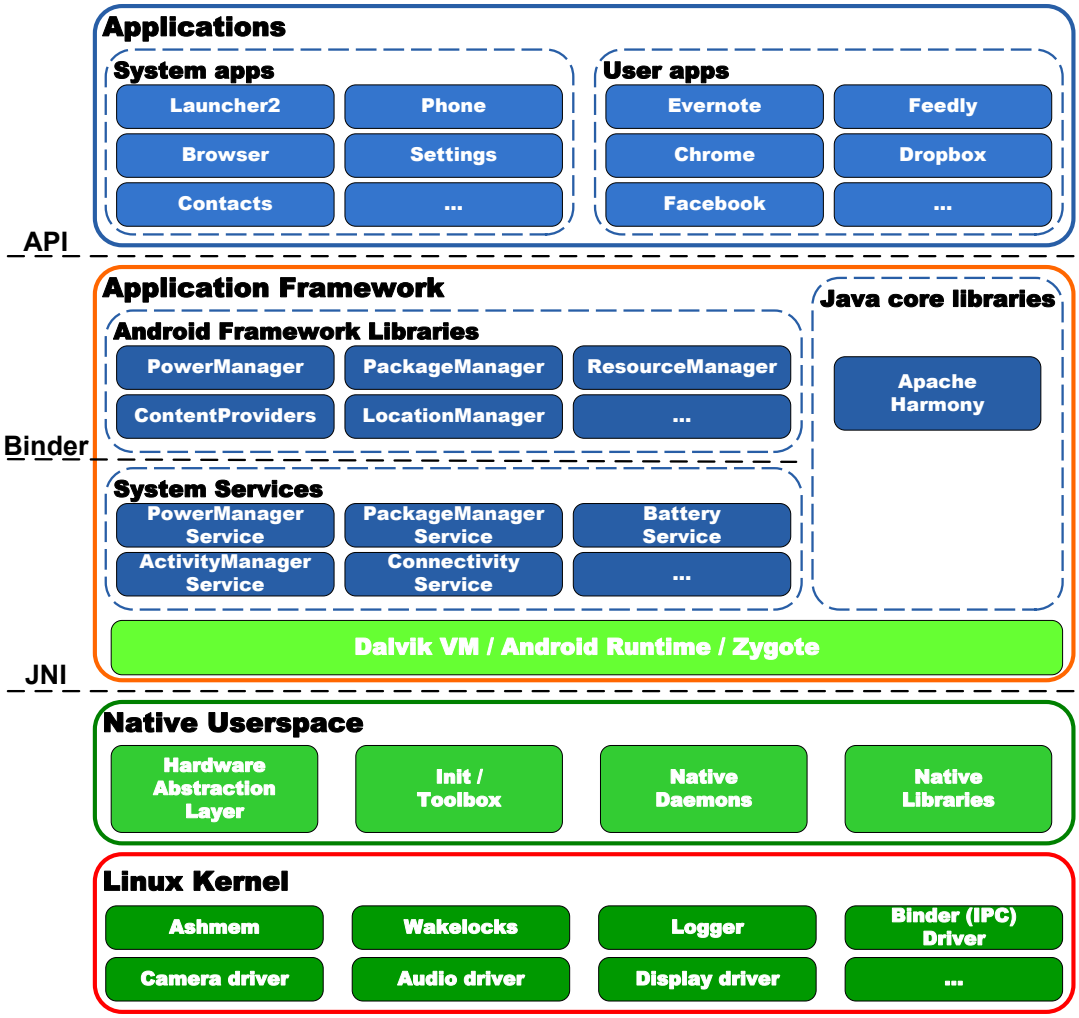

Figure 1.1: Android software stack

Linux Kernel. Before being acquainted by Google in 2005, Android was a startup product of the Android Inc. company. One of the features of startup companies is their tendency to maximise the reuse of already existing components to reduce the time and the cost of their product. So did Android Inc. selecting the *Linux Kernel* as a centerpiece of their new platform. In Android, *Linux Kernel* is responsible for process, memory, communication, filesystem management, etc. While Android mostly relies on the "vanilla" Linux Kernel functionality, several custom changes, which are required for the system operation, have been proposed to this level. Among them *Binder* (a driver, which provides the support for custom

RPC/IPC mechanism in Android), *Ashmem* (a replacement of the standard Linux shared memory functionality), *Wakelocks* (a mechanism that prevents the system from going to sleep) are the most notable ones [19]. Although these changes proved to be very useful in case of mobile operating systems, they are still out of the main branch of *Linux Kernel*.

Native Userspace. By the *Native Userspace* we understand all userspace components that run outside *Dalvik Virtual Machine* and do not belong to the *Linux Kernel* layer. The first component of this layer is *Hardware Abstraction Layer (HAL)* that is actually blurred between the *Linux Kernel* and *Native Userspace* layers. In Linux, drivers for hardware are either embedded into the kernel or loaded dynamically as modules. Although Android is built on top of *Linux Kernel* it exploits a very different approach to support new hardware. Instead, for each type of hardware Android defines an API that is used by upper layers to interact with this type of hardware. The suppliers of a hardware must provide a software module that is responsible for the implementation of the API defined in Android for this particular type of hardware. Thus, this solution allows Android not to embed all possible drivers into the kernel anymore and to disable the dynamic module loading kernel mechanism. The component, which provides this functionality, has been called *Hardware Abstraction Layer* in Android. Additionally, such architectural solution lets hardware suppliers to select the licence, under which their drivers are distributed [18, 19].

Kernel finishes its booting by starting only one userspace process called *init*. This process is responsible for starting all other processes and services in Android, along with performing some operations in the operating system. For instance, if a critical service stops answering in Android, the *init* process can reboot it. This process performs operations in accordance to the init.rc configuration file. *Toolbox* includes essential binaries, which provide shell utilities functionality in Android [19].

Android also relies on a number of key daemons. It starts them during system startup and preserves them running, when the system is work-

ing. For instance, *rild* (the Radio Interface Layer daemon, responsible for communications between baseband processor and other system), *servicemanager* (a daemon, which contains an index of all Binder services running in Android), *adbd* (Android Debug Bridge daemon that serves as a connection manager between host and target equipment), etc.

The last but not least component in *Native Userspace* is *Native Libraries*. There are two types of *Native Libraries*: native libraries that come from external projects, and developed within Android itself. These libraries are loaded dynamically and provide various functionality for Android processes [19].

Application Framework. *Dalvik* is Android's registry-based virtual machine. It allows the operating system to execute Android applications, which are written using Java language. During the built process, Java classes are compiled into a .dex file that are interpreted by the *Dalvik VM*. The *Dalvik VM* was specifically designed to be run in constrained environments. Additionally, the *Dalvik VM* provides functionality to interact with the rest of the system, including native binaries and libraries. To accelerate the process initialization procedure Android exploits a specific component called *Zygote*. This is a special "pre-warmed" process that has all core libraries linked in. When a new app is about to run, Android forks a new process from *Zygote* and sets the parameters of the process according to the specification of the launched application. This solution allows the operating system not to copy linked libraries into a new process, thus, speeding up app launching operation. *Java Core Libraries*, which are used in Android, are borrowed from Apache Harmony project.

System Services is one of the most important parts of Android. Android comes with a number of *System Services* that provide basic mobile operating system functionality to be used by Android app developers in their applications. For instance, `PackageManagerService` is responsible for managing (installation, update, deletion, etc.) Android packages within the operating system. Using *JNI* interfaces system services can

interact with the daemons, toolbox binaries and native libraries of the *Native Userspace* layer. The public API to *System Services* is provided via *Android Framework Libraries*. This API is used by application developers to interact with *System Services*.

Android Applications. Android applications are software applications that run on Android and provide most of the functionality available for the user. The stock Android operating system is shipped with a number of built-in apps called *System Applications*. These are applications compiled as a part of AOSP built process. Moreover, the user may install *User Applications* from numerous app markets to extend basic and introduce new functionality to the operating system.

1.2 Android General Security Description

The core security principle of Android is that an adversary app should not harm the operating system resources, the user and other applications. To procure the execution of this principle, Android being a layered operating system, exploits the provided security mechanisms of all the levels. Focusing on security, Android combines two levels of enforcement [**?**, **?**]: at the *Linux Kernel* level and at the *Application Framework* level (see Figure 1.2).

At the *Linux Kernel* level each application is run in special *Application Sandbox*. The kernel enforces the isolation of applications and operating system components exploiting standard Linux facilities (process separation and Discretionary Access Control over network sockets and filesystem). This isolation is imposed by assigning each application a separate Unix user (UID) and group (GID) identifiers. Such architectural decision enforces running each application in a separate Linux process. Thus, due to *Process Isolation* implemented in Linux, by default applications cannot interfere each other and have limited access to the facilities provided by the operating system. Therefore, *Application Sandbox* ensures that an application cannot drain the operating system resources and cannot interact

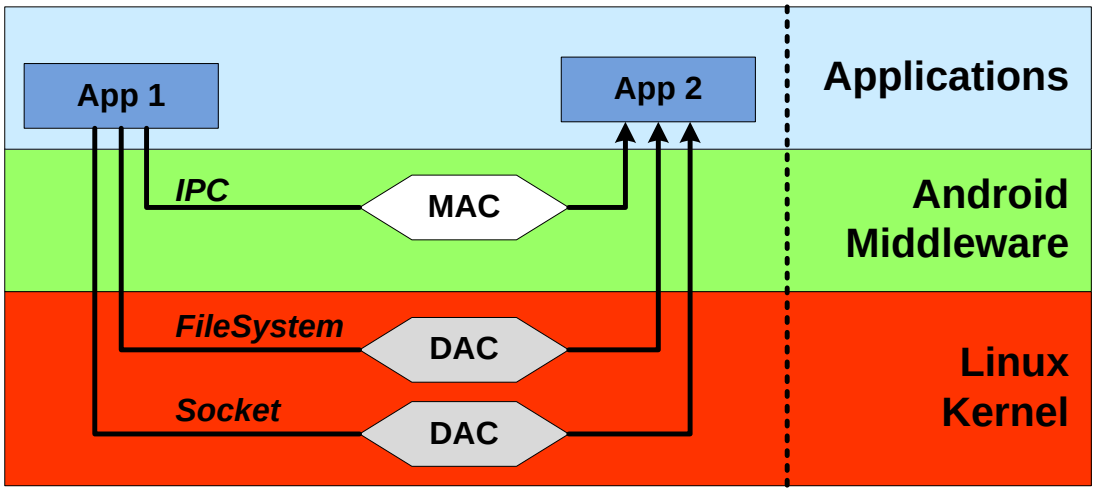

Figure 1.2: Two levels of Android security enforcement

with other apps [3].

The enforcement mechanism provided at the *Linux Kernel* layer effectively sandboxes an application from other apps and the system components. At the same time, an effective communicating protocol is required to allow developers to reuse application components and interact with the operating system units. This protocol is called *Inter-Process Communication* (IPC) because it facilitates the interactions between different processes. In case of Android, this protocol is implemented at the *Android Middleware* level (with a special driver released on the *Linux Kernel* level). The security on this level is provided by the *IPC Reference Monitor*. The reference monitor mediates all communications between processes and controls how applications access the components of the system and other apps. In Android, *IPC Reference Monitor* follows *Mandatory Access Control* (MAC) access control type.

All Android apps by default are run in low-privileged *Application Sandboxes*. Thus, an application has an access only to a limited set of system capabilities. The Android operating system controls the access of apps to the system resources that may adversely impact user experience [3]. This control is implemented in different forms, some of them are considered in details in following chapters. There is also a subset of protected system

features (for instance, camera, telephony or GPS functionality), the access to which should be provided to third-party apps. However, this access should be provided in controlled manner. In case of Android, such control is realized using *Permissions*. Basically, each sensitive API, which provides access to the protected system resources, is assigned with a *Permission* – unique security label. Moreover, protected features may also include components of other applications.

To make the use of protected features, the developer of an application must request the corresponding permissions in the file AndroidManifest.xml. During the installation of an application the Android OS parses this file and presents the user with the list of the permissions declared in this file. The installation of an application occurs according to "all or nothing" principle, meaning that the app is installed only if all permissions are accepted. Otherwise, the application will not be installed. The permissions are granted only at the installation time and can not be modified later. As an example of a permission, consider an application that needs to monitor incoming SMS messages. In this case, the AndroidManifest.xml file must contain in the `<uses-permission>` tag the following declaration: `"android.permission.RECEIVE_SMS"/>`.

An attempt of an application to use a feature, which permission has not been declared in the *Android Manifest* file, will typically result in a thrown security exception. The details of permission enforcement mechanism we consider in the following sections.

Chapter 2

Android Security on the Linux Kernel Level

One of the most widely known open-sources projects, Linux has proved itself as a secure, trusted and stable piece of software being researched, attacked and patched by thousands of people all over the world. Not surprisingly, *Linux Kernel* is the basis of the Android operating system [3]. Android relies on Linux not only for process, memory and filesystem management, it is also one of the most important components in the Android security architecture. In Android *Linux Kernel* is responsible for provisioning *Application Sandboxing* and enforcement of some permission.

2.1 Application Sandboxing

Let consider the process of an Android application installation in details. Android apps are distributed in the form of *Android Package* (.apk) files. A package consists of a Dalvik executable, resources, native libraries and a manifest file, and is signed by a developer signature. There are three main mediators that may install a package on a device in the stock Android operating system:

- **Google Play**.
- **Package Installer**.

- **adb install**.

Google Play is a special application that provides the user with a capability to look for an application uploaded to the market by third-party developers along with a possibility to install it. Although it is also a third-party application, Google Play app (because of being signed with the same signature as the operating system) has access to protected components of Android, which other third-party applications lack for. In case if the user installs applications from other sources she usually implicitly uses *Package Installer* app. This system application provides an interface that is used to start package installation process. The utility *adb install*, which is provided by Android, is mainly used by third-party app developers. While the former two mediators require a user to agree with the list of permissions during the installation process, the latter installs an app quietly. That is why it is mainly used in developer tools aiming at installing an application on a device for testing. This process is shown in the upper part of Figure 2.1. This figure shows more detailed overview of the Android security architecture. We will refer to it here and there in this work to explain the peculiarities of this operating system.

The process of provisioning *Application Sandbox* at the Linux kernel level is the following. During the installation, each package is assigned with a unique user identifier (*UID*) and a group identifier (*GID*) that are not changed during app life on a device. Thus, in Android each application has a corresponding Linux user. User name follows the format `app_x`, and *UID* of that user is equal to `Process.FIRST_APPLICATION_UID + x`, where `Process.FIRST_APPLICATION_UID` constant corresponds to 10000. For instance, in Figure 2.1 `ex1.apk` package during the installation receives `app_1` user name, and *UID* equal to 10001 .

In Linux, all files in memory are subject for Linux *Discretionary Access Control* (DAC). Access permissions are set by a creator or an owner of a file for three types of users: the owner of the file, the users who are in the same group with the owner and all other users. For each type of users, a

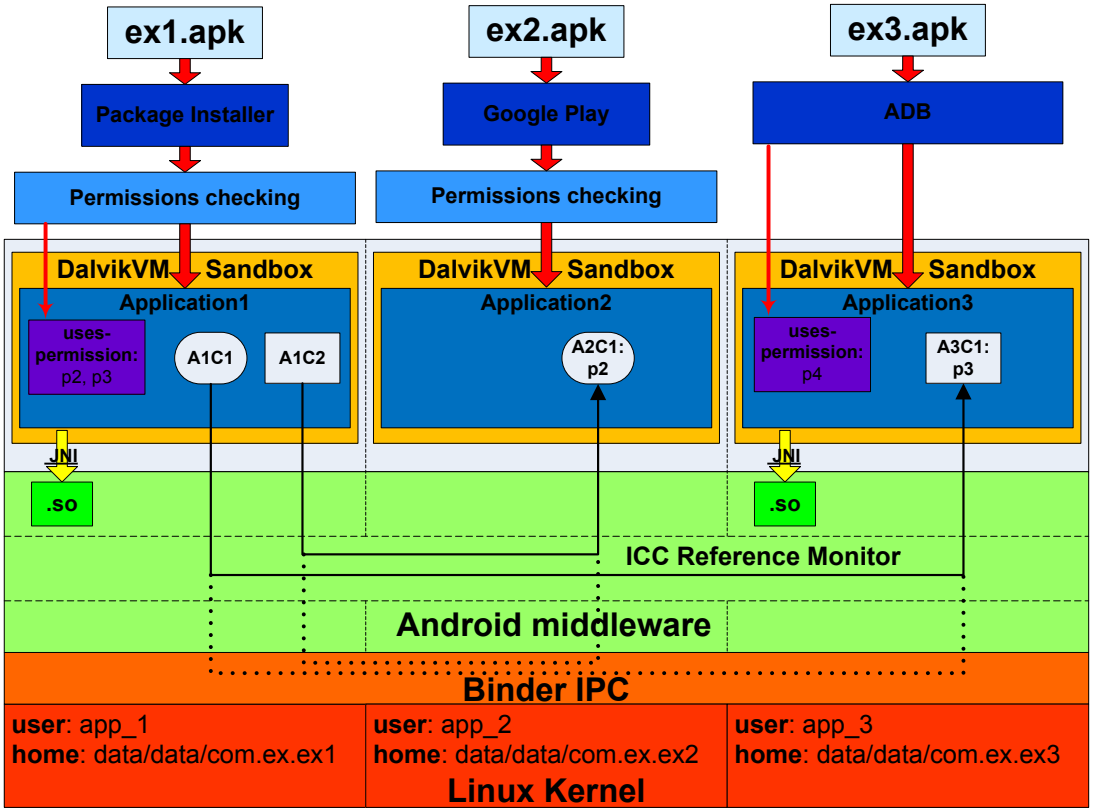

Figure 2.1: Android security architecture

tuple of read, write and execute (r-w-x) permissions are assigned. Hence, so as each application has its own UID and GID, Linux kernel enforces the app execution within its own isolated address space. Beside that, the app unique UIDs and GIDs are used by Linux kernel to enforce fair separation of device resources (memory, CPU, etc.) between different applications. Each application during the installation also receives its own home directory, for instance, /data/data/package_name, where package_name is the name of an Android package, for example, com.ex.ex1 In terms of Android, this folder is *Internal Storage*, where an application keeps its private data. Linux permissions assigned to this directory allows only the "owner" application to write to and read from this directory. It should be mentioned that there are some exceptions. The apps, which are signed with the same certificate, are able to share data between each other, may have the same UID or can even run in the same process.

These architectural decisions set up effective and efficient *Application Sandbox* on the *Linux Kernel* level. This type of sandbox is simple and based on the verified Linux *Discretionary Access Control* model. Luckily, so as the sandbox is enforced on the *Linux Kernel* level, native code and operating system applications are also subject to these constraints described in this chapter [3].

2.2 Permission Enforcement on the Linux Kernel level

It is possible to restrict the access to some system capabilities by assigning the Linux user and group owners to the components that implement this functionality. This type of restrictions can be applied to system resources like files, drivers and sockets. Android uses *Filesystem Permissions* and Android-specific kernel patches (known as *Paranoid Networking*) [13] to restrict the access to low-level system features like network sockets, camera device, external storage, possibility to read logs, etc.

Using filesystem permissions to files and device drivers, it is possible to limit processes in accessing some functionality of a device. For instance, such technique is applied to restrict access of applications to a device camera. The permissions to /dev/cam device driver is set to `0660`, with `root` owner and `camera` owner group. This means that only processes run as `root` or which are included in `camera` group, are able to read from and write to this device driver. Thus, only applications, which are included into `camera` group can interact with the camera. The mappings between permission labels and corresponding groups are defined in the file frameworks/base/data/etc/platform.xml, which excerpt is presented in Listing 2.1. Thus, during the installation if an app has requested the access to a camera feature and the user has approved it, this application is also assigned a `camera` Linux group GID (see corresponding Lines 8 and 9 in Listing 2.1). Therefore, this app receives a possibility to read information from /dev/cam device driver.

There are several points in Android where filesystem permissions to

```
1  ...
2  <permissions>
3  ...
4      <permission name="android.permission.INTERNET" >
5          <group gid="inet" />
6      </permission>
7
8      <permission name="android.permission.CAMERA" >
9          <group gid="camera" />
10     </permission>
11
12     <permission name="android.permission.READ_LOGS" >
13         <group gid="log" />
14     </permission>
15 ...
16 </permissions>
```

Listing 2.1: The mappings between permission labels and Linux groups

files, drivers and unix-sockets are set in: *init* program, init.rc configuration file(s), ueventd.rc configuration file(s) and *system ROM* filesystem config file. They are considered in details in Chapter 3.

In traditional Linux distributions, all processes are allowed to initiate network connections. At the same time, for mobile operating systems the access to networking capabilities has to be controlled. To implement this control in Android, special kernel patches have been added that limit the access to network facilities only to the processes that belong to specific Linux groups or have specific Linux capabilities. These Android-specific patches of the Linux kernel have obtained the name *Paranoid networking*. For instance, for `AF_INET` socket address family, which is responsible for network communication, this check is performed in kernel/net/ipv4/af_inet.c file (see the code extraction in Listing 2.2). The mappings between the Linux groups and permission labels for *Paranoid networking* are also set in platform.xml file (for instance, see Line 4 in Listing 2.1).

Similar *Paranoid Networking* patches are also applied to restrict the access to IPv6 and Bluetooth [19].

The constants used in these checks are hardcoded in the kernel and specified in the kernel/include/linux/android_aid.h file (see Listing 2.3).

Thus, at the *Linux Kernel* level the Android permissions are enforced

```
...
#ifdef CONFIG_ANDROID_PARANOID_NETWORK
#include <linux/android_aid.h>

static inline int current_has_network(void)
{
        return in_egroup_p(AID_INET) || capable(CAP_NET_RAW);
}
#else
static inline int current_has_network(void)
{
        return 1;
}
#endif
...

/*
 *      Create an inet socket.
 */

static int inet_create(struct net *net, struct socket *sock, int protocol,
                       int kern)
{
    ...
    if (!current_has_network())
                return -EACCES;
    ...
}
```

Listing 2.2: Paranoid networking patch

by checking if an application is included into a special predefined group. Only the members of this group have access to the protected functionality. During the installation of an app, if a user has agreed with the requested permission, the application is included into the corresponding Linux group and, hence, receives access to the protected functionality.

```
1  ...
2  #ifndef _LINUX_ANDROID_AID_H
3  #define _LINUX_ANDROID_AID_H
4
5  /* AIDs that the kernel treats differently */
6  #define AID_OBSOLETE_000 3001  /* was NET_BT_ADMIN */
7  #define AID_OBSOLETE_001 3002  /* was NET_BT */
8  #define AID_INET         3003
9  #define AID_NET_RAW      3004
10 #define AID_NET_ADMIN    3005
11 #define AID_NET_BW_STATS 3006  /* read bandwidth statistics */
12 #define AID_NET_BW_ACCT  3007  /* change bandwidth statistics accounting */
13
14 #endif
```

Listing 2.3: Android id constants hardcoded in Linux kernel

Chapter 3

Android Security on the Native Userspace Level

The *Native Userspace* level plays an important role in the security provisioning of the Android operating system. It is impossible to understand how the security architectural decisions are enforced in the system without the comprehension what happens on this layer. In this chapter the topics of the Android booting process and the filesystem peculiarities are considered along with the description how the security is enforced on the *Native Userspace* level.

3.1 Android Booting Process

To understand what procedures provision security on the *Native Userspace* level, at first the booting sequence of an Android device should be considered. It should be mentioned that during the first steps this sequence may vary on different devices but after the Linux kernel is loaded the process is usually the same. The flow of the booting process is shown in Figure 3.1.

When a user powers on a smartphone the CPU of the device will appear in a non-initialised state. In this case, a processor starts executing commands beginning from a hardwired address. This address points to a piece of code in the write-protected memory of the CPU, where *Boot ROM* is located (see Step 1 in Figure 3.1). The main aim of the code resided on Boot

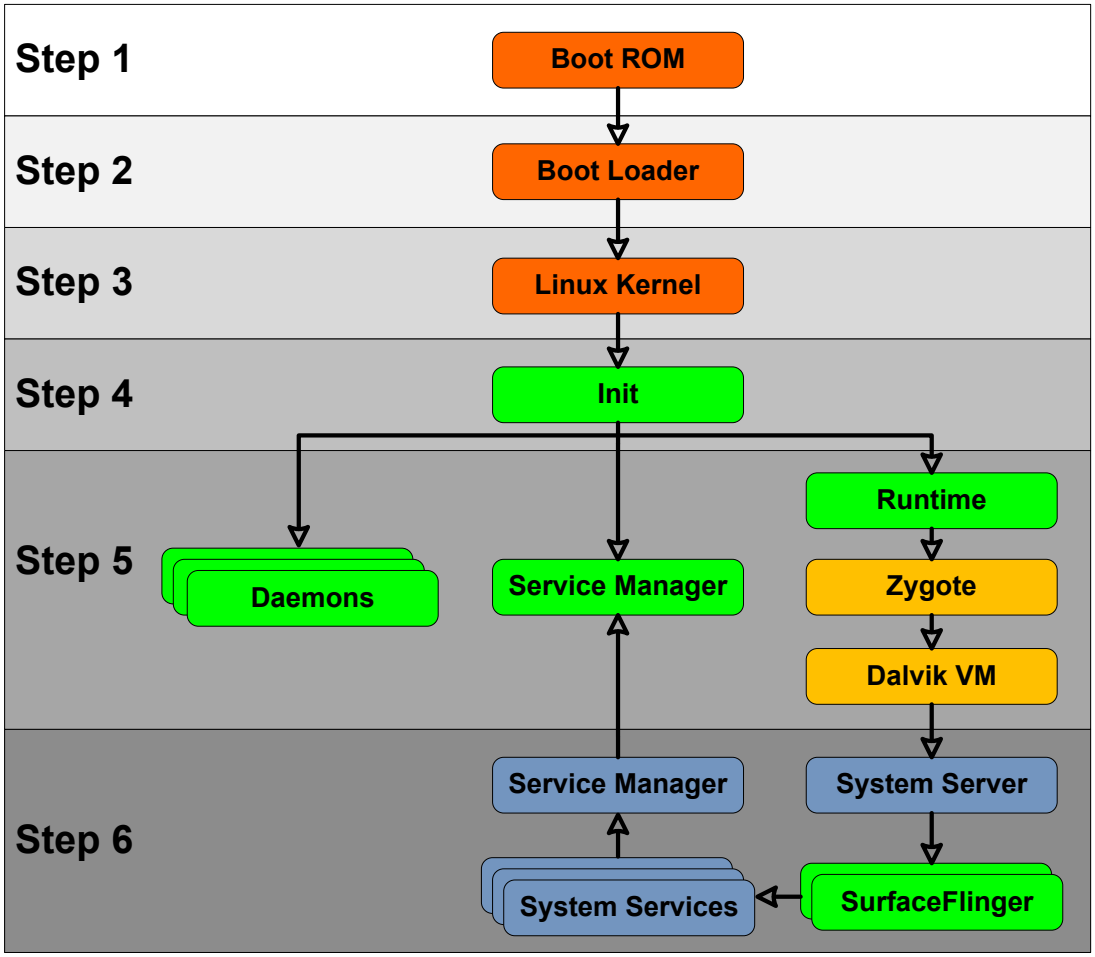

Figure 3.1: Android boot sequence

ROM is to detect a media, where *Boot Loader* is located [17]. When the detection is done, Boot ROM loads Boot Loader into the internal memory (which is only available after device power-on) and performs a jump to the loaded code of Boot Loader. On its turn, *Boot Loader* sets up external RAM, filesystem and network support. After that it loads *Linux Kernel* into the memory and passes the execution to it. *Linux Kernel* initialises the environment to run C code, activates interrupt controllers, sets up memory management units, defines scheduling, loads drivers and mounts root filesystem. When memory management units are initialized, the system is ready to use virtual memory and run user-space processes [17]. Actually, starting from this step the process does not differ from the one that occurs

on desktop computers running Linux.

The first user-space process, which is an ancestor of all processes in Android, is *init*. The executable of this program is located in the `root` directory of the Android filesystem. Listing 3.1 contains the focal points of the sources of this executable. It can be seen that the *init* binary is responsible for the creation of the basic filesystem entries (Lines from 7 to 16). After that (Line 18), the program parses the init.rc configuration file and executes the commands written there.

```
1  int main(int argc, char **argv)
2  {
3  ...
4      if (!strcmp(basename(argv[0]), "ueventd"))
5          return ueventd_main(argc, argv);
6  ...
7      mkdir("/dev", 0755);
8      mkdir("/proc", 0755);
9      mkdir("/sys", 0755);
10
11     mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
12     mkdir("/dev/pts", 0755);
13     mkdir("/dev/socket", 0755);
14     mount("devpts", "/dev/pts", "devpts", 0, NULL);
15     mount("proc", "/proc", "proc", 0, NULL);
16     mount("sysfs", "/sys", "sysfs", 0, NULL);
17 ...
18     init_parse_config_file("/init.rc");
19 ...
20 }
```

Listing 3.1: The sources of *init* program

The init.rc configuration file is written using a language called *Android Init Language* and located in the root directory. This configuration file can be imagined as a list of actions (sequence of commands), which execution is triggered by the predefined events. For instance, in Listing 3.2, `fs` (Line 1) is a *trigger*, while Lines 4 – 7 represent the *Actions*. The commands written in the init.rc configuration file defines system global variables, sets up basic kernel parameters for memory management, configures filesystem, etc. What is more important from the security perspective, it is also responsible for the basic filesystem structure creation and for the assignment of the owners and the filesystem permissions to the created nodes.

```
1 on fs
2     # mount mtd partitions
3     # Mount /system rw first to give the filesystem a chance to save a checkpoint
4     mount yaffs2 mtd@system /system
5     mount yaffs2 mtd@system /system ro remount
6     mount yaffs2 mtd@userdata /data nosuid nodev
7     mount yaffs2 mtd@cache /cache nosuid nodev
```

Listing 3.2: The list of actions performed on `fs` trigger in emulator

Additionally, the *init* program is responsible for starting several essential daemons and processes in Android (see Step 5 in Figure 3.1), the parameters of which are also defined in the init.rc file. An executed process in Linux by default is run with the same permissions (under the same UID) as an ancestor. In Android, *init* is started with the *root* privileges (`UID == 0`). This means that all descendant processes should run with the same UID. Luckily, the privileged processes may change their UIDs to the less privileged ones. Thus, all descendants of the *init* process may use this functionality specifying the UID and the GID of a forked process (the owner and group are also defined in the init.rc file).

One of the first daemons, which is forked from the *init* process, is the *ueventd* daemon. This service runs its own `main` function (see Line 5 in Listing 3.1) that reads the ueventd.rc and ueventd.[device_name].rc configuration files and replays the specified there kernel *uevent* hotplug events. These events set up the owners and permissions for different devices (see Listing 3.3). For instance, Line 5 shows how the filesystem permissions to `/dev/cam` device are set, which example was considered in Section 2.2. After that, the daemon waits listening for all future hotplug events.

```
1 ...
2 /dev/ashmem               0666   root       root
3 /dev/binder               0666   root       root
4 ...
5 /dev/cam                  0660   root       camera
6 ...
```

Listing 3.3: ueventd.rc file

One of the core services started by the *init* program is *servicemanager*

(see Step 5 in Figure 3.1). This service acts as an index of all services running in Android. It must be available on early phase because all system services, which are started afterward, should have a possibility to register themselves and, thus, become visible to the rest of the operating system [19].

Another core process launched by the *init* process is *Zygote*. *Zygote* is a special process that has been warmed-up. This means that the process has been initialised and linked against the core libraries. *Zygote* is an ancestor for all processes. When a new application is started, *Zygote* forks itself. After that, the parameters corresponding to a new application, for instance, UID, GIDs, nice-name, etc., are set for the forked child process. The acceleration of a new process creation is achieved because there is no need to copy core libraries into the new process. The memory of a new process has "copy-on-write" protection, meaning that the data will be copied from the zygote process to a new one only if the latter tries to write into the protected memory. So as core libraries cannot be changed, they are remained only in one place reducing memory consumption and the application startup time.

The first process, which is run using *Zygote* is *System Server* (Step 6 in Figure 3.1). This process, at first, runs native services, such as *SurfaceFlinger* and *SensorService*. After the services initialized, a callback is invoked, which starts the remaining services. All these services are then registered with *servicemanager*.

3.2 Android Filesystem

Although Android is based on *Linux Kernel*, its filesystem hierarchy does not comply with Filesystem Hierarchy Standard [10] that defines filesystem layout of Unix-like systems (see Listing 3.4). Some directories in Android and in Linux are the same, for instance, /dev, /proc, /sys, /etc, /mnt, etc. The purposes of these folders are the same as in Linux. At the same time, there are directories, such as /system, /data and /cache, which cannot be

found in the Linux systems. These folders are the core parts of Android. During the build of the Android operating system, three image files are created: system.img, userdata.img and cache.img. These images provide the core functionality of Android and are the ones that are flashed on a device. During the boot of the system the *init* program mounts these images to the predefined mounting points, like /system, /data and /cache correspondingly (see Listing 3.2).

```
1  drwxr-xr-x root     root              2013-04-10 08:13 acct
2  drwxrwx--- system   cache             2013-04-10 08:13 cache
3  dr-x------ root     root              2013-04-10 08:13 config
4  lrwxrwxrwx root     root              2013-04-10 08:13 d -> /sys/kernel/debug
5  drwxrwx--x system   system            2013-04-10 08:14 data
6  -rw-r--r-- root     root          116 1970-01-01 00:00 default.prop
7  drwxr-xr-x root     root              2013-04-10 08:13 dev
8  lrwxrwxrwx root     root              2013-04-10 08:13 etc -> /system/etc
9  -rwxr-x--- root     root       244536 1970-01-01 00:00 init
10 -rwxr-x--- root     root         2487 1970-01-01 00:00 init.goldfish.rc
11 -rwxr-x--- root     root        18247 1970-01-01 00:00 init.rc
12 -rwxr-x--- root     root         1795 1970-01-01 00:00 init.trace.rc
13 -rwxr-x--- root     root         3915 1970-01-01 00:00 init.usb.rc
14 drwxrwxr-x root     system            2013-04-10 08:13 mnt
15 dr-xr-xr-x root     root              2013-04-10 08:13 proc
16 drwx------ root     root              2012-11-15 05:31 root
17 drwxr-x--- root     root              1970-01-01 00:00 sbin
18 lrwxrwxrwx root     root              2013-04-10 08:13 sdcard -> /mnt/sdcard
19 d---r-x--- root     sdcard_r          2013-04-10 08:13 storage
20 drwxr-xr-x root     root              2013-04-10 08:13 sys
21 drwxr-xr-x root     root              2012-12-31 03:20 system
22 -rw-r--r-- root     root          272 1970-01-01 00:00 ueventd.goldfish.rc
23 -rw-r--r-- root     root         4024 1970-01-01 00:00 ueventd.rc
24 lrwxrwxrwx root     root              2013-04-10 08:13 vendor -> /system/vendor
```

Listing 3.4: Android filesystem

The /system partition incorporates the entire Android operating system except the Linux kernel, which itself is located on the /boot partition. This folder contains the subdirectories /system/bin and /system/lib that contain core native executables and shared libraries correspondingly. Additionally, this partition encompass all system applications that are prebuilt with the system image. The image is mounted in read only mode (see Line 5 in Listing 3.2). Hence, the content of this partition cannot be changed at runtime.

So as /system partition is mounted as read-only, it cannot be used for

storing data. For this purposes the separate partition /data is allocated that responsible for storing user data or information changing over the time. For instance, /data/app directory contains all apk files of installed applications, while /data/data folder encloses "home" directories of the apps.

The /cache partition is responsible for storing frequently accessed data and application components. Additionally, the operating system over-the-air updates are also stored on this partition before being run.

So as /system, /data and /cache are formed during the compilation of Android, the default rights and owners to the files and folders contained on these images have to be defined at compile time. This means that the user and groups UIDs and GIDs should be available during the compilation of this operating system. The android_filesystem_config.h file (see Listing 3.5) contains the list of predefined users and groups. It should be mentioned that the values in some lines (for instance, see Line 10) correspond to the ones already defined on the *Linux Kernel* level, described in Section 2.2.

Additionally, in this file the default rights, owners and owner groups of the files and folders are defined (see Listing 3.6). These rules are parsed and applied by `fs_config()` function, which is defined in the end of this file. This function is called during the assembly of the images.

3.2.1 Native Executables Protection

It can be mentioned in Listing 3.6 that some binaries are assigned with *setuid* and *setgid* access rights flags. For instance, the *su* program has them set. This well-known utility allows a user to run a program with the specified UID and GID. In Linux this functionality is usually used to run programs with superuser privileges. According to Listing 3.6, the binary /system/xbin/su is assigned with the access rights equal to "06755" (see Line 21). The first non-zero number "6" means that this binary has *setuid* and *setgid* (4 + 2) access rights flags set. Usually, in Linux an executable is run with the same privileges as the process that has started it. These

```
1  #define AID_ROOT             0 /* traditional unix root user */
2  #define AID_SYSTEM        1000 /* system server */
3  #define AID_RADIO         1001 /* telephony subsystem, RIL */
4  #define AID_BLUETOOTH     1002 /* bluetooth subsystem */
5  #define AID_GRAPHICS      1003 /* graphics devices */
6  #define AID_INPUT         1004 /* input devices */
7  #define AID_AUDIO         1005 /* audio devices */
8  #define AID_CAMERA        1006 /* camera devices */
9  ...
10 #define AID_INET          3003 /* can create AF_INET and AF_INET6 sockets */
11 ...
12 #define AID_APP          10000 /* first app user */
13 ...
14 static const struct android_id_info android_ids[] = {
15     { "root",       AID_ROOT, },
16     { "system",     AID_SYSTEM, },
17     { "radio",      AID_RADIO, },
18     { "bluetooth", AID_BLUETOOTH, },
19     { "graphics",   AID_GRAPHICS, },
20     { "input",      AID_INPUT, },
21     { "audio",      AID_AUDIO, },
22     { "camera",     AID_CAMERA, },
23 ...
24     { "inet",       AID_INET, },
25 ...
26 };
```

Listing 3.5: Android hard-coded UIDs and GIDs and their mapping to user names

flags allows a user to run a program with the privileges of executable owner or group [11]. Thus, in our case the binary /system/xbin/su will be run as *root* user. These *root* privileges allow the program to change its UID and GID to the ones specified by a user (see Line 15 in Listing 3.7). After that, *su* may start the provided program (for instance, see Line 22) with the specified UID and GID. Therefore, the program will be started with the required UID and GID.

In the case of privileged programs it is required to restrict the circle of applications that have access to such utilities. In our case, without such restrictions any app may run *su* program and obtain *root* level privileges. In Android, such restrictions on the *Native Userspace level* are implemented comparing the UID of the calling program with the list of the UIDs allowed to run it. Thus, in Line 9 the *su* executable obtains the current UID of the process, which is equal to the UID of the process calling it, and in Line 10 it compares this UID with the predefined list of allowed UIDs. Therefore,

```
/* Rules for directories.*/
static struct fs_path_config android_dirs[] = {
    { 00770, AID_SYSTEM, AID_CACHE,  "cache" },
    { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
...
    { 00777, AID_ROOT,   AID_ROOT,   "sdcard" },
    { 00755, AID_ROOT,   AID_ROOT,   0 },
};

/* Rules for files.*/
static struct fs_path_config android_files[] = {
...
    { 00644, AID_SYSTEM,      AID_SYSTEM,      "data/app/*" },
    { 00644, AID_MEDIA_RW,    AID_MEDIA_RW,    "data/media/*" },
    { 00644, AID_SYSTEM,      AID_SYSTEM,      "data/app-private/*" },
    { 00644, AID_APP,         AID_APP,         "data/data/*" },
    ...
    { 02755, AID_ROOT,        AID_NET_RAW,     "system/bin/ping" },
    { 02750, AID_ROOT,        AID_INET,        "system/bin/netcfg" },
    ...
    { 06755, AID_ROOT,        AID_ROOT,        "system/xbin/su" },
    ...
    { 06750, AID_ROOT,        AID_SHELL,       "system/bin/run-as" },
    { 00755, AID_ROOT,        AID_SHELL,       "system/bin/*" },
    ...
    { 00644, AID_ROOT,        AID_ROOT,         0 },
};
```

Listing 3.6: Default permissions and owners

only if the UID of the calling process is equal to `AID_ROOT` or `AID_SHELL` the *su* utility will be started. To perform such check, *su* imports the UID constants (see Line 1) defined in Android.

Additionally, in newer versions (starting from 4.3) the Android core developers started to use *Capabilities* Linux kernel system [4]. This allows them additionally restrict the privileges of the programs that are required to run with *root* privileges. For instance, in the considered case of the *su* program it is not required to have all privileges of the *root* user. For this program it is enough only to have a possibility to change current UID and GID. Therefore, this utility requires only `CAP_SETUID` and `CAP_SETGID` root capabilities to operate correctly.

```c
#include <private/android_filesystem_config.h>
...
int main(int argc, char **argv)
{
    struct passwd *pw;
    int uid, gid, myuid;

    /* Until we have something better, only root and the shell can use su. */
    myuid = getuid();
    if (myuid != AID_ROOT && myuid != AID_SHELL) {
        fprintf(stderr,"su: uid %d not allowed to su\n", myuid);
        return 1;
    }
    ...
    if(setgid(gid) || setuid(uid)) {
        fprintf(stderr,"su: permission denied\n");
        return 1;
    }

    /* User specified command for exec. */
    if (argc == 3 ) {
        if (execlp(argv[2], argv[2], NULL) < 0) {
            fprintf(stderr, "su: exec failed for %s Error:%s\n", argv[2],
                    strerror(errno));
            return -errno;
        }
    ...
}
```

Listing 3.7: Source code of *su* program

Chapter 4

Android Security on the Framework Level

As we described in Section 1.2 the security on the *Application Framework* level is enforced by *IPC Reference Monitor*. In Section 4.1 we start our consideration of the security mechanisms on this level from the description of the inter-process communication system used in Android. After that we introduce *permissions* in Section 4.2, while in Section 4.3 we describe the permission enforcement system implemented on this level.

4.1 Android Binder Framework

As we described in Section 2.1, all Android applications are run in *Application Sandboxes*. Roughly saying, the sandboxing of the apps is provisioned by running all apps in different processes with different Linux identities. Additionally, system services are also run in separate processes with more privileged identities that allow them to get access to different parts of the system protected using the Linux Kernel DAC capabilities (see Sections 2.1, 2.2 and 1.2). Thus, an Inter-Process Communication (IPC) framework is required to organize data and signals exchange between different processes. In Android, a special framework called *Binder* is used for inter-process communication [12]. The standard Posix *System V IPC*

framework is not supported [1] by the Android implementation of the *Bionic libc* library. Moreover, additionally to the *Binder* framework for some special cases Unix domain sockets are used (e.g., for communication with the *Zygote* daemon) but the consideration of these mechanisms is out of the scope of this work.

The *Binder* framework was specifically redeveloped to be used in Android. It provides the capabilities required to organize all types of communication between processes in this operating system. Basically, even the mechanisms, such as *Intents* and *ContentProviders*, well-known to application developers, are built on top of the *Binder* framework. This framework provides the variety of features, such as the possibility to invoke methods on remote objects as if they were local, synchronous and asynchronous method invocation, *link to death* [2], ability to send file descriptors across processes, etc. [12, 16].

The communication between the processes is organized according to synchronous client-server model. The client initiates a connection and waits for a reply from the server side. Thus, the communication between the client and the server may be imagined as they are executed in the same process thread. This provides a developer with the possibility to invoke methods on remote objects as if they were local. The communication model through Binder is presented in Figure 4.1. In this figure, the application in *Process A*, which acts as a Client, wants to use the behavior exposed by a Service, which runs in *Process B* [12].

All communications between clients and services using the *Binder* framework happens through a Linux kernel driver `/dev/binder`. The permissions to this device driver is set to world readable and writable (see Line 3 in Listing 3.3 located in Section 3.1). Hence, any application may write to and read from this device. To conceal the peculiarities of the *Binder* communication protocol, the *libbinder* library is used in Android. It provides

[1] `https://android.googlesource.com/platform/ndk/+/android-4.2.2_r1.2/docs/system/libc/SYSV-IPC.html`

[2] Link to Death is an automatic notification when a Binder of a certain process is terminated

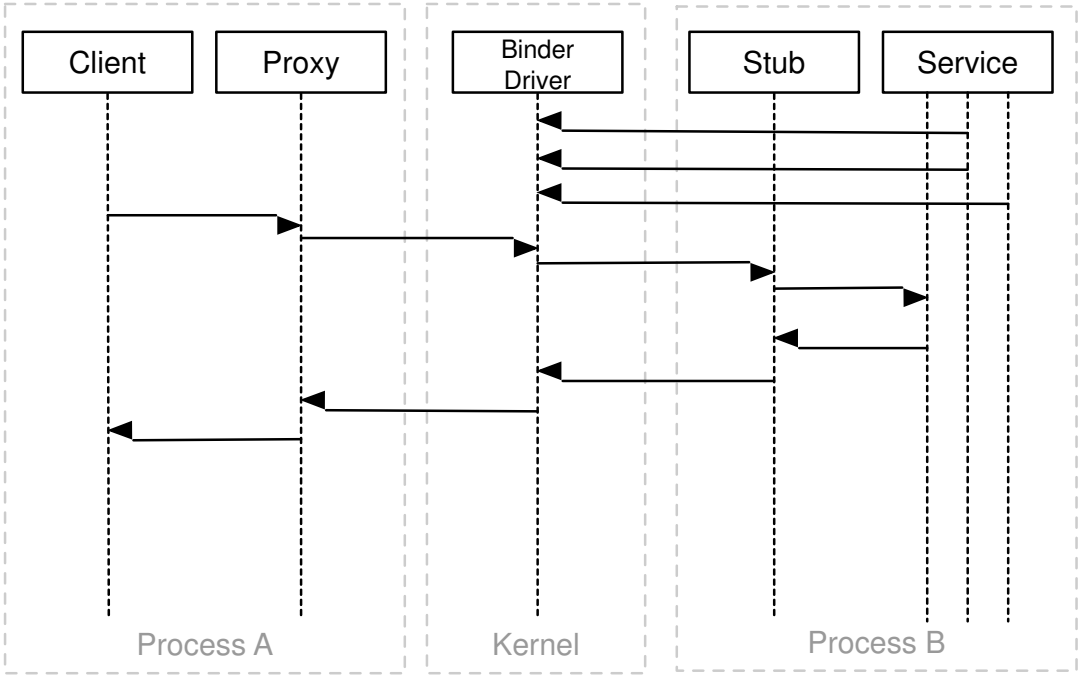

Figure 4.1: Android Binder communication model [12]

the facilities to make the process of interaction with the kernel driver transparent for an app developer. In particular, all communications between a Client and a Server happen through proxies on the client side and stubs on the server side. The proxies and the stubs are responsible for marshaling and unmarshaling the data and the commands sent over the *Binder* driver. To make use of proxies and stubs a developer just defines an *AIDL* interface that is transformed into a proxy and a stub during the compilation of the application. On the server side, a separate Binder thread is invoked to process a client request.

Technically, each Service (sometimes called as *Binder Service*) exposed using the *Binder* mechanism is assigned with a token. The kernel driver ensures that this 32 bit value is unique across all processes in the system. Thus, this token is used as a handle to a *Binder Service*. Having this handle it is possible to interact with the Service. However, to start using the Service the Client at first has to discover this value. The discovery of Service's handle occurs using Binder's *context manager* (*servicemanager* is

Android's implementation of Binder's *context manager*. Here we use these notions interchangeably). *Context manager* is a special *Binder Service* with the predefined handle value equal to `0` (the reference to which is obtained in Line 8 in Listing 4.1). So as it has a fixed handle value, any party can find it and call its methods. Basically, *context manager* acts as a name service providing the handle of a Service using the name of this Service. To achieve this goal, each Service must be registered with *context manager* (for instance, using the method `addService` of the `ServiceManager` class in Line 26). Thus, a Client has to know only the name of a Service to communicate with it. Resolving this name using *context manager* (see method `getService` Line 12) the Client receives the token that is later used for the interactions with the Service. The *Binder* driver allows only a single *context manager* to be registered. Therefore, *servicemanager* is one of the first services started by Android (see Section 3.1). The component *servicemanager* ensures that only the privileged system identities are allowed to register services.

The *Binder* framework does not impose any security by itself. At the same time, it provides the facilities to procure the security in Android. The *Binder* driver adds the UID and the PID of the sender process to each transaction. So as each application in the system has its own UID, this value may be used to identify the calling party. The receiver of the call may check the obtained values and decide if the transaction should be completed. The receiver may get the UID and the PID of the sender using the calls `android.os.Binder.getCallingUid()` and `android.os.Binder.getCallingPid()` [12]. Additionally, a *Binder* handle may also act as a security token due to its uniqueness across all the processes and the obscurity of its value [14].

4.2 Android Permissions

As we consider in Section 2.1, in Android each application by default obtains its own UID and GID system identities. Additionally, there are also

```
1  public final class ServiceManager {
2      ...
3      private static IServiceManager getIServiceManager() {
4          if (sServiceManager != null) {
5              return sServiceManager;
6          }
7          // Find the service manager
8          sServiceManager = ServiceManagerNative.asInterface(BinderInternal.
       getContextObject());
9          return sServiceManager;
10     }
11
12     public static IBinder getService(String name) {
13         try {
14             IBinder service = sCache.get(name);
15             if (service != null) {
16                 return service;
17             } else {
18                 return getIServiceManager().getService(name);
19             }
20         } catch (RemoteException e) {
21             Log.e(TAG, "error in getService", e);
22         }
23         return null;
24     }
25
26     public static void addService(String name, IBinder service, boolean allowIsolated) {
27         try {
28             getIServiceManager().addService(name, service, allowIsolated);
29         } catch (RemoteException e) {
30             Log.e(TAG, "error in addService", e);
31         }
32     }
33     ...
34 }
```

Listing 4.1: The sources of `ServiceManager`

a number of the identities hardcoded in the operating system (see Listing 3.5). These identities are used to separate the components of the Android operating system using the *DAC* enforced on the *Linux Kernel* level, thus, increasing the overall security of the operating system. Among these identities `AID_SYSTEM` stands out. This UID is used to run the *System Server* (`system_server`), the component that unites the services provided by the Android OS. The *System Server* has a privileged access to the operating system resources, and each service run within the *System Server* provides the controlled access to a particular functionality to other OS components and applications. This controlled access is backed by the

permission system.

As we consider in Section 4.1, the *Binder* framework provides the ability to get the UID and the PID of the sender on the receiver side. In general case, this functionality may be exploited by a service to control consumers that want to connect to the service. This can be achieved by comparing the UID and/or PID of a consumer with the list of UIDs allowed by the service. However, in Android this functionality is implemented in a slightly different manner. Each critical functionality of a service (or simply saying a method of a service) is guarded with a special label called *permission*. Roughly saying, before running such method a check if the calling process is assigned with the *permission*, is performed. If the calling process has the required permission then the service invocation will be allowed. Otherwise, a security check exception will be thrown (usually, `SecurityException`). For instance, if a developer wants to provide her app with a possibility to send SMS she has to add into app's `AndroidManifest.xml` file the following line `<uses-permission android:name="android.permission.SEND_SMS" />`. Android also provides a set of special calls that allow to check at runtime if a service consumer has been assigned with a permission.

The permission model described so far provides an effective way to enforce security. At the same time, this model is ineffective because it considers all the permissions as equal. At the same time, in the case of mobile operating systems the provided capabilities may not be always equal in the security sense. For instance, the capability to install applications is more critical then the ability to send SMSes, which in turn is more dangerous then the setting an alarm or vibrating.

This problem is addressed in Android by introducing the security levels of permissions. There are four possible levels of permissions: `normal`, `dangerous`, `signature` and `signatureOrSystem`. The level of permissions is either hardcoded into the Android operating system (for system permissions) or assigned by a developer of a third-party app in the declaration of a custom permission. This level influences on a decision whether to grant

the permission to a requesting application. To be granted, the `normal` permissions have to be just requested in application's AndroidManifest.xml file. The `dangerous` permissions, besides to be requested in the manifest file, have to be also approved by a user. In this case, during the installation of an app the user is displayed with the set of permissions requested by the package. If the user approves them, then the application will be installed. Otherwise, the installation will be canceled. The `signature` permission is granted by the system if the app requested the permission is signed with the same signatures as the application that has declared it (the usage of app signatures in Android is considered in Section 6.1). The `signatureOrSystem` permission is granted either if the apps requesting and the declaring the permission are signed with the same certificates or the requesting application is located on the system image. Thus, for our example the vibrating capability will be protected with the permission of the `normal` level, send SMSes functionality will be guarded with the `dangerous` permission level and package installation ability will be secured with the `signatureOrSystem` permission level.

4.2.1 System Permission Definitions

System permissions, which are used to protect Android operating system functionality, are defined in framework's AndroidManifest.xml file located in frameworks/base/core/res folder of the Android sources. An excerpt of this file with several permission definition examples is shown in Listing 4.2. In these examples the permission declarations are shown used to protect sending SMSes, vibrator and package installation functionality.

By default the developers of third-party applications do not have access to the functionality protected with system permissions of levels `signature` and `signatureOrSystem`. This behaviour is ensured in the following way. The *Application Framework* package is signed with the *platform* certificate. Thus, the applications requiring the functionality protected with the permissions of these levels must be signed with the same *platform* certificate.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="android" coreApp="true" android:sharedUserId="android.uid.system"
    android:sharedUserLabel="@string/android_system_label">
    ...
    <!-- Allows an application to send SMS messages. -->
    <permission android:name="android.permission.SEND_SMS"
        android:permissionGroup="android.permission-group.MESSAGES"
        android:protectionLevel="dangerous"
        android:permissionFlags="costsMoney"
        android:label="@string/permlab_sendSms"
        android:description="@string/permdesc_sendSms" />
    ...
    <!-- Allows access to the vibrator -->
    <permission android:name="android.permission.VIBRATE"
        android:permissionGroup="android.permission-group.AFFECTS_BATTERY"
        android:protectionLevel="normal"
        android:label="@string/permlab_vibrate"
        android:description="@string/permdesc_vibrate" />
    ...
    <!-- Allows an application to install packages. -->
    <permission android:name="android.permission.INSTALL_PACKAGES"
        android:label="@string/permlab_installPackages"
        android:description="@string/permdesc_installPackages"
        android:protectionLevel="signature|system" />
    ...
</manifest>
```

Listing 4.2: The definitions of system permissions

However, the access to the private key of this certificate is available only to the builders of the operating system, usually hardware producers (who make their own customization of Android) or telecommunication operators (who distribute the phones with their modified images of operating systems).

4.2.2 Permission Management

The system service `PackageManagerService` is responsible for the application management in Android. This service assists the installation, uninstallation and update of applications in the operating system. Another important role of this service is permission management. Basically, it can be considered as a policy administration point. It stores the information that allows to check if an Android package is assigned with a particular permission. Additionally, during the installation and upgrade of applications it performs a bunch of checks to ensure that the integrity of permission

model is not violated during these processes. Moreover, it also acts as a policy decision point. The methods of this service (as we will show later) are the last elements in the chain of the permission checks. We will not consider the operation of `PackageManagerService` here. However, the interested reader may refer to [15, 19] to get some more details how the installation of applications is performed.

`PackageManagerService` stores all information related to permissions of third-party applications in the /data/system/packages.xml [7]. This file is used as a persistent storage between the restarts of the system. However, at runtime all information about permissions is preserved in RAM allowing to increase the responsiveness of the system. This information is collected during the boot using the data stored in the packages.xml file for third-party applications and through parsing system apps.

4.3 Permission Enforcement on the Application Framework level

To understand how Android enforces permissions on the *Application Framework* level, for instance, let consider the *Vibrator Service*. In Listing 4.3 in Line 6 an example how the *Vibrator Service* protects its method `vibrate` is shown. In this line the check is performed if a calling component is assigned with the label `android.permission.VIBRATE` defined by the constant `android.Manifest.permission.VIBRATE`. Android provides several methods to check if a sender (or service consumer) has been assigned with a permission. In our case, these facilities are represented by the method `checkCallingOrSelfPermission`. Additionally to this method, there are also a number of other methods that can be used to check the permissions of the service caller.

The implementation of the method `checkCallingOrSelfPermission` is shown in Listing 4.4. In Line 24 the method `checkPermission` is called. It takes the `uid` and the `pid` as parameters that are provided by the *Binder*

```
1  public class VibratorService extends IVibratorService.Stub
2          implements InputManager.InputDeviceListener {
3      ...
4      public void vibrate(long milliseconds, IBinder token) {
5          if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
6                  != PackageManager.PERMISSION_GRANTED) {
7              throw new SecurityException("Requires VIBRATE permission");
8          }
9          ...
10     }
11     ...
12 }
```

Listing 4.3: The check of a permission

framework.

In Line 11, the check is redirected to the `ActivityManagerService` class that in turn performs the actual check in the method `checkComponentPermission` of the `ActivityManager` component. The code of this method is presented in Listing 4.5. In Line 4 it checks if the caller UID belongs to the privileged ones. The components with the *root* and *system* UIDs are granted by the system with all permissions.

In Line 26 in Listing 4.5 the permission check is redirected to *Package Manager* that forwards it to `PackageManagerService`. As we explained before, this service knows what permissions are assigned to Android packages. The `PackageManagerService` method, which performs the permission check, is presented in Listing 4.6. In Line 7 the exact check is performed if a permission is granted to the Android app defined by its UID.

```
class ContextImpl extends Context {
    ...
    @Override
    public int checkPermission(String permission, int pid, int uid) {
        if (permission == null) {
            throw new IllegalArgumentException("permission is null");
        }

        try {
            return ActivityManagerNative.getDefault().checkPermission(
                    permission, pid, uid);
        } catch (RemoteException e) {
            return PackageManager.PERMISSION_DENIED;
        }
    }

    @Override
    public int checkCallingOrSelfPermission(String permission) {
        if (permission == null) {
            throw new IllegalArgumentException("permission is null");
        }

        return checkPermission(permission, Binder.getCallingPid(),
                Binder.getCallingUid());
    }
    ...
}
```

Listing 4.4: The excerpt of `ContextImpl` class

```
public static int checkComponentPermission(String permission, int uid,
        int owningUid, boolean exported) {
    // Root, system server get to do everything.
    if (uid == 0 || uid == Process.SYSTEM_UID) {
        return PackageManager.PERMISSION_GRANTED;
    }
    // Isolated processes don't get any permissions.
    if (UserId.isIsolated(uid)) {
        return PackageManager.PERMISSION_DENIED;
    }
    // If there is a uid that owns whatever is being accessed, it has
    // blanket access to it regardless of the permissions it requires.
    if (owningUid >= 0 && UserId.isSameApp(uid, owningUid)) {
        return PackageManager.PERMISSION_GRANTED;
    }
    // If the target is not exported, then nobody else can get to it.
    if (!exported) {
        Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid=" +
    owningUid);
        return PackageManager.PERMISSION_DENIED;
    }
    if (permission == null) {
        return PackageManager.PERMISSION_GRANTED;
    }
    try {
        return AppGlobals.getPackageManager()
                .checkUidPermission(permission, uid);
    } catch (RemoteException e) {
        // Should never happen, but if it does... deny!
        Slog.e(TAG, "PackageManager is dead?!?", e);
    }
    return PackageManager.PERMISSION_DENIED;
}
```

Listing 4.5: Method `checkComponentPermission` of the `ActivityManager`

```
public int checkUidPermission(String permName, int uid) {
    final boolean enforcedDefault = isPermissionEnforcedDefault(permName);
    synchronized (mPackages) {
        Object obj = mSettings.getUserIdLPr(UserHandle.getAppId(uid));
        if (obj != null) {
            GrantedPermissions gp = (GrantedPermissions)obj;
            if (gp.grantedPermissions.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        } else {
            HashSet<String> perms = mSystemPermissions.get(uid);
            if (perms != null && perms.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        }
        if (!isPermissionEnforcedLocked(permName, enforcedDefault)) {
            return PackageManager.PERMISSION_GRANTED;
        }
    }
    return PackageManager.PERMISSION_DENIED;
}
```

Listing 4.6: The method `checkUidPermission` of `PackageManagerService`

Chapter 5

Android Security on the Application Level

Although in this section we describe the security on the *Application* level, the actual security enforcement usually happens on lower layers described so far. However, it is easier to explain some security features of Android after introducing the *Application* level.

5.1 Application Components

Android apps are distributed in the form of Android Package (.apk) files. A package consists of Dalvik executable files, resources files, a manifest file and native libraries, and is signed by the developer of the applications using self-signed certificate.

Each Android application consists of several components of four component types: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. The separation of an application into the components supports the reuse of application parts between the apps.

Activity . An *Activity* is an element of user interface. Generally speaking, the activity often represents a screen.

Service . A *Service* is a background worker in Android. The service can run indefinite time. The most famous example of a service is media

player that plays music in the background even if the user leaves the activity that has started this service.

Broadcast receiver . A *Broadcast Receiver* is a component of an application that receives broadcast messages and starts a workflow according to the obtained message.

Content provider . A *Content Provider* is a component that provides an application with abilities to store and retrieve data. It also permits to share a set of data with another application.

So as Android applications consist of different components, there is no central entry point unlike Java programs with the `main` method. Having no central point, all components (with an exception to broadcast receivers that may also be defined dynamically) need to be declared by the developer of an application in the AndroidManifest.xml file. The separation into components makes possible to use parts in other applications. For instance, in Listing 5.1 an example of app's AndroidManifest.xml file is shown. This application consists of one *Activity* declared in Line 21. Other applications may call this activity integrating the functionality of this component into their apps.

Android provides a variety of methods to invoke the components of applications. A new *Activity* is started by using the methods `startActivity` and `startActivityForResult`. *Services* are started through the method `startService`. In this case, called service invokes its method `onStart`. When a developer is going to establish a connection between a component and a service she invokes the `bindService` method and the `onBind` method is invoked in the called service. *Broadcast receivers* are started when an app or system component send special messages using the methods `sendBroadcast`, `sendOrderedBroadcast` and `sendStickyBroadcast`.

Content providers are invoked by the requests from content resolvers. All other component types are activated through *Intents*. *Intents* is a special mean of communication in Android based on the *Binder* framework.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.testpackage.testapp"
    android:versionCode="1"
    android:versionName="1.0"
    android:sharedUserId="com.testpackage.shareduid"
    android:sharedUserLabel="@string/sharedUserId" >

    <uses-sdk android:minSdkVersion="10" />

    <permission android:name="com.testpackage.permission.mypermission"
        android:label="@string/mypermission_string"
        android:description="@string/mypermission_descr_string"
        android:protectionLevel="dangerous" />

    <uses-permission android:name="android.permission.SEND_SMS"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity android:name=".TestActivity"
            android:label="@string/app_name"
            android:permission="com.testpackage.permission.mypermission" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter >
                <action android:name="com.testpackage.testapp.MY_ACTION" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 5.1: Example of the AndroidManifest.xml file

Intents are passed into the methods that perform component invocation. The called component can be invoked by two different types of intents. To show the differences of these types, let consider an example. For instance, a user wants to choose a picture in an application. The developer of the application can use an *Explicit Intent* or an *Implicit Intent* to invoke a component that selects a picture. For the first intent type, the developer realizes picking functionality in the component of his application and calls this component using the *Component Name* data field of the explicit intent. Of course, the developer can invoke a component of other application, but, in this case, he has to be sure that this application is installed in the system. Generally, from the developer's point of view, there is no

difference between the interactions of components inside one application or among components of different applications. For the second intent type, the developer transfers the right to choose the appropriate component to the operating system. The intent object contains some information in its *Action*, *Data* and *Category* fields. According to this information, using *Intent Filters* the operating system chooses the proper component that may process the intent. An intent filter defines the "template" of intents the component can process. Of course, the same application can define an intent filter that will process intents from other component.

5.2 Permissions on the Application Level

Permissions are used not only for protecting the access to the system resources. The developers of third-party applications may also use custom permissions to guard the access to the components of their applications. An example of custom permission declaration is shown in Listing 5.1 in Line 11. The declaration of custom permissions is similar to the one of the system permissions.

To illustrate the usage of custom permissions let refer to Figure 5.1. The *Application 2* consisting of 3 components wants to protect the access to two of them: $C1$ and $C2$. To achieve this goal the developer of the *Application 2* has to declare two permission labels $p1$, $p2$ and assign them to protected components correspondingly. If a developer of the *Application 1* wants ot obtain access to component $C1$ of the *Application 2* she must define that her app requires permission $p1$. In this case, the *Application 1* receives a possibility to use the component $C1$ of the *Application 2*. If the app has not specified the required permission, the access to the component guarded with this permission is prohibited (see the case of the component $C2$ in Figure 5.1). Referring back to our example of the `AndroidManifest.xml` file in Listing 5.1, the activity `TestActivity` is protected with the permission `com.testpackage.permission.mypermission`, which is declared in the same application manifest file. If another application wants to use

the functionality provided by `TestActivity`, it must request the usage of this permission, similarly to how it is done in Line 16.

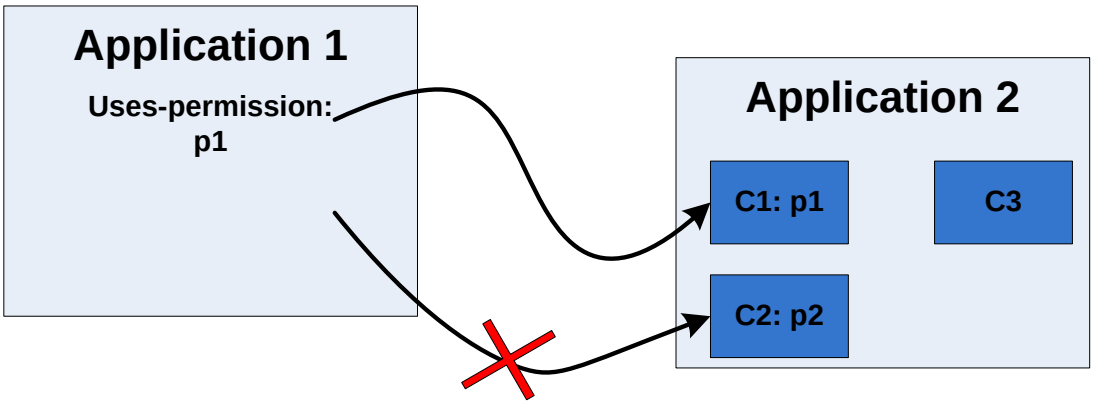

Figure 5.1: Permission enforcement to guard the components of third-party applications

`ActivityManagerService` is responsible for the invocation of the components of application. To enforce the security of app components, in the framework methods (e.g., `startActivity` described in Section 5.1), which are used to invoke the components, the special hooks are placed. These hooks check if an application has permission to call the component. These checks end in `PackageManagerServer` class with the `checkUidPermission` method (see Listing 4.6). Thus, the actual permission enforcement happens on the *Application Framework* level that is considered as a trusted part of the Android operating system. Hence, the check cannot be bypassed by applications. More information about how the components are called and permission checks can be found in [8].

Chapter 6

Other topics on Android security

In this chapter we consider other topics related to Android security that do not directly belong to any topics already considered.

6.1 Application Signing Process

Android applications are spread across the devices in the form of Android *Application Package files* (.apk files). As programs for this platform are mainly written in Java, not surprisingly this format has a lot in common with the Java packaging format – jar (Java ARchive), which is used to combine code, resource and metadata (from an optional META-INF directory) files into one file using the *zip* archiving algorithm. The META-INF directory stores package and extension configuration data, including security, versioning, extension and services [5]. Basically, in the case of Android the *apkbuilder* tool zips together built project files [1] and then this archive is signed with the standard Java utility *jarsigner* [6]. During the application signing process `jarsigner` creates the META-INF directory that usually contains the following files in case of Android: *manifest file* (MANIFEST.MF), *signature files* (with .SF extension) and *signature block files* (.RSA or .DSA).

The *manifest file* (MANIFEST.MF) consists of the main attributes section and per-entry attributes, one entry for each file contained in the unsigned apk. These per-entry attributes store information about the file

name and a digest of the file contents encoded using the `base64` format. On Android, the `SHA1` algorithm is used to compute the digest. An excerpt from a manifest file is presented in Listing 6.1.

```
Manifest-Version: 1.0
Created-By: 1.6.0_41 (Sun Microsystems Inc.)

Name: res/layout/main.xml
SHA1-Digest: NJ1YLN3mBEKTPibVXbFO8eRCAr8=

Name: AndroidManifest.xml
SHA1-Digest: wBoSXxhOQ2LR/pJY7Bczu1sWLy4=
```

Listing 6.1: An excerpt from a manifest file

The content of the *signature file* (.SF), which contains data to be singed, is similar to the one of MANIFEST.MF. An example of this file is presented in Listing 6.2. Main section contains a digest of the main attributes (`SHA1-Digest-Manifest-Main-Attributes`) and a digest of the content (`SHA1-Digest-Manifest`) of the manifest file. Per-entry section contains digests of entries in the manifest file with the corresponding file names.

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: nl/DtR972nRpjey6ocvNKvmjvw8=
Created-By: 1.6.0_41 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: Ej5guqx3DYaOLOm3Kh89ddgEJW4=

Name: res/layout/main.xml
SHA1-Digest: Z871jZHrhRKHDaGf2K4p4fKgztk=

Name: AndroidManifest.xml
SHA1-Digest: hQtlGk+tKFLSXufjNaTwd9qd4Cw=
...
```

Listing 6.2: An excerpt from a signature file

The last part in the chain is the *signature block file* (.DSA or .RSA). This binary file contains a signed version of the signature file; it has the same name as the corresponding .SF file. Depending on the used algorithm (RSA or DSA) it has different extensions.

It is possible to sign the same apk file with several different certificates. In this case in the `META-INF` directory there will be several .SF and .DSA

or .RSA files (their number will be equal to the number of times the application was signed).

6.1.1 App Signature Check in Android

Most of Android apps are sealed with a developer-signed certificate (notice that for Android "certificate" and "signature" can be used interchangeably). This certificate is used for assurance that the code of the original application and its update come from the same place, and to establish trust relationships between applications of the same developer. To perform this check Android simply compares binary representations of certificates, which were used to sign an application and its update (in the first case) and collaborating applications (in the second).

This check of certificates is implemented in `PackageManagerService` by the method `int compareSignatures(Signature[] s1, Signature[] s2)`, which code is presented in Listing 6.3. In the previous section we noted that in Android it is possible to sign the same application with several different certificates. This explains why the method takes two arrays of signatures as parameters. Despite the fact that this method takes the central place in the Android security provision, its behaviour strongly depends on the version of the platform. In the newer versions (starting from Android 2.2) this method compares two arrays of `Signature`, and if both arrays are not equal to `null` returns `SIGNATURE_MATCH` value if all `s2` signatures are contained in `s1`, and `SIGNATURE_NO_MATCH` otherwise. Before the version 2.2, this method checked if array `s1` is contained in `s2`. That behaviour allowed the system to install upgrades even if they had been signed only with a subset of certificates of the original application [2].

Trust relationships between applications of the same developer are required in several cases. The first case is connected with the permissions of the levels `signature` and `signatureOrSystem`. To use the functionality protected with the permissions of these levels, the packages declaring the permission and requesting it must be signed with the same set of cer-

```
static int compareSignatures(Signature[] s1, Signature[] s2) {
    if (s1 == null) {
        return s2 == null
                ? PackageManager.SIGNATURE_NEITHER_SIGNED
                : PackageManager.SIGNATURE_FIRST_NOT_SIGNED;
    }
    if (s2 == null) {
        return PackageManager.SIGNATURE_SECOND_NOT_SIGNED;
    }
    HashSet<Signature> set1 = new HashSet<Signature>();
    for (Signature sig : s1) {
        set1.add(sig);
    }
    HashSet<Signature> set2 = new HashSet<Signature>();
    for (Signature sig : s2) {
        set2.add(sig);
    }
    // Make sure s2 contains all signatures in s1.
    if (set1.equals(set2)) {
        return PackageManager.SIGNATURE_MATCH;
    }
    return PackageManager.SIGNATURE_NO_MATCH;
}
```

Listing 6.3: The method `compareSignatures` of `PackageManagerService`

tificates. The second case related to Android's capability to run different applications with the same UID or even in the same Linux process. In this case, applications requested such behavior must be signed with the same signature.

Bibliography

[1] Android application: Building and Running. Available Online. http://developer.android.com/tools/building/index.html.

[2] Android Security Discussions: Multiple Certificates and Upgrade process. Available Online. https://groups.google.com/forum/?fromgroups#!topic/android-security-discuss/sY7Ormv3uWk.

[3] Android Security Overview. Available Online. http://source.android.com/devices/tech/security/index.html.

[4] capabilities(7) - Linux man page. Available Online. http://linux.die.net/man/7/capabilities.

[5] Jar File Specification. Available Online. http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html.

[6] jarsigner - JAR Signing and Verification Tool. Available Online. http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html.

[7] Permissions for System Apps (not in /data/system/packages.xml?). Forum Discussion. https://groups.google.com/forum/#!topic/android-developers/ZOrtSBG5_XA.

[8] System Permissions. Available Online. http://developer.android.com/guide/topics/security/permissions.html.

[9] The Android Open Source Project. Available Online. http://source.android.com/index.html.

[10] Filesystem Hierarchy Standard, version 2.3. Available Online, January 2004. `http://www.pathname.com/fhs/pub/fhs-2.3.html`.

[11] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, 2002.

[12] Aleksandar Gargenta. Deep Dive into Android IPC/Binder Framework. Available Online. `https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm`.

[13] Marko Gargenta. Android Security Underpinnings. Available Online. `https://thenewcircle.com/s/post/1518/Android_Security_Underpinnings.htm`.

[14] Alex Lockwood. Binders & Window Tokens. Available Online, July 2013. `http://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html`.

[15] Ketan Parmar. In Depth: Android Package Manager and Package Installer. Available Online, October 2012. `http://www.kpbird.com/2012/10/in-depth-android-package-manager-and.html`.

[16] Thorsten Schreiber. Android Binder. Android Interprocess Communication. Master's thesis, Ruhr University Bochum, 2011.

[17] Enea Android team. The Android boot process from power on. `http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html`, June 2009. Available Online.

[18] Karim Yaghmour. Extending Android HAL. Available Online, September 2012. `http://www.opersys.com/blog/extending-android-hal`.

[19] Karim Yaghmour. *Embedded Android*. O'Reilly Media, Inc., 2013.

[20] Yury Zhauniarovich. *Improving the security of the Android ecosystem.* PhD thesis, University of Trento, April 2014. To appear in April 2015.

History

Version 1.00: Initial version of the book is almost completely taken from the thesis [20].

Version 1.01: Modification of the main page, removed list of figures and list of listings, typos corrections.