1

ANDROID'S SECURITY MODEL

This chapter will first briefly introduce Android's architecture, inter-process communication (IPC) mechanism, and main components. We then describe Android's security model and how it relates to the underlying Linux security infrastructure and code signing. We conclude with a brief overview of some newer additions to Android's security model, namely multi-user support, mandatory access control (MAC) based on SELinux, and verified boot. Android's architecture and security model are built on top of the traditional Unix process, user, and file paradigm, but this paradigm is not described from scratch here. We assume a basic familiarity with Unix-like systems, particularly Linux.

Android's Architecture

Let's briefly examine Android's architecture from the bottom up. Figure 1-1 shows a simplified representation of the Android stack.

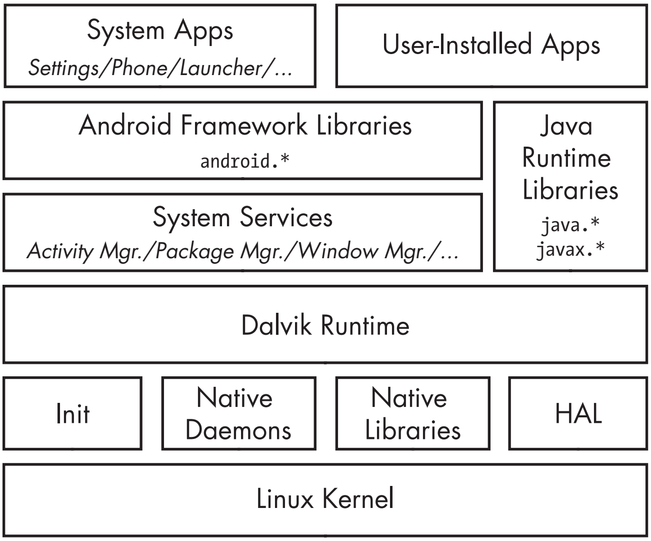

Figure 1-1: The Android architecture

Linux Kernel

As you can see in Figure 1-1, Android is built on top of the Linux kernel. As in any Unix system, the kernel provides drivers for hardware, networking, filesystem access, and process management. Thanks to the Android Mainlining Project,[1] you can now run Android with a recent vanilla kernel (with some effort), but an Android kernel is slightly different from a "regular" Linux kernel that you might find on a desktop machine or a non-Android embedded device. The differences are due to a set of new features (sometimes called *Androidisms*[2]) that were originally added to support Android. Some of the main Androidisms are the low memory killer, wakelocks (integrated as part of wakeup sources support in the mainline Linux kernel), anonymous shared memory (ashmem), alarms, paranoid networking, and Binder.

The most important Androidisms for our discussion are Binder and paranoid networking. Binder implements IPC and an associated security mechanism, which we discuss in more detail on page 5. Paranoid networking restricts access to network sockets to applications that hold specific permissions. We delve deeper into this topic in Chapter 2.

Native Userspace

On top of the kernel is the native userspace layer, consisting of the *init* binary (the first process started, which starts all other processes), several native daemons, and a few hundred native libraries that are used throughout the system. While the presence of an *init* binary and daemons is reminiscent

[1] *Android Mainlining Project*, *http://elinux.org/Android_Mainlining_Project*

[2] For a more detailed discussion of Androidisms, see Karim Yaghmour's *Embedded Android*, O'Reilly, 2013, pp. 29–38.

of a traditional Linux system, note that both *init* and the associated startup scripts have been developed from scratch and are quite different from their mainline Linux counterparts.

Dalvik VM

The bulk of Android is implemented in Java and as such is executed by a Java Virtual Machine (JVM). Android's current Java VM implementation is called *Dalvik* and it is the next layer in our stack. Dalvik was designed with mobile devices in mind and cannot run Java bytecode (*.class* files) directly: its native input format is called *Dalvik Executable (DEX)* and is packaged in *.dex* files. In turn, *.dex* files are packaged either inside system Java libraries (JAR files), or inside Android applications (APK files, discussed in Chapter 3).

Dalvik and Oracle's JVM have different architectures—register-based in Dalvik versus stack-based in the JVM—and different instruction sets. Let's look at a simple example to illustrate the differences between the two VMs (see Listing 1-1).

```
public static int add(int i, int j) {
    return i + j;
}
```

Listing 1-1: Static Java method that adds two integers

When compiled for each VM, the `add()` static method, which simply adds two integers and returns the result, would generate the bytecode shown in Figure 1-2.

JVM Bytecode

```
 public static int add(int, int);
   Code:
      0: iload_0❶
      1: iload_1❷
      2: iadd❸
      3: ireturn❹
```

Dalvik Bytecode

```
.method public static add(II)I

    add-int v0, p0, p1❺


    return v0❻
.end method
```

Figure 1-2: JVM and Dalvik bytecode

Here, the JVM uses two instructions to load the parameters onto the stack (❶ and ❷), then executes the addition ❸, and finally returns the result ❹. In contrast, Dalvik uses a single instruction to add parameters (in registers *p0* and *p1*) and puts the result in the *v0* register ❺. Finally, it returns the contents of the *v0* register ❻. As you can see, Dalvik uses fewer instructions to achieve the same result. Generally speaking, register-based VMs use fewer instructions, but the resulting code is larger than the corresponding code in a stack-based VM. However, on most architectures, loading code is less expensive than instruction dispatch, so register-based VMs can be interpreted more efficiently.[3]

In most production devices, system libraries and preinstalled applications do not contain device-independent DEX code directly. As a performance optimization, DEX code is converted to a device-dependent format and stored in an Optimized DEX (*.odex*) file, which typically resides in the same directory as its parent JAR or APK file. A similar optimization process is performed for user-installed applications at install time.

Java Runtime Libraries

A Java language implementation requires a set of runtime libraries, defined mostly in the `java.*` and `javax.*` packages. Android's core Java libraries are originally derived from the Apache Harmony project[4] and are the next layer on our stack. As Android has evolved, the original Harmony code has changed significantly. In the process, some features have been replaced entirely (such as internationalization support, the cryptographic provider, and some related classes), while others have been extended and improved. The core libraries are developed mostly in Java, but they have some native code dependencies as well. Native code is linked into Android's Java libraries using the standard *Java Native Interface (JNI)*,[5] which allows Java code to call native code and vice versa. The Java runtime libraries layer is directly accessed both from system services and applications.

System Services

The layers introduced up until now make up the plumbing necessary to implement the core of Android—system services. *System services* (79 as of version 4.4) implement most of the fundamental Android features, including display and touch screen support, telephony, and network connectivity. Most system services are implemented in Java; some fundamental ones are written in native code.

With a few exceptions, each system service defines a remote interface that can be called from other services and applications. Coupled with the service discovery, mediation, and IPC provided by Binder, system services effectively implement an object-oriented OS on top of Linux.

Let's look at how Binder enables IPC on Android in detail, as this is one of the cornerstones of Android's security model.

Inter-Process Communication

As mentioned previously, Binder is an inter-process communication (IPC) mechanism. Before getting into detail about how Binder works, let's briefly review IPC.

[3] Yunhe Shi et al., *Virtual Machine Showdown: Stack Versus Registers*, *https://www.usenix.org/legacy/events/vee05/full_papers/p153-yunhe.pdf*

[4] The Apache Software Foundation, *Apache Harmony*, *http://harmony.apache.org/*

[5] Oracle, *Java™ Native Interface*, *http://docs.oracle.com/javase/7/docs/technotes/guides/jni/*

As in any Unix-like system, processes in Android have separate address spaces and a process cannot directly access another process's memory (this is called *process isolation*). This is usually a good thing, both for stability and security reasons: multiple processes modifying the same memory can be catastrophic, and you don't want a potentially rogue process that was started by another user to dump your email by accessing your mail client's memory. However, if a process wants to offer some useful service(s) to other processes, it needs to provide some mechanism that allows other processes to discover and interact with those services. That mechanism is referred to as *IPC*.

The need for a standard IPC mechanism is not new, so several options predate Android. These include files, signals, sockets, pipes, semaphores, shared memory, message queues, and so on. While Android uses some of these (such as local sockets), it does not support others (namely System V IPCs like semaphores, shared memory segments, and message queues).

Binder

Because the standard IPC mechanisms weren't flexible or reliable enough, a new IPC mechanism called *Binder* was developed for Android. While Android's Binder is a new implementation, it's based on the architecture and ideas of OpenBinder.[6]

Binder implements a distributed component architecture based on abstract interfaces. It is similar to Windows Common Object Model (COM) and Common Object Broker Request Architectures (CORBA) on Unix, but unlike those frameworks, it runs on a single device and does not support remote procedure calls (RPC) across the network (although RPC support could be implemented on top of Binder). A full description of the Binder framework is outside the scope of this book, but we introduce its main components briefly in the following sections.

Binder Implementation

As mentioned earlier, on a Unix-like system, a process cannot access another process's memory. However, the kernel has control over all processes and therefore can expose an interface that enables IPC. In Binder, this interface is the */dev/binder* device, which is implemented by the Binder kernel driver. The *Binder driver* is the central object of the framework, and all IPC calls go through it. Inter-process communication is implemented with a single `ioctl()` call that both sends and receives data through the `binder_write_read` structure, which consists of a `write_buffer` containing commands for the driver, and a `read_buffer` containing commands that the userspace needs to perform.

But how is data actually passed between processes? The Binder driver manages part of the address space of each process. The Binder driver-managed chunk of memory is read-only to the process, and all writing

[6] PalmSource, Inc., *OpenBinder, http://www.angryredplanet.com/~hackbod/openbinder/docs/html/*

is performed by the kernel module. When a process sends a message to another process, the kernel allocates some space in the destination process's memory, and copies the message data directly from the sending process. It then queues a short message to the receiving process telling it where the received message is. The recipient can then access that message directly (because it is in its own memory space). When a process is finished with the message, it notifies the Binder driver to mark the memory as free. Figure 1-3 shows a simplified illustration of the Binder IPC architecture.

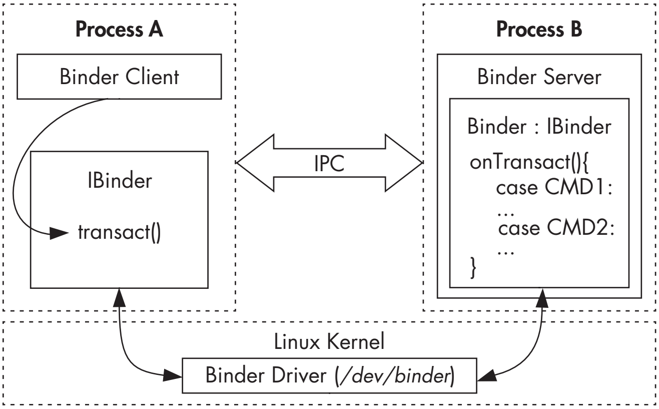

Figure 1-3: Binder IPC

Higher-level IPC abstractions in Android such as *Intents* (commands with associated data that are delivered to components across processes), *Messengers* (objects that enable message-based communication across processes), and *ContentProviders* (components that expose a cross-process data management interface) are built on top of Binder. Additionally, service interfaces that need to be exposed to other processes can be defined using the *Android Interface Definition Language (AIDL)*, which enables clients to call remote services as if they were local Java objects. The associated aidl tool automatically generates *stubs* (client-side representations of the remote object) and *proxies* that map interface methods to the lower-level transact() Binder method and take care of converting parameters to a format that Binder can transmit (this is called *parameter marshalling/unmarshalling*). Because Binder is inherently typeless, AIDL-generated stubs and proxies also provide type safety by including the target interface name in each Binder transaction (in the proxy) and validating it in the stub.

Binder Security

On a higher level, each object that can be accessed through the Binder framework implements the IBinder interface and is called a *Binder object.* Calls to a Binder object are performed inside a *Binder transaction*, which contains a reference to the target object, the ID of the method to execute, and a data buffer. The Binder driver automatically adds the process ID (PID) and effective user ID (EUID) of the calling process to the transaction data. The called process (*callee*) can inspect the PID and EUID and decide whether it should execute the requested method based on its internal logic or system-wide metadata about the calling application.

Since the PID and EUID are filled in by the kernel, caller processes cannot fake their identity to get more privileges than allowed by the system (that is, Binder prevents *privilege escalation*). This is one of the central pieces of Android's security model, and all higher-level abstractions, such as permissions, build upon it. The EUID and PID of the caller are accessible via the `getCallingPid()` and `getCallingUid()` methods of the `android.os.Binder` class, which is part of Android's public API.

NOTE *The calling process's EUID may not map to a single application if more than one application is executing under the same UID (see Chapter 2 for details). However, this does not affect security decisions, as processes running under the same UID are typically granted the same set of permissions and privileges (unless process-specific SELinux rules have been defined).*

Binder Identity

One of the most important properties of Binder objects is that they maintain a unique identity across processes. Thus if process A creates a Binder object and passes it to process B, which in turn passes it to process C, calls from all three processes will be processed by the same Binder object. In practice, process A will reference the Binder object directly by its memory address (because it is in process A's memory space), while process B and C will receive only a handle to the Binder object.

The kernel maintains the mapping between "live" Binder objects and their handles in other processes. Because a Binder object's identity is unique and maintained by the kernel, it is impossible for userspace processes to create a copy of a Binder object or obtain a reference to one unless they have been handed one through IPC. Thus a Binder object is a unique, unforgeable, and communicable object that can act as a security *token*. This enables the use of capability-based security in Android.

Capability-Based Security

In a *capability-based security model*, programs are granted access to a particular resource by giving them an unforgeable *capability* that both references the target object and encapsulates a set of access rights to it. Because capabilities are unforgeable, the mere fact that a program possesses a capability is sufficient to give it access to the target resource; there is no need to maintain access control lists (ACLs) or similar structures associated with actual resources.

Binder Tokens

In Android, Binder objects can act as capabilities and are called *Binder tokens* when used in this fashion. A Binder token can be both a capability and a target resource. The possession of a Binder token grants the owning process full access to a Binder object, enabling it to perform Binder transactions on the target object. If the Binder object implements multiple actions (by selecting the action to perform based on the `code` parameter of the Binder transaction), the caller can perform any action when it has a reference to that Binder object. If more granular access control is required, the implementation of each action needs to implement the necessary permission checks, typically by utilizing the PID and EUID of the caller process.

A common pattern in Android is to allow all actions to callers running as *system* (UID 1000) or *root* (UID 0), but perform additional permission checks for all other processes. Thus access to important Binder objects such as system services is controlled in two ways: by limiting who can get a reference to that Binder object and by checking the caller identity before performing an action on the Binder object. (This check is optional and implemented by the Binder object itself, if required.)

Alternatively, a Binder object can be used only as a capability without implementing any other functionality. In this usage pattern, the same Binder object is held by two (or more) cooperating processes, and the one acting as a server (processing some kind of client requests) uses the Binder token to authenticate its clients, much like web servers use session cookies.

This usage pattern is used internally by the Android framework and is mostly invisible to applications. One notable use case of Binder tokens that is visible in the public API is *window tokens*. The top-level window of each activity is associated with a Binder token (called a window token), which Android's window manager (the system service responsible for managing application windows) keeps track of. Applications can obtain their own window token but cannot get access to the window tokens of other applications. Typically you don't want other applications adding or removing windows on top of your own; each request to do so must provide the window token associated with the application, thus guaranteeing that window requests are coming from your own application or from the system.

Accessing Binder Objects

Although Android controls access to Binder objects for security purposes, and the only way to communicate with a Binder object is to be given a reference to it, some Binder objects (most notably system services) need to be universally accessible. It is, however, impractical to hand out references to all system services to each and every process, so we need some mechanism that allows processes to discover and obtain references to system services as needed.

In order to enable service discovery, the Binder framework has a single *context manager*, which maintains references to Binder objects. Android's context manager implementation is the *servicemanager* native daemon. It is started very early in the boot process so that system services can register with it as they start up. Services are registered by passing a service name and a Binder reference to the service manager. Once a service is registered, any client can obtain its Binder reference by using its name. However, most system services implement additional permission checks, so obtaining a reference does not automatically guarantee access to all of its functionality. Because anyone can access a Binder reference when it is registered with the service manager, only a small set of whitelisted system processes can register system services. For example, only a process executing as UID 1002 (`AID_BLUETOOTH`) can register the *bluetooth* system service.

You can view a list of registered services by using the `service list` command, which returns the name of each registered service and the implemented `IBinder` interface. Sample output from running the command on an Android 4.4 device is shown in Listing 1-2.

```
$ service list
service list
Found 79 services:
0       sip: [android.net.sip.ISipService]
1       phone: [com.android.internal.telephony.ITelephony]
2       iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
3       simphonebook: [com.android.internal.telephony.IIccPhoneBook]
4       isms: [com.android.internal.telephony.ISms]
5       nfc: [android.nfc.INfcAdapter]
6       media_router: [android.media.IMediaRouterService]
7       print: [android.print.IPrintManager]
8       assetatlas: [android.view.IAssetAtlas]
9       dreams: [android.service.dreams.IdreamManager]
--snip--
```

Listing 1-2: Obtaining a list of registered system services with the `service list` command

Other Binder Features

While not directly related to Android's security model, two other notable Binder features are reference counting and death notification (also known as link to death). *Reference counting* guarantees that Binder objects are automatically freed when no one references them and is implemented in the kernel driver with the `BC_INCREFS`, `BC_ACQUIRE`, `BC_RELEASE`, and `BC_DECREFS` commands. Reference counting is integrated at various levels of the Android framework but is not directly visible to applications.

Death notification allows applications that use Binder objects that are hosted by other processes to be notified when those processes are killed by the kernel and to perform any necessary cleanup. Death notification is implemented with the `BC_REQUEST_DEATH_NOTIFICATION` and `BC_CLEAR_DEATH_NOTIFICATION` commands in the kernel driver and the `linkToDeath()` and `unlinkToDeath()` methods of the `IBinder` interface[7] in the framework. (Death notifications for local binders are not sent, because local binders cannot die without the hosting process dying as well.)

[7] Google, *Android APIs Reference*, "IBinder," *http://developer.android.com/reference/android/os/IBinder.html*

Android Framework Libraries

Next on the stack are the Android framework libraries, sometimes called just "the framework." The framework includes all Java libraries that are not part of the standard Java runtime (`java.*`, `javax.*`, and so on) and is for the most part hosted under the `android` top-level package. The framework includes the basic blocks for building Android applications, such as the base classes for activities, services, and content providers (in the `android.app.*` packages); GUI widgets (in the `android.view.*` and `android.widget` packages); and classes for file and database access (mostly in the `android.database.*` and `android.content.*` packages). It also includes classes that let you interact with device hardware, as well as classes that take advantage of higher-level services offered by the system.

Even though almost all Android OS functionality above the kernel level is implemented as system services, it is not exposed directly in the framework but is accessed via facade classes called *managers*. Typically, each manager is backed by a corresponding system service; for example, the `BluetoothManager` is a facade for the `BluetoothManagerService`.

Applications

On the highest level of the stack are *applications* (or *apps*), which are the programs that users directly interact with. While all apps have the same structure and are built on top of the Android framework, we distinguish between system apps and user-installed apps.

System Apps

System apps are included in the OS image, which is read-only on production devices (typically mounted as */system*), and cannot be uninstalled or changed by users. Therefore, these apps are considered secure and are given many more privileges than user-installed apps. System apps can be part of the core Android OS or can simply be preinstalled user applications, such as email clients or browsers. While all apps installed under */system* were treated equally in earlier versions of Android (except by OS features that check the app signing certificate), Android 4.4 and higher treat apps installed in */system/priv-app/* as privileged applications and will only grant permissions with protection level *signatureOrSystem* to privileged apps, not to all apps installed under */system*. Apps that are signed with the platform signing key can be granted system permissions with the *signature* protection level, and thus can get OS-level privileges even if they are not preinstalled under */system*. (See Chapter 2 for details on permissions and code signing.)

While system apps cannot be uninstalled or changed, they can be updated by users as long as the updates are signed with the same private key, and some can be overridden by user-installed apps. For example, a user can choose to replace the preinstalled application launcher or input method with a third-party application.

User-Installed Apps

User-installed apps are installed on a dedicated read-write partition (typically mounted as */data*) that hosts user data and can be uninstalled at will. Each application lives in a dedicated security sandbox and typically cannot affect other applications or access their data. Additionally, apps can only access resources that they have explicitly been granted a permission to use. Privilege separation and the principle of least privilege are central to Android's security model, and we will explore how they are implemented in the next section.

Android App Components

Android applications are a combination of loosely coupled *components* and, unlike traditional applications, can have more than one entry point. Each component can offer multiple entry points that can be reached based on user actions in the same or another application, or triggered by a system event that the application has registered to be notified about.

Components and their entry points, as well as additional metadata, are defined in the application's manifest file, called *AndroidManifest.xml*. Like most Android resource files, this file is compiled into a binary XML format (similar to ASN.1) before bundling it in the application package (APK) file in order to decrease size and speed up parsing. The most important application property defined in the manifest file is the application package name, which uniquely identifies each application in the system. The package name is in the same format as Java package names (reverse domain name notation; for example, *com.google.email*).

The *AndroidManifest.xml* file is parsed at application install time, and the package and components it defines are registered with the system. Android requires each application to be signed using a key controlled by its developer. This guarantees that an installed application cannot be replaced by another application that claims to have the same package name (unless it is signed with the same key, in which case the existing application is updated). We'll discuss code signing and application packages in Chapter 3.

The main components of Android apps are listed below.

Activities

An *activity* is a single screen with a user interface. Activities are the main building blocks of Android GUI applications. An application can have multiple activities and while they are usually designed to be displayed in a particular order, each activity can be started independently, potentially by a different app (if allowed).

Services

A *service* is a component that runs in the background and has no user interface. Services are typically used to perform some long-running operation, such as downloading a file or playing music, without blocking the user interface. Services can also define a remote interface using

AIDL and provide some functionality to other apps. However, unlike system services, which are part of the OS and are always running, application services are started and stopped on demand.

Content providers

Content providers provide an interface to app data, which is typically stored in a database or files. Content providers can be accessed via IPC and are mainly used to share an app's data with other apps. Content providers offer fine-grained control over what parts of data are accessible, allowing an application to share only a subset of its data.

Broadcast receivers

A *broadcast receiver* is a component that responds to systemwide events, called *broadcasts*. Broadcasts can originate from the system (for example, announcing changes in network connectivity), or from a user application (for example, announcing that background data update has completed).

Android's Security Model

Like the rest of the system, Android's security model also takes advantage of the security features offered by the Linux kernel. Linux is a multiuser operating system and the kernel can isolate user resources from one another, just as it isolates processes. In a Linux system, one user cannot access another user's files (unless explicitly granted permission) and each process runs with the identity (*user* and *group ID*, usually referred to as *UID* and *GID*) of the user that started it, unless the set-user-ID or set-group-ID (SUID and SGID) bits are set on the corresponding executable file.

Android takes advantage of this user isolation, but treats users differently than a traditional Linux system (desktop or server) does. In a traditional system, a UID is given either to a physical user that can log into the system and execute commands via the shell, or to a system service (daemon) that executes in the background (because system daemons are often accessible over the network, running each daemon with a dedicated UID can limit the damage if one is compromised). Android was originally designed for smartphones, and because mobile phones are personal devices, there was no need to register different physical users with the system. The physical user is implicit, and UIDs are used to distinguish applications instead. This forms the basis of Android's application sandboxing.

Application Sandboxing

Android automatically assigns a unique UID, often called an *app ID*, to each application at installation and executes that application in a dedicated process running as that UID. Additionally, each application is given a dedicated data directory which only it has permission to read and write to. Thus, applications are isolated, or *sandboxed*, both at the process level (by having each run in a dedicated process) and at the file level (by having a private data directory). This creates a kernel-level application sandbox, which applies to all applications, regardless of whether they are executed in a native or virtual machine process.

System daemons and applications run under well-defined and constant UIDs, and very few daemons run as the root user (UID 0). Android does not have the traditional */etc/password* file and its system UIDs are statically defined in the *android_filesystem_config.h* header file. UIDs for system services start from 1000, with 1000 being the *system* (`AID_SYSTEM`) user, which has special (but still limited) privileges. Automatically generated UIDs for applications start at 10000 (`AID_APP`), and the corresponding usernames are in the form *app_XXX* or *uY_aXXX* (on Android versions that support multiple physical users), where *XXX* is the offset from *AID_APP* and *Y* is the Android user ID (not the same as UID). For example, the 10037 UID corresponds to the *u0_a37* username and may be assigned to the Google email client application (*com.google.android.email* package). Listing 1-3 shows that the email application process executes as the *u0_a37* user ❶, while other application processes execute as different users.

```
$ ps
--snip--
u0_a37    16973 182   941052  60800 ffffffff 400d073c S com.google.android.email❶
u0_a8     18788 182   925864  50236 ffffffff 400d073c S com.google.android.dialer
u0_a29    23128 182   875972  35120 ffffffff 400d073c S com.google.android.calendar
u0_a34    23264 182   868424  31980 ffffffff 400d073c S com.google.android.deskclock
--snip--
```

Listing 1-3: Each application process executes as a dedicated user on Android

The data directory of the email application is named after its package name and is created under */data/data/* on single-user devices. (Multi-user devices use a different naming scheme as discussed in Chapter 4.) All files inside the data directory are owned by the dedicated Linux user, *u0_a37*, as shown in Listing 1-4 (with timestamps omitted). Applications can optionally create files using the `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` flags to allow direct access to files by other applications, which effectively sets the `S_IROTH` and `S_IWOTH` access bits on the file, respectively. However, the direct sharing of files is discouraged, and those flags are deprecated in Android versions 4.2 and higher.

```
# ls -l /data/data/com.google.android.email
drwxrwx--x u0_a37   u0_a37           app_webview
drwxrwx--x u0_a37   u0_a37           cache
drwxrwx--x u0_a37   u0_a37           databases
drwxrwx--x u0_a37   u0_a37           files
--snip--
```

Listing 1-4: Application directories are owned by the dedicated Linux user

Application UIDs are managed alongside other package metadata in the */data/system/packages.xml* file (the canonical source) and also written to the */data/system/packages.list* file. (We discuss package management and the *packages.xml* file in Chapter 3.) Listing 1-5 shows the UID assigned to the *com.google.android.email* package as it appears in *packages.list*.

```
# grep 'com.google.android.email' /data/system/packages.list
com.google.android.email 10037 0 /data/data/com.google.android.email default 3003,1028,1015
```

Listing 1-5: The UID corresponding to each application is stored in /data/system/packages.list

Here, the first field is the package name, the second is the UID assigned to the application, the third is the debuggable flag (`1` if debuggable), the fourth is the application's data directory path, and the fifth is the *seinfo* label (used by SELinux). The last field is a list of the supplementary GIDs that the app launches with. Each GID is typically associated with an Android permission (discussed next) and the GID list is generated based on the permissions granted to the application.

Applications can be installed using the same UID, called a *shared user ID*, in which case they can share files and even run in the same process. Shared user IDs are used extensively by system applications, which often need to use the same resources across different packages for modularity. For example, in Android 4.4 the system UI and keyguard (lockscreen implementation) share UID 10012 (see Listing 1-6).

```
# grep ' 10012 ' /data/system/packages.list
com.android.keyguard 10012 0 /data/data/com.android.keyguard platform 1028,1015,1035,3002,3001
com.android.systemui 10012 0 /data/data/com.android.systemui platform 1028,1015,1035,3002,3001
```

Listing 1-6: System packages sharing the same UID

While the shared user ID facility is not recommended for non-system apps, it's available to third-party applications as well. In order to share the same UID, applications need to be signed by the same code signing key. Additionally, because adding a shared user ID to a new version of an installed app causes it to change its UID, the system disallows this (see Chapter 2). Therefore, a shared user ID cannot be added retroactively, and apps need to be designed to work with a shared ID from the start.

Permissions

Because Android applications are sandboxed, they can access only their own files and any world-accessible resources on the device. Such a limited application wouldn't be very interesting though, and Android can grant additional, fine-grained access rights to applications in order to allow for richer functionality. Those access rights are called *permissions*, and they can control access to hardware devices, Internet connectivity, data, or OS services.

Applications can request permissions by defining them in the *AndroidManifest.xml* file. At application install time, Android inspects the list of requested permissions and decides whether to grant them or not. Once granted, permissions cannot be revoked and they are available to the application without any additional confirmation. Additionally, for features such as private key or user account access, explicit user confirmation is required for each accessed object, even if the requesting application has been granted the corresponding permission (see Chapters 7 and 8). Some permission can only be granted to applications that are part of the Android OS, either because they're preinstalled or signed with the same key as the OS. Third-party applications can define custom permissions and define similar restrictions known as permission *protection levels*, thus restricting access to an app's services and resources to apps created by the same author.

Permission can be enforced at different levels. Requests to lower-level system resources, such as device files, are enforced by the Linux kernel by checking the UID or GID of the calling process against the resource's owner and access bits. When accessing higher-level Android components, enforcement is performed either by the Android OS or by each component (or both). We discuss permissions in Chapter 2.

IPC

Android uses a combination of a kernel driver and userspace libraries to implement IPC. As discussed in "Binder" on page 5, the Binder kernel driver guarantees that the UID and PID of callers cannot be forged, and many system services rely on the UID and PID provided by Binder to dynamically control access to sensitive APIs exposed via IPC. For example, the system Bluetooth manager service only allows system applications to enable Bluetooth silently if the caller is running with the *system* UID (1000) by using the code shown in Listing 1-7. Similar code is found in other system services.

```
public boolean enable() {
    if ((Binder.getCallingUid() != Process.SYSTEM_UID) &&
        (!checkIfCallerIsForegroundUser())) {
        Log.w(TAG,"enable(): not allowed for non-active and non-system user");
        return false;
     }
--snip--
}
```

Listing 1-7: Checking that the caller is running with the system *UID*

More coarse-grained permissions that affect all methods of a service exposed via IPC can be automatically enforced by the system by specifying a permission in the service declaration. As with requested permissions, required permissions are declared in the *AndroidManifest.xml* file. Like the dynamic permission check in the example above, per-component permissions are also implemented by consulting the caller UID obtained from Binder under the hood. The system uses the package database to determine the permission required by the callee component, and then maps the caller UID to a package name and retrieves the set of permissions granted to the caller. If the required permission is in that set, the call succeeds. If not, it fails and the system throws a `SecurityException`.

Code Signing and Platform Keys

All Android applications must be signed by their developer, including system applications. Because Android APK files are an extension of the Java JAR package format,[8] the code signing method used is also based on JAR signing. Android uses the APK signature to make sure updates for an app are coming from the same author (this is called the *same origin policy*) and to establish trust relationships between applications. Both of these security features are implemented by comparing the signing certificate of the currently installed target app with the certificate of the update or related application.

System applications are signed by a number of *platform keys*. Different system components can share resources and run inside the same process when they are signed with the same platform key. Platform keys are generated and controlled by whoever maintains the Android version installed on a particular device: device manufacturers, carriers, Google for Nexus devices, or users for self-built open source Android versions. (We'll discuss code signing and the APK format in Chapter 3.)

Multi-User Support

Because Android was originally designed for handset (smartphone) devices that have a single physical user, it assigns a distinct Linux UID to each installed application and traditionally does not have a notion of a physical user. Android gained support for multiple physical users in version 4.2, but multi-user support is only enabled on tablets, which are more likely to be shared. Multi-user support on handset devices is disabled by setting the maximum number of users to 1.

Each user is assigned a unique user ID, starting with 0, and users are given their own dedicated data directory under */data/system/users/<user ID>/*, which is called the user's *system directory*. This directory hosts user-specific settings such as homescreen parameters, account data, and a list of currently installed applications. While application binaries are shared between users, each user gets a copy of an application's data directory.

To distinguish applications installed for each user, Android assigns a new effective UID to each application when it is installed for a particular user. This effective UID is based on the target physical user's user ID and the app's UID in a single-user system (the *app ID*). This composite structure of the granted UID guarantees that even if the same application is installed by two different users, both application instances get their own sandbox. Additionally, Android guarantees dedicated shared storage (hosted on an SD card for older devices), which is world-readable, to each physical user.

[8] Oracle, *JAR File Specification*, *http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html*

The user to first initialize the device is called the *device owner*, and only they can manage other users or perform administrative tasks that influence the whole device (such as factory reset). (We discuss multi-user support in greater detail in Chapter 4.)

SELinux

The traditional Android security model relies heavily on the UIDs and GIDs granted to applications. While those are guaranteed by the kernel, and by default each application's files are private, nothing prevents an application from granting world access to its files (whether intentionally or due to a programming error).

Similarly, nothing prevents malicious applications from taking advantage of the overly permissive access bits of system files or local sockets. In fact, inappropriate permissions assigned to application or system files have been the source of a number of Android vulnerabilities. Those vulnerabilities are unavoidable in the default access control model employed by Linux, known as *discretionary access control (DAC)*. *Discretionary* here means that once a user gets access to a particular resource, they can pass it on to another user at their discretion, such as by setting the access mode of one of their files to world-readable. In contrast, *mandatory access control (MAC)* ensures that access to resources conforms to a system-wide set of *authorization rules* called a *policy*. The policy can only be changed by an administrator, and users cannot override or bypass it in order to, for example, grant everyone access to their own files.

Security Enhanced Linux (SELinux) is a MAC implementation for the Linux kernel and has been integrated in the mainline kernel for more than 10 years. As of version 4.3, Android integrates a modified SELinux version from the Security Enhancements for Android (SEAndroid) project[9] that has been augmented to support Android-specific features such as Binder. In Android, SELinux is used to isolate core system daemons and user applications in different security *domains* and to define different access policies for each domain. As of version 4.4, SELinux is deployed in *enforcing mode* (violations to the system policy generate runtime errors), but policy enforcement is only applied to core system daemons. Applications still run in *permissive mode* and violations are logged but do not cause runtime errors. (We give more details about Android's SELinux implementation in Chapter 12.)

System Updates

Android devices can be updated over-the-air (OTA) or by connecting the device to a PC and pushing the update image using the standard Android debug bridge (ADB) client or some vendor-provided application with similar functionality. Because in addition to system files, an Android update might need to modify the baseband (modem) firmware, bootloader, and

[9] SELinux Project, *SE for Android*, *http://selinuxproject.org/page/SEAndroid*

other parts of the device that are not directly accessible from Android, the update process typically uses a special-purpose, minimal OS with exclusive access to all device hardware. This is called a *recovery OS* or simply *recovery*.

OTA updates are performed by downloading an OTA package file (typically a ZIP file with an added code signature), which contains a small script file to be interpreted by the recovery, and rebooting the device in *recovery mode*. Alternatively, the user can enter recovery mode by using a device-specific key combination when booting the device, and apply the update manually by using the menu interface of the recovery, which is usually navigated using the hardware buttons (Volume up/down, Power, and so on) of the device.

On production devices, the recovery accepts only updates signed by the device manufacturer. Update files are signed by extending the ZIP file format to include a signature over the whole file in the comment section (see Chapter 3), which the recovery extracts and verifies before installing the update. On some devices (including all Nexus devices, dedicated developer devices, and some vendor devices), device owners can replace the recovery OS and disable system update signature verification, allowing them to install updates by third parties. Switching the device bootloader to a mode that allows replacing the recovery and system images is called *bootloader unlocking* (not to be confused with SIM-unlocking, which allows a device to be used on any mobile network) and typically requires wiping all user data (factory reset) in order to make sure that a potentially malicious third-party system image does not get access to existing user data. On most consumer devices, unlocking the bootloader has the side effect of voiding the device's warranty. (We discuss system updates and recovery images in Chapter 13.)

Verified Boot

As of version 4.4, Android supports verified boot using the *verity* target[10] of Linux's Device-Mapper. Verity provides transparent integrity checking of block devices using a cryptographic hash tree. Each node in the tree is a cryptographic hash, with leaf nodes containing the hash value of a physical data block and intermediary nodes containing hash values of their child nodes. Because the hash in the root node is based on the values of all other nodes, only the root hash needs to be trusted in order to verify the rest of the tree.

Verification is performed with an RSA public key included in the boot partition. Device blocks are checked at runtime by calculating the hash value of the block as it is read and comparing it to the recorded value in the hash tree. If the values do not match, the read operation results in an I/O error indicating that the filesystem is corrupted. Because all checks are performed by the kernel, the boot process needs to verify the integrity of the kernel in order for verified boot to work. This process is device-specific and is typically implemented by using an unchangeable, hardware-specific key that

[10] Linux kernel source tree, *dm-verity*, *http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/verity.txt*

is "burned" (written to write-only memory) into the device. That key is used to verify the integrity of each bootloader level and eventually the kernel. (We discuss verified boot in Chapter 10.)

Summary

Android is a privilege-separated operating system based on the Linux kernel. Higher-level system functions are implemented as a set of cooperating system services that communicate using an IPC mechanism called Binder. Android isolates applications from each other by running each with a distinct system identity (Linux UID). By default, applications are given very few privileges and have to request fine-grained permission in order to interact with system services, hardware devices, or other applications. Permissions are defined in each application's manifest file and are granted at install time. The system uses the UID of each application to find out what permissions it has been granted and to enforce them at runtime. In recent versions, system processes isolation takes advantage of SELinux to further constrain the privileges given to each process.