Security

Android Security

Elenkov

Nikolay Elenkov

Inter nals

Foreword by Jon Sawyer

An In-Depth Guide to Android's Security

Architecture

aNDROID sECURITY INTERNaLs

an In-Depth Guide to android's security architecture

by Nikolay Elenkov

San Francisco

Printed in USA

First printing

18 17 16 15 14 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-581-1 ISBN-13: 978-1-59327-581-5

Publisher: William Pollock Production Editor: Alison Law Cover Illustration: Garry Booth Interior Design: Octopod Studios Developmental Editor: William Pollock Technical Reviewer: Kenny Root Copyeditor: Gillian McGarvey Compositor: Susan Glinert Stevens Proofreader: James Fraleigh Indexer: BIM Proofreading & Indexing Services

SFI-00000

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc. 245 8th Street, San Francisco, CA 94103 phone:
415.863.9900; info@nostarch.com www.nostarch.com

Library of Congress Control Number: 2014952666

about the author

Nikolay Elenkov has been working on enterprise security projects for the past 10 years. He has developed security software on various plat- forms, ranging from smart cards and HSMs to Windows and Linux servers. He became interested in Android shortly after the initial public release and has been developing applications for it since version 1.5. Nikolay's interest in Android internals intensified after the release of Android 4.0 (Ice Cream Sandwich), and for the past three years he's been documenting his findings and writing about Android security on his blog, *http://nelenkov.blogspot.com/*.

about the Technical Reviewer

Kenny Root has been a core contributor to the Android platform at Google since 2009, where his focus has been primarily on security and cryptography. He is the author of ConnectBot, the first SSH app for Android, and is an avid open source contributor. When he's not hack- ing on software, he's spending time with his wife and two boys. He is an alumnus of Stanford University, Columbia University, Chinese University of Hong Kong, and Baker College, but he's originally from Kansas City, which has the best barbecue.

BRIEf CONTENTs

CONTENTs IN DETaIL

5 CRYPTOgRaPhIC PROvIDERs 115

6 NETwORk sECURITY aND PkI 145

fOREwORD

I first became aware of the quality of Nikolay's work in Android security with the release of Android 4.0, Ice Cream Sandwich. I needed a better explanation of the new Android backup format; I was struggling to exploit a vulnerability I had found, because I didn't have a full grasp of the new feature and format. His clear, in-depth expla- nation helped me understand the issue, exploit the vulnerability, and get a patch into production devices quickly. I have since been a frequent visitor to his blog, often referring to it when I need a refresher.

While I was honored to be asked to write this foreword, I honestly didn't believe I'd learn much from the book because I've been working on Android security for many years. This belief could not have been more wrong. As I read and digested new information regarding subjects I thought I knew thoroughly, my mind whirled with thoughts of what I had missed and what I could have done better. Why wasn't a reference like this available when I first engrossed myself in Android?

This book exposes the reader to a wide range of security topics, from Android permissions and sandboxing to the Android SELinux implementa- tion, SEAndroid. It provides excellent explanations of minute details and rarely seen features such as dm-verify. Like me, you'll walk away from this book with a better understanding of Android security features.

Android Security Internals has earned a permanent spot on my office bookshelf.

Jon “jcase” Sawyer CTO, Applied Cybersecurity LLC Port Angeles, WA

aCkNOwLEDGmENTs

I would like to thank everyone at No Starch Press who worked on this book. Special thanks to Bill Pollock for making my ramblings readable and to Alison Law for her patience in turning them into an actual book.

A big thanks to Kenny Root for reviewing all chapters and sharing the backstories behind some of Android’s security features.

Thanks to Jorrit “Chainfire” Jongma for maintaining SuperSU, which has been an invaluable tool for poking at Android’s internals, and for reviewing my coverage of it in Chapter 13.

Thanks to Jon “jcase” Sawyer for continuing to challenge our assump- tions about Android security and for contributing a foreword to my book.

INTRODUCTION

In a relatively short period of time, Android has become the world's most popular mobile platform. Although originally designed for smartphones, it now powers tablets, TVs, and wearable devices, and will soon even be found in cars. Android is being developed at a breathtaking pace, with an average of two major releases per year. Each new release brings a better UI, performance improvements, and a host of new user-facing features which are typically blogged about and dissected in excruciating detail by Android enthusiasts.

One aspect of the Android platform that has seen major improvements over the last few years, but which has received little public attention, is secu- rity. Over the years, Android has become more resistant to common exploit techniques (such as buffer overflows), its application isolation (sandboxing) has been reinforced, and its attack surface has been considerably reduced by aggressively decreasing the number of system processes that run as root. In addition to these exploit mitigations, recent versions of Android have introduced major new security features such as restricted user support,

full-disk encryption, hardware-backed credential storage, and support for centralized device management and provisioning. Even more enterprise- oriented features and security improvements such as managed profile support, improved full-disk encryption, and support for biometric authen- tication have been announced for the next Android release (referred to as *Android L* as I write this).

As with any new platform feature, discussing cutting-edge security improvements is exciting, but it's arguably more

important to understand Android's security architecture from the bottom up because each new secu- rity feature builds upon and integrates with the platform's core security model. Android's sandboxing model (in which each application runs as a separate Linux user and has a dedicated data directory) and permission sys- tem (which requires each application to explicitly declare the platform fea- tures it requires) are fairly well understood and documented. However, the internals of other fundamental platform features that have an impact on device security, such as package management and code signing, are largely treated as a black box beyond the security research community.

One of the reasons for Android's popularity is the relative ease with which a device can be "flashed" with a custom build of Android, "rooted" by applying a third-party update package, or otherwise customized. Android enthusiast forums and blogs feature many practical "How to" guides that take users through the steps necessary to unlock a device and apply various customization packages, but they offer very little structured information about how such system updates operate under the hood and what risks they carry. This books aims to fill these gaps by providing an exploration of how Android works by describing its security architecture from the bottom up and delving deep into the implementation of major Android subsystems and components that relate to device and data security. The coverage includes broad topics that affect all applications, such as package and user manage- ment, permissions and device policy, as well as more specific ones such as cryptographic providers, credential storage, and support for secure elements. It's not uncommon for entire Android subsystems to be replaced or rewritten between releases, but security-related development is conserva- tive by nature, and while the described behavior might be changed or aug- mented across releases, Android's core security architecture should remain fairly stable in future releases.

who This Book Is For

This book should be useful to anyone interested in learning more about Android's security architecture. Both security researchers looking to evalu- ate the security level of Android as a whole or of a specific subsystem and platform developers working on customizing and extending Android will find the high-level description of each security feature and the provided implementation details to be a useful starting point for understanding the underlying platform source code. Application developers can gain a deeper understanding of how the platform works, which will enable them to write more secure applications and take better advantage of the security- related APIs that the platform provides. While some parts of the book are accessible to a non-technical audience, the bulk of the discussion is closely tied to Android source code or system files, so familiarity with the core con- cepts of software development in a Unix environment is useful.

Prerequisites

The book assumes basic familiarity with Unix-style operating systems, pref- erably Linux, and does not explain common concepts such as processes, user groups, file permissions, and so on. Linux-specific or recently added OS features (such as capability and mount namespaces) are generally intro- duced briefly before discussing Android subsystems that use them. Most of the presented platform code comes from core Android daemons (usu- ally implemented in C or C++) and system services (usually implemented in Java), so basic familiarity with at least one of these languages is also required. Some code examples feature sequences of Linux system calls, so familiarity with Linux system programming can be helpful in understand- ing the code, but is not absolutely required. Finally, while the basic struc- ture and core components (such as activities and services) of Android apps are briefly described in the initial chapters, basic understanding of

Android development is assumed.

android versions

The description of Android's architecture and implementation in this book (except for several proprietary Google features) is based on source code publicly released as part of the Android Open Source Project (AOSP). Most of the discussion and code excerpts reference Android 4.4, which is the lat- est publicly available version released with source code at the time of this writing. The master branch of AOSP is also referenced a few times, because commits to master are generally a good indicator of the direction future Android releases will take. However, not all changes to the master branch are incorporated in public releases as is, so it's quite possible that future releases will change and even remove some of the presented functionality.

A developer preview version of the next Android release (Android L, mentioned earlier) was announced shortly after the draft of this book was completed. However, as of this writing, the full source code of Android L is not available and its exact public release date is unknown. While the pre- view release does include some new security features, such as improvements to device encryption, managed profiles, and device management, none of these features are final and so are subject to change. That is why this book does not discuss any of these new features. Although we could introduce some of Android L's security improvements based on their observed behav- ior, without the underlying source code, any discussion about their imple- mentation would be incomplete and speculative.

how Is This Book Organized?

This book consists of 13 chapters that are designed to be read in sequence. Each chapter discusses a different aspect or feature of Android security, and subsequent chapters build on the concepts introduced by their pre- decessors. Even if you're already familiar with Android's architecture and security model and are looking for details about a specific topic, you should at least skim Chapters 1 through 3 because the topics they cover form the foundation for the rest of the book.

• **Chapter 1: Android's Security Model** gives a high-level overview of Android's architecture and security model.

• **Chapter 2: Permissions** describes how Android permissions are declared, used, and enforced by the system.

• **Chapter 3: Package Management** discusses code signing and details how Android's application installation and management process works.

• **Chapter 4: User Management** explores Android's multi-user support and describes how data isolation is implemented on multi-user devices.

• **Chapter 5: Cryptographic Providers** gives an overview of the Java Cryptography Architecture (JCA) framework and describes Android's JCA cryptographic providers.

• **Chapter 6: Network Security and PKI** introduces the architecture of the Java Secure Socket Extension (JSSE) framework and delves into its Android implementation.

• **Chapter 7: Credential Storage** explores Android's credential store and introduces the APIs it provides to applications that need to store crypto- graphic keys securely.

• **Chapter 8: Online Account Management** discusses Android's online account management framework and shows how support for Google accounts is integrated into Android.

• **Chapter 9: Enterprise Security** presents Android's device management framework, details how VPN support is implemented, and delves into Android's support for the Extensible Authentication Protocol (EAP).

• **Chapter 10: Device Security** introduces verified boot, disk encryption, and Android's lockscreen implementation, and shows how secure USB debugging and encrypted device backups are implemented.

• **Chapter 11: NFC and Secure Elements** gives an overview of Android's NFC stack, delves into secure element (SE) integration and APIs, and introduces host-based card emulation (HCE).

• **Chapter 12: SELinux** starts with a brief introduction to SELinux's archi- tecture and policy language, details the changes made to SELinux in order to integrate it in Android, and gives an overview of Android's base SELinux policy.

• **Chapter 13: System Updates and Root Access** discusses how Android's bootloader and recovery OS are used to perform full system updates, and details how root access can be obtained on both engineering and production Android builds.

Conventions

Because the main topic of this book is Android's architecture and implemen- tation, it contains multiple code excerpts and file listings, which are exten- sively referenced in the sections that follow each listing or code example. A few format conventions are used to set those references (which typically include multiple OS or programming language constructs) apart from the rest of the text.

Commands; function and variable names; XML attributes; and SQL object names are set in monospace (for example: "the `id` command," "the `getCallingUid()` method," "the `name` attribute," and so on). The names of files and directories, Linux users and groups, processes, and other OS objects are set in *italic* (for example: "the *packages.xml* file," "the *system* user," "the *vold* daemon," and so on). String literals are also set in *italic* (for example: "the *AndroidOpenSSL* provider"). If you use such string literals in a program, you typically need to enclose them in double or single quotes (for example: `Signature.getInstance("SHA1withRSA", "AndroidOpenSSL")`).

Java class names are typically in their unqualified format without the package name (for example: "the `Binder` class"); fully qualified names are only used when multiple classes with the same name exist in the discussed API or package, or when specifying the containing package is otherwise important (for example: "the `javax.net.ssl.SSLSocketFactory` class"). When referenced in the text, function and method names are shown with paren- theses, but their parameters are typically omitted for brevity (for example: "the `getInstance()` factory method"). See the relevant reference documenta- tion for the full function or method signature.

Most chapters include diagrams that illustrate the architecture or struc- ture of the discussed security subsystem or component. All diagrams follow an informal "boxes and arrows" style and do not conform strictly to a par- ticular format. That said, most diagrams borrow ideas from UML class and deployment diagrams, and boxes typically represent classes or objects, while arrows represent dependency or communication paths.

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.

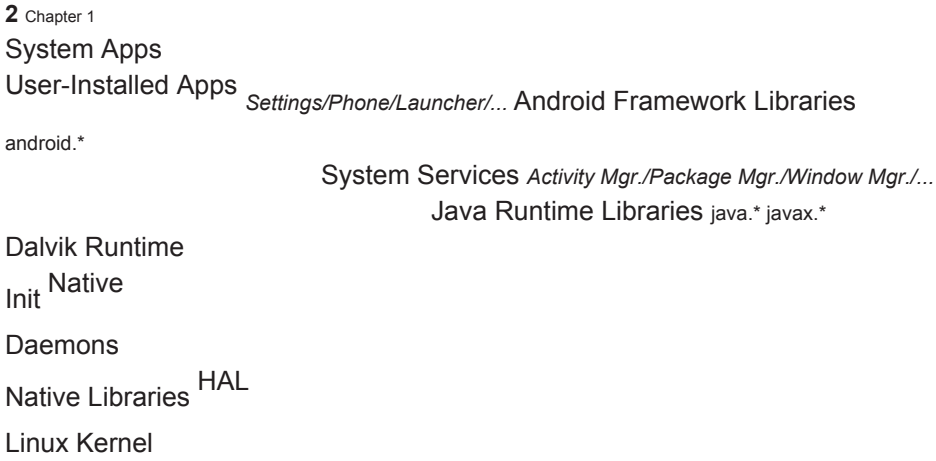

Figure 1-1: The Android architecture

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, file- system 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 embed- ded 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), anony- mous 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 net- working restricts access to network sockets to applications that hold spe- cific 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.

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 cor- responding 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 applica- tions do not contain device-independent DEX code directly. As a perfor- mance 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 librar- ies 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, includ- ing 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, *JavaTM 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 com- ponents 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 pro- cess'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.

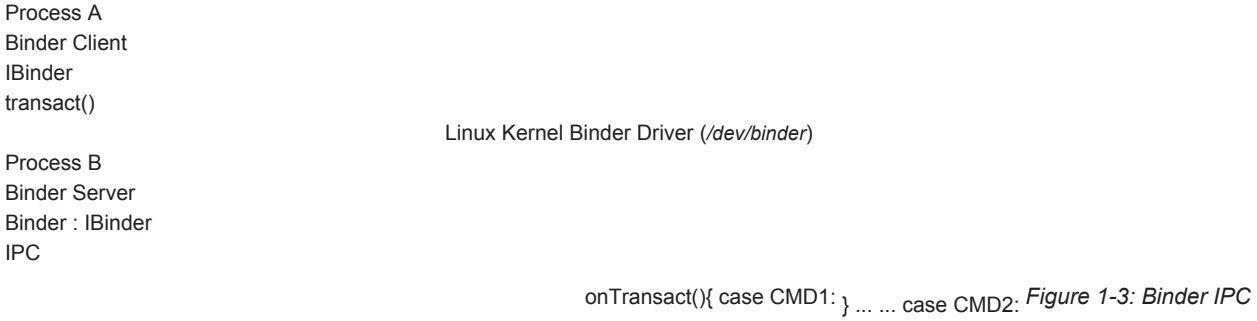

Figure 1-3: Binder IPC

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 ser- vices 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 includ- ing 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 sys- tem (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.

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 main- tain 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 capabili- ties 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 main- tain 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 trans actions 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 permis- sion 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 win- dow 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 refer- ence 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 functional- ity. 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` com- mand, which returns the name of each registered service and the imple- mented `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 auto- matically freed when no one references them and is implemented in the kernel driver with the BC_INCREFS, BC_ACQUIRE, BC_RELEASE, and BC_DECREFS com- mands. 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 ser- vices 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 produc- tion 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 (typi- cally 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 pack- age 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 dis- played 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 block- ing 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, appli- cation 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 acces- sible, 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 appli- cation (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 multi- user 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 dedi- cated 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 ser- vices 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 cor- responds 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 option- ally 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 writ- ten 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 permis- sion (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 imple- mentation) 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 confirma- tion 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 ker- nel 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 sys- tem 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 specify- ing 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 permis- sions are also implemented by consulting the caller UID obtained from Binder under the hood. The system uses the package database to deter- mine 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 sys- tem 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 secu- rity 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 gener- ated 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 physi- cal 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 set- tings 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 appli- cation from granting world access to its files (whether intentionally or due to a programming error).

Similarly, nothing prevents malicious applications from taking advan- tage 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 vulner- abilities 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 *authoriza- tion rules* called a *policy*. The policy can only be changed by an administra- tor, 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 applica- tions in different security *domains* and to define different access policies for each domain. As of version 4.4, SELinux is deployed in *enforcing mode* (viola- tions 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 sim- ilar 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 (typi- cally 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 navi- gated 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 for- mat 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 recov- ery 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 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.)

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

summary

Android is a privilege-separated operating system based on the Linux kernel. Higher-level system functions are implemented as a set of cooper- ating 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.

2

PERmIssIONs

In the previous chapter, we gave an overview of Android's security model and briefly introduced permissions. In this chapter we'll provide more details about permissions, focusing on their imple- mentation and enforcement. We will then discuss how to define custom permissions and apply them to each of Android's components. Finally, we'll say a few words about *pending intents*, which are tokens that allow an application to start an intent with the identity and privileges of another application.

The Nature of Permissions

As we learned in Chapter 1, Android applications are sandboxed and by default can access only their own files and a very limited set of system ser- vices. In order to interact with the system and other applications, Android applications can request a set of additional permissions that are granted at install time and cannot be changed (with some exceptions, as we'll discuss later in this chapter).

In Android, a *permission* is simply a string denoting the ability to per- form a particular operation. The target operation can be anything from accessing a physical resource (such as the device's SD card) or shared data (such as the list of registered contacts) to the ability to start or access a component in a third-party application. Android comes with a built-in set of predefined permissions. New permissions that correspond to new fea- tures are added in each version.

New built-in permissions, which lock down functionality that previously didn't require a permission, are applied conditionally, depending on the targetSdkVersion *speci- fied in an app's manifest: applications targeting Android versions that were released before the new permission was introduced cannot be expected to know about it, and therefore the permission is usually granted implicitly (without being requested). However, implicitly granted permissions are still shown in the list of permissions on the app installer screen so that users can be aware of them. Apps targeting later ver- sions need to explicitly request the new permission.*

Built-in permissions are documented in the platform API reference.[1] Additional permissions, called *custom permissions*, can be defined by both system and user-installed applications.

To view a list of the permissions currently known to the system, use the pm list permissions command (see Listing 2-1). To display additional infor- mation about permissions, including the defining package, label, description, and protection level, add the -f parameter to the command.

```
$ pm list permissions All Permissions:

permission:android.permission.REBOOT❶ permission:android.permission.BIND_VPN_SERVICE❷
permission:com.google.android.gallery3d.permission.GALLERY_PROVIDER❸
permission:com.android.launcher3.permission.RECEIVE_LAUNCH_BROADCASTS❹ --snip--
```

Listing 2-1: Getting a list of all permissions

Permission names are typically prefixed with their defining package concatenated with the string *.permission*. Because built-in permissions are defined in the android package, their names start with *android.permission*. For example, in Listing 2-1, the REBOOT ❶ and BIND_VPN_SERVICE ❷ are built-in permissions, while GALLERY_PROVIDER ❸ is defined by the Gallery application (package com.google.android.gallery3d) and RECEIVE_LAUNCH_BROADCASTS ❹ is defined by the default launcher application (package com.android.launcher3).

1. Google, *Android API Reference*, "Manifest.permission class," *http://developer.android.com/reference/android/Manifest.permission.html*

Requesting Permissions

Applications request permissions by adding one or more <uses-permission> tags to their *AndroidManifest.xml* file and can define new permissions with the <permission> tag. Listing 2-2 shows an example manifest file that requests the INTERNET and WRITE_EXTERNAL_STORAGE permissions. (We show how to define custom permission in "Custom Permissions" on page 42.)

```
<?xml version="1.0" encoding="utf-8"?> <manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" package="com.example.app"
    android:versionCode="1" android:versionName="1.0" >

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

android:name="android.permission.WRITE_EXTERNAL_STORAGE" /> --*snip*-- <application android:name="SampleApp" ...> --*snip*-- </application> </manifest>

Listing 2-2: Requesting permissions using the application manifest file

Permission management

Permissions are assigned to each application (as identified by a unique *package name*) at install time by the system *package manager* service. The package manager maintains a central database of installed packages, both pre installed and user-installed, with information about the install path, version, signing certificate, and assigned permissions of each package, as well as a list of all permissions defined on a device. (The `pm list permissions` command introduced in the previous section obtains this list by query- ing the package manager.) This package database is stored in the XML file */data/system/packages.xml*, which is updated each time an application is installed, updated, or uninstalled. Listing 2-3 shows a typical application entry from *packages.xml*.

```
<package name="com.google.android.apps.translate"
        codePath="/data/app/com.google.android.apps.translate-2.apk"
        nativeLibraryPath="/data/app-lib/com.google.android.apps.translate-2" flags="4767300" ft="1430dfab9e0"
        it="142cdf04d67" ut="1430dfabd8d" version="30000028" userId="10204"❶ installer="com.android.vending">

    <sigs count="1">
<cert index="7" />❷ </sigs>
```

<perms>❸

<item name="android.permission.READ_EXTERNAL_STORAGE" /> <item
name="android.permission.USE_CREDENTIALS" /> <item name="android.permission.READ_SMS" />
<item name="android.permission.CAMERA" /> <item
name="android.permission.WRITE_EXTERNAL_STORAGE" /> <item
name="android.permission.INTERNET" /> <item name="android.permission.MANAGE_ACCOUNTS"

```
/> <item name="android.permission.GET_ACCOUNTS" /> <item
name="android.permission.ACCESS_NETWORK_STATE" /> <item
name="android.permission.RECORD_AUDIO" /> </perms>

    <signing-keyset identifier="17" /> <signing-keyset identifier="6" />
                                                        </package>
```

Listing 2-3: Application entry in packages.xml

We discuss the meaning of most tags and attributes in Chapter 3, but for now let's focus on the ones that are related to permissions. Each pack- age is represented by a <package> element, which contains information about the assigned UID (in the userId attribute ❶), signing certificate (in the <cert> tag ❷), and assigned permissions (listed as children of the <perms> tag ❸). To get information about an installed package programmatically, use the getPackageInfo() method of the android.content.pm.PackageManager class, which returns a PackageInfo instance that encapsulates the informa- tion contained in the <package> tag.

If all permissions are assigned at install time and cannot be changed or revoked without uninstalling the application, how does the package manager decide whether it should grant the requested permissions? To understand this, we need to discuss permission protection levels.

Permission Protection Levels

According to the official documentation,[2] a permission's *protection level* "characterizes the potential risk implied in the permission and indicates the procedure that the system should follow when determining whether or not to grant the permission." In practice, this means that whether a permission is granted or not depends on its protection level. The following sections discuss the four protection levels defined in Android and how the system handles each.

normal This is the default value. It defines a permission with low risk to the sys- tem or other applications. Permissions with protection level *normal* are

2. Google, *Android API Guides, "*App Manifest: <permission> tag," *http://developer.android.com/guide/topics/manifest/permission-element.html#plevel*

dialog

Figure 2-2: Google Play Store client application install confirmation dialog

Figure 2-1: Default Android applica- tion install confirmation

automatically granted without requiring user confirmation. Examples are ACCESS_NETWORK_STATE (allows applications to access information about networks) and GET_ACCOUNTS (allows access to the list of accounts in the Accounts Service).

dangerous Permissions with the *dangerous* protection level give access to user data or some form of control over the device. Examples are READ_SMS (allows an appli- cation to read SMS messages) and CAMERA (gives applications access to the camera device). Before granting dangerous permissions, Android shows a confirmation dialog that displays information about the requested permis- sions. Because Android requires that all requested permission be granted at install time, the user can either agree to install the app,

thus granting the requested *dangerous* permission(s), or cancel the application install. For example, for the application shown in Listing 2-3 (Google Translate), the system confirmation dialog will look like the one shown in Figure 2-1.

Google Play and other application market clients display their own dialog, which is typically styled differently. For the same application, the Google Play Store client displays the dialog shown in Figure 2-2. Here, all *dangerous* permissions are organized by permission group (see “System Permissions” on page 37) and normal permissions are not displayed.

signature A *signature* permission is only granted to applications that are signed with the same key as the application that declared the permission. This is the "strongest" permission level because it requires the possession of a crypto- graphic key, which only the app (or platform) owner controls. Thus, applica- tions using *signature* permissions are typically controlled by the same author. Built-in signature permissions are typically used by system applications that perform device management tasks. Examples are NET_ADMIN

(configure net- work interfaces, IPSec, and so on) and ACCESS_ALL_EXTERNAL_STORAGE (access all multi-user external storage). We'll discuss *signature* permissions in more detail in "Signature Permissions" on page 39.

signatureOrSystem Permissions with this protection level are somewhat of a compromise: they are granted to applications that are either part of the system image, or that are signed with the same key as the app that declared the permis- sion. This allows vendors that have their applications preinstalled on an Android device to share specific features that require a permission without having to share signing keys. Until Android 4.3, any application installed on the *system* partition was granted *signatureOrSystem* permissions automatically. Since Android 4.4, applications need to be installed in the */system/priv-app/* directory in order to be granted permissions with this protection level.

Permission assignment

Permissions are enforced at various layers in Android. Higher-level compo- nents such as applications and system services query the package manager to determine which permissions have been assigned to an application and decide whether to grant access. Lower-level components like native daemons typically do not have access to the package manager and rely on the UID, GID, and supplementary GIDs assigned to a process in order to determine which privileges to grant it. Access to system resources like device files, Unix domain sockets (local sockets), and network sockets is regulated by the kernel based on the owner and access mode of the target resource and the UID and GIDs of the accessing process.

We'll look into framework-level permission enforcement in "Permission Enforcement" on page 30. Let's first discuss how permissions are mapped to OS-level constructs such as UID and GIDs and how these process IDs are used for permission enforcement.

Permissions and Process Attributes As in any Linux system, Android processes have a number of associated pro- cess attributes, most importantly real and effective UID and GID, and a set of supplementary GIDs.

As discussed in Chapter 1, each Android application is assigned a unique UID at install time and executes in a dedicated process. When the application is started, the process's UID and GID are set to the application UID assigned by the installer (the package manager service). If additional permissions have been assigned to the application, they are mapped to GIDs and assigned as supplementary GIDs to the process. Permission to GID mappings for built-in permissions are defined in the */etc/permission/ platform.xml* file. Listing 2-4 shows an excerpt from the *platform.xml* file found on an Android 4.4 device.

```
<?xml version="1.0" encoding="utf-8"?> <permissions>
    --snip-- <permission name="android.permission.INTERNET" >❶
<group gid="inet" /> </permission>

    <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >❷
            <group gid="sdcard_r" /> <group gid="sdcard_rw" />
                                        </permission>

    <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS"
                                    uid="media" />❸ <assign-permission
    name="android.permission.ACCESS_SURFACE_FLINGER"
```

```
uid="media" />❹ --snip-- </permissions>
```

Listing 2-4: Permission to GID mapping in platform.xml

Here, the INTERNET permission is associated with the *inet* GID ❶, and the WRITE_EXTERNAL_STORAGE permission is associated with the *sdcard_r* and *sdcard_rw* GIDs ❷. Thus any process for an app that has been granted the INTERNET permission is associated with the supplementary GID correspond- ing to the *inet* group, and processes with the WRITE_EXTERNAL_STORAGE permis- sion have the GIDs of *sdcard_r* and *sdcard_rw* added to the list of associated supplementary GIDs.

The <assign-permission> tag serves the opposite purpose: it is used to assign higher-level permissions to system processes running under a spe- cific UID that do not have a corresponding package. Listing 2-4 shows that processes running with the *media* UID (in practice, this is the *mediaserver* daemon) are assigned the MODIFY_AUDIO_SETTINGS ❸ and ACCESS_SURFACE_FLINGER ❹ permissions.

Android does not have an */etc/group* file, so the mapping from group names to GIDs is static and defined in the *android_filesystem_config.h* header file. Listing 2-5 shows an excerpt containing the *sdcard_rw* ❶, *sdcard_r* ❷, and *inet* ❸ groups.

```
--snip-- #define AID_ROOT 0 /* traditional unix root user */ #define AID_SYSTEM 1000 /* system server */
--snip--
```

#define AID_SDCARD_RW 1015 /* external storage write access */ #define AID_SDCARD_R 1028 /* external storage read access */ #define AID_SDCARD_ALL 1035 /* access all users external storage */ --*snip*-- #define AID_INET 3003 /* can create AF_INET and AF_INET6 sockets */ --*snip*--

struct android_id_info {

```
const char *name; unsigned aid; };static const struct android_id_info android_ids[] = {
{ "root", AID_ROOT, }, { "system", AID_SYSTEM, }, --snip-- { "sdcard_rw", AID_SDCARD_RW, },❶ { "sdcard_r",
AID_SDCARD_R, },❷ { "sdcard_all", AID_SDCARD_ALL, }, --snip-- { "inet", AID_INET, },❸ };
```

Listing 2-5: Static user and group name to UID/GID mapping in android_filesystem_config.h

The *android_filesystem_config.h* file also defines the owner, access mode, and associated capabilities (for executables) of core Android system direc- tories and files.

The package manager reads *platform.xml* at startup and maintains a list of permissions and associated GIDs. When it grants permissions to a package during installation, the package manager checks whether each permission has an associated GID(s). If so, the GID(s) is added to the list of supple- mentary GIDs associated with the application. The supplementary GID list is written as the last field of the *packages.list* file (see Listing 1-5 on page 14).

Process Attribute Assignment Before we see how the kernel and lower-level system services check and enforce permissions, we need to examine how Android application pro- cesses are started and assigned process attributes.

As discussed in Chapter 1, Android applications are implemented in Java and are executed by the Dalvik VM. Thus each application pro- cess is in fact a Dalvik VM process executing the application's bytecode. In order to reduce the application memory footprint and improve startup time, Android does not start a new Dalvik VM process for each application. Instead, it uses a partially initialized process called *zygote* and forks it (using the fork() system call[3]) when it needs to start a new application. However,

3. For detailed information about process management functions like fork(), setuid(), and so on, see the respective man pages or a Unix programming text, such as W. Richard Stevens and Stephen A. Rago's *Advanced Programming in the UNIX Environment (3rd edition)*, Addison- Wesley Professional, 2013.

instead of calling one of the exec() functions like it does when starting a native process, it merely executes the main() function of the specified Java class. This process is called *specialization*, because the generic *zygote* process is turned into a specific application process, much like cells originating from the zygote cell specialize into cells that perform different functions. Thus the forked process inherits the memory image of the *zygote* process, which has pre loaded most core and application framework Java classes. Because those classes never change and Linux uses a copy-on-write mechanism when fork- ing processes, all child processes of *zygote* (that is, all Android applications) share the same copy of framework Java classes.

The *zygote* process is started by the *init.rc* initialization script and receives commands on a Unix-domain socket, also named *zygote*. When *zygote* receives a request to start a new application process, it forks itself, and the child process executes roughly the following code (abbreviated from forkAndSpecializeCommon() in *dalvik_system_Zygote.cpp*) in order to spe- cialize itself as shown in Listing 2-6.

```
pid = fork();

if (pid == 0) {
int err; /* The child process */ err = setgroupsIntarray(gids);❶ err = setrlimitsFromArray(rlimits);❷ err = setresgid(gid,
gid, gid);❸ err = setresuid(uid, uid, uid);❹ err = setCapabilities(permittedCapabilities, effectiveCapabilities);❺ err =
set_sched_policy(0, SP_DEFAULT);❻ err = setSELinuxContext(uid, isSystemServer, seInfo, niceName);❼
enableDebugFeatures(debugFlags);❽ }
```

Listing 2-6: Application process specialization in zygote

As shown here, the child process first sets its supplementary GIDs (corre- sponding to permissions) using setgroups(),

called by setgroupsIntarray() at ❶. Next, it sets resource limits using setrlimit(), called by setrlimitsFromArray() at ❷, then sets the real, effective, and saved user and group IDs using setresgid() ❸ and setresuid() ❹.

The child process is able to change its resource limits and all process attributes because it initially executes as root, just like its parent process, *zygote*. After the new process attributes are set, the child process executes with the assigned UIDs and GIDs and cannot go back to executing as root because the saved user ID is not 0.

After setting the UIDs and GIDs, the process sets its capabilities[4] using capset(), called from setCapabilities() ❺. Then, it sets its scheduling policy

4. For a discussion of Linux capabilities, see Chapter 39 of Michael Kerrisk's *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, 2010.

by adding itself to one of the predefined control groups ❻.[5] At ❼, the pro- cess sets its nice name (displayed in the process list, typically the applica- tion's package name) and *seinfo* tag (used by SELinux, which we discuss in Chapter 12). Finally, it enables debugging if requested ❽.

Android 4.4 introduces a new, experimental runtime called Android RunTime (ART),

which is expected to replace Dalvik in a future version. While ART brings many changes to the current execution environment, most importantly ahead-of-time (AOT) compilation, it uses the same zygote-based app process execution model as Dalvik.

The process relationship between *zygote* and application process is evident in the process list obtained with the ps command, as shown in Listing 2-7.

```
$ ps USER PID PPID VSIZE RSS WCHAN PC NAME root 1 0 680 540 ffffffff 00000000 S /init❶ --snip-- root 181 1 858808
38280 ffffffff 00000000 S zygote❷ --snip-- radio 1139 181 926888 46512 ffffffff 00000000 S com.android.phone nfc 1154
181 888516 36976 ffffffff 00000000 S com.android.nfc u0_a7 1219 181 956836 48012 ffffffff 00000000 S
com.google.android.gms
```

Listing 2-7: zygote *and application process relationship*

Here, the PID column denotes the process ID, the PPID column denotes the parent process ID, and the NAME column denotes the process name. As you can see, *zygote* (PID 181 ❷) is started by the *init* process (PID 1 ❶) and all application processes have *zygote* as their parent (PPID 181). Each process executes under a dedicated user, either built-in (*radio*, *nfc*), or auto- matically assigned (*u0_a7*) at install time. The process names are set to the package name of each application (com.android.phone, com.android.nfc, and com.google.android.gms).

Permission Enforcement

As discussed in the previous section, each application process is assigned a UID, GID, and supplementary GIDs when it is forked from *zygote*. The ker- nel and system daemons use these process identifiers to decide whether to grant access to a particular system resource or function.

Kernel-Level Enforcement Access to regular files, device nodes, and local sockets is regulated just as it is in any Linux system. One Android-specific addition is requiring processes that want to create network sockets to belong to the group *inet*. This Android kernel addition is known as "paranoid network security" and is implemented as an additional check in the Android kernel, as shown in Listing 2-8.

5. Linux Kernel Archives, *CGROUPS*, *https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt*

```
#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);❶} #else static
inline int current_has_network(void) { return 1;❷ }#endif --snip-- static int inet_create(struct net *net, struct socket *sock,
int protocol,
                                                    int kern) {
        --snip-- if (!current_has_network())
    return -EACCES;❸ --snip-- }
```

Listing 2-8: Paranoid network security implementation in the Android kernel

Caller processes that do not belong to the AID_INET (GID 3003, name *inet*) group and do not have the CAP_NET_RAW capability (allowing the use of RAW and PACKET sockets) receive an access denied error (❶ and ❸). Non-Android kernels do not define CONFIG_ANDROID_PARANOID_NETWORK and thus no special group membership is required to create a socket ❷. In order for the *inet* group to be assigned to an application

process, it needs to be granted the INTERNET permission. As a result, only applications with the INTERNET per- mission can create network sockets. In addition to checking process creden- tials when creating sockets, Android kernels also grant certain capabilities to processes executing with specific GIDs: processes that execute with the AID_NET_RAW (GID 3004) are given the CAP_NET_RAW capability, and those exe- cuting with AID_NET_ADMIN (GID 3005) are given the CAP_NET_ADMIN capability.

Paranoid network security is also used to control access to Bluetooth sockets and the kernel tunneling driver (used for VPN). A full list of Android GIDs that the kernel treats in a special way can be found in the *include/linux/android_aid.h* file in the kernel source tree.

Native Daemon-Level Enforcement While Binder is the preferred IPC mechanism in Android, lower-level native daemons often use Unix domain sockets (local sockets) for IPC. Because Unix domain sockets are represented as nodes on the filesystem, standard filesystem permission can be used to control access.

As most sockets are created with an access mode that only allows access to their owner and group, clients running under a different UID and GID cannot connect to the socket. Local sockets for system daemons are defined

in *init.rc* and created by *init* on startup with the specified access mode. For example, Listing 2-9 shows how the volume management daemon (*vold*) is defined in *init.rc*:

```
service vold /system/bin/vold
    class core socket vold stream 0660 root mount❶ ioprio be 2
```

Listing 2-9: vold *daemon entry in* init.rc

vold declares a socket called *vold* with the 0660 access mode, owned by *root* and with group set to *mount* ❶. The *vold* daemon needs to run as root in order to mount or unmount volumes, but members of the *mount* group (AID_MOUNT, GID 1009) can send it commands via the local socket without needing to run as the superuser. Local sockets for Android daemons are created in the */dev/socket/* directory. Listing 2-10 shows that the *vold* socket ❶ has the owner and permission specified in *init.rc*.

```
$ ls -l /dev/socket

srw-rw---- system system 1970-01-18 14:26 adbd srw------- system system 1970-01-18 14:26 installd srw-rw---- root
system 1970-01-18 14:26 netd --snip-- srw-rw-rw- root root 1970-01-18 14:26 property_service srw-rw---- root radio
1970-01-18 14:26 rild srw-rw---- root mount 1970-01-18 14:26 vold❶ srw-rw---- root system 1970-01-18 14:26 zygote
```

Listing 2-10: Local sockets for core system daemons in /dev/socket/

Unix domain sockets allow the passing and querying of client creden- tials using the SCM_CREDENTIALS control message and the SO_PEERCRED socket option. Like the effective UID and effective GUID that are part of a Binder transaction, the peer credentials associated with a local socket are checked by the kernel and cannot be forged by user-level processes. This allows native daemons to implement additional, fine-grained control over the operations that they allow for a particular client, as shown in Listing 2-11 using the *vold* daemon as an example.

```
int CommandListener::CryptfsCmd::runCommand(SocketClient *cli,
                                            int argc, char **argv) { if ((cli->getUid() != 0) && (cli->getUid() !=
    AID_SYSTEM)) {❶
        cli->sendMsg(ResponseCode::CommandNoPermission,
"No permission to run cryptfs commands", false); return 0; } --snip-- }
```

Listing 2-11: Fine-grained access control based on socket client credentials in vold

The *vold* daemon only allows encrypted container management com- mands to clients running as the *root* (UID 0) or *system* (AID_SYSTEM, UID 1000) users. Here, the UID returned by SocketClient->getUid() ❶ is initialized with the client UID obtained using getsockopt(SO_PEERCRED) as shown in Listing 2-12 at ❶.

```
void SocketClient::init(int socket, bool owned, bool useCmdNum) {
    --snip-- struct ucred creds; socklen_t szCreds = sizeof(creds);
    memset(&creds, 0, szCreds);

    int err = getsockopt(socket, SOL_SOCKET, SO_PEERCRED, &creds, &szCreds);❶ if (err == 0) {
mPid = creds.pid; mUid = creds.uid; mGid = creds.gid; } }
```

Listing 2-12: Obtaining local socket client credentials using getsockopt()

Local socket connection functionality is encapsulated in the android.net.LocalSocket class and is available to Java applications as well, allowing higher-level system services to communicate with native daemons without using JNI code. For example, the MountService framework class uses LocalSocket to send commands to the *vold* daemon.

Framework-Level Enforcement As discussed in the introduction to Android permissions, access to Android components can be controlled using permissions by declaring the required permissions in the manifest of the enclosing application. The system keeps track of the permissions associated with each component and checks to see whether callers have been granted the required permissions before allowing access. Because components cannot change the permissions they require at runtime, enforcement by the system is *static*. Static permissions are an example of declarative security. When using declarative security, security attributes such as roles and permissions are placed in the metadata of a component (the *AndroidManifest.xml* file in Android), rather than in the component itself, and are enforced by the container or runtime environ- ment. This has the advantage of isolating security decisions from business logic but can be less flexible than implementing securing checks within the component.

Android components can also check to see whether a calling process has been granted a certain permission without declaring the permissions in the manifest. This *dynamic permission enforcement* requires more work but allows for more fine-grained access control. Dynamic permission enforce- ment is an example of imperative security, because security decisions are made by each component rather than being enforced by the runtime environment.

Let's look at how dynamic and static permission enforcement are imple- mented in more detail.

Dynamic Enforcement As discussed in Chapter 1, the core of Android is implemented as a set of cooperating system services that can be called from other processes using the Binder IPC mechanism. Core services register with the service manager and any application that knows their registration name can obtain a

Binder reference. Because Binder does not have a built-in access control mechanism, when clients have a reference they can call any method of the underlying system service by passing the appropriate parameters to Binder.transact(). Therefore, access control needs to be implemented by each system service.

In Chapter 1, we showed that system services can regulate access to exported operations by directly checking the UID of the caller obtained from Binder.getCallingUid() (see Listing 1-7 on page 15). However, this method requires that the service knows the list of allowed UIDs in advance, which only works for well-known fixed UIDs such as those of *root* (UID 0) and *system* (UID 1000). Also, most services do not care about the actual UID of the caller; they simply want to check if it has been granted a certain permission.

Because each application UID in Android is associated with a unique package (unless it is part of a shared user ID), and the package manager keeps track of the permissions granted to each package, this is made possible by querying the package manager service. Checking to see whether the caller has a certain permission is a very common operation, and Android provides a number of helper methods in the android.content.Context class that can per- form this check.

Let's first examine how the int Context.checkPermission(String permission, int pid, int uid) method works. This method returns PERMISSION_GRANTED if the passed UID has the permission, and returns PERMISSION_DENIED otherwise. If the caller is *root* or *system*, the permission is automatically granted. As a performance optimization, if the requested permission has been declared by the calling app, it is granted without examining the actual permission. If that is not the case, the method checks to see whether the target com- ponent is public (exported) or private, and denies access to all private components. (We'll discuss component export in "Public and Private Components" on page 43.) Finally, the code queries the package man- ager service to see if the caller has been granted the requested permission. The relevant code from the PackageManagerService class is shown in Listing 2-13.

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

Listing 2-13: UID-based permission check in PackageManagerService

Here the PackageManagerService first determines the *app ID* of the appli- cation based on the passed UID ❶ (the same application can be assigned multiple UIDs when installed for different users, which we discuss in detail in Chapter 4) and then obtains the set of granted permissions. If the GrantedPermission class (which holds the actual java.util.Set<String> of permission names) contains the target permission, the method returns PERMISSION_GRANTED ❷. If not, it checks whether the target permission should be automatically assigned to the passed-in UID ❸ (based on the <assign-permission> tags in platform.xml, as shown in Listing 2-4). If this check fails as well, it finally returns PERMISSION_DENIED.

The other permission-check helper methods in the Context class follow the same procedure. The int checkCallingOrSelfPermission(String permission) method calls Binder.getCallingUid() and Binder.getCallingPid() for us, and then calls checkPermission(String permission, int pid, int uid) using the obtained values. The enforcePermission(String permission, int pid, int uid, String message) method does not return a result but instead throws a SecurityException with the specified message if the permission is not granted. For example, the

BatterStatsService class guarantees that only apps that have the BATTERY_STATS permission can obtain battery statistics by calling enforceCallingPermission() before executing any other code, as shown in Listing 2-14. Callers that have not been granted the permission receive a SecurityException.

```
public byte[] getStatistics() {
    mContext.enforceCallingPermission(
android.Manifest.permission.BATTERY_STATS, null); Parcel out = Parcel.obtain(); mStats.writeToParcel(out,
0); byte[] data = out.marshall(); out.recycle(); return data; }
```

Listing 2-14: Dynamic permission check in BatteryStatsService

Static Enforcement Static permission enforcement comes into play when an application tries to interact with a component declared by another application. The

enforcement process takes into account the permissions declared for each target component (if any) and allows the interaction if the caller process has been granted the required permission.

Android uses intents to describe an operation it needs to perform, and intents that fully specify the target component (by package and class name) are called *explicit*. On the other hand, *implicit* intents contain some data (often only an abstract action such as ACTION_SEND) that allows the system to find a matching component, but they

do not fully specify a target component.

When the system receives an implicit intent, it first resolves it by search- ing for matching components. If more than one matching component is found, the user is presented with a selection dialog. When a target compo- nent has been selected, Android checks to see whether it has any associated permissions, and if it does, checks whether they have been granted to the caller. The general process is similar to dynamic enforcement: the UID and PID of the caller are obtained using Binder.getCallingUid() and Binder.getCallingPid(), the caller UID is mapped to a package name, and the associated permissions are retrieved. If the set of caller permissions contains the ones required by the target component, the component is started; otherwise, a SecurityException is thrown.

Permission checks are performed by the ActivityManagerService, which resolves the specified intent and checks to see whether the target compo- nent has an associated permission attribute. If so, it delegates the permission check to the package manager. The timing and concrete sequence of per- mission checks is slightly different depending on the target component. (Next, we'll examine how checks are performed for each component.)

Activity and Service Permission Enforcement Permission checks for activities are performed if the intent passed to Context.startActivity() or startActivityForResult() resolves to an activ- ity that declares a permission. A SecurityException is thrown if the caller does not have the required permission. Because Android services can be started, stopped, and bound to, calls to Context.startService(), stopService(), and bindService() are all subject to permission checks if the target service declares a permission.

Content Provider Permission Enforcement Content provider permissions can either protect the whole component or a particular exported URI, and different permissions can be specified for reading and writing. (You'll learn more about permission declaration in "Content Provider Permissions" on page 46.) If different permissions for reading and writing have been specified, the read permission con- trols who can call ContentResolver.query() on the target provider or URI, and the write permission controls who can call ContentResolver.insert(), ContentResolver.update(), and ContentResolver.delete() on the provider or one of its exported URIs. The checks are performed synchronously when one of these methods is called.

Broadcast Permission Enforcement When sending a broadcast, applications can require that receivers hold a particular permission by using the Context.sendBroadcast (Intent intent, String receiverPermission) method. Because broadcasts are asynchronous, no permission check is performed when calling this method. The check is performed when delivering the intent to registered receivers. If a tar- get receiver does not hold the required permission, it is skipped and does not receive the broadcast, but no exception is thrown. In turn, broadcast receivers can require that broadcasters hold a specific permission in order to be able to target them.

The required permission is specified in the manifest or when regis- tering a broadcast dynamically. This permission check is also performed when delivering the broadcast and does not result in a SecurityException. Thus delivering a broadcast might require two permission checks: one for the broadcast sender (if the receiver specified a permission) and one for the broadcast receiver (if the sender specified a permission).

Protected and Sticky Broadcasts Some system broadcasts are declared as *protected* (for example, BOOT_COMPLETED and PACKAGE_INSTALLED) and can only be sent by a system process running as one of SYSTEM_UID, PHONE_UID, SHELL_UID, BLUETOOTH_UID, or *root*. If a process run- ning under a different UID tries to send a protected broadcast, it receives a SecurityException when calling one of the sendBroadcast() methods. Sending "sticky" broadcasts (if marked as sticky, the system preserves the sent Intent object after

the broadcast is complete) requires that the sender holds BROADCAST_STICKY permission; otherwise, a SecurityException is thrown and the broadcast is not sent.

system Permissions

Android's built-in permissions are defined in the android package, some- times also referred to as "the framework" or "the platform." As we learned in Chapter 1, the core Android framework is the set of classes shared by system services, with some exposed via the public SDK as well. Framework classes are packaged in JAR files found in */system/framework/* (about 40 in latest releases).

Besides JAR libraries, the framework contains a single APK file, *framework-res.apk*. As the name implies, it packages framework resources (animation, drawables, layouts, and so on), but no actual code. Most importantly, it defines the android package and system permissions. As *framework-res.apk* is an APK file, it contains an *AndroidManifest.xml* file where permission groups and permissions are declared (see Listing 2-15).

```
<?xml version="1.0" encoding="utf-8"?> <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">
```

```
--snip-- <protected-broadcast android:name="android.intent.action.BOOT_COMPLETED" />❶ <protected-broadcast
android:name="android.intent.action.PACKAGE_INSTALL" /> --snip-- <permission-group
android:name="android.permission-group.MESSAGES"
android:label="@string/permgrouplab_messages"
android:icon="@drawable/perm_group_messages"
android:description="@string/permgroupdesc_messages"
```

```
    android:permissionGroupFlags="personalInfo" android:priority="360"/>❷ <permission
    android:name="android.permission.SEND_SMS"
    android:permissionGroup="android.permission-group.MESSAGES"❸
    android:protectionLevel="dangerous" android:permissionFlags="costsMoney"
    android:label="@string/permlab_sendSms" android:description="@string/permdesc_sendSms" /> --snip--
    <permission android:name="android.permission.NET_ADMIN"
    android:permissionGroup="android.permission-group.SYSTEM_TOOLS" android:protectionLevel="signature"
    />❹ --snip-- <permission android:name="android.permission.MANAGE_USB"
    android:permissionGroup="android.permission-group.HARDWARE_CONTROLS"
    android:protectionLevel="signature|system"❺ android:label="@string/permlab_manageUsb"
    android:description="@string/permdesc_manageUsb" /> --snip-- <permission
    android:name="android.permission.WRITE_SECURE_SETTINGS"
  android:permissionGroup="android.permission-group.DEVELOPMENT_TOOLS"
  android:protectionLevel="signature|system|development"❻ android:label="@string/permlab_writeSecureSettings"
  android:description="@string/permdesc_writeSecureSettings" /> --snip-- </manifest>
```

Listing 2-15: System permission definitions in the manifest of frameworkgres.apk

As shown in this listing, the *AndroidManifest.xml* file also declares the system's protected broadcasts ❶. A *permission group* ❷ specifies a name for a set of related permissions. Individual permission can be added to a group by specifying the group name in their permissionGroup attribute ❸.

Permission groups are used to display related permissions in the system UI, but each permission still needs to be requested individually. That is, appli- cations cannot request that they be granted all the permissions in a group.

Recall that each permission has an associated protection level declared using the protectionLevel attribute, as shown at ❹.

Protection levels can be combined with *protection flags* to further con- strain how permissions are granted. The currently defined flags are system (0x10) and development (0x20). The system flag requires that applications be part of the system image (that is, installed on the read-only *system* partition) in order to be granted a permission. For example, the MANAGE_USB permis- sion, which allows applications to manage preferences and permissions for

USB devices, is only granted to applications that are both signed with the platform signing key and installed on the *system* partition ❺. The development flag marks development permissions ❻, which we'll discuss after presenting signature permissions.

Signature Permissions As discussed in Chapter 1, all Android applications are required to be code signed with a signature key controlled by the developer. This applies to sys- tem applications and the framework resource package as well. We discuss package signing in detail in Chapter 3, but for now let's say a few words about how system applications are signed.

System applications are signed by a *platform key*. By default, there are four different keys in the current Android source tree: *platform*, *shared*, *media*, and *testkey* (*releasekey* for release builds). All packages considered part of the core platform (System UI, Settings, Phone, Bluetooth, and so on) are signed with the *platform* key; the search- and contacts-related packages with the *shared* key; the gallery app and media related providers with the *media* key; and everything else (including packages that don't explicitly specify the signing key in their makefile) with the *testkey* (or *releasekey*). The *framework-res.apk* APK that defines system permissions is signed with the *platform* key. Thus any app trying to request a system permission with *signature* protection level needs to be signed with the same key as the framework resource package.

For example, the NET_ADMIN permission shown in Listing 2-15 (which allows a granted application to control

network interfaces), is declared with the *signature* protection level ❹ and can only be granted to applications signed with the *platform* key.

The Android open source repository (AOSP) includes pregenerated test keys that are used by default when signing compiled packages. They should never be used for production builds because they are public and available to anyone who down- loads Android source code. Release builds should be signed with newly generated private keys that belong only to the build owner. Keys can be generated using the make_key *script, which is included in the* development/tools/ *AOSP directory. See the* build/target/product/security/README *file for details on platform key generation.*

Development Permissions Traditionally, the Android permission model does not allow for dynamically granting and revoking permissions, and the set of granted permission for an application is fixed at install time. However, since Android 4.2, this rule has been relaxed a little by adding a number of *development permissions* (such as READ_LOGS and WRITE_SECURE_SETTINGS). Development permission can be granted or revoked on demand using the *pm grant* and *pm revoke* commands on the Android shell.

Of course, this operation is not available to everyone and is protected by the GRANT_REVOKE_PERMISSIONS signature permission. It is granted to the android .uid.shell *shared user ID (UID 2000), and to all processes started from the Android shell (which also runs as UID 2000).*

shared User ID

Android applications signed with the same key can request the ability to run as the same UID, and optionally in the same process. This feature is referred to as *shared user ID* and is extensively used by core framework services and system applications. Because it can have subtle effects on process accounting and application management, the Android team does not recommend that third-party applications use it, but it is available to user-installed applications as well. Additionally, switching an existing appli- cations that does not use a shared user ID to a shared user ID is not sup- ported, so cooperating applications that need to use shared user ID should be designed and released as such from the start.

Shared user ID is enabled by adding the sharedUserId attribute to *AndroidManifest.xml*'s root element. The user ID specified in the mani- fest needs to be in Java package format (containing at least one dot [.]) and is used as an identifier, much like package names for applications. If the specified shared UID does not exist, it is created. If another package with the same shared UID is already installed, the signing certificate is compared to that of the existing package, and if they do not match, an INSTALL_FAILED_SHARED_USER_INCOMPATIBLE error is returned and installation fails.

Adding the sharedUserId attribute to a new version of an installed app will cause it to change its UID, which would result in losing access to its own files (that was the case in some early Android versions). Therefore, this is disallowed by the system, which will reject the update with the INSTALL_FAILED_UID_CHANGED error. In short, if you plan to use shared UID for your apps, you have to design for it from the start, and must have used it since the very first release.

The shared UID itself is a first class object in the system's package database and is treated much like applications: it has an associated signing certificate(s) and permissions. Android has five built-in shared UIDs, which are

automatically added when the system is bootstrapped:

- *android.uid.system* (SYSTEM_UID, 1000)
- *android.uid.phone* (PHONE_UID, 1001)
- *android.uid.bluetooth* (BLUETOOH_UID, 1002)
- *android.uid.log* (LOG_UID, 1007)
- *android.uid.nfc* (NFC_UID, 1027)

Listing 2-16 shows how the *android.uid.system* shared user is defined:

```
<shared-user name="android.uid.system" userId="1000"> <sigs count="1">
```



```
<cert index="4" /> </sigs> <perms> <item name="android.permission.MASTER_CLEAR" /> <item
name="android.permission.CLEAR_APP_USER_DATA" /> <item
name="android.permission.MODIFY_NETWORK_ACCOUNTING" /> --snip-- <shared-user/>
```

Listing 2-16: Definition of the android.uid.system *shared user*

As you can see, apart from having a bunch of scary permissions (about 66 on a 4.4 device), the definition is very similar to the package declara- tions shown earlier. Conversely, packages that are part of a shared user do not have an associated granted permission list. Instead, they inherit the permissions of the shared user, which are a union of the permissions requested by all currently installed packages with the same shared user ID. One side effect of this is that if a package is part of a shared user, it can access APIs that it hasn't explicitly requested permissions for, as long as some package with the same shared user ID has already requested them. Permissions are dynamically removed from the <shared-user> definition as packages are installed or uninstalled though, so the set of available permis- sions is neither guaranteed nor constant.

Listing 2-17 shows how the declaration of the KeyChain system app that runs under a shared user ID looks like. As you can see, it references the shared user with the sharedUserId attribute and lacks explicit permission declarations:

```
<package name="com.android.keychain"
        codePath="/system/app/KeyChain.apk" nativeLibraryPath="/data/app-lib/KeyChain"
        flags="540229" ft="13cd65721a0" it="13c2d4721f0" ut="13cd65721a0"
        version="19" sharedUserId="1000"> <sigs count="1">
<cert index="4" /> </sigs> <signing-keyset identifier="1" /> </package>
```

Listing 2-17: Package declaration of an application that runs under a shared user ID

The shared UID is not just a package management construct; it actu- ally maps to a shared Linux UID at runtime as well. Listing 2-18 shows an example of two system apps running as the *system* user (UID 1000):

```
system 5901 9852 845708 40972 ffffffff 00000000 S com.android.settings system 6201 9852 824756 22256 ffffffff
00000000 S com.android.keychain
```

Listing 2-18: Applications running under a shared UID (system)

Applications that are part of a shared user can run in the same pro- cess, and because they already have the same Linux UID and can access the same system resources, this typically does not require any additional

modifications. A common process can be requested by specifying the same process name in the process attribute of the <application> tag in the mani- fests of all apps that need to run in one process. While the obvious result of this is that the apps can share memory and communicate directly instead of using IPC, some system services allow special access to components running in the same process (for example, direct access to cached passwords or get- ting authentication tokens without showing UI prompts). Google applica- tions (such as Play Services and the Google location service) take advantage of this by requesting to run in the same process as the Google login service in order to be able to sync data in the background without user interaction. Naturally, they are signed with the same certificate and are part of the *com.google.uid.shared* shared user.

Custom Permissions

Custom permissions are simply permissions declared by third-party applica- tions. When declared, they can be added to application components for static enforcement by the system, or the application can dynamically check to see if callers have been granted the permission using the checkPermission() or enforcePermission() methods of the Context class. As with built-in permissions, applications can define permission groups that their custom permissions are added to. For example, Listing 2-19 shows the declaration of a permission group ❷ and the permission belonging to that group ❸.

```
<?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.app" android:versionCode="1"
    android:versionName="1.0" > --snip-- <permission-tree
        android:name="com.example.app.permission"
        android:label="@string/example_permission_tree_label" />❶

        <permission-group
            android:name="com.example.app.permission-group.TEST_GROUP"
            android:label="@string/test_permission_group_label"
            android:description="@string/test_permission_group_desc"/>❷

        <permission
            android:name="jcom.example.app.permission.PERMISSION1" android:label="@string/permission1_label"
            android:description="@string/permission1_desc"
            android:permissionGroup="com.example.app.permission-group.TEST_GROUP"
            android:protectionLevel="signature" />❸
--snip-- </manifest>
```

Listing 2-19: Custom permission tree, permission group, and permission declaration

As with system permissions, if the protection level is *normal* or *danger- ous*, custom permission will be granted automatically when the user okays the confirmation dialog. In order to be able to control which applications are granted a custom permission, you need to declare it with the *signature* protection level to guarantee that it will only be granted to applications signed with the same key.

The system can only grant a permission that it knows about, which means that appli- cations that define custom permissions need to be installed before the applications that make use of those permissions are installed. If an application requests a permission unknown to the system, it is ignored and not granted.

Applications can also add new permissions dynamically using the android.content.pm.PackageManager.addPermission() API

and remove them with the matching removePermision() API. Such dynamically added permissions must belong to a *permission tree* defined by the application. Applications can only add or remove permissions from a permission tree in their own pack- age or another package running as the same shared user ID.

Permission tree names are in reverse domain notation and a per- mission is considered to be in a permission tree if its name is pre- fixed with the permission tree name plus a dot (.). For example, the com.example.app.permission.PERMISSION2 permission is a member of the com.example.app.permission tree defined in Listing 2-19 at ❶. Listing 2-20 shows how to add a dynamic permission programmatically.

```
PackageManager pm = getPackageManager(); PermissionInfo permission = new PermissionInfo();
permission.name = "com.example.app.permission.PERMISSION2"; permission.labelRes =
R.string.permission_label; permission.protectionLevel = PermissionInfo.PROTECTION_SIGNATURE;
boolean added = pm.addPermission(permission); Log.d(TAG, "permission added: " + added);
```

Listing 2-20: Adding a dynamic permission programmatically

Dynamically added permissions are added to the package database (*/data/system/packages.xml*). They persist across reboots, just like permis- sions defined in the manifest, but they have an additional type attribute set to *dynamic*.

Public and Private Components

Components defined in the *AndroidManifest.xml* file can be public or pri- vate. Private components can be called only by the declaring application, while public ones are available to other applications as well.

With the exception of content providers, all components are private by default. Because the purpose of content providers is to share data with other applications, content providers were initially public by default, but this behavior changed in Android 4.2 (API Level 17). Applications that tar- get API Level 17 or later now get private content providers by default, but they are kept public for backward compatibility when targeting a lower API level.

Components can be made public by explicitly setting the exported attri- bute to true, or implicitly by declaring an intent filter. Components that have an intent filter but that do not need to be public can be made private by set- ting the exported attribute to false. If a component is not exported, calls from external applications are blocked by the activity manager, regardless of the permissions the calling process has been granted (unless it is running as *root* or *system*). Listing 2-21 shows how to keep a component private by setting the exported attribute to false.

```
<service android:name=".MyService" android:exported="false" >
    <intent-filter>
<action android:name="com.example.FETCH_DATA" /> </intent-filter> </service>
```

Listing 2-21: Keeping a component private by setting exported="false"

Unless explicitly intended for public consumption, all public compo- nents should be protected by a custom permission.

activity and service Permissions

Activities and services can each be protected by a single permission set with the permission attribute of the target component. The activity permis- sion is checked when other applications call Context.startActivity() or Con text.startActivityForResult() with an intent that resolves to that activity. For services, the permission is checked when other applications call one of Context.startService(), stopService(), or bindService() with an intent that resolves to the service.

For example, Listing 2-22 shows two custom permissions, START_MY_ACTIVITY and USE_MY_SERVICE, set to an activity ❶ and service ❷, respectively. Applications that want to use these components need to request the respective permissions using the <uses-permission> tag in their manifest.

```
<?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp" ... > <permission
    android:name="com.example.permission.START_MY_ACTIVITY"
    android:protectionLevel="signature" android:label="@string/start_my_activity_perm_label"
    android:description="@string/start_my_activity_perm_desc" /> <permission
    android:name="com.example.permission.USE_MY_SERVICE"
        android:protectionLevel="signature" android:label="@string/use_my_service_perm_label"
        android:description="@string/use_my_service_perm_desc" />

     --snip-- <activity android:name=".MyActivity"
     android:label="@string/my_activity" android:permission="com.example.permission.START_MY_ACTIVITY">❶
     <intent-filter> --snip-- </intent-filter> </activity> <service android:name=".MyService"
android:permission="com.example.permission.USE_MY_SERVICE">❷ <intent-filter> --snip-- </intent-filter>
</service> --snip-- </manifest>
```

Listing 2-22: Protecting activities and services with custom permissions

Broadcast Permissions

Unlike activities and services, permissions for broadcast receivers can be specified both by the receiver

itself and by the application sending the broadcast. When sending a broadcast, applications can either use the Context.sendBroadcast(Intent intent) method to send a broadcast to be delivered to all registered receives, or limit the scope of components that receive the broadcast by using the Context.sendBroadcast(Intent intent, String receiverPermission). The receiverPermission parameter specifies the permission that interested receivers need to hold in order to receive the broadcast. Alternatively, starting with Android 4.0, senders can use the Intent.setPackage(String packageName) to limit the scope of receivers to those defined in the specified package. On multi-user devices, system applications that hold the INTERACT_ACROSS_USERS permission can send a broadcast that is delivered only to a specific user by the using the sendBroadcastAsUser(Intent intent, UserHandle user) and sendBroadcastAsUser(Intent intent, UserHandle user, String receiverPermission) methods.

Receivers can limit who can send them broadcasts by specifying a per- mission using the permission attribute of the <receiver> tag in the manifest for statically registered receivers, or by passing the required permission to the Context.registerReceiver(BroadcastReceiver receiver, IntentFilter filter, String broadcastPermission, Handler scheduler) method for dynamically regis- tered receivers.

Only broadcasters that have been granted the required permission will be able to send a broadcast to that receiver. For example, device administration applications that enforce systemwide security policies (we discuss device administration in Chapter 9) require the BIND_DEVICE_ADMIN

permission in order to receive the DEVICE_ADMIN_ENABLED broadcast. Because this is a system permission with protection level *signature*, requiring the per- mission guarantees that only the system can activate device administration applications. For example, Listing 2-23 shows how the default Android Email application specifies the BIND_DEVICE_ADMIN ❶ permission for its PolicyAdmin receiver.

```
<?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.email" android:versionCode="500060" >
        --snip-- <receiver
            android:name=".SecurityPolicy$PolicyAdmin" android:label="@string/device_admin_label"
            android:description="@string/device_admin_description"
            android:permission="android.permission.BIND_DEVICE_ADMIN" >❶ <meta-data
            android:name="android.app.device_admin" android:resource="@xml/device_admin"
            /> <intent-filter> <action
android:name="android.app.action.DEVICE_ADMIN_ENABLED" /> </intent-filter> </receiver> --snip-- </manifest>
```

Listing 2-23: Specifying a permission for a statically registered broadcast receiver

As with other components, private broadcast receivers can only receive broadcasts originating from the same application.

Content Provider Permissions

As mentioned in "The Nature of Permissions" on page 21, content pro- viders have a more complex permission model than other components, as we'll describe in detail in this section.

Static Provider Permissions While a single permissions that controls access to the whole provider can be specified using the `permission` attribute, most providers employ different per- mission for reading and writing, and can also specify per-URI permissions. One example of a provider that uses different permissions for reading and writing is the built-in `ContactsProvider`. Listing 2-24 shows the declaration of its `ContactsProvider2` class.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.providers.contacts" android:sharedUserId="android.uid.shared"
        android:sharedUserLabel="@string/sharedUserLabel">
          --snip-- <provider android:name="ContactsProvider2"
            android:authorities="contacts;com.android.contacts" android:label="@string/provider_label"
            android:multiprocess="false" android:exported="true"
            android:readPermission="android.permission.READ_CONTACTS"❶
            android:writePermission="android.permission.WRITE_CONTACTS">❷ --snip-- <path-permission
 android:pathPattern="/contacts/.*/photo" android:readPermission="android.permission.GLOBAL_SEARCH" />❸
 <grant-uri-permission android:pathPattern=".*" /> </provider> --snip-- </manifest>
```

Listing 2-24: `ContactsProvider` permission declarations

The provider uses the `readPermission` attribute to specify one permission for reading data (`READ_CONTACTS` ❶), and a separate permission for writing data using the `writePermission` attribute (`WRITE_CONTACTS` ❷). Thus, applications that only hold the `READ_CONTACTS` permission can only call the `query()` method of the provider, and calls to `insert()`, `update()`, or `delete()` require the caller to hold the `WRITE_CONTACTS` permission. Applications that need to both read and write to the contacts provider need to hold both permissions.

When the global read and write permission are not sufficiently flexible, providers can specify per-URI permissions to protect a certain subset of their data. Per-URI permissions have higher priority than the component-level permission (or read and write permissions, if specified separately). Thus if an application wants to access a content provider URI that has an associated permission, it needs to hold only the target URI's permission,

and not the component-level permission. In Listing 2-24, the ContactsProvider2 uses the <path-permission> tag to require that applications trying to read photos of con- tacts hold the GLOBAL_SEARCH permission ❸. As per-URI permissions override the global read permission, interested applications do not need to hold the READ_CONTACTS permission. In practice, the GLOBAL_SEARCH permission is used to grant read-only access to some of the system providers' data to Android's search system, which cannot be expected to hold read permissions to all providers.

Dynamic Provider Permissions While statically defined per-URI permissions can be quite powerful, appli- cations sometimes need to grant temporary access to a particular piece of data (referred to by its URI) to other apps, without requiring that they hold a particular permission. For example, an email or messaging applica- tion may need to cooperate with an image viewer app in order to display an attachment. Because the app cannot know the URIs of attachments in advance, if it used static per-URI permissions, it would need to grant read access to all attachments to the image viewer app, which is undesirable.

To avoid this situation and potential security concern, applications can dynamically grant temporary per-URI access using the Context .grantUriPermission(String toPackage, Uri uri, int modeFlags) method and revoke access using the matching revokeUriPermission(Uri uri, int modeFlags) method. Temporary per-URI access is enabled by setting the global grantUriPermissions attribute to true or by adding a <grant-uri-permission> tag in order to enable it for a specific URI. For example, Listing 2-25 shows how the Email application uses the grantUriPermissions attribute

❶ to allow tempo- rary access to attachments without requiring the READ_ATTACHMENT permission.

```
<?xml version="1.0" encoding="utf-8"?> <manifest
xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.android.email" android:versionCode="500060" >
           <provider
android:name=".provider.AttachmentProvider" android:authorities="com.android.email.attachmentprovider"
android:grantUriPermissions="true"❶ android:exported="true"
android:readPermission="com.android.email.permission.READ_ATTACHMENT"/> --snip-- </manifest>
```

Listing 2-25: AttachmentProvider declaration from the Email app

In practice, applications rarely use the Context.grantPermission() and revokePermission() methods directly to allow per-URI access. Instead, they set the FLAG_GRANT_READ_URI_PERMISSION or FLAG_GRANT_WRITE_URI_PERMISSION flags to the intent used to start the cooperating application (image viewer in our example). When those flags are set, the recipient of the intent is granted per- mission to perform read or write operations on the URI in the intent's data.

Beginning with Android 4.4 (API Level 19), per-URI access grants can be persisted across device reboots with the ContentResolver .takePersistableUriPermission() method, if the received intent has the FLAG_GRANT_PERSISTABLE_URI_PERMISSION flag set. Grants are persisted to the */data/system/urigrants.xml* file and can be revoked by calling the releasePersistableUriPermission() method. Both transient and persistent per-URI access grants are managed by the system ActivityManagerService, which APIs related to per-URI access call internally.

Beginning with Android 4.1 (API level 16), applications can use the ClipData facility[6] of intents to add more than one content URI to temporar- ily be granted access to.

Per-URI access is granted using one of the FLAG_GRANT_* intent flags, and automatically revoked when the task of the called application finishes, so there is no need to call revokePermission(). Listing 2-26 shows how the Email application creates an intent that launches an attachment viewer application.

6. Google, *Android API Reference*, "ClipData," *http://developer.android.com/reference/android/ content/ClipData.html*

```
public Intent getAttachmentIntent(Context context, long accountId) {
      Uri contentUri = getUriForIntent(context, accountId); Intent intent = new
      Intent(Intent.ACTION_VIEW); intent.setDataAndType(contentUri, mContentType);
      intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION |
      Intent.FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET); return intent; }
```

Listing 2-26: Using the FLAG_GRANT_READ_URI_PERMISSION flag to start a viewer application

Pending Intents

Pending intents are neither an Android component nor a permission, but because they allow an application to grant its own permissions to another application, we discuss them here.

Pending intents encapsulate an intent and a target action to perform with it (start an activity, send a broadcast, and so on). The main difference from "regular" intents is that pending intents also include the identity of the applications that created them. This allows pending intents to be handed to other applications, which can use

them to perform the specified action using the identity and permissions of the original application. The identity stored in pending intents is guaranteed by the system ActivityManagerService, which keeps track of the currently active pending intents.

Pending intents are used to implement alarms and notifications in Android. Alarms and notifications allow any application to specify an action that needs to be performed on its behalf, either at a specified time for alarms, or when the user interacts with a system notification. Alarms and notifications can be triggered when the application that created them is no longer running, and the system uses the information in the pending intent to start it and per- form the intent action on its behalf. Listing 2-27 shows how the Email applica- tion uses a pending intent created with the PendingIntent.getBroadcast() ❶ to schedule broadcasts that trigger email synchronization.

```
private void setAlarm(long id, long millis) {
    --snip-- Intent i = new Intent(this, MailboxAlarmReceiver.class); i.putExtra("mailbox", id);
    i.setData(Uri.parse("Box" + id)); pi = PendingIntent.getBroadcast(this, 0, i, 0);❶
    mPendingIntents.put(id, pi); AlarmManager am =
(AlarmManager)getSystemService(Context.ALARM_SERVICE); m.set(AlarmManager.RTC_WAKEUP,
System.currentTimeMillis() + millis, pi); --snip-- }
```

Listing 2-27: Using a pending intent to schedule an alarm

Pending intents can be handed to non-system applications as well. The same rules apply: applications that receive a PendingIntent instance can perform the specified operation with the same permissions and iden- tity as creator applications. Therefore, care should be taken when building the base intent, and base intents should generally be as specific as possible (with component name explicitly specified) to ensure that the intent is received by the intended components.

The implementation of pending intents is rather complex, but it is based on the same IPC and sandboxing principles that other Android components are built upon. When an application creates a pending intent, the system retrieves its UID and PID using Binder.getCallingUid() and Binder.getCallingPid(). Based on those, the system retrieves the package name and user ID (on multi-user devices) of the creator and stores them in a PendingIntentRecord along with the base intent and any additional meta- data. The activity manager keeps a list of active pending intents by storing the corresponding PendingIntentRecords, and when triggered, retrieves the necessary record. It then uses the information in the record to assume the identity of the pending intent creator and execute the specified action. From there, the process is the same as when starting any Android compo- nent and the same permission checks are performed.

summary Android runs each application in a restricted sandbox and requires that applications request specific permissions in order to interact with other apps or the system. Permissions are strings that denote the ability to per- form a particular action. They are granted at application install time and (with the exception of development permissions) remain fixed during an application's lifetime. Permissions can be mapped to Linux supplemen- tary group IDs, which the kernel checks before granting access to system resources.

Higher-level system services enforce permissions by obtaining the UID of the calling application using Binder and looking up the permissions it holds in the package manager database. Permissions associated with a com- ponent declared in an application's manifest file are automatically enforced by the system, but applications can also choose to perform additional per- mission checks dynamically. In addition to using built-in permissions, appli- cations can also define custom permissions and associate them with their components in order to control access.

Each Android component can require a permission, and content pro- viders can additionally specify read and write permissions on a per-URI basis. Pending intents encapsulate the identity of the application that created them as well as an intent and an action to perform, which allows the system or third-party applications to perform actions on behalf of the original applications with the same identity and permissions.

3

PaCkaGE maNaGEmENT

In this chapter, we take an in-depth look at Android package management. We begin with a description of Android's package format and code signing imple- mentation, and then detail the APK install process. Next, we explore Android's support for encrypted APKs and secure application containers, which are used to implement a form of DRM for paid applications. Finally, we describe Android's pack- age verification mechanism and its most widely used implementation: the Google Play application verification service.

android application Package Format

Android applications are distributed and installed in the form of application package (APK) files, which are usually referred to as *APK files*. APK files are container files that include both application code and resources, as well as the application manifest file. They can also include a code signature. The

APK format is an extension of the Java JAR format,[1] which in turn is an exten- sion of the popular ZIP file format. APK files typically have the *.apk* extension and are associated with the *application/vnd.android.package-archive* MIME type.

Because APK files are simply ZIP files, you can easily examine their con- tents by extracting them with any compression utility that supports the ZIP format. Listing 3-1 shows the contents of a typical APK file after it

has been extracted.

```
apk/ |-- AndroidManifest.xml❶ |-- classes.dex❷ |--
resources.arsc❸ |-- assets/❹ |-- lib/❺ | |-- armeabi/ | | `--
libapp.so | `-- armeabi-v7a/ | `-- libapp.so |-- META-INF/❻ |
|-- CERT.RSA | |-- CERT.SF | `-- MANIFEST.MF `-- res/❼
    |-- anim/ |-- color/ |-- drawable/ |-- layout/ |--
    menu/ |-- raw/ `-- xml/
```

Listing 3-1: Contents of a typical APK file

Every APK file includes an *AndroidManifest.xml* file ❶ which declares the application's package name, version, components, and other metadata. The *classes.dex* file ❷ contains the executable code of the application and is in the native DEX format of the Dalvik VM. The *resources.arsc* ❸ packages all of the application's compiled resources such as strings and styles. The *assets* directory ❹ is used to bundle raw asset files with the application, such as fonts or music files.

Applications that take advantage of native libraries via JNI contain a *lib* directory ❺, with subdirectories for each supported platform architecture. Resources that are directly referenced from Android code, either directly using the android.content.res.Resources class or indirectly via higher-level APIs, are stored in the *res* directory ❼, with separate directories for each resource type (animations, images, menu definitions, and so on). Like JAR files, APK files also contain a *META-INF* directory ❻, which hosts the pack- age manifest file and code signatures. We'll describe the contents of this directory in the next section.

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

Code signing

As we learned in Chapter 2, Android uses APK code signing, in particular the APK signing certificate, in order to control which applications can be granted permission with the *signature* protection level. The APK signing certificate is also used for various checks during the application installa- tion process, so before we get into details about APK installation, we should become more familiar with code signing in Android. This section provides some details about Java code signing in general and highlights the differ- ences with Android's implementation.

Let's start with a few words about code signing in general. Why would anyone want to sign code? For the usual reasons: integrity and authenticity. Before executing any third-party program, you want to make sure that it hasn't been tampered with (integrity) and that it was actually created by the entity that it claims to come from (authenticity). These features are usually implemented by a digital signature scheme, which guarantees that only the entity owning the signing key can produce a valid code signature.

The signature verification process verifies both that the code has not been tampered with and that the signature was produced with the expected key. But one problem that code signing doesn't solve directly is whether the code signer (software publisher) can be trusted. The usual way to estab- lish trust is to require that the code signer holds a digital certificate and attaches it to the signed code. Verifiers decide whether to trust the certifi- cate based on a trust model (such as PKI or web of trust) or on a case-by- case basis.

Another problem that code signing does not even attempt to solve is whether the signed code is safe to run. As Flame[2] and other code-signed malware have demonstrated, even code that appears to have been signed by a trusted third party might not be safe.

Java Code Signing Java code signing is performed at the JAR file level. It reuses and extends JAR

manifest files in order to add a code signature to the JAR archive. The main JAR manifest file (*MANIFEST.MF*) has entries with the filename and digest value of each file in the archive. For example, Listing 3-2 shows the start of the JAR manifest file of a typical APK file. (We'll use APKs instead of regular JARs for all examples in this section.)

```
Manifest-Version: 1.0 Created-By: 1.0 (Android)

Name: res/drawable-xhdpi/ic_launcher.png SHA1-Digest:
K/0Rd/lt0qSlgDD/9DY7aCNlBvU=
```

2. Microsoft Corporation, *Flame malware collision attack explained*, *http://blogs.technet.com/b/srd/archive/2012/06/06/more-information-about-the-digital-certificates-used-to-sign-the-flame-malware.aspx*