Capitolo 1

Modello di sicurezza di Android

In questo capitolo

- **Architettura di Android**
- **Modello di sicurezza di Android**
- **Riepilogo**

In questo capitolo vengono presentati in breve l'architettura di Android, il meccanismo di comunicazione tra processi (IPC, *Inter-Process Communication*) e i componenti principali. A seguire vengono descritti il modello di sicurezza di Android e la sua relazione con l'infrastruttura di protezione Linux sottostante e con la firma del codice. Il capitolo si conclude con una breve introduzione alle nuove aggiunte al modello di sicurezza di Android, in particolare il supporto multiutente, il controllo d'accesso obbligatorio (MAC, *Mandatory Access Control*) basato su SELinux e il boot verificato. L'architettura e il modello di sicurezza di Android si basano sul tradizionale paradigma Unix di processi, utenti e file, che pertanto eviteremo di spiegare da zero. Presumiamo infatti che i lettori abbiano una certa familiarità con i sistemi Unix, in particolare con Linux.

Architettura di Android

Esaminiamo brevemente l'architettura di Android partendo dalle fondamenta. Nella Figura 1.1 è mostrata una rappresentazione semplificata dello stack di Android.

Kernel di Linux

La Figura 1.1 illustra come Android è costruito sul kernel Linux. Come in qualunque sistema Unix, il kernel fornisce i driver per l'hardware, il networking, l'accesso al file system e la gestione dei processi.

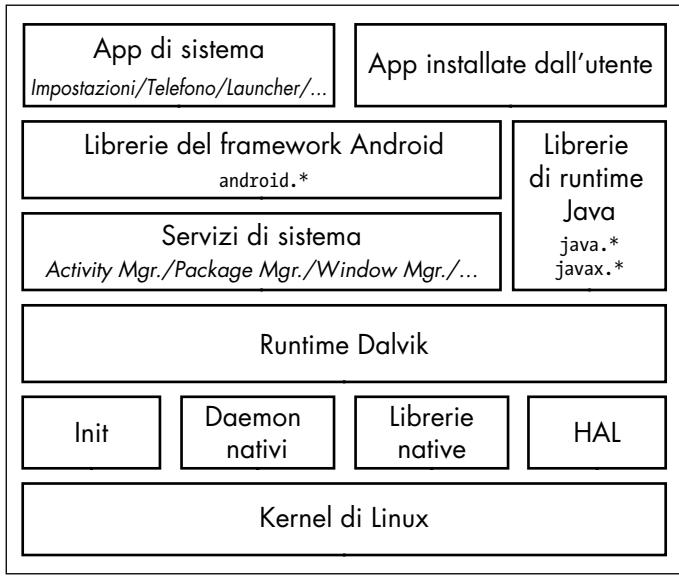

Figura 1.1 L'architettura di Android.

Grazie all'Android Mainlining Project (`http://elinux.org/Android_Mainlining_Project`), con un po' di impegno ora è possibile eseguire Android con un kernel vanilla recente; tuttavia, un kernel Android è leggermente diverso da un kernel Linux "normale" che è possibile trovare su un computer desktop o su un dispositivo integrato non Android. Le differenze sono dovute a un set di nuove funzionalità (a volte chiamate *androidismi*; vedi *Embedded Android* di Karim Yaghmour, O'Reilly, 2013) aggiunto in origine per il supporto di Android. Alcuni dei principali androidismi sono il low memory killer, i wakelock (integrati come parte del supporto delle origini di wakeup nel kernel Linux mainline), la memoria condivisa anonima (ashmem), gli allarmi, il paranoid networking e Binder. Gli androidismi più importanti per la nostra discussione sono Binder e il paranoid networking. Binder implementa IPC e un meccanismo di sicurezza associato, di cui si parla nel dettaglio nel paragrafo "Binder" di questo capitolo. Il paranoid networking limita l'accesso ai socket di rete alle applicazioni che dispongono di autorizzazioni specifiche. L'argomento è approfondito nel Capitolo 2.

Userspace nativo

Sopra il kernel si trova il livello dello userspace nativo, composto dal binario `init` (il primo processo avviato che a sua volta avvia tutti gli altri processi), diversi daemon nativi e qualche centinaio di librerie native utilizzate in tutto il sistema. Anche se la presenza di un binario `init` e di daemon ricorda il sistema Linux tradizionale, va osservato che sia `init` sia gli script di avvio associati sono stati sviluppati da zero e sono piuttosto diversi dalle controparti Linux mainline.

Dalvik VM

Il grosso di Android è implementato in Java e di conseguenza è eseguito da una *Java Virtual Machine* (JVM). L'attuale implementazione della Java VM in Android è chiamata *Dalvik* e corrisponde al livello successivo dello stack. Dalvik è stato progettato pensando ai dispositivi mobili e non può eseguire direttamente il bytecode Java (file `.class`): il suo formato di input nativo è chiamato *Dalvik Executable* (DEX) ed è fornito in package con estensione `.dex`. A loro volta i file `.dex` sono inseriti in package all'interno delle librerie Java di sistema (file JAR) o delle applicazioni Android (file APK, descritti nel Capitolo 3). Le JVM Dalvik e Oracle presentano architetture diverse (basata sul registro in Dalvik e sullo stack in JVM) e set di istruzioni diversi. Ecco un semplice esempio che illustra le differenze tra le due VM (Listato 1.1).

Listato 1.1 Metodo Java statico per la somma di due integer.

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

La compilazione per ogni VM del metodo statico `add()` (che somma due integer e restituisce il risultato dell'operazione) produce il bytecode mostrato nella Figura 1.2.

Bytecode JVM

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

Bytecode Dalvik

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

    add-int v0, p0, p1❺


    return v0❻
.end method
```

Figura 1.2 Bytecode di JVM e di Dalvik.

Qui JVM usa due istruzioni per caricare i parametri nello stack (❶ e ❷), esegue la somma ❸ e infine restituisce il risultato ❹. Dalvik, invece, usa una singola istruzione per sommare i parametri (nei registri `p0` e `p1`), inserisce il risultato nel registro `v0` ❺ e infine restituisce il contenuto del registro `v0` ❻. Come potete notare, Dalvik utilizza un numero inferiore di istruzioni per ottenere lo stesso risultato. In generale, le VM basate sui registri utilizzano meno istruzioni, ma il codice risultante ha dimensioni superiori rispetto al codice corrispondente in una VM basata sullo stack. Tuttavia, nella maggior parte delle architetture il caricamento del codice risulta meno oneroso rispetto al dispatching delle istruzioni, pertanto le VM basate sui registri possono essere interpretate in maniera più efficiente (vedi Yunhe Shi *et al.*, *Virtual Machine Showdown: Stack Versus Registers*, `http://bit.ly/1wLLHxB`).

Nella maggior parte dei dispositivi di produzione le librerie di sistema e le applicazioni preinstallate non contengono direttamente codice DEX indipendente dal dispositivo. Per ottimizzare le prestazioni il codice DEX viene convertito in un formato dipendente dal dispositivo e salvato in un file Optimized DEX (`.odex`), che generalmente risiede nella

stessa directory del file JAR o APK padre. Un processo di ottimizzazione simile viene eseguito in fase di installazione per le applicazioni installate dall'utente.

Librerie di runtime Java

Un'implementazione del linguaggio Java richiede un set di librerie di runtime definite perlopiù nei package `java.*` e `javax.*`. Le librerie Java fondamentali di Android sono state derivate in origine dal progetto Apache Harmony (`http://harmony.apache.org/`) e rappresentano il livello successivo del nostro stack. Con l'evoluzione di Android il codice Harmony originale è cambiato notevolmente. Alcune funzionalità (come il supporto dell'internazionalizzazione, il provider di crittografia e alcune classi correlate) sono state interamente sostituite, altre sono state estese e migliorate. Le librerie fondamentali sono sviluppate principalmente in Java, ma presentano anche alcune dipendenze dal codice nativo. Il codice nativo è collegato alle librerie Java di Android attraverso la *Java Native Interface*, JNI, standard (`http://bit.ly/1rxE750`), che consente al codice Java di chiamare il codice nativo (e viceversa). Il livello delle librerie di runtime Java è direttamente accessibile sia dalle applicazioni sia dai servizi di sistema.

Servizi di sistema

I livelli introdotti finora forniscono i collegamenti necessari per implementare l'elemento fondamentale di Android, ovvero i servizi di sistema. I *servizi di sistema* (79 nella versione 4.4) implementano pressoché tutte le funzionalità base di Android, tra cui il supporto del display e del touch screen, la telefonia e la connettività di rete. La maggior parte dei servizi di sistema è implementata in Java; alcuni di quelli fondamentali sono scritti in codice nativo.
A parte poche eccezioni, ogni servizio di sistema definisce un'interfaccia remota che può essere chiamata da altri servizi e applicazioni. Insieme al service discovery, alla mediation e a IPC, forniti da Binder, i servizi di sistema implementano con efficacia un sistema operativo orientato agli oggetti su Linux.
Vediamo quindi nei dettagli in che modo Binder consente l'uso di IPC su Android, visto che IPC è una delle pietre miliari del modello di sicurezza di Android.

Comunicazione tra processi

Come spiegato in precedenza, Binder è un meccanismo di comunicazione tra processi (IPC, *Inter-Process Communication*). Prima di entrare nei dettagli del funzionamento di Binder è quindi utile riesaminare brevemente IPC.
Come in qualunque sistema Unix, i processi in Android presentano spazi degli indirizzi separati: un processo non può accedere direttamente alla memoria di un altro processo (si parla di *isolamento dei processi*). Solitamente questa scelta è ottima a fini di stabilità e sicurezza: una modifica della stessa memoria da parte di più processi può risultare catastrofica, e certo non vorrete che un altro utente avvii un processo dannoso per scaricare la vostra posta elettronica accedendo alla memoria del vostro client principale. Tuttavia, se un processo vuole offrire servizi utili ad altri processi, deve fornire un meccanismo

che consenta agli altri processi di scoprire e interagire con tali servizi. Questo meccanismo è detto IPC.
L'esigenza di un meccanismo IPC standard non è una novità, e per questo molte soluzioni risalgono a tempi precedenti ad Android. Tra queste soluzioni sono inclusi file, segnali, socket, pipe, semafori, memoria condivisa, code di messaggi e così via. Android ne utilizza alcune (per esempio i socket locali) e non ne supporta altre (nello specifico i meccanismi IPC System V quali semafori, segmenti di memoria condivisa e code di messaggi).

Binder

Dal momento che i meccanismi IPC standard non erano sufficientemente flessibili o affidabili, per Android è stato sviluppato un nuovo meccanismo IPC chiamato *Binder*. Pur essendo una nuova implementazione, Binder di Android è basato sull'architettura e sulle idee di OpenBinder (`http://bit.ly/ZG3BqX`). Binder implementa un'architettura a componenti distribuiti basata su interfacce astratte. È simile al *Common Object Model* (COM) di Windows e alle *Common Object Broker Request Architectures* (COBRA) di Unix, ma a differenza di questi framework viene eseguito su un solo device e non supporta le *Remote Procedure Call* (RPC) sulla rete (sebbene sia possibile implementare il supporto RPC al di sopra di Binder). Una descrizione completa del framework Binder va oltre l'ambito di questo libro; tuttavia, nei paragrafi seguenti saranno presentati brevemente i suoi componenti principali.

Implementazione di Binder

Come affermato in precedenza, su un sistema Unix un processo non può accedere alla memoria di un altro processo. Tuttavia, il kernel ha il controllo su tutti i processi e pertanto può esporre un'interfaccia che abiliti IPC. In Binder questa interfaccia è il device */dev/binder*, implementato dal driver del kernel Binder. Il *driver Binder* è l'oggetto centrale del framework, attraverso cui passano tutte le chiamate IPC. La comunicazione tra processi viene implementata con una singola chiamata `ioctl()` che invia e riceve i dati attraverso la struttura `binder_write_read`, costituita da un `write_buffer` contenente i comandi per il driver e da un `read_buffer` con i comandi necessari per l'esecuzione dello userspace.
A questo punto è probabile che ci si chieda come vengono effettivamente passati i dati tra i processi. Il driver Binder gestisce parte dello spazio degli indirizzi di ogni processo. Il blocco di memoria gestito dal driver Binder è di sola lettura per il processo, e tutta la scrittura è eseguita dal modulo del kernel. Quando un processo invia un messaggio a un altro processo, il kernel assegna spazio nella memoria del processo di destinazione e copia i dati del messaggio direttamente dal processo di invio. Accoda quindi al processo ricevente un breve messaggio che comunica dove si trova il messaggio ricevuto. Il destinatario può così accedere direttamente a tale messaggio, che si trova nel suo spazio di memoria. Quando un processo viene terminato con il messaggio, il driver Binder riceve una notifica per segnare la memoria come disponibile. Nella Figura 1.3 è mostrata un'illustrazione semplificata dell'architettura IPC di Binder.
Le astrazioni IPC di livello più alto in Android, come *Intent* (comandi con dati associati che vengono forniti ai componenti attraverso i processi), *Messenger* (oggetti che abilitano la comunicazione basata su messaggi tra i processi) e *ContentProvider* (componenti che espongono un'interfaccia di gestione dei dati tra processi), sono create al di sopra di Binder.

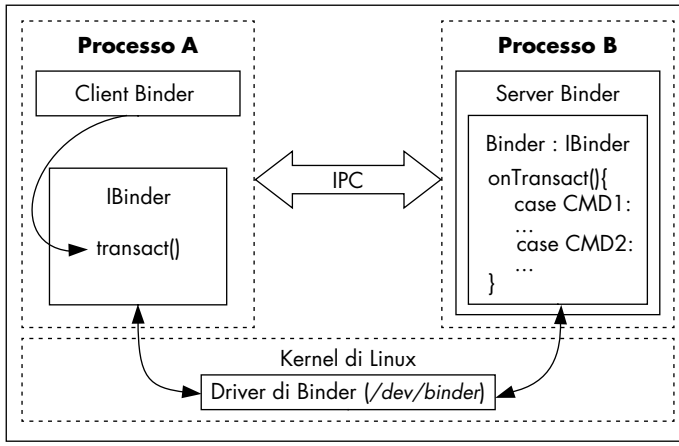

Figura 1.3 Meccanismo IPC di Binder.

Inoltre, le interfacce dei servizi che devono essere esposte ad altri processi possono essere definite utilizzando *Android Interface Definition Language* (AIDL), che permette ai client di chiamare i servizi remoti come se fossero oggetti Java locali. Lo strumento `aidl` associato genera automaticamente *stub* (rappresentazioni lato client dell'oggetto remoto) e *proxy* che associano i metodi di interfaccia al metodo Binder di livello inferiore `transact()` e si occupano della conversione dei parametri in un formato che può essere trasmesso da Binder (in questo caso si parla di *marshalling/unmarshalling dei parametri*). Binder è per natura typeless, pertanto stub e proxy generati da AIDL forniscono anche l'indipendenza dai tipi includendo il nome dell'interfaccia target in ogni transazione di Binder (nel proxy) e convalidandolo nello stub.

Sicurezza di Binder

A un livello più alto, ogni oggetto a cui si può accedere attraverso il framework Binder implementa l'interfaccia `IBinder` ed è chiamato *oggetto Binder*. Le chiamate a un oggetto Binder vengono eseguite all'interno di una *transazione Binder*, che contiene un riferimento all'oggetto target, l'ID del metodo da eseguire e un buffer dei dati. Il driver Binder aggiunge automaticamente ai dati della transazione il *process ID* (PID) e l'*effective user ID* (EUID) del processo chiamante. Il processo chiamato (*callee*) può esaminare il PID e l'EUID e stabilire se eseguire il metodo richiesto in base alla logica interna o ai metadati a livello di sistema in relazione all'applicazione chiamante.
Il PID e l'EUID sono forniti dal kernel, pertanto i processi chiamanti non possono falsificare la loro identità per ottenere più privilegi rispetto a quanto concesso dal sistema (in pratica, Binder previene l'*escalation dei privilegi*). Questo è uno degli elementi centrali del modello di sicurezza di Android, su cui si basano tutte le astrazioni di livello superiore, quali i permessi. L'EUID e il PID del chiamante sono accessibili con i metodi `getCallingPid()` e `getCallingUid()` della classe `android.os.Binder`, che fa parte dell'API pubblica di Android.

NOTA

L'EUID del processo chiamante potrebbe non essere associato a una singola applicazione se sono in esecuzione più applicazioni con lo stesso UID (vedi il Capitolo 2 per i dettagli).

Tuttavia, questo non influenza le decisioni relative alla sicurezza, perché ai processi in esecuzione con lo stesso UID viene generalmente concesso lo stesso set di permessi e privilegi (tranne qualora siano definite regole SELinux specifiche per il processo).

Identità Binder

Una delle proprietà più importanti degli oggetti Binder è la capacità di mantenere un'identità univoca tra i processi. Se il processo A crea un oggetto Binder e lo passa al processo B, che a sua volta lo passa al processo C, le chiamate da tutti i tre processi saranno elaborate dallo stesso oggetto Binder. In pratica, il processo A farà riferimento all'oggetto Binder direttamente con il suo indirizzo di memoria (perché si trova nello spazio di memoria del processo A), mentre i processi B e C riceveranno solamente un handle all'oggetto Binder.
Il kernel mantiene l'associazione tra gli oggetti Binder "live" e i loro handle negli altri processi. Visto che l'identità di un oggetto Binder è univoca e gestita dal kernel, è impossibile che i processi dello userspace creino una copia di un oggetto Binder oppure ottengano un riferimento a un oggetto, a meno che non ne sia stato passato loro uno tramite IPC. Per questo motivo un oggetto Binder è un oggetto univoco, non falsificabile e comunicabile che può agire come *token* di protezione, e per questo consente l'uso della sicurezza basata sulle capability in Android.

Sicurezza basata sulle capability

In un *modello di sicurezza basata sulle capability*, i programmi possono accedere a una particolare risorsa quando viene concessa loro una *capability* non falsificabile che fa riferimento all'oggetto target e vi incapsula un set di diritti di accesso. Visto che le capability non sono falsificabili, il solo fatto che un programma ne possieda una è sufficiente a consentirgli l'accesso alla risorsa target; non è necessario mantenere liste di controllo degli accessi (ACL) o strutture simili associate alle risorse vere e proprie.

Token Binder

In Android gli oggetti Binder possono agire come capability e sono detti *token Binder* quando sono utilizzati in questo modo. Un token Binder può essere sia una capability sia una risorsa target. Il possesso di un token Binder garantisce al processo proprietario l'accesso completo a un oggetto Binder e la possibilità di eseguire transazioni Binder su tale oggetto target. Se l'oggetto Binder implementa più azioni (scegliendo l'azione da eseguire in base al parametro `code` della transazione Binder), il chiamante può eseguire qualunque azione nel momento in cui dispone di un riferimento a tale oggetto Binder. Qualora sia richiesto un controllo di accesso più granulare, l'implementazione di ogni azione deve applicare i controlli necessari sui permessi, tipicamente utilizzando il PID e l'EUID del processo chiamante.
Uno schema comune in Android prevede di consentire tutte le azioni ai chiamanti in esecuzione come *system* (UID 1000) o *root* (UID 0), ma di eseguire ulteriori controlli sui permessi per tutti gli altri processi. Di conseguenza, l'accesso a oggetti Binder importanti come i servizi di sistema è controllato in due modi: limitando chi può ottenere un riferimento a tale oggetto Binder e verificando l'identità del chiamante prima di eseguire un'azione sull'oggetto Binder (questo controllo è facoltativo e implementato dall'oggetto Binder stesso, se richiesto).

In alternativa, un oggetto Binder può essere utilizzato solo come capability, senza implementare altre funzionalità. In questo modello di utilizzo, lo stesso oggetto Binder è detenuto da due o più processi che collaborano; quello che agisce come server (elaborando qualche tipo di richiesta client) utilizza il token Binder per autenticare i suoi client (in modo analogo ai server web che utilizzano i cookie di sessione).
Questo schema è utilizzato internamente dal framework Android ed è pressoché invisibile alle applicazioni. Un caso di utilizzo importante dei token Binder, visibile nell'API pubblica, è quello dei *window token*. La finestra di primo livello di ogni activity è associata a un token Binder (chiamato window token), di cui viene tenuta traccia dal window manager di Android (il servizio di sistema responsabile della gestione delle finestre delle applicazioni). Le applicazioni possono ottenere un proprio window token, ma non possono accedere ai window token di altre applicazioni. In genere è preferibile che altre applicazioni non possano aggiungere o rimuovere finestre sopra la propria; ogni richiesta in tal senso deve fornire il window token associato all'applicazione, garantendo così che le richieste di finestre provengano dalla propria applicazione o dal sistema.

Accesso agli oggetti Binder

Nonostante Android controlli l'accesso agli oggetti Binder per questioni di sicurezza, e che l'unico modo per comunicare con un oggetto Binder sia con un riferimento allo stesso, alcuni oggetti Binder (nello specifico i servizi di sistema) devono essere universalmente accessibili. È tuttavia poco pratico trasferire i riferimenti a tutti i servizi di sistema a ogni processo, pertanto occorre un meccanismo che consenta ai processi di individuare e ottenere riferimenti ai servizi di sistema in base alle necessità.
Per abilitare l'individuazione dei servizi, il framework Binder dispone di un singolo *context manager* che mantiene i riferimenti agli oggetti Binder. L'implementazione del context manager di Android è il daemon nativo *servicemanager*, avviato nelle primissime fasi del processo di boot affinché i servizi di sistema possano registrarsi durante l'avvio. I servizi vengono registrati passando al service manager il nome del servizio e un riferimento Binder. Dopo la registrazione di un servizio i client possono ottenerne il riferimento Binder utilizzando il relativo nome. Tuttavia, la maggior parte dei servizi di sistema implementa controlli supplementari delle autorizzazioni, quindi il recupero di un riferimento non garantisce automaticamente l'accesso a tutte le funzionalità del servizio. Visto che chiunque può accedere a un riferimento Binder registrato con il service manager, solo un piccolo gruppo di processi di sistema nella whitelist può registrare i servizi di sistema. Per esempio, solo un processo in esecuzione con UID 1002 (`AID_BLUETOOTH`) può registrare il servizio di sistema *bluetooth*.
È possibile vedere un elenco dei servizi registrati utilizzando il comando `service list`, che restituisce il nome di ogni servizio registrato e l'interfaccia `IBinder` implementata. Un output di esempio ottenuto con l'esecuzione del comando su un dispositivo Android 4.4 è disponibile nel Listato 1.2.

Listato 1.2 Recupero di un elenco di servizi di sistema registrati con il comando service list.

```
$ 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]
--altro codice--
```

Altre funzionalità di Binder

Per quanto non siano direttamente correlate al modello di sicurezza di Android, esistono altre due importanti funzionalità di Binder chiamate reference counting e death notification (o link to death). Il *reference counting* (o conteggio dei riferimenti) garantisce che gli oggetti Binder siano liberati automaticamente quando nessuno vi fa riferimento ed è implementato nel driver del kernel con i comandi `BC_INCREFS`, `BC_ACQUIRE`, `BC_RELEASE` e `BC_DECREFS`. Il reference counting è integrato in vari livelli del framework Android ma non è direttamente visibile alle applicazioni.
La *death notification* permette alle applicazioni che usano oggetti Binder ospitati da altri processi di ricevere una notifica qualora questi processi vengano rimossi dal kernel e di eseguire la pulizia necessaria. La death notification è implementata con i comandi `BC_REQUEST_DEATH_NOTIFICATION` e `BC_CLEAR_DEATH_NOTIFICATION` nel driver del kernel e con i metodi `linkToDeath()` e `unlinkToDeath()` dell'interfaccia `IBinder` (`http://bit.ly/1pfJiaa`) nel framework. Le death notification per i Binder locali non vengono inviate, perché questi Binder non possono essere terminati senza che venga terminato anche il processo di hosting.

Librerie del framework Android

Nella successiva posizione dello stack si trovano le librerie del framework Android, a volte definite semplicemente "il framework". Il framework include tutte le librerie Java che non sono parte del runtime Java standard (`java.*`, `javax.*` e così via) ed è per la maggior parte ospitato nei package `android` di primo livello. Il framework contiene i blocchi fondamentali per la costruzione di applicazioni Android, per esempio le classi di base per activity, servizi e content provider (nei package `android.app.*`), i widget GUI (nei package `android.view.*` e `android.widget`) e le classi per l'accesso a file e database (per la maggior parte nei package `android.database.*` e `android.content.*`). Include inoltre le classi che permettono di interagire con l'hardware del dispositivo, nonché le classi che sfruttano i servizi di alto livello offerti dal sistema.
Sebbene quasi tutte le funzionalità del sistema operativo Android poste sopra il kernel siano implementate come servizi di sistema, queste non vengono esposte direttamente nel framework, ma sono accessibili tramite classi di facciata definite *manager*. Generalmente, ogni manager è sostenuto da un servizio di sistema corrispondente: per esempio, `BluetoothManager` è un manager di facciata per `BluetoothManagerService`.

Applicazioni

Al livello più alto dello stack si trovano le *applicazioni*, o *app*, vale a dire i programmi con cui l'utente interagisce in maniera diretta. Sebbene tutte le app abbiano la stessa

struttura e siano create sul framework Android, occorre distinguere tra app di sistema e app installate dall'utente.

App di sistema

Le app di sistema sono incluse nell'immagine del sistema operativo, che è di sola lettura sui dispositivi di produzione (generalmente montata come */system*), e non possono essere disinstallate o modificate dagli utenti. Per questo motivo sono considerate sicure e ricevono molti più privilegi delle app installate dall'utente. Le app di sistema possono essere parte del sistema core, oppure possono essere applicazioni utente preinstallate come client e-mail o browser. Anche se tutte le app installate in */system* erano trattate allo stesso modo nelle precedenti versioni di Android (fatta eccezione per le funzionalità del sistema operativo che verificano il certificato di firma dell'app), Android 4.4 e versioni successive trattano le app installate in */system/priv-app/* come applicazioni privilegiate; i permessi con livello di protezione *signatureOrSystem* vengono concessi solo alle app privilegiate, non a tutte le app installate in */system*. Le app firmate con il codice di firma della piattaforma possono ottenere permessi di sistema con il livello di protezione *signature*, e di conseguenza possono ricevere privilegi a livello di sistema operativo anche se non sono preinstallate in */system*. Consultate il Capitolo 2 per i dettagli sui permessi e sulla firma del codice.
Per quanto le app di sistema non possano essere disinstallate o modificate, possono essere aggiornate dagli utenti (purché gli aggiornamenti siano firmati con la stessa chiave privata) e alcune possono essere sostituite da app installate dall'utente. Per esempio, un utente può scegliere di sostituire il launcher o il metodo di input di un'applicazione preinstallata con un'applicazione di terze parti.

App installate dall'utente

Le app installate dall'utente vengono configurate in una partizione di lettura/scrittura dedicata (generalmente montata come */data*) che ospita i dati utente e possono essere disinstallate a piacere. Ogni applicazione risiede in una sandbox di sicurezza dedicata e in genere non influenza le altre applicazioni né accede ai loro dati. Inoltre, le app possono accedere unicamente alle risorse per cui hanno ottenuto un'esplicita autorizzazione all'uso. La separazione dei privilegi e il principio detto del *least privilege* sono fondamentali per il modello di sicurezza di Android; la loro implementazione è descritta nel paragrafo successivo.

Componenti delle app Android

Le applicazioni Android sono una combinazione di *componenti loosely coupled* e, a differenza delle applicazioni tradizionali, possono disporre di più punti di ingresso. Ogni componente può offrire molteplici punti di ingresso raggiungibili in base alle azioni dell'utente nella stessa o in un'altra applicazione; tali punti di ingresso possono inoltre essere attivati da un evento di sistema per cui l'applicazione ha chiesto di ricevere notifiche.
I componenti e i loro punti di ingresso, insieme ai metadati supplementari, sono definiti nel file manifest dell'applicazione, chiamato `AndroidManifest.xml`. Come la maggior parte dei file di risorse Android, questo file è compilato in un formato XML binario (simile ad ASN.1) prima dell'inserimento nel file APK (package dell'applicazione) al fine di ridurre le dimensioni e accelerare il parsing. La più importante proprietà delle applicazioni

definita nel file manifest è il nome del package dell'applicazione, che identifica in modo univoco ogni applicazione nel sistema. Il nome del package ha lo stesso formato dei nomi dei package Java (notazione a nome di dominio inverso, per esempio `com.google.email`). Il file `AndroidManifest.xml` viene sottoposto a parsing in fase di installazione dell'applicazione, quando il package e i componenti definiti vengono registrati nel sistema. Android richiede che ogni applicazione sia firmata con una chiave controllata dal suo sviluppatore: questo garantisce che un'applicazione installata non possa essere sostituita da un'altra applicazione che utilizza lo stesso nome di package (a meno che non sia firmata con la stessa chiave, caso in cui l'applicazione esistente viene aggiornata). La firma del codice e i package delle applicazioni sono spiegati nel Capitolo 3.
Di seguito sono elencati i componenti principali delle app Android.

Activity
Un'*activity* è una singola schermata con un'interfaccia utente. Le activity sono i blocchi fondamentali utilizzati per creare le applicazioni GUI di Android; un'applicazione può disporre di più activity. Sebbene generalmente siano progettate per la visualizzazione in un ordine specifico, le activity possono essere avviate in maniera indipendente, volendo anche da un'app diversa (se consentito).

Servizi
Un *servizio* è un componente che viene eseguito in background e non dispone di un'interfaccia utente. I servizi sono normalmente utilizzati per eseguire operazioni di lunga durata, come il download di un file o la riproduzione di musica, senza bloccare l'interfaccia utente. I servizi possono inoltre definire un'interfaccia remota utilizzando AIDL e fornire alcune funzionalità alle altre app. Tuttavia, a differenza dei servizi di sistema che sono parte del sistema operativo e sono sempre in esecuzione, i servizi delle applicazioni vengono avviati e arrestati su richiesta.

Content provider
I content provider offrono un'interfaccia ai dati delle app, generalmente archiviati in un database o in più file. I content provider, a cui si può accedere tramite IPC, sono utilizzati principalmente per condividere i dati di un'app con altre app. I content provider offrono un controllo preciso sulle parti dei dati accessibili, permettendo a un'applicazione di condividere solo un sottoinsieme dei suoi dati.

Broadcast receiver
Un *broadcast receiver* è un componente che risponde a eventi a livello di sistema chiamati *broadcast*. I broadcast possono essere creati dal sistema (che per esempio annuncia cambiamenti a livello di connettività di rete) o da un'applicazione utente (che segnala per esempio il completamento dell'aggiornamento in background dei dati).

Modello di sicurezza di Android

Analogamente al resto del sistema, anche il modello di sicurezza di Android sfrutta i vantaggi delle funzionalità di protezione offerte dal kernel Linux. Linux è un sistema operativo multiutente e il suo kernel può isolare le risorse di un utente da quelle di un

altro, proprio come isola i processi. In un sistema Linux un utente non può accedere ai file di un altro utente (salvo dietro concessione di autorizzazioni esplicite) e ogni processo viene eseguito con l'identità (*user ID* e *group ID*, generalmente chiamati UID e GID) dell'utente che lo ha avviato, a meno che per il file eseguibile corrispondente non siano impostati i bit *set-user-ID* o *set-group-ID* (SUID e SGID).
Android sfrutta questo isolamento degli utenti, ma tratta gli utenti in maniera diversa rispetto a un tradizionale sistema Linux (desktop o server). In un sistema tradizionale, viene assegnato un UID a ogni utente fisico che può accedere al sistema ed eseguire comandi dalla shell o a un servizio di sistema (daemon) che viene eseguito in background (perché i daemon di sistema sono spesso accessibili in rete; l'esecuzione di ogni daemon con un UID dedicato può limitare i danni in caso di compromissione di un daemon). Android in origine è stato progettato per gli smartphone e, visto che i telefoni cellulari sono dispositivi personali, non era necessario registrare utenti fisici diversi sul sistema. L'utente fisico è implicito e gli UID sono pertanto usati per distinguere le applicazioni. Questo metodo forma le basi del sandboxing delle applicazioni di Android.

Sandboxing delle applicazioni

In fase di installazione Android assegna automaticamente a ogni applicazione un UID univoco, spesso definito *app ID*, ed esegue tale applicazione in un processo dedicato in esecuzione con tale UID. Inoltre, a ogni applicazione viene assegnata una directory dati dedicata in cui può leggere e scrivere solo l'applicazione specifica. Le applicazioni sono quindi isolate, o *in sandbox*, sia a livello di processo (ognuna viene eseguita in un processo dedicato) sia a livello di file (ognuna ha una directory dati privata). Si crea così una sandbox delle applicazioni a livello di kernel, che si applica a tutte le applicazioni indipendentemente dalla modalità di esecuzione (processo nativo o di macchina virtuale). Le applicazioni e i daemon di sistema vengono eseguiti con UID ben definiti e costanti, e ben pochi daemon sono eseguiti con l'utente root (UID 0). Android non dispone del tradizionale file */etc/password* e i suoi UID di sistema sono definiti in maniera statica nel file header `android_filesystem_config.h`. Gli UID per i servizi di sistema partono da 1000; 1000 corrisponde all'utente `system` (`AID_SYSTEM`) che dispone di privilegi speciali (ma pur sempre limitati). Gli UID generati automaticamente per le applicazioni partono da 10000 (`AID_APP`) e i nomi utente corrispondenti sono nella forma `app_XXX` o `uY_aXXX` (nelle versioni di Android che supportano più utenti fisici), dove `XXX` è l'offset rispetto ad `AID_APP` e `Y` è lo user ID Android (che non corrisponde all'UID). Per esempio, l'UID 10037 corrisponde al nome utente `u0_a37` e può essere assegnato all'applicazione client e-mail Google (package `com.google.android.email`). Il Listato 1.3 mostra che il processo dell'applicazione e-mail viene eseguito con l'utente `u0_a37` ❶, mentre gli altri processi applicativi vengono eseguiti con altri utenti.

Listato 1.3 Ogni processo applicativo viene eseguito con un utente dedicato su Android.

```
$ ps
--altro codice--
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
--altro codice--
```

La directory dati dell'applicazione e-mail prende il nome dal suo package e viene creata in */data/data/* sui dispositivi monoutente. I dispositivi multiutente usano uno schema di denominazione diverso, descritto nel Capitolo 4. Tutti i file nella directory dati sono di proprietà dell'utente Linux dedicato, `u0_a37`, come mostrato nel Listato 1.4 (in cui sono stati omessi i timestamp). Facoltativamente, le applicazioni possono creare file utilizzando i flag `MODE_WORLD_READABLE` e `MODE_WORLD_WRITEABLE`, che consentono l'accesso diretto ai file da parte delle altre applicazioni e che impostano rispettivamente i bit di accesso `S_IROTH` e `S_IWOTH` sul file. Tuttavia, la condivisione diretta dei file è sconsigliata e questi flag sono deprecati in Android versioni 4.2 e successive.

Listato 1.4 Le directory delle applicazioni sono di proprietà dell'utente Linux dedicato.

```
# 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
--altro codice--
```

Gli UID delle applicazioni sono gestiti insieme agli altri metadati del package nel file `/data/system/packages.xml` (l'origine canonica) e sono scritti anche nel file `/data/system/packages.list`. La gestione dei package e il file `packages.xml` sono presentati nel Capitolo 3. Il Listato 1.5 mostra l'UID assegnato al package `com.google.android.email` come appare in `packages.list`.

Listato 1.5 L'UID corrispondente a ogni applicazione è memorizzato in /data/system/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
```

Qui il primo campo è il nome del package, il secondo è l'UID assegnato all'applicazione, il terzo è il flag di debug (`1` se si può eseguire il debug), il quarto è il percorso della directory dati dell'applicazione e il quinto è l'etichetta `seinfo` (usata da SELinux). L'ultimo campo è un elenco di GID supplementari con cui viene avviata l'app. Ogni GID è tipicamente associato a un permesso Android (argomento affrontato in seguito) e l'elenco dei GID viene generato in base ai permessi concessi all'applicazione.
Le applicazioni possono essere installate utilizzando lo stesso UID, definito *user ID condiviso*, e in questo caso possono condividere i file e persino essere eseguite nello stesso processo. Gli user ID condivisi sono utilizzati in maniera estesa dalle applicazioni di sistema, che spesso necessitano di utilizzare le stesse risorse tra package diversi per ragioni di modularità. Per esempio, in Android 4.4 l'interfaccia di sistema e il keyguard (implementazione del blocco dello schermo) condividono l'UID 10012 (Listato 1.6).

Listato 1.6 Package di sistema che condividono lo stesso UID.

```
# 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
```

Anche se la struttura di user ID condivisi non è consigliata per le app non di sistema, è disponibile anche per le applicazioni di terze parti. Per condividere lo stesso UID le applicazioni devono essere firmate con la stessa chiave di firma del codice. Inoltre, poiché l'aggiunta di un nuovo user ID condiviso a una nuova versione di un'app installata provoca una modifica del suo UID, il sistema vieta questa operazione (consultate il Capitolo 2). Di conseguenza, uno user ID condiviso non può essere aggiunto in maniera retroattiva e le app devono essere progettate per funzionare con un ID condiviso sin dall'inizio.

Permessi

Visto che le applicazioni Android sono in sandbox, possono accedere unicamente ai propri file e a qualsiasi risorsa accessibile in maniera globale sul dispositivo. Un'applicazione così limitata non sarebbe tuttavia molto interessante: per questo Android può concedere alle applicazioni altri diritti di accesso specifici per consentire funzionalità superiori. Questi diritti di accesso sono chiamati *permessi* (o *autorizzazioni*) e possono controllare l'accesso a dispositivi hardware, connettività Internet, dati e servizi del sistema operativo.
Le applicazioni possono richiedere i permessi definendoli nel file `AndroidManifest.xml`. In fase di installazione dell'applicazione, Android esamina l'elenco di autorizzazioni richieste e decide se concederle o meno. Dopo la concessione le autorizzazioni non possono essere revocate e sono disponibili all'applicazione senza necessità di ulteriore conferma. Inoltre, per le funzionalità come la chiave privata o l'accesso all'account utente, è necessaria una conferma esplicita dell'utente per ogni oggetto, anche se all'applicazione richiedente è stato concesso il permesso corrispondente (leggete i Capitoli 7 e 8). Alcune autorizzazioni possono essere concesse solo alle applicazioni che sono parte del sistema operativo Android, sia perché sono preinstallate sia perché sono firmate con la stessa chiave del sistema operativo. Le applicazioni di terze parti possono definire permessi personalizzati e restrizioni simili chiamate *livelli di protezione dei permessi*, in grado di limitare l'accesso a servizi e risorse di un'app alle app create dallo stesso autore.
I permessi possono essere applicati a livelli diversi. Le richieste alle risorse di sistema di livello inferiore, quali i file del dispositivo, sono gestite dal kernel di Linux confrontando l'UID o il GID del processo chiamante con quello del proprietario della risorsa e con i bit di accesso. Per l'accesso ai componenti Android di livello superiore, la gestione viene eseguita sia dal sistema operativo Android sia da ogni componente. I permessi sono affrontati nel Capitolo 2.

IPC

Android usa una combinazione di driver del kernel e librerie dello userspace per implementare IPC. Come spiegato nel paragrafo "Binder" di questo capitolo, il driver del kernel Binder garantisce che l'UID e il PID dei chiamanti non possano essere falsificati; molti servizi di sistema fanno affidamento su UID e PID forniti da Binder per controllare dinamicamente l'accesso alle API sensibili esposte tramite IPC. Per esempio, grazie al codice mostrato nel Listato 1.7, il servizio di sistema Bluetooth Manager consente alle applicazioni di sistema di eseguire Bluetooth senza interventi manuali se il chiamante è in esecuzione con l'UID `system` (1000). Un codice simile è presente negli altri servizi di sistema.

Listato 1.7 Verifica che il chiamante sia in esecuzione con l'UID system.

```
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;
     }
--altro codice--
}
```

Permessi meno dettagliati, che interessano tutti i metodi di un servizio esposto tramite IPC, possono essere applicati automaticamente dal sistema specificandoli nella dichiarazione del servizio. Come i permessi richiesti, quelli obbligatori sono dichiarati nel file `AndroidManifest.xml`. Analogamente al controllo dinamico mostrato nell'esempio sopra, i permessi per componente sono implementati consultando l'UID del chiamante ottenuto da Binder dietro le quinte. Il sistema usa il database dei package per determinare l'autorizzazione richiesta dal componente chiamato, quindi associa l'UID del chiamante al nome del package e recupera il set di permessi concessi al chiamante. Se l'autorizzazione richiesta è presente nel set, la chiamata ha esito positivo. In caso contrario, la chiamata non riesce e viene generata una `SecurityException`.

Firma del codice e chiavi della piattaforma

Tutte le applicazioni Android devono essere firmate dal loro sviluppatore, comprese le applicazioni di sistema. Visto che i file APK di Android sono un'estensione del formato di package Java JAR (`http://bit.ly/11rmJtR`), anche il metodo usato per la firma del codice è basato sulla firma JAR. Android utilizza la firma APK per garantire che gli aggiornamenti di un'app provengano dallo stesso autore (in questo caso di sparla di *criterio della stessa origine*) e per stabilire relazioni di fiducia tra le applicazioni. Entrambe le funzionalità di protezione vengono implementate confrontando il certificato di firma dell'app attualmente installata con il certificato dell'aggiornamento o dell'applicazione correlata.
Le applicazioni di sistema sono firmate da diverse *chiavi della piattaforma*. Componenti di sistema diversi possono condividere le risorse ed essere eseguiti nello stesso processo se sono firmati con la medesima chiave della piattaforma. Le chiavi della piattaforma vengono generate e controllate da chi mantiene la versione di Android installata su un particolare dispositivo, vale a dire produttori di dispositivo, gestori telefonici, Google per i device Nexus o gli utenti delle versioni di Android open source realizzate autonomamente. La firma del codice e il formato APK sono descritti nel Capitolo 3.

Supporto multiutente

Android in origine è stato progettato per gli smartphone, associati a un unico utente fisico; per questo, assegna un UID Linux distinto a ogni applicazione installata e per tradizione non usa la nozione di utente fisico. Android ha ottenuto il supporto per più utenti fisici nella versione 4.2, ma il supporto multiutente è disponibile esclusivamente sui tablet, che è più facile vengano condivisi. Il supporto multiutente sui dispositivi mobili può essere disabilitato impostando il numero massimo di utenti a 1.

A ogni utente viene assegnato uno user ID univoco, partendo da 0, e gli utenti ricevono una propria directory dati dedicata in */data/system/users/<user ID>/*: questa è definita *directory di sistema* dell'utente. La directory ospita impostazioni specifiche per l'utente quali parametri della schermata iniziale, dati dell'account e un elenco delle applicazioni attualmente installate. Se i binari delle applicazioni sono condivisi tra gli utenti, ogni utente riceve una copia della directory dati di un'applicazione.
Per distinguere le applicazioni installate per ogni utente, Android assegna un nuovo effective UID a ogni applicazione nella fase di installazione per un utente specifico. Questo effective UID è basato sullo user ID dell'utente fisico di destinazione e sull'UID dell'app in un sistema monoutente (l'*app ID*). Questa struttura composita dell'UID concesso garantisce che, anche qualora la stessa applicazione venga installata da due utenti diversi, entrambe le istanze dell'applicazione ricevano la loro sandbox. Inoltre, Android garantisce a ogni utente uno spazio di archiviazione condiviso dedicato (sulla scheda SD per i dispositivi meno recenti), leggibile da tutti. L'utente che per primo inizializza il dispositivo viene definito come *proprietario del dispositivo* ed è il solo a poter gestire gli altri utenti o eseguire attività amministrative che interessano l'intero dispositivo (come il ripristino alle impostazioni di fabbrica). Il supporto multiutente è descritto nei dettagli nel Capitolo 4.

SELinux

Il tradizionale modello di sicurezza di Android si affida agli UID e ai GID concessi alle applicazioni. Per quanto questi siano garantiti dal kernel, e considerando che per impostazione predefinita i file di ogni applicazione sono privati, nulla impedisce a un'applicazione di concedere l'accesso illimitato ai suoi file (intenzionalmente o a causa di un errore di programmazione).
Analogamente, nulla vieta alle applicazioni dannose di sfruttare i bit di accesso eccessivamente permissivi dei file di sistema o dei socket locali. In effetti, l'assegnazione di permessi inappropriati ai file di sistema o delle applicazioni è stata la causa di numerose vulnerabilità di Android. Queste vulnerabilità non possono essere evitate nel modello di controllo di accesso predefinito utilizzato da Linux, noto come *Discretionary Access Control* (DAC). La parola *discretionary* segnala che, una volta che l'utente ha ottenuto l'accesso a una particolare risorsa, può trasferirlo a sua discrezione a un altro utente, per esempio impostando la modalità di accesso di uno dei file sulla leggibilità globale. Al contrario, il modello *Mandatory Access Control* (MAC) garantisce che l'accesso alle risorse sia conforme a un set di *regole di autorizzazione*, definite *policy*, esteso a livello di sistema. La policy può essere modificata solamente da un amministratore; gli utenti non possono sostituirla o ignorarla, per esempio per concedere l'accesso illimitato ai propri file.
Security Enhanced Linux (SELinux) è un'implementazione di MAC per il kernel Linux che è stata integrata nel kernel mainline per oltre dieci anni. A partire dalla versione 4.3 Android dispone di una versione di SELinux modificata dal progetto *Security Enhancements for Android* (SEAndroid, `http://seandroid.bitbucket.org/`), migliorata per supportare le funzionalità specifiche per Android come Binder. In Android, SELinux è usato per isolare i daemon del sistema core e le applicazioni utente in diversi *domini* di protezione e per definire policy di accesso diverse per ogni dominio. A partire dalla versione 4.4 SELinux è distribuito nella *modalità di enforcing* (le violazioni alle policy di sistema generano errori di runtime), ma l'enforcing delle policy avviene solo nei daemon del sistema core. Le

applicazioni vengono tuttora eseguite nella *modalità permissive* e le violazioni vengono registrate senza causare errori di runtime. Maggiori dettagli sull'implementazione SE-Linux di Android sono disponibili nel Capitolo 12.

Aggiornamenti del sistema

I dispositivi Android possono essere aggiornati *over-the-air* (OTA) o collegando il device a un PC e inviando l'immagine dell'aggiornamento utilizzando il client *Android Debug Bridge* (ADB) standard o un'applicazione fornita da altri produttori con funzionalità simili. Oltre ai file di sistema, un aggiornamento di Android potrebbe dover modificare il firmware baseband (modem), il bootloader e altre parti del dispositivo non direttamente accessibili da Android; per questo di solito il processo di aggiornamento utilizza un sistema operativo minimo e specializzato con accesso esclusivo a tutto l'hardware del dispositivo, che è detto *sistema operativo di recovery* o semplicemente *recovery*.
Gli aggiornamenti OTA vengono eseguiti scaricando un package OTA (in genere un file ZIP con una firma del codice), che contiene un piccolo file di script interpretabile dal recovery, e riavviando il dispositivo nella *modalità di recovery*. In alternativa, l'utente può accedere alla modalità di recovery utilizzando una combinazione di tasti specifica del dispositivo durante l'avvio dello stesso e applicare manualmente l'aggiornamento utilizzando l'interfaccia di menu del recovery, generalmente ricorrendo ai tasti fisici (volume, accensione e così via) del device.
Nei dispositivi di produzione il recovery accetta unicamente gli aggiornamenti firmati dal produttore. I file di aggiornamento vengono firmati estendendo il formato di file ZIP per includere una firma dell'intero file nella sezione dei commenti (vedere il Capitolo 3), che il recovery estrae e verifica prima di installare l'aggiornamento. Su alcuni dispositivi (compresi tutti i Nexus, i dispositivi per sviluppatori dedicati e i dispositivi di alcuni produttori), i proprietari dei dispositivi possono sostituire il sistema operativo di recovery e disabilitare la verifica della firma per gli aggiornamenti di sistema, consentendo l'installazione di aggiornamenti di terzi. Il passaggio del bootloader del device a una modalità che consente la sostituzione del sistema operativo di recovery e delle immagini di sistema è detto *sblocco del bootloader* (da non confondersi con lo sblocco della SIM, che consente di utilizzare un dispositivo su qualunque rete mobile), e di solito richiede la cancellazione di tutti i dati utente (ripristino delle impostazioni di fabbrica) per garantire che un'immagine di sistema di terze parti potenzialmente dannosa non possa accedere ai dati utente esistenti. Sulla maggior parte dei dispositivi consumer, lo sblocco del bootloader ha l'effetto collaterale di invalidare la garanzia del dispositivo. Gli aggiornamenti del sistema e le immagini di recovery sono trattati nel Capitolo 13.

Boot verificato

A partire dalla versione 4.4 Android supporta il boot verificato tramite il target *verity* (`http://bit.ly/1CoIXIf`) del Device-Mapper di Linux.Verity offre un controllo trasparente dell'integrità dei dispositivi di blocco utilizzando un albero di hashtree crittografici. Ogni nodo dell'albero è un hash crittografico, con i nodi foglia che contengono il valore hash di un blocco dati fisico e i nodi intermediari che contengono i valori hash dei loro nodi figlio.Visto che l'hash nel nodo radice è basato sui valori di tutti gli altri nodi, è necessario che sia ritenuto attendibile l'hash radice per verificare il resto dell'albero.

La verifica viene eseguita con una chiave pubblica RSA inclusa nella partizione di boot. I blocchi del dispositivo vengono verificati in fase di esecuzione calcolando il valore hash del blocco letto e confrontandolo con il valore registrato nell'albero degli hash. Se i valori non corrispondono, l'operazione di lettura provoca un errore di I/O che indica che il file system è danneggiato. Tutti i controlli vengono eseguiti dal kernel, pertanto il processo di boot deve verificare l'integrità del kernel affinché il boot verificato funzioni. Questo processo è specifico per il dispositivo e normalmente viene implementato con una chiave invariabile specifica per l'hardware che viene scritta nella memoria di sola scrittura del dispositivo. La chiave è utilizzata per verificare l'integrità di ogni livello del bootloader e alla fine del kernel. Il boot verificato è descritto nel Capitolo 10.

Riepilogo

Android è un sistema operativo separato da privilegi basato sul kernel di Linux. Le funzioni di sistema di livello più alto sono implementate come set di servizi di sistema cooperanti che comunicano mediante un meccanismo IPC chiamato Binder. Android isola tra loro le applicazioni eseguendole con un'identità di sistema distinta (UID di Linux). Per impostazione predefinita le applicazioni ricevono pochissimi privilegi, pertanto devono richiedere permessi specifici per interagire con i servizi di sistema, i dispositivi hardware e le altre applicazioni. I permessi sono definiti nel file manifest di ogni applicazione e sono concessi in fase di installazione. Il sistema usa l'UID di ogni applicazione per scoprire quali permessi sono stati concessi e per applicarli in fase di esecuzione. Nelle versioni recenti l'isolamento dei processi di sistema sfrutta SELinux per vincolare ulteriormente i privilegi assegnati a ogni processo.