Nikolay Elenkov

Android™

guida alla sicurezza per hacker e sviluppatori

ESAMINARE
l'architettura di sicurezza della piattaforma Android

CONOSCERE
i framework per la gestione protetta dei dati

"I nostri sistemi ogni giorno fanno passi avanti, ma continuiamo a invitare la comunità a lavorare con noi per tenere al sicuro Android."
– Hiroshi Lockheimer

APOGEO

ANDROID

GUIDA ALLA SICUREZZA PER HACKER E SVILUPPATORI

Nikolay Elenkov

APOGEO

Ho scoperto la qualità del lavoro di Nikolay sulla sicurezza di Android con il rilascio di Android 4.0, Ice Cream Sandwich. Avevo bisogno di una spiegazione migliore del nuovo formato di backup di Android; non riuscivo infatti a sfruttare una vulnerabilità che avevo trovato perché non capivo appieno la nuova funzionalità e il formato. La sua spiegazione chiara e approfondita mi ha aiutato a comprendere il problema, a sfruttare la vulnerabilità e a ottenere rapidamente una patch per i dispositivi di produzione. Da allora sono un visitatore assiduo del suo blog, a cui mi rivolgo quando ho bisogno di un ripasso.

Per quanto fossi onorato della richiesta di scrivere questa prefazione, onestamente non credevo di poter imparare molto da questo libro, visto che mi occupo della sicurezza di Android da parecchi anni. Non avrei potuto sbagliarmi di più. Mentre leggevo e digerivo nuove informazioni su argomenti che pensavo di conoscere approfonditamente, continuavo a pensare quanto mi fossi perso e cosa avrei potuto fare meglio. Perché non c'era una guida come questa quando ho iniziato a interessarmi ad Android?

Questo libro presenta al lettore un'ampia varietà di argomenti legati alla sicurezza, dal sandboxing e dai permessi di Android fino all'implementazione SELinux di Android, chiamata SEAndroid. Offre inoltre spiegazioni eccellenti di dettagli minimi e di funzionalità raramente affrontate, come dm-verify. Alla fine del libro avrete una comprensione migliore delle funzionalità di sicurezza di questo sistema operativo.

Android – guida alla sicurezza per hacker e sviluppatori si è guadagnato un posto fisso nella libreria del mio ufficio.

Jon "jcase" Sawyer
CTO, Applied Cybersecurity LLC

Android ha impiegato un tempo relativamente breve per divenire la piattaforma mobile più diffusa nel mondo. Sebbene in origine sia stato progettato per gli smartphone, oggi è utilizzato su tablet, televisori e dispositivi indossabili; a breve sarà disponibile anche sulle automobili.

Android viene sviluppato a ritmi incessanti: in media assistiamo a due release principali l'anno, ognuna delle quali introduce una nuova interfaccia utente, miglioramenti a livello di prestazioni e una serie di nuove funzionalità utente ampiamente discusse nei blog ed esaminate fin nei più piccoli dettagli dagli appassionati del sistema.

Uno degli aspetti della piattaforma Android che ha subìto notevoli miglioramenti negli ultimi anni, pur non suscitando grande attenzione tra il pubblico, è la sicurezza. Negli anni Android è divenuto sempre più resistente alle tecniche comuni di exploit (come l'overflow del buffer), il suo isolamento delle applicazioni (sandboxing) è stato rafforzato e la sua superficie attaccabile si è notevolmente ridotta grazie a una diminuzione importante del numero di processi di sistema eseguiti come root. Inoltre, nelle ultime versioni sono state introdotte importanti funzionalità di protezione quali i profili utente con una migliore e differenziata gestione dei permessi, la crittografia dell'intero disco, l'archiviazione delle credenziali con supporto hardware e il supporto per il provisioning, la gestione dei dispositivi centralizzati. Con Android 5 sono stati annunciati altri miglioramenti alla sicurezza e nuove funzionalità *enterprise*, come il supporto a una gestione separata dei profili utente e delle relative app associate, il potenziamento della crittografia dell'intero disco e il supporto per l'autenticazione biometrica.

Parlare dei miglioramenti all'avanguardia nel campo della sicurezza è sicuramente interessante, ma è molto più importante capire l'architettura di protezione di Android esaminandola dal basso verso l'alto, visto che

ogni nuova funzionalità di sicurezza viene costruita partendo dal modello di protezione della piattaforma e integrata al suo interno. Il modello di sandboxing di Android (in cui ogni applicazione viene eseguita come un utente Linux distinto e ha una directory dati dedicata) e il suo sistema di autorizzazioni (che richiede a ogni applicazione di dichiarare esplicitamente le funzionalità della piattaforma che intende utilizzare) sono ben documentati e chiari. Tuttavia, i componenti interni di altre funzionalità fondamentali della piattaforma che incidono sulla sicurezza dei dispositivi, come la gestione dei package e la firma del codice, sono generalmente considerati come una "scatola chiusa" dalla community che si occupa di ricerca nell'ambito della sicurezza.

Tra i motivi per cui Android è così popolare vi è sicuramente la facilità relativa con cui un dispositivo può essere "flashato" con una build personalizzata di Android, sottoposto a "root" con l'applicazione di un package di aggiornamento di terze parti o comunque personalizzato con altre tecniche. I forum e i blog degli appassionati di Android sono pieni di numerose guide pratiche che accompagnano gli utenti nelle procedure necessarie per sbloccare un device e applicare vari package personalizzati; tuttavia, queste fonti offrono informazioni molto poco strutturate sul funzionamento "dietro le quinte" degli aggiornamenti di sistema e sui relativi rischi.

Questo libro mira a colmare queste lacune con una descrizione dell'architettura di protezione di Android dal basso verso l'alto, addentrandosi nell'implementazione dei principali sottosistemi Android e nei componenti legati alla sicurezza dei dati e dei dispositivi. Tra gli argomenti affrontati vi sono questioni che interessano tutte le applicazioni, come la gestione di utenti e package, i permessi e le policy del dispositivo, ma anche spiegazioni più specifiche come quelle sui provider di crittografia, l'archiviazione delle credenziali e il supporto di elementi sicuri.

Non è raro che tra una release e l'altra interi sottosistemi Android vengano sostituiti o riscritti; tuttavia, lo sviluppo relativo alla sicurezza è conservativo per natura. Quindi, anche se il comportamento descritto

potrebbe cambiare o essere integrato da una release all'altra, l'architettura di sicurezza fondamentale di Android dovrebbe rimanere piuttosto stabile anche nelle versioni future.

Destinatari del libro

Questo libro è utile per chiunque sia interessato a saperne di più sull'architettura di sicurezza di Android. Le descrizioni di alto livello di ogni funzionalità di sicurezza e i dettagli di implementazione forniti sono un punto di partenza utile sia per i ricercatori nel campo della sicurezza, interessati a valutare il livello di sicurezza di Android nel suo complesso o di un sottosistema specifico, sia per gli sviluppatori della piattaforma, che si occupano della personalizzazione e dell'estensione di Android, entrambi interessati a comprendere il codice sorgente della piattaforma sottostante. Gli sviluppatori di applicazioni possono comprendere meglio il funzionamento della piattaforma per scrivere applicazioni più sicure e sfruttare meglio le API di sicurezza che questa fornisce. Anche se parte del libro è fruibile da un pubblico senza elevate competenze tecniche, la maggior parte della trattazione è strettamente legata al codice sorgente o ai file di sistema di Android, pertanto è utile una conoscenza dei concetti base relativi allo sviluppo del software in ambiente Unix.

Prerequisiti

Il libro presume che i lettori conoscano almeno i fondamenti dei sistemi operativi in stile Unix, preferibilmente Linux; i concetti comuni quali processi, gruppi di utenti, permessi dei file e così via non verranno quindi spiegati. Le funzionalità del sistema operativo specifiche di Linux o aggiunte di recente sono generalmente introdotte brevemente prima di parlare dei sottosistemi di Android che le utilizzano. La maggior parte del codice della piattaforma presentato proviene dai daemon del core (solitamente implementati in C o C++) e dai servizi di sistema (generalmente implementati in Java) di Android, pertanto è richiesta una certa dimestichezza con almeno uno di questi linguaggi. Alcuni esempi di codice contengono sequenze di chiamate del sistema Linux, quindi può essere utile conoscere la programmazione in ambiente Linux per capire il codice (sebbene non sia assolutamente obbligatorio). Infine, anche se la struttura di base e i componenti fondamentali (quali activity e servizi) delle app di Android sono brevemente descritti nei capitoli iniziali, si presume che il lettore conosca almeno le basi dello sviluppo di questo sistema operativo.

Versioni di Android

La descrizione dell'architettura e dell'implementazione di Android proposta in questo libro (fatta eccezione per numerose funzionalità di proprietà di Google) si basa su codice sorgente rilasciato al pubblico come parte dell'*Android Open Source Project* (AOSP). La maggior parte della trattazione e dei frammenti di codice si riferisce ad Android 4.4. A volte si fa riferimento anche al *ramo master* di AOSP, in quanto l'adesione a esso è in genere una valida indicazione della direzione che prenderanno le release future di Android. Va però sottolineato che non tutte le modifiche al ramo master vengono integrate senza variazioni nelle release al pubblico, pertanto è possibile che nelle versioni future alcune delle funzionalità presentate vengano modificate o persino rimosse.

Una versione di anteprima per sviluppatori di Android 5 è stata annunciata mentre questo libro veniva completato. Al momento di questa stesura il codice sorgente di Android 5 non era disponibile e la data esatta di rilascio al pubblico era sconosciuta. Per quanto la release di anteprima includesse nuove funzionalità di sicurezza, quali miglioramenti alla crittografia del dispositivo, alla gestione dei profili e dei dispositivi, nessuna di queste caratteristiche era definitiva. Ecco perché nel libro queste nuove funzionalità non vengono affrontate. Per quanto sia possibile introdurre alcuni dei miglioramenti alla protezione di Android 5 basandosi sul comportamento osservato, senza il codice sorgente sottostante una qualunque descrizione della relativa implementazione sarebbe stata incompleta e teorica.

Organizzazione del libro

Il libro prevede 13 capitoli da leggere in sequenza. In ogni capitolo viene affrontato un aspetto o una caratteristica della sicurezza di Android, e i capitoli successivi si basano sui concetti introdotti in quelli precedenti. Anche se conoscete già l'architettura e il modello di sicurezza di Android e siete interessati soltanto ai dettagli relativi a un argomento specifico, è consigliabile che leggiate comunque i Capitoli da 1 a 3, in quanto i concetti che contengono formano le basi per tutto il resto del libro.

- Il Capitolo 1, "Modello di sicurezza di Android", offre un'introduzione di alto livello all'architettura e al modello di sicurezza di Android.
- Nel Capitolo 2, "Permessi", vengono spiegate la dichiarazione, l'uso e l'applicazione dei permessi Android.
- Il Capitolo 3, "Gestione dei package", affronta la firma del codice e i dettagli relativi al funzionamento dell'installazione e della gestione di applicazioni Android.
- Nel Capitolo 4, "Gestione degli utenti", viene esaminato il supporto multiutente di Android e viene descritta l'implementazione dell'isolamento dei dati nei dispositivi multiutente.
- Il Capitolo 5, "Provider di crittografia", offre informazioni generali sul framework *Java Cryptography Architecture* (JCA) e descrive i provider di crittografia JCA di Android.
- Il Capitolo 6, "Sicurezza di rete e PKI", introduce l'architettura del framework *Java Secure Socket Extension* (JSSE) e ne esamina l'implementazione in Android.
- Il Capitolo 7, "Archiviazione delle credenziali", descrive lo store delle credenziali di Android e introduce le API fornite alle applicazioni che necessitano di memorizzare in sicurezza le chiavi di crittografia.
- Nel Capitolo 8, "Gestione degli account online", si parla del framework di gestione degli account online di Android e

dell'integrazione in Android del supporto per gli account Google.

- Il Capitolo 9, "Sicurezza aziendale", si occupa invece del framework di gestione dei dispositivi di Android, descrivendo nei dettagli l'implementazione del supporto VPN e addentrandosi nel supporto per l'*Extensible Authentication Protocol* (EAP).
- Il Capitolo 10, "Sicurezza del dispositivo", introduce il boot verificato, la crittografia del disco e l'implementazione della schermata di blocco di Android, mostrando anche l'implementazione del debug USB sicuro e del backup del dispositivo crittografato.
- Nel Capitolo 11, "NFC ed elementi sicuri", vengono presentati lo stack NFC di Android, l'integrazione di elementi sicuri (SE, *Secure Element*), le API e l'emulazione di schede basata su host (HCE, *Host-based Card Emulation*).
- Il Capitolo 12, "SELinux", inizia con una descrizione dell'architettura e del linguaggio per le policy di SELinux, spiega nei dettagli i cambiamenti apportati a SELinux per l'integrazione in Android e presenta la policy SELinux base di Android.
- Nel Capitolo 13, "Aggiornamenti di sistema e accesso root", si parla dell'uso del bootloader e del sistema operativo di recovery di Android per l'esecuzione di aggiornamenti sull'intero sistema e delle modalità con cui ottenere l'accesso root sulle build Android di engineering e produzione.

Convenzioni

Questo libro si occupa principalmente dell'architettura e dell'implementazione di Android, pertanto contiene numerosi frammenti di codice e listati, a cui si fa riferimento in maniera costante nei paragrafi che li seguono. Per distinguere questi riferimenti (che di solito includono più costrutti del sistema operativo o del linguaggio di programmazione) dal resto del testo sono state adottate alcune convenzioni di formattazione.

Comandi, nomi di funzioni e variabili, attributi XML, nomi di oggetti SQL, parole chiave, nomi di file, utenti, gruppi di Linux, URL, processi e altri oggetti del sistema operativo sono resi in `monospaziato` (per esempio, "il file `packages.xml` ", "l'utente `system`", "il daemon `vold`" e così via). Anche i valori letterali stringa sono in questo formato; quando li utilizzate in un programma, dovete generalmente racchiuderli tra virgolette singole o doppie (per esempio, `Signature.getInstance("SHA1withRSA", "AndroidOpenSSL")`).

I nomi delle classi Java sono generalmente nel formato non qualificato, senza il nome del package (per esempio, "la classe `Binder`"); i nomi completi sono utilizzati solamente quando nell'API o nel package in discussione sono presenti più classi con lo stesso nome, o quando per altre ragioni è importante specificare il package contenitore (per esempio, "la classe `javax.net.ssl.SSLSocketFactory`"). Quando referenziati nel testo, i nomi di metodi e funzioni sono mostrati con le parentesi, ma i loro parametri sono omessi per ragioni di spazio (per esempio, "il metodo factory `getInstance()`"). Consultate la documentazione di riferimento pertinente per la firma completa della funzione o del metodo.

I percorsi, le cartelle/directory, le partizioni, i nomi degli elementi dell'interfaccia, come menu, opzioni, finestre di dialogo, le chiavi, i permessi, i livelli di protezione, le librerie e i moduli sono invece riportati in *corsivo*.

La maggior parte dei capitoli contiene diagrammi che illustrano l'architettura o la struttura del sottosistema o del componente di

protezione in discussione. Tutti i diagrammi seguono lo stile informale fatto di caselle e frecce, ma non si conformano rigorosamente a un formato specifico. Detto questo, molti prendono in prestito idee dai diagrammi di distribuzione e dalla classe UML; le caselle generalmente rappresentano classi e oggetti, mentre le frecce indicano le dipendenze o i percorsi di comunicazione.

Infine, in molti listati sono inseriti degli indicatori numerici come riferimento a particolari righe di codice in modo da richiamarle e meglio commentarle nel testo. Questi indicatori sono resi graficamente con uno sfondo nero circolare, ❶ - ❷, oppure in bold tra parentesi tonde, **(1) - (2)**.

Ringraziamenti

Desidero ringraziare tutto il personale di No Starch Press che ha collaborato a questo libro. Un ringraziamento speciale va a Bill Pollock, che ha reso intelligibili i miei sproloqui, e ad Alison Law, per la sua pazienza durante la trasformazione degli stessi in un libro vero e proprio.

Un grande ringraziamento è riservato a Kenny Root, che si è occupato della revisione dei capitoli e ha condiviso alcuni aneddoti relativi alle funzionalità di sicurezza di Android.

Ringrazio Jorrit "Chainfire" Jongma per la manutenzione di SuperSU, uno strumento impareggiabile per frugare tra i componenti interni di Android, e per aver rivisto la mia descrizione dello stesso nel Capitolo 13.

Grazie a Jon "jcase" Sawyer per aver continuato a mettere in dubbio le nostre supposizioni sulla sicurezza di Android e per aver contribuito al mio libro scrivendone la prefazione.

L’autore

Nikolay Elenkov ha lavorato a progetti di sicurezza per grandi aziende negli ultimi dieci anni. Ha sviluppato software per la sicurezza su varie piattaforme, da smart card e HSM a server Windows e Linux. Si è interessato ad Android subito dopo il rilascio iniziale al pubblico e sviluppa applicazioni per questo sistema operativo dalla versione 1.5. L’interesse di Nikolay per i componenti interni di Android si è intensificato con il rilascio di Android 4.0 (Ice Cream Sandwich). Negli ultimi tre anni ha documentato le sue scoperte nel suo blog dedicato alla sicurezza Android, consultabile all’indirizzo http://nelenkov.blogspot.com/.

Il revisore tecnico

Kenny Root è un collaboratore fondamentale di Google nell'ambito della piattaforma Android dal 2009 e rivolge la sua attenzione prevalentemente alla sicurezza e alla crittografia. È l'autore di ConnectBot, la prima app SSH per Android; contribuisce inoltre allo sviluppo del mondo open source. Quando non si sta occupando di hacking, trascorre il tempo con la moglie e i due figli. Ha studiato alla Stanford University, alla Columbia University, alla Chinese University di Hong Kong e al Baker College, ma proviene da Kansas City, la città delle grigliate migliori del mondo.

Modello di sicurezza di Android

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.

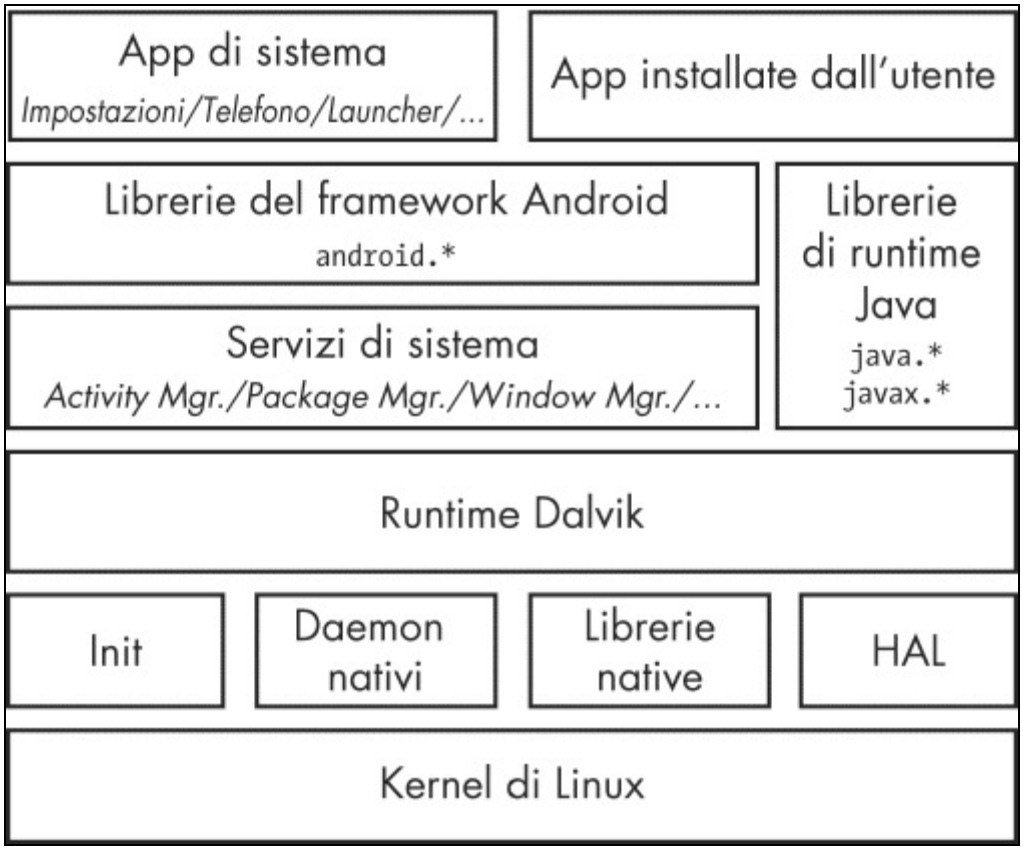

Figura 1.1 L'architettura di Android.

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 (**(1)** e **(2)**), esegue la somma **(3)** e infine restituisce il risultato **(4)**. Dalvik, invece, usa una singola istruzione per sommare i parametri (nei registri `p0` e `p1`), inserisce il risultato nel registro `v0` **(5)** e infine restituisce il contenuto del registro `v0` **(6)**. 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.

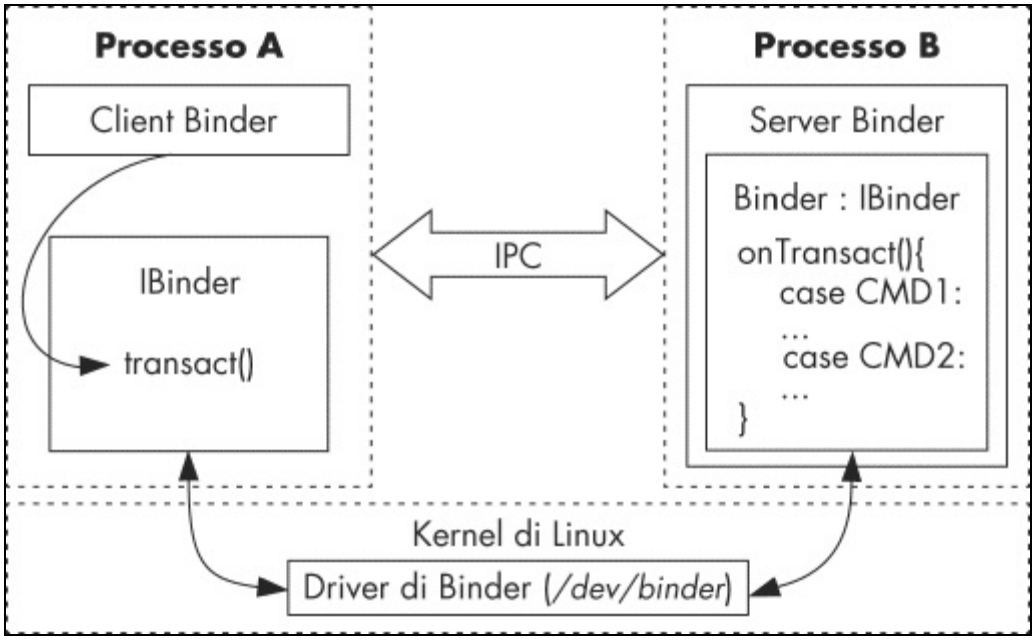

Figura 1.3 Meccanismo IPC di Binder.

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` **(1)**, 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.email(1)
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 SELinux 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.

Permessi

Nel capitolo precedente sono state offerte un'introduzione al modello di sicurezza di Android e una breve presentazione dei permessi. In questo capitolo forniremo maggiori dettagli sui permessi, concentrandoci sull'implementazione e sull'applicazione. Vedremo quindi come definire permessi personalizzati e applicarli ai diversi componenti di Android. Infine, accenneremo ai *pending intent*, token che consentono a un'applicazione di avviare un intent con l'identità e i privilegi di un'altra applicazione.

Natura dei permessi

Come è stato spiegato nel Capitolo 1, le applicazioni Android sono in sandbox e per impostazione predefinita possono accedere unicamente ai propri file e a un set limitato di servizi di sistema. Per interagire con il sistema e le altre applicazioni, possono richiedere un set di permessi aggiuntivi concessi in fase di installazione e non modificabili (con alcune eccezioni che vedremo più avanti nel capitolo).

In Android un *permesso* (o *autorizzazione*, come può apparire quando si scarica e installa un'applicazione) non è altro che una stringa che denota la capacità di eseguire una specifica operazione. L'operazione target può spaziare dall'accesso a una risorsa fisica (per esempio la scheda SD del dispositivo) o ai dati condivisi (quale un elenco di contatti registrati) alla capacità di avviare o accedere a un componente in un'applicazione di terze parti. Android è fornito con un set integrato di permessi predefiniti; in ogni versione vengono aggiunti nuovi permessi corrispondenti alle nuove funzionalità.

NOTA

I nuovi permessi integrati, che bloccano funzionalità che in precedenza non richiedevano un permesso, vengono applicati in maniera condizionale in base al valore `targetSdkVersion` specificato nel manifest di un'app: le applicazioni destinate a versioni di Android rilasciate prima dell'introduzione del nuovo permesso non possono ovviamente conoscere quest'ultimo, pertanto il permesso viene concesso in maniera implicita (senza che sia richiesto). In ogni caso, i permessi concessi in maniera implicita sono tuttora mostrati nell'elenco di permessi della schermata di installazione dell'app (a volte con la dicitura "autorizzazioni", anziché "permessi"), affinché gli utenti possano esserne a conoscenza. Le app destinate a versioni successive devono richiedere esplicitamente il nuovo permesso.

I permessi integrati sono documentati nella guida di riferimento all'API della piattaforma (`http://bit.ly/1nX6vCm`). I permessi supplementari, detti *permessi personalizzati*, possono essere definiti sia dal sistema sia dalle applicazioni installate dall'utente.

Per visualizzare un elenco dei permessi attualmente noti al sistema è possibile utilizzare il comando `pm list permissions` (Listato 2.1). Per visualizzare ulteriori informazioni sui permessi, compresi il package di

definizione, l'etichetta, la descrizione e il livello di protezione, è sufficiente aggiungere il parametro `-f` al comando.

Listato 2.1 Recupero di un elenco di tutti i permessi.

```
$ pm list permissions
All Permissions:

permission:android.permission.REBOOT(1)
permission:android.permission.BIND_VPN_SERVICE(2)
permission:com.google.android.gallery3d.permission.GALLERY_PROVIDER(3)
permission:com.android.launcher3.permission.RECEIVE_LAUNCH_BROADCASTS(4)
--altro codice--
```

I nomi dei permessi sono generalmente preceduti dal nome del package che li definisce concatenato alla stringa `.permission`. I permessi integrati sono definiti nel package `android`, pertanto i loro nomi iniziano con `android.permission`. Per esempio, nel Listato 2.1, `REBOOT` **(1)** e `BIND_VPN_SERVICE` **(2)** sono permessi integrati, mentre `GALLERY_PROVIDER` **(3)** è definito dall'applicazione Gallery (package `com.google.android.gallery3d`) e `RECEIVE_LAUNCH_BROADCASTS` **(4)** è definito dall'applicazione launcher predefinita (package `com.android.launcher3`).

Richiesta dei permessi

Le applicazioni richiedono i permessi aggiungendo uno o più tag `<uses-permission>` al loro file `AndroidManifest.xml` e possono definire nuovi permessi con il tag `<permission>`. Nel Listato 2.2 è mostrato un file manifest di esempio che richiede i permessi `INTERNET` e `WRITE_EXTERNAL_STORAGE`. La definizione di permessi personalizzati è descritta nel paragrafo "Permessi personalizzati" di questo capitolo.

Listato 2.2 Richiesta di permessi con il file manifest dell'applicazione.

```
<?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" />
    --altro codice--
    <application android:name="SampleApp" ...>
    --altro codice--
    </application>
</manifest>
```

Gestione dei permessi

I permessi vengono assegnati a ogni applicazione (come indicato da un *nome di package* univoco) da un servizio di *package manager* di sistema in fase di installazione. Il package manager mantiene un database centrale dei package installati (sia preinstallati, sia installati dall'utente), con informazioni sul percorso di installazione, sulla versione, sul certificato di firma e sui permessi assegnati di ogni package, e un elenco di tutti i permessi definiti su un dispositivo. Il comando `pm list permissions` presentato nel paragrafo precedente recupera questo elenco interrogando il package manager. Questo database dei package è salvato nel file XML `/data/system/packages.xml` e viene aggiornato ogni volta che un'applicazione viene installata, aggiornata o disinstallata. Nel Listato 2.3 è mostrata la voce tipica per un'applicazione in `packages.xml`.

Listato 2.3 Voce di un'applicazione in 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"(1)
         installer="com.android.vending">

    <sigs count="1">
        <cert index="7" />(2)
    </sigs>

    <perms>(3)
        <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>
```

Il significato della maggior parte dei tag e degli attributi è descritto nel Capitolo 3; per il momento concentriamoci su quelli relativi ai permessi. Ogni package è rappresentato da un elemento `<package>` che contiene informazioni sull'UID assegnato (nell'attributo `userId` **(1)**), sul certificato

di firma (nel tag `<cert>` **(2)**) e sui permessi assegnati (elencati come figli del tag `<perms>` **(3)**). Per ottenere informazioni su un package installato tramite un programma è possibile utilizzare il metodo `getPackageInfo()` della classe `android.content.pm.PackageManager`, il quale restituisce un'istanza di `PackageInfo` che incapsula le informazioni contenute nel tag `<package>`.

Abbiamo detto che tutti i permessi sono assegnati in fase di installazione e che non possono essere modificati o revocati senza disinstallare l'applicazione. Come fa allora il package manager a decidere se concedere i permessi richiesti? Per comprenderlo dobbiamo prima affrontare i livelli di protezione dei permessi.

Livelli di protezione dei permessi

Secondo la documentazione ufficiale (http://bit.ly/1ree17k), il *livello di protezione* di un permesso caratterizza il rischio implicito nel permesso stesso e indica la procedura che il sistema deve seguire per determinare se concedere o meno il permesso. In pratica, il fatto che un permesso venga o meno concesso dipende dal suo livello di protezione. Nei paragrafi che seguono sono descritti i quattro livelli di protezione definiti in Android e le modalità con cui il sistema li gestisce.

normal

È il valore predefinito, che definisce un permesso a basso rischio per il sistema o le altre applicazioni. I permessi con il livello di protezione *normal* vengono concessi automaticamente senza chiedere conferma all'utente. Due esempi sono `ACCESS_NETWORK_STATE` (che consente alle applicazioni di accedere alle informazioni sulle reti) e `GET_ACCOUNTS` (che permette l'accesso all'elenco di account nel servizio Accounts).

dangerous

I permessi con livello di protezione *dangerous* permettono l'accesso ai dati dell'utente o qualche forma di controllo sul dispositivo. Due esempi sono `READ_SMS` (che consente a un'applicazione di leggere gli SMS) e `CAMERA` (che permette alle applicazioni di accedere alla fotocamera). Prima di concedere permessi pericolosi (è proprio questo il significato di *dangerous*), Android mostra una finestra di dialogo di conferma contenente informazioni sui permessi (o autorizzazioni) richiesti. Visto che Android richiede che tutti i permessi richiesti siano concessi in fase di installazione, l'utente può accettare di installare l'app, concedendo pertanto i permessi *dangerous* richiesti, o annullare l'installazione. Per esempio, per l'applicazione mostrata nel Listato 2.3 (Google Translate), la finestra di conferma del sistema è simile a quella mostrata nella Figura 2.1.

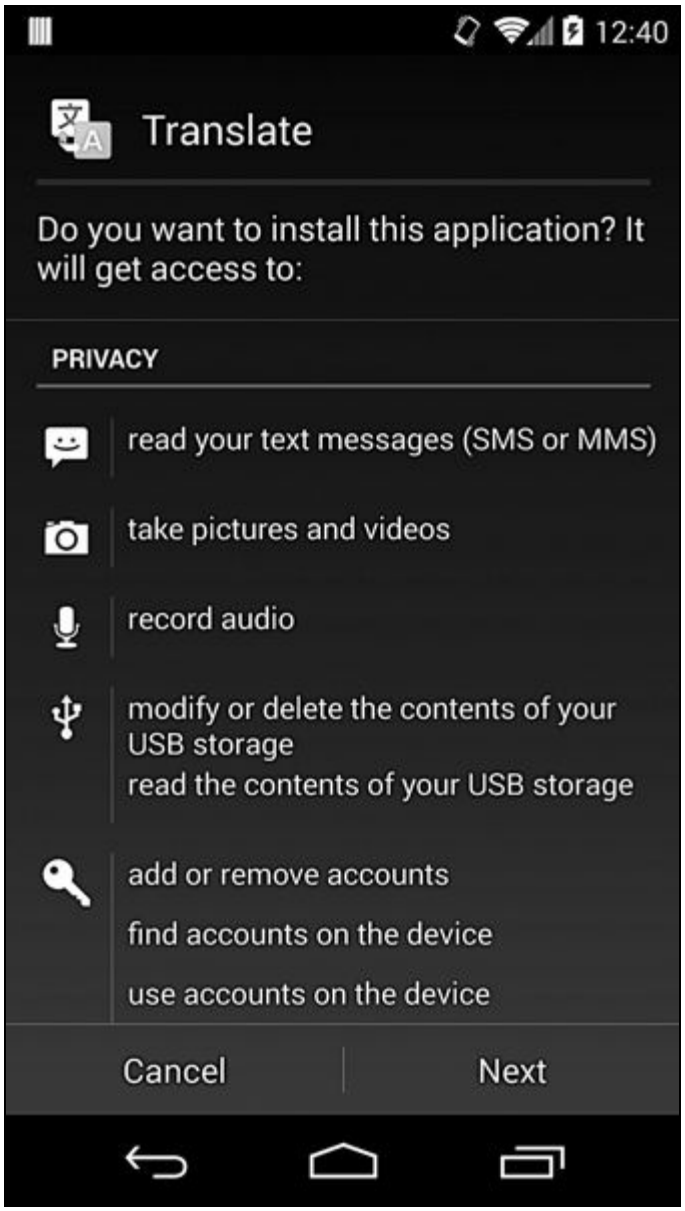

Figura 2.1 Finestra di conferma predefinita per l'installazione di applicazioni Android.

Google Play e altri store di applicazioni mostrano una finestra di dialogo personalizzata, che in genere presenta uno stile differente. Per la stessa applicazione, lo store Google Play mostra la finestra di dialogo della Figura 2.2. Qui tutti i permessi *dangerous* sono organizzati per gruppo di permessi (consultate il paragrafo "Permessi di sistema" in questo capitolo) e i permessi *normal* non sono visualizzati.

signature

Un permesso *signature* viene concesso unicamente alle applicazioni firmate con la stessa chiave dell'applicazione che dichiara il permesso. Questo è il livello di permesso più "forte", perché richiede il possesso di

una chiave di crittografia controllata esclusivamente dal proprietario dell'app (o della piattaforma). Di conseguenza, le applicazioni che usano i permessi *signature* sono generalmente controllate dal medesimo autore. I permessi *signature* integrati sono tipicamente utilizzati da applicazioni di sistema che eseguono attività di gestione dei dispositivi. Due esempi sono `NET_ADMIN` (che configura le interfacce di rete, IPSec e così via) e `ACCESS_ALL_EXTERNAL_STORAGE` (per l'accesso a tutto lo storage esterno multiutente). I permessi *signature* sono trattati nei dettagli nel paragrafo "Permessi signature" di questo capitolo.

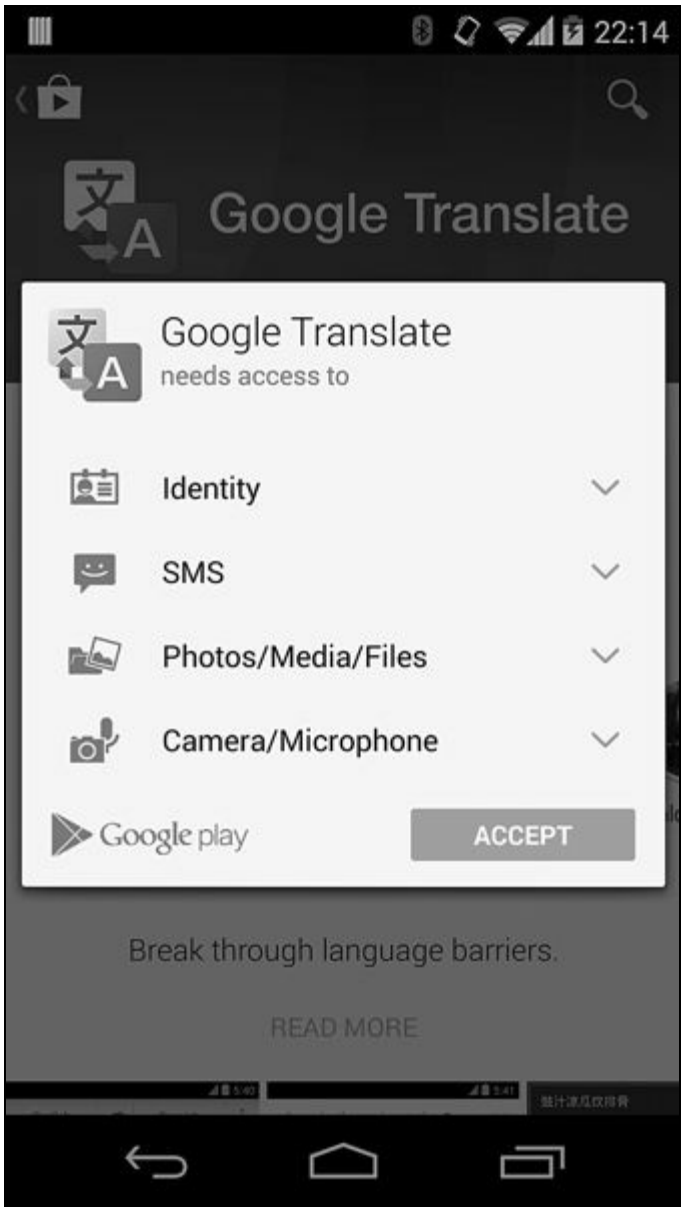

Figura 2.2 Finestra di conferma dello store Google Play per l'installazione di applicazioni.

signatureOrSystem

I permessi con questo livello di protezione rappresentano una sorta di compromesso: sono concessi alle applicazioni che sono parte dell'immagine del sistema o che sono firmate con la stessa chiave dell'app che ha dichiarato il permesso. In questo modo i vendor le cui applicazioni sono preinstallate su un dispositivo Android possono condividere funzionalità specifiche che richiedono un permesso senza dover condividere le chiavi di firma. Fino ad Android 4.3, qualunque applicazione installata nella partizione *system* riceveva automaticamente i permessi `signatureOrSystem`; a partire da Android 4.4, le applicazioni devono essere installate nella directory */system/priv-app/* per ottenere permessi con questo livello di protezione.

Assegnazione dei permessi

I permessi sono applicati a vari livelli in Android. I componenti di livello più alto, quali applicazioni e servizi di sistema, interrogano il package manager per determinare quali permessi sono stati assegnati a un'applicazione e decidere se concedere l'accesso. I componenti di livello più basso, come i daemon nativi, in genere non hanno accesso al package manager e fanno affidamento su UID, GID e GID supplementari assegnati a un processo per determinare quali privilegi concedere. L'accesso alle risorse di sistema quali file di dispositivo, socket di dominio Unix (socket locali) e socket di rete è regolato dal kernel in base al proprietario e alla modalità di accesso della risorsa target e all'UID e ai GID dei processi che tentano l'accesso.

L'applicazione di permessi a livello di framework è descritta nel paragrafo "Applicazione dei permessi" di questo capitolo. Occupiamoci prima del mapping dei permessi ai costrutti a livello di sistema operativo, quali UID e GID, osservando come sono utilizzati questi process ID per l'applicazione dei permessi.

Permessi e attributi di processo

Come in qualunque sistema Linux, ai processi Android sono associati numerosi attributi di processo, da UID e GID reali ed effective a un set di GID supplementari.

Nel Capitolo 1 è stato affermato che in fase di installazione a ogni applicazione Android viene assegnato un UID univoco e che la stessa viene eseguita in un processo dedicato. All'avvio dell'applicazione, l'UID e il GID del processo vengono impostati sull'UID dell'applicazione assegnato dal programma di installazione (il servizio del package manager). Se all'applicazione sono stati assegnati permessi aggiuntivi, questi vengono mappati ai GID e assegnati come GID supplementari al processo. Il permesso ai mapping GID per i permessi integrati è definito

nel file /etc/permission/platform.xml. Nel Listato 2.4 è mostrato un frammento del file platform.xml di un dispositivo Android 4.4.

Listato 2.4 Permesso al mapping GID in platform.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<permissions>
    --altro codice--
    <permission name="android.permission.INTERNET" >(1)
        <group gid="inet" />
    </permission>

    <permission name="android.permission.WRITE_EXTERNAL_STORAGE" >(2)
        <group gid="sdcard_r" />
        <group gid="sdcard_rw" />
    </permission>

    <assign-permission name="android.permission.MODIFY_AUDIO_SETTINGS"
                                                uid="media" />(3)
    <assign-permission name="android.permission.ACCESS_SURFACE_FLINGER"
                                                uid="media" />(4)
    --altro codice--
</permissions>
```

Qui il permesso INTERNET è associato al GID inet **(1)**, mentre il permesso WRITE_EXTERNAL_STORAGE è associato ai GID sdcard_r e sdcard_rw **(2)**. Di conseguenza, qualunque processo di un'app a cui è stato concesso il permesso INTERNET è associato al GID supplementare corrispondente al gruppo inet, mentre per i processi con il permesso WRITE_EXTERNAL_STORAGE vengono aggiunti all'elenco di GID supplementari associati i GID di sdcard_r e sdcard_rw.

Il tag <assign-permission> serve per lo scopo opposto: è utilizzato per assegnare permessi di livello più alto ai processi di sistema in esecuzione con uno UID specifico per cui non esiste un package corrispondente. Nel Listato 2.4 è mostrato come i processi in esecuzione con l'UID *media* (in pratica, questo è il daemon mediaserver) ricevono i permessi MODIFY_AUDIO_SETTINGS **(3)** e ACCESS_SURFACE_FLINGER **(4)**.

Android non dispone del tradizionale file /etc/group, pertanto il mapping tra nomi di gruppo e GID è statico e definito nel file header android_filesystem_config.h. Nel Listato 2.5 è mostrato un frammento di codice contenente i gruppi sdcard_rw **(1)**, sdcard_r **(2)** e inet **(3)**.

Listato 2.5 Mapping tra nomi di utente e gruppo statici e UID/GID in android_filesystem_config.h.

```
--altro codice--
#define AID_ROOT             0  /* tradizionale utente root unix */
#define AID_SYSTEM        1000  /* server di sistema */
--altro codice--
#define AID_SDCARD_RW     1015  /* accesso in scrittura allo storage esterno */
#define AID_SDCARD_R      1028  /* accesso in lettura allo storage esterno */
#define AID_SDCARD_ALL    1035  /* accesso allo storage esterno di tutti gli utenti */
```

```
--altro codice--
#define AID_INET          3003  /* può creare i socket AF_INET e AF_INET6 */
--altro codice--

struct android_id_info {
    const char *name;
    unsigned aid;
};

static const struct android_id_info android_ids[] = {
    { "root",           AID_ROOT, },
    { "system",         AID_SYSTEM, },
    --altro codice--
    { "sdcard_rw",      AID_SDCARD_RW, },(1)
    { "sdcard_r",       AID_SDCARD_R, },(2)
    { "sdcard_all",     AID_SDCARD_ALL, },
    --altro codice--
    { "inet",           AID_INET, },(3)
};
```

Il file `android_filesystem_config.h` definisce anche il proprietario, la modalità di accesso e le capability associate (per i file eseguibili) dei file e delle directory di sistema di Android core.

Il package manager legge `platform.xml` all'avvio e mantiene un elenco dei permessi e dei GID associati. Quando concede i permessi a un package durante l'installazione, il package manager verifica se ogni permesso è associato a uno o più GID. In questo caso, i GID vengono aggiunti all'elenco di GID supplementari associati all'applicazione. L'elenco di GID supplementari viene scritto come ultimo campo del file `packages.list` (Listato 1.5 nel Capitolo 1).

Assegnazione degli attributi di processo

Prima di vedere come il kernel e i servizi di sistema di livello inferiore verificano e applicano i permessi, è necessario esaminare il modo in cui vengono avviati i processi applicativi Android e come vengono loro assegnati gli attributi di processo.

Come spiegato nel Capitolo 1, le applicazioni Android sono implementate in Java ed eseguite dalla Dalvik VM. Di conseguenza, ogni processo applicativo è in effetti un processo Dalvik VM che esegue il bytecode dell'applicazione. Per ridurre il consumo di memoria dell'applicazione e migliorare il tempo di avvio, Android non avvia un nuovo processo Dalvik VM per ogni applicazione, ma utilizza un processo parzialmente inizializzato, chiamato *zygote*, e lo biforca utilizzando la chiamata di sistema `fork()`. Per informazioni dettagliate sulle

funzioni di gestione dei processi come `fork()`, `setuid()` e così via, consultate le rispettive pagine man o un testo sulla programmazione Unix, come il libro di W. Richard Stevens e Stephen A. Rago *Advanced Programming in the UNIX Environment, (*terza edizione, Addison-Wesley Professional, 2013). Ad ogni modo, invece di chiamare una delle funzioni `exec()` come all'avvio di un processo nativo, Android esegue semplicemente la funzione `main()` della classe Java specificata. Questo processo è definito *specializzazione*, perché il processo *zygote* generico viene trasformato in un processo applicativo specifico, così come le cellule originate dalla cellula zigote si trasformano in cellule specializzate che adempiono a funzioni diverse. Il processo di fork eredita quindi l'immagine di memoria del processo *zygote*, che ha precaricato la maggior parte delle classi Java del core e del framework dell'applicazione. Visto che queste classi non cambiano mai e che Linux usa un meccanismo di "copia in scrittura" per il fork dei processi, tutti i processi figlio di *zygote* (vale a dire tutte le applicazioni Android) condividono la stessa copia delle classi Java del framework.

Il processo *zygote* viene avviato dallo script di inizializzazione `init.rc` e riceve i comandi in un socket con dominio Unix, anch'esso chiamato *zygote*. Quando *zygote* riceve la richiesta di avviare un nuovo processo applicativo, avviene la biforcazione, e il processo figlio esegue il codice riportato di seguito (una forma abbreviata del codice di `forkAndSpecializeCommon()` in `dalvik_system_Zygote.cpp`) per specializzarsi (Listato 2.6).

Listato 2.6 Specializzazione di un processo applicativo in zygote.

```
pid = fork();

if (pid == 0) {
    int err;
    /* Processo figlio */
    err = setgroupsIntarray(gids);(1)
    err = setrlimitsFromArray(rlimits);(2)
    err = setresgid(gid, gid, gid);(3)
    err = setresuid(uid, uid, uid);(4)
    err = setCapabilities(permittedCapabilities, effectiveCapabilities);(5)
    err = set_sched_policy(0, SP_DEFAULT);(6)
    err = setSELinuxContext(uid, isSystemServer, seInfo, niceName);(7)
    enableDebugFeatures(debugFlags);(8)
}
```

Come mostrato, il processo figlio configura per prima cosa i suoi GID supplementari (corrispondenti ai permessi) utilizzando `setgroups()`,

chiamato da `setgroupsIntarray()` in **(1)**. A seguire, imposta i limiti delle risorse utilizzando `setrlimit()`, chiamato da `setrlimitsFromArray()` in **(2)** e imposta gli user ID e i group ID reali, effective e salvati utilizzando `setresgid()` **(3)** e `setresuid()` **(4)**.

Il processo figlio può cambiare i suoi limiti delle risorse e tutti gli attributi di processo in quanto inizialmente viene eseguito come root, proprio come il suo processo padre *zygote*. Dopo l'impostazione dei nuovi attributi di processo, il processo figlio viene eseguito con UID e GID assegnati e non può essere nuovamente eseguito come root perché lo user ID salvato non è 0.

Dopo l'impostazione di UID e GID il processo configura le sue capability utilizzando `capset()`, chiamato da `setCapabilities()` **(5)**. Per una descrizione delle capability di Linux, leggete il Capitolo 39 del libro di Michael Kerrisk *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* (No Starch Press, 2010). A questo punto imposta la sua policy di programmazione aggiungendo sé stesso a uno dei gruppi di controllo predefiniti **(6)** (fate riferimento a Linux Kernel Archives, *CGROUPS*, `http://bit.ly/1u8cwcI`). Nel punto **(7)**, il processo imposta il suo nome descrittivo (visualizzato nell'elenco dei processi e generalmente corrispondente al nome del package dell'applicazione) e il tag `seinfo` (utilizzato da SELinux, come spiegato nel Capitolo 12). Per finire, abilita il debug, se richiesto **(8)**.

NOTA

Android 4.4 introduce un nuovo runtime sperimentale chiamato *Android RunTime* (ART), che si prevede andrà a sostituire Dalvik in una versione futura. Anche se ART apporta numerose modifiche all'ambiente di esecuzione attuale, la più importante delle quali è la compilazione AOT (*ahead-of-time*), utilizza lo stesso modello di esecuzione dei processi applicativi basato su *zygote* presente in Dalvik.

La relazione a livello di processo tra *zygote* e il processo applicativo è evidente nell'elenco di processi ottenuto con il comando `ps`, come mostrato nel Listato 2.7.

Listato 2.7 Relazione tra zygote e processi applicativi.

```
$ ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     680    540   ffffffff 00000000 S /init(1)
--altro codice--
```

```
root      181   1     858808 38280 ffffffff 00000000 S zygote(2)
--altro codice--
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
```

Qui la colonna `PID` denota il process ID, la colonna `PPID` il process ID padre e la colonna `NAME` il nome del processo. Come potete vedere, *zygote* (`PID 181` **(2)**) viene avviato dal processo `init` (`PID 1` **(1)**) e tutti i processi applicativi hanno *zygote* come padre (`PPID 181`). Ogni processo viene eseguito da un utente dedicato, sia predefinito (`radio`, `nfc`), sia assegnato automaticamente (`u0_a7`) in fase di installazione. I nomi dei processi corrispondono al nome del package di ogni applicazione (`com.android.phone`, `com.android.nfc` e `com.google.android.gms`).

Applicazione dei permessi

Come già spiegato nel paragrafo precedente, a ogni processo applicativo vengono assegnati un UID, un GID e alcuni GID supplementari a seguito del fork da *zygote*. I daemon del kernel e del sistema utilizzano questi identificatori di processo per decidere se concedere l'accesso a una particolare risorsa o funzione di sistema.

Applicazione a livello di kernel

L'accesso ai file normali, ai nodi dei dispositivi e ai socket locali è regolamentato come in qualunque sistema Linux. Un'aggiunta specifica di Android è la richiesta che i processi che desiderano creare socket di rete appartengano al gruppo `inet`. Questa aggiunta del kernel di Android è nota come *sicurezza di rete paranoid* ed è implementata come verifica supplementare nel kernel di Android, come mostrato nel Listato 2.8.

Listato 2.8 Implementazione della sicurezza di rete paranoid nel kernel di Android.

```
#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);(1)}
#else
static inline int current_has_network(void)
{        return 1;(2)
}
#endif
--altro codice--
static int inet_create(struct net *net, struct socket *sock, int protocol,
                                                        int kern)
{
        --altro codice--
        if (!current_has_network())
                 return -EACCES;(3)
        --altro codice--
}
```

I processi chiamanti che non appartengono al gruppo `AID_INET` (GID 3003, nome `inet`) e non possiedono la capability `CAP_NET_RAW` (che consente l'uso dei socket RAW e PACKET) ricevono un errore di accesso negato (**(1)** e **(3)**). I kernel non Android non definiscono `CONFIG_ANDROID_PARANOID_NETWORK`, di conseguenza non è richiesta l'appartenenza a gruppi speciali per creare un socket **(2)**. Affinché il gruppo `inet` sia assegnato a un processo applicativo, è necessario che gli sia concesso il

permesso INTERNET. Di conseguenza, solo le applicazioni con il permesso INTERNET possono creare socket di rete. Oltre a verificare le credenziali dei processi, durante la creazione dei socket i kernel di Android concedono capability specifiche ai processi in esecuzione con GID specifici: i processi in esecuzione con AID_NET_RAW (GID 3004) ricevono la capability CAP_NET_RAW, mentre quelli in esecuzione con AID_NET_ADMIN (GID 3005) ottengono la capability CAP_NET_ADMIN.

La sicurezza di rete paranoid è utilizzata anche per controllare l'accesso ai socket Bluetooth e il driver di tunneling del kernel (utilizzato per le VPN). Un elenco completo dei GID di Android trattati in maniera speciale dal kernel è disponibile nel file include/linux/android_aid.h nella struttura ad albero di origine del kernel.

Applicazione a livello di daemon nativo

Per quanto sia Binder il meccanismo IPC preferito in Android, i daemon nativi di livello inferiore spesso usano i socket del dominio Unix (socket locali) per IPC. Questi socket sono rappresentati da nodi nel file system, pertanto consentono di utilizzare i permessi standard del file system per il controllo di accesso.

La maggior parte dei socket viene creata con una modalità di accesso che permette l'accesso solamente al proprietario e al relativo gruppo, pertanto i client eseguiti con un UID o un GID diverso non possono connettersi al socket. I socket locali per i daemon di sistema sono definiti in init.rc e sono creati da init all'avvio con la modalità di accesso specificata. A titolo di esempio, il Listato 2.9 mostra la definizione del daemon di gestione dei volumi (vold) in init.rc.

Listato 2.9 Daemon vold in init.rc.

```
service vold /system/bin/vold
    class core
    socket vold stream 0660 root mount(1)
    ioprio be 2
```

vold dichiara un socket chiamato vold con la modalità di accesso 0660, di proprietà di root e con gruppo impostato su mount **(1)**. Il daemon vold deve essere eseguito come root per montare e smontare i volumi, ma i membri

del gruppo `mount` (`AID_MOUNT`, GID 1009) possono inviare comandi attraverso il socket locale senza essere in esecuzione come superuser. I socket locali per i daemon Android sono creati nella directory */dev/socket/*. Nel Listato 2.10 è mostrato che il socket `vold` **(1)** dispone del proprietario e dei permessi specificati in `init.rc`.

Listato 2.10 Socket locali per i daemon del sistema core in /dev/socket/.

```
$ 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
--altro codice--
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 vold(1)
srw-rw---- root     system           1970-01-18 14:26 zygote
```

I socket del dominio Unix consentono il passaggio e il recupero delle credenziali client tramite il messaggio di controllo `SCM_CREDENTIALS` e l'opzione socket `SO_PEERCRED`. Analogamente all'effective UID e all'effective GUID che sono parte di una transazione Binder, le credenziali del peer associate a un socket locale vengono verificate dal kernel e non possono essere falsificate dai processi a livello utente. In questo modo i daemon nativi possono implementare un preciso controllo supplementare sulle operazioni consentite a un client specifico, come mostrato nel Listato 2.11 che utilizza il daemon `vold` come esempio.

Listato 2.11 Controllo di accesso preciso basato sulle credenziali client del socket in vold.

```
int CommandListener::CryptfsCmd::runCommand(SocketClient *cli,
                                            int argc, char **argv) {
    if ((cli->getUid() != 0) && (cli->getUid() != AID_SYSTEM)) {(1)
        cli->sendMsg(ResponseCode::CommandNoPermission,
               "No permission to run cryptfs commands", false);
        return 0;
    }
    --altro codice--
}
```

Il daemon `vold` consente esclusivamente i comandi di gestione del contenitore crittografato inviati ai client in esecuzione come utenti `root` (UID 0) o `system` (`AID_SYSTEM`, UID 1000). Qui l'UID restituito da `SocketClient->getUid()` **(1)** viene inizializzato con l'UID del cliente ottenuto utilizzando `getsockopt(SO_PEERCRED)`, come mostrato nel Listato 2.12 in corrispondenza del punto **(1)**.

Listato 2.12 Recupero delle credenziali client del socket locale con getsockopt().

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

    int err = getsockopt(socket, SOL_SOCKET, SO_PEERCRED, &creds, &szCreds);(1)
    if (err == 0) {
        mPid = creds.pid;
        mUid = creds.uid;
        mGid = creds.gid;
    }
}
```

La funzionalità di connessione al socket locale è incapsulata nella classe `android.net.LocalSocket` ed è disponibile anche per le applicazioni Java; consente ai servizi di sistema di livello più alto di comunicare con i daemon nativi senza utilizzare il codice JNI. Per esempio, la classe framework `MountService` usa `LocalSocket` per inviare comandi al daemon `vold`.

Applicazione a livello di framework

Come spiegato nell'introduzione ai permessi di Android, l'accesso ai componenti Android può essere controllato dai permessi dichiarando quelli richiesti nel manifest dell'applicazione contenitore. Il sistema tiene traccia dei permessi associati a ogni componente e, prima di consentire l'accesso, verifica se ai chiamanti sono stati concessi i permessi richiesti. I componenti non possono cambiare i permessi necessari in fase di runtime, pertanto l'applicazione da parte del sistema è *statica*. I permessi statici sono un esempio di sicurezza dichiarativa. Quando si utilizza la sicurezza dichiarativa, gli attributi di sicurezza quali ruoli e permessi vengono inseriti nei metadati di un componente (il file `AndroidManifest.xml` in Android) anziché nel componente stesso, e sono applicati dal contenitore o dall'ambiente di runtime. Si ottiene così il vantaggio di isolare le decisioni legate alla sicurezza dalla logica del business, per quanto il risultato sia meno flessibile rispetto all'implementazione dei controlli di sicurezza all'interno del componente.

I componenti Android possono inoltre verificare se a un processo chiamante è stato concesso un permesso specifico senza dichiarare i permessi nel manifest. Questa applicazione *dinamica* dei permessi

richiede una maggiore quantità di lavoro, ma permette un controllo di accesso più preciso. È un esempio di sicurezza imperativa, perché le decisioni sulla sicurezza sono prese da ogni componente, invece di essere applicate dall'ambiente di runtime.

Vediamo ora nei dettagli l'implementazione dell'applicazione statica e dinamica dei permessi.

Applicazione dinamica

Come già spiegato nel Capitolo 1, il core di Android viene implementato come un set di servizi di sistema cooperanti che possono essere chiamati da altri processi utilizzando il meccanismo IPC di Binder. I servizi del core si registrano nel service manager e qualunque applicazione che conosce il loro nome di registrazione può ottenere un riferimento Binder. Tuttavia, Binder non dispone di un meccanismo di controllo di accesso integrato; di conseguenza, quando i client ottengono un riferimento possono chiamare qualunque metodo del servizio di sistema sottostante passando i parametri appropriati a `Binder.transact()`. Di conseguenza, il controllo di accesso deve essere implementato da ogni servizio di sistema.

Nel Capitolo 1 è stato spiegato come i servizi di sistema possono regolamentare l'accesso alle operazioni esportate controllando direttamente l'UID del chiamante ottenuto da `Binder.getCallingUid()` (Listato 1.7). Tuttavia, questo metodo richiede che il servizio conosca anticipatamente l'elenco di UID consentiti, soluzione attuabile solo per UID fissi e ben noti come quelli di `root` (UID 0) e `system` (UID 1000). Inoltre, la maggior parte dei servizi non si preoccupa dell'UID effettivo del chiamante; desidera semplicemente verificare se gli è stato concesso un permesso specifico.

Ogni UID dell'applicazione in Android è associato a un package univoco (a meno che non sia parte di un user ID condiviso) e il package manager tiene traccia dei permessi concessi a ogni package grazie all'interrogazione del servizio package manager. La verifica della disponibilità di un particolare permesso per il chiamante è un'operazione

comune, pertanto Android mette a disposizione numerosi metodi helper per eseguire questo controllo nella classe `android.content.Context`.

Vediamo allora il funzionamento del metodo `int Context.checkPermission(String permission, int pid, int uid)`. Questo metodo restituisce `PERMISSION_GRANTED` se l'UID passato dispone del permesso, oppure `PERMISSION_DENIED` in caso contrario. Se il chiamante è `root` o `system`, il permesso viene concesso automaticamente. Per ottimizzare le prestazioni, se il permesso richiesto è stato dichiarato dall'app chiamante, questo viene concesso senza esaminare il permesso vero e proprio; in caso contrario, il metodo verifica se il componente target è pubblico (esportato) o privato e nega l'accesso a tutti i componenti privati. L'esportazione dei componenti è affrontata nel paragrafo "Componenti pubblici e privati" di questo capitolo. Infine, il codice interroga il servizio del package manager per valutare se al chiamante è stato concesso il permesso richiesto. Il codice pertinente della classe `PackageManagerService` è mostrato nel Listato 2.13.

Listato 2.13 Verifica dei permessi basata su UID in PackageManagerService.

```
public int checkUidPermission(String permName, int uid) {
    synchronized (mPackages) {
        Object obj = mSettings.getUserIdLPr((1)UserHandle.getAppId(uid));
        if (obj != null) {
            GrantedPermissions gp = (GrantedPermissions)obj;(2)
            if (gp.grantedPermissions.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        } else {
            HashSet<String> perms = mSystemPermissions.get(uid);(3)
            if (perms != null && perms.contains(permName)) {
                return PackageManager.PERMISSION_GRANTED;
            }
        }
    }
    return PackageManager.PERMISSION_DENIED;
}
```

`PackageManagerService` determina per prima cosa l'*app ID* dell'applicazione in base all'UID passato **(1)** (alla stessa applicazione possono essere assegnati più UID se viene installata per utenti diversi, come sarà spiegato nei dettagli nel Capitolo 4) e poi ottiene il set di permessi concessi. Se la classe `GrantedPermission` (che contiene `java.util.Set<String>` dei nomi dei permessi) contiene il permesso target, il metodo restituisce `PERMISSION_GRANTED` **(2)**. In caso contrario, verifica se il permesso target deve essere assegnato automaticamente all'UID passato **(3)** (in base ai tag `<assign-permission>` in

platform.xml, come mostrato nel Listato 2.4). Se anche questa verifica ha esito negativo, viene restituito PERMISSION_DENIED.

Gli altri metodi helper per la verifica dei permessi nella classe Context si attengono alla stessa procedura. Il metodo int checkCallingOrSelfPermission(String permission) chiama Binder.getCallingUid() e Binder.getCallingPid(), quindi chiama checkPermission(String permission, int pid, int uid) utilizzando i valori ottenuti. Il metodo enforcePermission(String permission, int pid, int uid, String message) non restituisce un risultato, ma genera una SecurityException con il messaggio specificato qualora il permesso non venga concesso. Per esempio, la classe BatterStatsService garantisce che solo le app che possiedono il permesso BATTERY_STATS possano ottenere le statistiche sulla batteria chiamando enforceCallingPermission() prima di eseguire qualunque altro codice, come mostrato nel Listato 2.14. I chiamanti a cui non è stato concesso il permesso ricevono una SecurityException.

Listato 2.14 Verifica dinamica dei permessi in BatteryStatsService.

```
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;
}
```

Applicazione dinamica

L'applicazione statica dei permessi viene utilizzata quando un'applicazione prova a interagire con un componente dichiarato da un'altra applicazione. Il processo di applicazione tiene conto dei permessi dichiarati per ogni componente target (se presente) e consente l'interazione se il processo chiamante possiede il permesso richiesto.

Android usa gli intent per descrivere un'operazione da eseguire; gli intent che specificano per intero il componente target (con nome del package e della classe) sono detti *explicit*. Gli *implicit* intent, invece, contengono alcuni dati (spesso un'azione astratta come ACTION_SEND) che

consentono al sistema di individuare un componente corrispondente ma non specificano per intero il componente target.

Quando il sistema riceve un implicit intent, per prima cosa lo risolve cercando i componenti corrispondenti: se ne trova più di uno, viene mostrata all'utente una finestra di dialogo di selezione. Una volta selezionato un componente target, Android verifica se è associato a qualche permesso e, in questo caso, controlla se i permessi sono stati concessi al chiamante.

Il processo generico è simile all'applicazione dinamica: UID e PID del chiamante vengono ottenuti utilizzando `Binder.getCallingUid()` e `Binder.getCallingPid()`, l'UID del chiamante viene associato a un nome di package e infine vengono recuperati i permessi associati. Se il set di permessi del chiamante contiene i permessi richiesti dal componente target, il componente viene avviato; in caso contrario, viene generata una `SecurityException`.

Le verifiche dei permessi sono svolte da `ActivityManagerService`, che risolve l'intent specificato e verifica se al componente target è associato un attributo di permesso: in questo caso, la verifica dei permessi viene delegata al package manager. Le tempistiche e la sequenza concreta della verifica sono leggermente diverse per ogni componente target (più avanti sarà descritta la modalità di verifica specifica per ognuno di essi).

Applicazione di permessi per activity e servizi

La verifica dei permessi per le activity viene eseguita quando l'intent passato a `Context.startActivity()` o `startActivityForResult()` viene risolto in un'activity che dichiara un permesso. Se il chiamante non dispone del permesso richiesto viene generata una `SecurityException`. Dal momento che i servizi Android possono essere avviati, arrestati e associati, le chiamate a `Context.startService()`, `stopService()` e `bindService()` sono tutte soggette alla verifica dei permessi se il servizio target dichiara un permesso.

Applicazione di permessi ai content provider

I permessi dei content provider possono proteggere sia l'intero componente sia un particolare URI esportato; è inoltre possibile specificare permessi diversi per la lettura e la scrittura. Ulteriori informazioni sulla dichiarazione dei permessi sono disponibili nel paragrafo "Permessi dei content provider" di questo capitolo. Se sono stati specificati permessi diversi per la lettura e la scrittura, il permesso di lettura controlla chi può chiamare `ContentResolver.query()` sul provider o sull'URI target, mentre il permesso di scrittura controlla chi può chiamare `ContentResolver.insert()`, `ContentResolver.update()` e `ContentResolver.delete()` sul provider o su uno dei suoi URI esportati. I controlli sono eseguiti in maniera sincrona quando viene chiamato uno di questi metodi.

Applicazione di permessi ai broadcast

Durante l'invio di un broadcast, le applicazioni possono richiedere che i receiver dispongano di un particolare permesso utilizzando il metodo `Context.sendBroadcast (Intent intent, String receiverPermission)`. I broadcast sono asincroni, pertanto non viene eseguita alcuna verifica dei permessi durante la chiamata del metodo; la verifica viene invece eseguita durante il trasferimento dell'intent ai receiver registrati. Se un receiver target non possiede il permesso richiesto, viene ignorato e non riceve il broadcast, ma non viene generata alcuna eccezione. A loro volta, i broadcast receiver possono richiedere che i broadcaster possiedano un permesso specifico per sceglierli come target.

Il permesso richiesto viene specificato nel manifest o durante la registrazione dinamica di un broadcast. Questa verifica dei permessi viene eseguita anche durante il recapito del broadcast e non genera una `SecurityException`. Di conseguenza, il recapito di un broadcast può richiedere due controlli dei permessi: uno per il broadcast sender (se il receiver ha specificato un permesso) e uno per il broadcast receiver (se il sender ha specificato un permesso).

Broadcast protetti e sticky

Alcuni broadcast di sistema sono dichiarati come *protetti* (per esempio `BOOT_COMPLETED` e `PACKAGE_INSTALLED`) e possono essere inviati solo da un processo di sistema in esecuzione con uno degli UID `SYSTEM_UID`, `PHONE_UID`, `SHELL_UID`, `BLUETOOTH_UID` o `root`. Se un processo in esecuzione con un altro UID tenta di inviare un broadcast protetto, riceve una `SecurityException` durante la chiamata a uno dei metodi `sendBroadcast()`. L'invio di broadcast "sticky" (per i quali il sistema conserva l'oggetto `Intent` inviato al completamento del broadcast) richiede che il sender possieda il permesso `BROADCAST_STICKY`; diversamente, viene generata una `SecurityException` e il broadcast non viene trasmesso.

Permessi di sistema

I permessi predefiniti di Android sono definiti nel package `android`, a volte chiamato "il framework" o "la piattaforma". Come già spiegato nel Capitolo 1, il framework core di Android è il set di classi condivise dai servizi di sistema, alcune delle quali sono esposte anche dall'SDK pubblico. Le classi del framework sono inserite nei file JAR della directory */system/framework/* (circa 40 nelle ultime release).

Oltre alle librerie JAR, il framework contiene un singolo file APK chiamato `framework-res.apk`, che contiene le risorse del framework (animazioni, drawable, layout e così via) ma non codice vero e proprio. L'aspetto più importante è che definisce il package `android` e i permessi di sistema. `framework-res.apk` è un file APK, pertanto contiene un file `AndroidManifest.xml` in cui sono dichiarati i permessi e i gruppi di permessi (Listato 2.15).

Listato 2.15 Permessi di sistema definiti nel manifest di framework-res.apk.

```
<?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">
    --altro codice--
    <protected-broadcast android:name="android.intent.action.BOOT_COMPLETED" />(1)
    <protected-broadcast android:name="android.intent.action.PACKAGE_INSTALL" />
    --altro codice--
    <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"/>(2)
    <permission android:name="android.permission.SEND_SMS"
        android:permissionGroup="android.permission-group.MESSAGES"(3)
        android:protectionLevel="dangerous"
        android:permissionFlags="costsMoney"
        android:label="@string/permlab_sendSms"
        android:description="@string/permdesc_sendSms" />
    --altro codice--
    <permission android:name="android.permission.NET_ADMIN"
        android:permissionGroup="android.permission-group.SYSTEM_TOOLS"
        android:protectionLevel="signature" />(4)
    --altro codice--
    <permission android:name="android.permission.MANAGE_USB"
        android:permissionGroup="android.permission-group.HARDWARE_CONTROLS"
        android:protectionLevel="signature|system"(5)
        android:label="@string/permlab_manageUsb"
        android:description="@string/permdesc_manageUsb" />
    --altro codice--
    <permission android:name="android.permission.WRITE_SECURE_SETTINGS"
        android:permissionGroup="android.permission-group.DEVELOPMENT_TOOLS"
        android:protectionLevel="signature|system|development"(6)
        android:label="@string/permlab_writeSecureSettings"
        android:description="@string/permdesc_writeSecureSettings" />
```

```
    --altro codice--
</manifest>
```

Come mostrato nel listato, il file `AndroidManifest.xml` dichiara anche i broadcast protetti del sistema **(1)**. Un *gruppo di permessi* **(2)** specifica un nome per un set di permessi correlati. I singoli permessi possono essere aggiunti a un gruppo specificando il nome di questo nel relativo attributo `permissionGroup` **(3)**.

I gruppi di permessi sono utilizzati per visualizzare i permessi correlati nell'interfaccia di sistema, ma ogni permesso deve tuttora essere richiesto singolarmente: in pratica, le applicazioni non possono richiedere che vengano concessi loro tutti i permessi di un gruppo.

Come sappiamo, a ogni permesso è associato un livello di protezione dichiarato utilizzando l'attributo `protectionLevel`, come mostrato al punto **(4)**.

I livelli di protezione possono essere combinati con i *flag di protezione* per vincolare ulteriormente la concessione dei permessi. I flag attualmente definiti sono `system` (0x10) e `development` (0x20). Il flag `system` richiede che le applicazioni siano parte dell'immagine del sistema (vale a dire installate nella partizione *system* di sola lettura) affinché sia concesso loro un permesso. Per esempio, il permesso `MANAGE_USB`, che consente alle applicazioni di gestire preferenze e permessi per i dispositivi USB, è concesso solo alle applicazioni firmate con la chiave di firma della piattaforma e installate nella partizione *system* **(5)**. Il flag `development` segnala i permessi di sviluppo **(6)**, descritti dopo la presentazione dei permessi di firma.

Permessi di firma

Come spiegato nel Capitolo 1, tutte le applicazioni Android devono disporre di un codice firmato con una chiave di firma controllata dallo sviluppatore. Questo vale sia per le applicazioni di sistema sia per i package di risorse del framework. I dettagli della firma dei package sono disponibili nel Capitolo 3; per il momento concentriamoci sulla firma delle applicazioni di sistema.

Le applicazioni di sistema sono firmate da una *chiave della piattaforma*. Per impostazione predefinita, nella struttura sorgente Android attuale sono presenti quattro chiavi diverse: *platform*, *shared*, *media* e *testkey* (*releasekey* per le build di release). Tutti i package considerati parte della piattaforma core (interfaccia di sistema, Impostazioni, Telefono, Bluetooth e così via) sono firmati con la chiave *platform*; i package per la ricerca e i contatti con la chiave *shared*; i provider relativi all'app Galleria e agli elementi multimediali con la chiave *media*; tutto il resto (compresi i package che non specificano esplicitamente la chiave di firma nel makefile) con la chiave *testkey* (o *releasekey*). L'APK `framework-res.apk` che definisce i permessi di sistema è firmato con la chiave *platform*, di conseguenza ogni app che prova a richiedere un permesso di sistema con il livello di protezione *signature* deve essere firmato con la stessa chiave del package di risorse del framework.

Per esempio, il permesso `NET_ADMIN` mostrato nel Listato 2.15 (che consente a un'applicazione di controllare le interfacce di rete) è dichiarato con il livello di protezione *signature* **(4)** e può essere concesso unicamente alle applicazioni firmate con la chiave *platform*.

NOTA

Android Open Source Repository (AOSP) contiene chiavi di test pre-generate, utilizzate per impostazione predefinita durante la firma dei package compilati. Non dovrebbero mai essere utilizzate per le build di produzione, perché sono pubbliche e disponibili a chiunque scarichi il codice sorgente di Android. Le build di release dovrebbero essere firmate unicamente con chiavi private generate da zero e appartenenti al proprietario della build. Le chiavi possono essere generate utilizzando lo script `make_key`, incluso nella directory *development/tools/* di AOSP. Fate riferimento al file `build/target/product/security/README` per i dettagli sulla generazione di chiavi della piattaforma.

Permessi di sviluppo

Per tradizione, il modello di permessi di Android non consente la concessione e la revoca dinamiche dei permessi, quindi il set di permessi concessi a un'applicazione è fisso in fase di installazione. Tuttavia, a partire da Android 4.2 questa regola è stata resa meno rigorosa con l'aggiunta di alcuni *permessi di sviluppo* (quali `READ_LOGS` e

`WRITE_SECURE_SETTINGS`). I permessi di sviluppo possono essere concessi o revocati su richiesta utilizzando i comandi `pm grant` e `pm revoke` nella shell di Android.

NOTA

Ovviamente, questa operazione non è disponibile per tutti ed è protetta dal permesso di firma `GRANT_REVOKE_PERMISSIONS`, che viene concesso allo user ID condiviso `android.uid.shell` (UID 2000) e a tutti i processi avviati dalla shell di Android (eseguita anch'essa con UID 2000).

User ID condiviso

Le applicazioni Android firmate con la stessa chiave possono richiedere la capacità di essere eseguite con lo stesso UID e, facoltativamente, nello stesso processo. Questa funzionalità è chiamata *user ID condiviso* ed è utilizzata in maniera estensiva dai servizi del framework core e dalle applicazioni di sistema. Dal momento che può influenzare il conteggio dei processi e la gestione delle applicazioni, il team di Android ne sconsiglia l'uso alle applicazioni di terzi, ma è comunque disponibile anche per le applicazioni installate dall'utente. Inoltre, il passaggio allo user ID condiviso di applicazioni esistenti che non usano uno user ID condiviso non è supportato, quindi le applicazioni cooperanti che necessitano di user ID condivisi devono essere progettate e rilasciate come tali fin dall'inizio.

Lo user ID condiviso viene abilitato aggiungendo l'attributo `sharedUserId` all'elemento `root` di `AndroidManifest.xml`. Lo user ID specificato nel manifest deve avere il formato dei package Java (contenente almeno un punto [.]) ed è usato come identificatore, proprio come i nomi dei package per le applicazioni. Se l'UID condiviso specificato non esiste, viene creato. Se è già installato un altro package con lo stesso UID condiviso, il certificato di firma viene confrontato con quello del package esistente e, qualora non corrispondano, viene restituito un errore `INSTALL_FAILED_SHARED_USER_INCOMPATIBLE` e l'installazione non riesce.

L'aggiunta dell'attributo `sharedUserId` a una nuova versione di un'app installata provoca la modifica del suo UID, causando la perdita dell'accesso ai suoi file (accadeva in alcune delle prime versioni di Android). Per questo l'aggiunta non è consentita dal sistema, che rifiuta l'aggiornamento con l'errore `INSTALL_FAILED_UID_CHANGED`. In breve, se prevedete di utilizzare gli UID condivisi per le vostre app, dovrete progettarle in tal senso fin dall'inizio e utilizzare questa tecnica sin dalla primissima release.

L'UID condiviso stesso è un oggetto di prima classe nel database dei package del sistema ed è trattato in maniera analoga alle applicazioni: dispone infatti di uno o più certificati di firma associati e di permessi. Android dispone di cinque UID condivisi predefiniti, aggiunti automaticamente durante il bootstrap del sistema:

- `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).

Nel Listato 2.16 è mostrata la definizione dell'utente condiviso `android.uid.system`.

Listato 2.16 Definizione dell'utente condiviso android.uid.system.

```
<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" />
--altro codice--
<shared-user/>
```

Come potete osservare, a parte alcuni permessi "preoccupanti" (circa 66 su un dispositivo 4.4), la definizione è molto simile alle dichiarazioni dei package viste in precedenza. Al contrario, i package che sono parte di un utente condiviso non dispongono di un elenco associato di permessi concessi, ma ereditano i permessi dell'utente condiviso, che costituiscono l'unione dei permessi richiesti da tutti i package attualmente installati con il medesimo user ID condiviso. Un effetto collaterale relativo al caso in cui il package è parte di un utente condiviso riguarda il fatto che il package può accedere ad API per cui non ha richiesto esplicitamente i permessi, purché qualche package con lo stesso user ID condiviso li abbia già richiesti. I permessi vengono rimossi dinamicamente dalla definizione `<shared-user>` nel momento in cui i package vengono installati o disinstallati, pertanto il set di permessi disponibili non è né garantito né costante.

Nel Listato 2.17 è mostrata la dichiarazione dell'app di sistema `KeyChain` eseguita con uno user ID condiviso. Come potete osservare, fa riferimento all'utente condiviso con l'attributo `sharedUserId` ed è priva di dichiarazioni di permesso esplicite.

Listato 2.17 Dichiarazione del package per un'applicazione eseguita con uno user ID condiviso.

```
<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>
```

L'UID condiviso non è solo un costrutto di gestione dei package, ma effettua anche il mapping vero e proprio con un UID Linux condiviso in fase di runtime. Nel Listato 2.18 è mostrato un esempio di due app di sistema eseguite dall'utente `system` (UID 1000).

Listato 2.18 Applicazioni in esecuzione con uno UID condiviso (system).

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

Le applicazioni che sono parte di un utente condiviso possono essere eseguite nello stesso processo; inoltre, poiché dispongono già dello stesso UID Linux e possono accedere alle medesime risorse di sistema, in genere non richiedono ulteriori modifiche. È possibile richiedere un processo comune specificando lo stesso nome di processo nell'attributo `process` del tag `<application>` nei manifest di tutte le app che devono essere eseguite in un solo processo. Il risultato più ovvio è la condivisione della memoria tra le app e la comunicazione diretta (anziché mediante IPC); inoltre, alcuni servizi di sistema concedono un accesso speciale ai componenti in esecuzione nello stesso processo (per esempio l'accesso diretto alle password nella cache o la ricezione di token di autenticazione senza la visualizzazione dei prompt nell'interfaccia). Le applicazioni Google (come Play Services e il servizio di geolocalizzazione) sfruttano questa possibilità richiedendo l'esecuzione nello stesso processo del servizio di login Google, affinché sia possibile sincronizzare i dati in

background senza interagire con l'utente. Naturalmente queste applicazioni sono firmate con lo stesso certificato e sono parte dell'utente condiviso `com.google.uid.shared`.

Permessi personalizzati

I *permessi personalizzati* non sono altro che permessi dichiarati da applicazioni di terze parti. Una volta dichiarati, possono essere aggiunti ai componenti delle app per l'applicazione statica da parte del sistema; inoltre, l'applicazione può verificare dinamicamente se i chiamanti hanno ottenuto il permesso utilizzando i metodi `checkPermission()` o `enforcePermission()` della classe `Context`. Come nel caso dei permessi predefiniti, le applicazioni possono definire gruppi di permessi a cui aggiungere i permessi personalizzati. A titolo di esempio, il Listato 2.19 mostra la dichiarazione di un gruppo di permessi **(2)** e il permesso appartenente a tale gruppo **(3)**.

Listato 2.19 Dichiarazione della struttura dei permessi personalizzati, del gruppo di permessi e dei permessi.

```
<?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" >
    --altro codice--
    <permission-tree
        android:name="com.example.app.permission"
        android:label="@string/example_permission_tree_label" />(1)

    <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"/>(2)

    <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" />(3)
    --altro codice--
</manifest>
```

Come avviene per i permessi di sistema, se il livello di protezione è *normal* o *dangerous*, il permesso personalizzato viene concesso automaticamente quando l'utente seleziona *OK* nella finestra di conferma. Per controllare le applicazioni a cui viene concesso un permesso personalizzato è necessario dichiararlo con il livello di protezione *signature* per garantire che sia concesso solo alle applicazioni firmate con la medesima chiave.

NOTA

Il sistema può concedere un permesso solo se lo conosce; di conseguenza, l'applicazione che definisce permessi personalizzati deve essere installata prima delle applicazioni che fanno uso di tali permessi. Se un'applicazione richiede un permesso sconosciuto al sistema, il permesso viene ignorato e di conseguenza negato.

Le applicazioni possono inoltre aggiungere dinamicamente nuovi permessi utilizzando l'API `android.content.pm.PackageManager.addPermission()` e rimuoverli con l'API corrispondente `removePermission()`. I permessi aggiunti dinamicamente devono appartenere a una *struttura di permessi* definita dall'applicazione. Le applicazioni possono aggiungere o rimuovere permessi solamente da una struttura di permessi presente nel loro package o in un altro package in esecuzione con lo stesso user ID condiviso.

I nomi delle strutture di permessi utilizzano la notazione a dominio inverso e un permesso è considerato parte della relativa struttura se il suo nome è preceduto dal nome della struttura di permessi e da un punto (.). Per esempio, il permesso `com.example.app.permission.PERMISSION2` è un membro della struttura `com.example.app.permission` definita nel Listato 2.19 nel punto **(1)**. Nel Listato 2.20 è mostrata l'aggiunta di un permesso dinamico tramite codice di programma.

Listato 2.20 Aggiunta di un permesso dinamico tramite codice di programma.

```
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);
```

I permessi aggiunti dinamicamente sono aggiunti al database dei package (`/data/system/packages.xml`). Persistono dopo i riavvii, come i permessi definiti nel manifest, ma possiedono un attributo supplementare `type` impostato su `dynamic`.

Componenti pubblici e privati

I componenti definiti nel file `AndroidManifest.xml` possono essere *pubblici* o *privati*. I componenti privati possono essere chiamati solo dall'applicazione dichiarante, mentre quelli pubblici sono disponibili anche ad altre applicazioni.

Fatta eccezione per i content provider, tutti i componenti sono privati per impostazione predefinita. Visto che lo scopo dei content provider è condividere dati con altre applicazioni, questi erano pubblici di default, ma questo comportamento è cambiato in Android 4.2 (API livello 17). Le applicazioni per API livello 17 o versioni successive oggi ricevono content provider privati per impostazione predefinita; questi componenti restano invece pubblici (per la compatibilità con le versioni precedenti) quando l'app è destinata a un livello API inferiore.

I componenti possono essere resi pubblici impostando esplicitamente l'attributo `exported` su `true`, o dichiarando implicitamente un filtro intent. I componenti con un filtro intent che non necessitano di essere pubblici possono essere resi privati impostando l'attributo `exported` su `false`. Se un componente non viene esportato, le chiamate dalle applicazioni esterne vengono bloccate dall'activity manager, indipendentemente dai permessi concessi al processo chiamante (a meno che non sia in esecuzione come `root` o `system`). Il Listato 2.21 mostra come mantenere privato un componente impostandone l'attributo `exported` su `false`.

Listato 2.21 Configurazione di un componente privato impostando exported="false".

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

Tutti i componenti pubblici, tranne quelli esplicitamente destinati all'uso da parte del pubblico, dovrebbero essere protetti da un permesso personalizzato.

Permessi per activity e servizi

Le activity e i servizi possono essere protetti da un singolo permesso impostato con l'attributo `permission` del componente target. Il permesso dell'activity viene verificato quando le altre applicazioni chiamano `Context.startActivity()` o `Context.startActivityForResult()` con un intent che viene risolto nell'activity in questione. Per i servizi, il permesso viene verificato quando le altre applicazioni chiamano `Context.startService()`, `stopService()` o `bindService()` con un intent che viene risolto nel servizio.

A titolo di esempio, il Listato 2.22 mostra due permessi personalizzati, `START_MY_ACTIVITY` e `USE_MY_SERVICE`, impostati rispettivamente su un'activity **(1)** e un servizio **(2)**. Le applicazioni che intendono utilizzare questi componenti devono richiedere i permessi appropriati utilizzando il tag `<uses-permission>` nel loro manifest.

Listato 2.22 Protezione di activity e servizi con permessi personalizzati.

```
<?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" />

    --altro codice--
    <activity android:name=".MyActivity"
        android:label="@string/my_activity"
        android:permission="com.example.permission.START_MY_ACTIVITY">(1)
        <intent-filter>
        --altro codice--
       </intent-filter>
    </activity>
    <service android:name=".MyService"
        android:permission="com.example.permission.USE_MY_SERVICE">(2)
        <intent-filter>
        --altro codice--
        </intent-filter>
    </service>
    --altro codice--
</manifest>
```

Permessi per i broadcast

A differenza dei permessi per activity e servizi, i permessi per i broadcast receiver possono essere specificati sia dal receiver stesso sia dall'applicazione che trasmette il broadcast. Durante l'invio di un broadcast, le applicazioni possono utilizzare il metodo `Context.sendBroadcast(Intent intent)` per inviare un broadcast da recapitare a tutti i receiver registrati, oppure limitare l'ambito dei componenti che ricevono il broadcast utilizzando `Context.sendBroadcast(Intent intent, String receiverPermission)`. Il parametro `receiverPermission` specifica che il permesso che i receiver interessati devono possedere per ricevere il broadcast. In alternativa, a partire da Android 4.0, i sender possono utilizzare `Intent.setPackage(String packageName)` per limitare l'ambito ai receiver definiti nel package specificato. Sui dispositivi multiutente, le applicazioni di sistema con il permesso `INTERACT_ACROSS_USERS` possono inviare un broadcast che viene recapitato solamente a un utente specifico utilizzando i metodi `sendBroadcastAsUser(Intent intent, UserHandle user)` e `sendBroadcastAsUser(Intent intent, UserHandle user, String receiverPermission)`.

I receiver possono definire chi può inviare loro broadcast specificando un permesso con l'attributo `permission` del tag `<receiver>` nel manifest (per i receiver registrati in maniera statica) oppure passando il permesso richiesto al metodo `Context.registerReceiver(BroadcastReceiver receiver, IntentFilter filter, String broadcastPermission, Handler scheduler)` (per i receiver registrati dinamicamente).

Solo i broadcaster con il permesso richiesto potranno inviare un broadcast a tale receiver. Per esempio, le applicazioni di amministrazione del dispositivo che applicano policy di sicurezza a livello di sistema (l'amministrazione del dispositivo è presentata nel Capitolo 9) necessitano del permesso `BIND_DEVICE_ADMIN` per ricevere il broadcast `DEVICE_ADMIN_ENABLED`. Poiché si tratta di un permesso di sistema con livello di protezione *signature*, la richiesta del permesso garantisce che solo il sistema possa attivare le applicazioni di amministrazione del dispositivo.

Come esempio, il Listato 2.23 mostra in che modo l'applicazione Email predefinita di Android specifica il permesso `BIND_DEVICE_ADMIN` **(1)** per il suo receiver `PolicyAdmin`.

Listato 2.23 Specifica di un permesso per un broadcast receiver registrato in maniera statica.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.email"
    android:versionCode="500060" >
        --altro codice--
        <receiver
            android:name=".SecurityPolicy$PolicyAdmin"
            android:label="@string/device_admin_label"
            android:description="@string/device_admin_description"
            android:permission="android.permission.BIND_DEVICE_ADMIN" >(1)
            <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>
        --altro codice--
</manifest>
```

Come altri componenti, i broadcast receiver privati possono ricevere solamente i broadcast originati dalla stessa applicazione.

Permessi per i content provider

Come già spiegato nel paragrafo "Natura dei permessi", i content provider dispongono di un modello di permessi più complesso rispetto agli altri componenti; tale modello è descritto nei dettagli in questo paragrafo.

Permessi per i provider statici

Anche se è possibile specificare un singolo permesso che controlli l'accesso all'intero provider utilizzando l'attributo `permission`, la maggior parte dei provider utilizza permessi diversi per la lettura e la scrittura e può specificare anche permessi "per URI". Un provider che usa permessi diversi per la lettura e la scrittura è `ContactsProvider`; il Listato 2.24 mostra la dichiarazione della sua classe `ContactsProvider2`.

Listato 2.24 Dichiarazione dei permessi di ContactsProvider2.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.android.providers.contacts"
        android:sharedUserId="android.uid.shared"
        android:sharedUserLabel="@string/sharedUserLabel">
         --altro codice--
         <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"(1)
            android:writePermission="android.permission.WRITE_CONTACTS">(2)
            --altro codice--
            <path-permission
                    android:pathPattern="/contacts/.*/photo"
                    android:readPermission="android.permission.GLOBAL_SEARCH" />(3)
            <grant-uri-permission android:pathPattern=".*" />
        </provider>
        --altro codice--
</manifest>
```

Il provider usa l'attributo `readPermission` per specificare un permesso di lettura dei dati (`READ_CONTACTS` **(1)**) e un permesso separato per la scrittura di dati con l'attributo `writePermission` (`WRITE_CONTACTS`) **(2)**. Di conseguenza, le applicazioni che possiedono solo il permesso `READ_CONTACTS` possono chiamare soltanto il metodo `query()` del provider, mentre le chiamate a `insert()`, `update()` o `delete()` richiedono che il chiamante abbia il permesso

WRITE_CONTACTS. Le applicazioni che devono leggere e scrivere sul provider dei contatti necessitano di entrambi i permessi.

Se il permesso di lettura e scrittura globale non è abbastanza flessibile, i provider possono specificare permessi "per URI" al fine di proteggere un particolare sottoinsieme dei dati. I permessi per URI hanno una priorità superiore ai permessi a livello di componente (o ai permessi di lettura e scrittura, se specificati separatamente). In pratica, se un'applicazione vuole accedere all'URI di un content a cui è associato un permesso, deve detenere solamente il permesso dell'URI target e non il permesso a livello di componente. Nel Listato 2.24 ContactsProvider2 usa il tag <path-permission> per richiedere che le applicazioni che provano a leggere le foto dei contatti abbiano il permesso GLOBAL_SEARCH **(3)**. Visto che i permessi per URI sostituiscono il permesso di lettura globale, le applicazioni interessate non necessitano del permesso READ_CONTACTS. In pratica, il permesso GLOBAL_SEARCH è utilizzato per concedere al sistema di ricerca Android, il quale non può possedere i permessi di lettura per tutti i provider, l'accesso in sola lettura ad alcuni dati dei provider di sistema.

Permessi per i provider dinamici

Anche se i permessi per URI definiti in maniera statica possono essere piuttosto potenti, le applicazioni a volte devono concedere l'accesso temporaneo a un particolare dato (definito dal suo URI) alle altre app, senza richiedere che esse abbiano un permesso specifico. Per esempio, un'applicazione per e-mail o SMS potrebbe dover cooperare con un visualizzatore di immagini per mostrare un allegato. Poiché l'app non può conoscere anticipatamente gli URI degli allegati, l'uso dei permessi per URI statici imporrebbe di concedere al visualizzatore di immagini l'accesso in lettura a tutti gli allegati (un comportamento sicuramente non desiderabile).

Per evitare questa situazione e il relativo problema di sicurezza, le applicazioni possono concedere temporaneamente l'accesso per URI utilizzando il metodo `Context.grantUriPermission(String toPackage, Uri uri, int modeFlags)`

e revocare poi l'accesso con il metodo `revokeUriPermission(Uri uri, int modeFlags)`. L'accesso per URI temporaneo viene abilitato impostando l'attributo globale `grantUriPermissions` su `true` o aggiungendo un tag `<grant-uri-permission>` per abilitarlo solo per un URI specifico. Per esempio, il Listato 2.25 mostra come l'applicazione Email usa l'attributo `grantUriPermissions` **(1)** per consentire l'accesso temporaneo agli allegati senza richiedere il permesso `READ_ATTACHMENT`.

Listato 2.25 Dichiarazione AttachmentProvider dall'app Email.

```
<?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"(1)
           android:exported="true"
           android:readPermission="com.android.email.permission.READ_ATTACHMENT"/>
        --altro codice--
</manifest>
```

In pratica, le applicazioni usano raramente i metodi `Context.grantPermission()` e `revokePermission()` in maniera diretta per concedere l'accesso per URI; preferiscono invece impostare i flag `FLAG_GRANT_READ_URI_PERMISSION` o `FLAG_GRANT_WRITE_URI_PERMISSION` sull'intent usato per avviare l'applicazione collaborativa (nell'esempio il visualizzatore di immagini). Con l'impostazione di questi flag al destinatario dell'intent viene concesso il permesso di eseguire operazioni in lettura o scrittura sull'URI nei dati dell'intent.

A partire da Android 4.4 (API livello 19), le concessioni di accesso per URI possono essere rese persistenti tra i riavvii del dispositivo con il metodo `ContentResolver.takePersistableUriPermission()`, se per l'intent ricevuto è impostato il flag `FLAG_GRANT_PERSISTABLE_URI_PERMISSION`. Le concessioni persistenti sono inserite nel file `/data/system/urigrants.xml` e possono essere revocate chiamando il metodo `releasePersistableUriPermission()`. Le concessioni di accesso per URI, temporanee e persistenti, sono gestite dal servizio di sistema `ActivityManagerService`, chiamando internamente le API relative all'accesso per URI.

A partire da Android 4.1 (API livello 16), le applicazioni possono utilizzare la facility `ClipData` degli intent per aggiungere più di un URI di contenuto a cui concedere temporaneamente l'accesso (http://bit.ly/ZG3LhP). L'accesso per URI viene concesso con uno dei flag `FLAG_GRANT_*` dell'intent e viene revocato automaticamente al termine del task dell'applicazione chiamante; non è quindi necessario chiamare `revokePermission()`. Il Listato 2.26 mostra come l'applicazione Email crea un intent che avvia un visualizzatore di allegati.

Listato 2.26 Uso del flag FLAG_GRANT_READ_URI_PERMISSION per avviare un visualizzatore.

```
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;
}
```

Pending intent

I pending intent non sono né un componente di Android né un permesso, ma poiché consentono a un'applicazione di concedere i suoi permessi a un'altra applicazione è utile affrontarli in questo capitolo.

I pending intent incapsulano un intent e un'azione target da eseguire con esso (avviare un'activity, inviare un broadcast e così via). La differenza principale rispetto agli intent "normali" è che i pending intent includono anche l'identità delle applicazioni che li hanno creati. In questo modo possono essere passati ad altre applicazioni, che li usano per eseguire l'azione specificata con l'identità e i permessi dell'applicazione originale. L'identità memorizzata nei pending intent è garantita dal servizio di sistema `ActivityManagerService`, che tiene traccia dei pending intent attualmente attivi.

I pending intent sono usati per implementare allarmi e notifiche in Android: questi consentono a qualunque applicazione di specificare un'azione da eseguire per suo conto, sia a un orario specificato (per gli allarmi) sia quando l'utente interagisce con una notifica di sistema. Gli allarmi e le notifiche possono essere attivati quando l'applicazione che li ha creati non è più in esecuzione e il sistema usa le informazioni nel pending intent per avviarla ed eseguire l'azione dell'intent per suo conto. Il Listato 2.27 mostra come l'applicazione Email usa un pending intent creato con `PendingIntent.getBroadcast()` **(1)** per pianificare i broadcast che attivano la sincronizzazione e-mail.

Listato 2.27 Uso di un pending intent per pianificare un allarme.

```
private void setAlarm(long id, long millis) {
    --altro codice--
    Intent i = new Intent(this, MailboxAlarmReceiver.class);
    i.putExtra("mailbox", id);
    i.setData(Uri.parse("Box" + id));
    pi = PendingIntent.getBroadcast(this, 0, i, 0);(1)
    mPendingIntents.put(id, pi);
    AlarmManager am =
            (AlarmManager)getSystemService(Context.ALARM_SERVICE);
    m.set(AlarmManager.RTC_WAKEUP,
          System.currentTimeMillis() + millis, pi);
    --altro codice--
}
```

I pending intent possono essere passati anche ad applicazioni non di sistema. Valgono le stesse regole: le applicazioni che ricevono un'istanza di `PendingIntent` possono eseguire l'operazione specificata con gli stessi permessi e la stessa identità delle applicazioni creatrici. Di conseguenza, è necessario prestare attenzione durante la creazione dell'intent di base, che in genere dovrebbe essere il più possibile specifico (con il nome del componente dichiarato in maniera esplicita) per garantire che sia ricevuto dai componenti previsti.

L'implementazione dei pending intent è piuttosto complessa, ma si basa sugli stessi principi di IPC e sandbox su cui si basano gli altri componenti di Android. Quando un'applicazione crea un pending intent, il sistema ne recupera UID e PID con `Binder.getCallingUid()` e `Binder.getCallingPid()`. Grazie a queste informazioni il sistema recupera il nome del package e lo user ID (sui dispositivi multiutente) dell'autore e li memorizza in un `PendingIntentRecord` insieme all'intent di base e a eventuali metadati aggiuntivi. L'activity manager mantiene un elenco dei pending intent attivi memorizzando i `PendingIntentRecord` corrispondenti e, all'attivazione, recupera il record necessario. Usa poi le informazioni nel record per assumere l'identità dell'autore del pending intent ed eseguire l'azione specificata. Da quel momento, il processo è lo stesso di quando si avvia un componente Android e si esegue la verifica dei permessi.

Riepilogo

Android esegue ogni applicazione in una sandbox limitata e richiede che le applicazioni chiedano permessi specifici per interagire con altre app o con il sistema. I permessi sono stringhe che denotano la capacità di eseguire una specifica operazione; vengono concessi in fase di installazione dell'applicazione e, fatta eccezione per i permessi di sviluppo, restano fissi per tutta la durata di un'applicazione. Possono essere mappati a group ID supplementari di Linux, verificati dal kernel prima di concedere l'accesso alle risorse di sistema.

I servizi di sistema di livello più alto applicano i permessi ottenendo l'UID dell'applicazione chiamante tramite Binder e ricercano i permessi detenuti nel database del package manager. I permessi associati a un componente dichiarati nel file manifest di un'applicazione vengono applicati automaticamente dal sistema, ma le applicazioni possono decidere di effettuare altri controlli sui permessi in maniera dinamica. Oltre a utilizzare i permessi predefiniti, le applicazioni possono definire permessi personalizzati e associarli ai componenti per controllare l'accesso.

Ogni componente di Android può richiedere un permesso; in più, i content provider possono specificare permessi di lettura e scrittura o permessi per URI. I pending intent incapsulano l'identità dell'applicazione che li ha creati, oltre che un intent e un'azione da eseguire; in questo modo il sistema o un'applicazione di terze parti può eseguire azioni per conto delle applicazioni originali con la stessa identità e gli stessi permessi.

Gestione dei package

In questo capitolo parleremo in maniera approfondita della gestione dei package in Android. Inizieremo con una descrizione del loro formato e dell'implementazione della firma del codice, dopodiché vedremo nei dettagli il processo di installazione dell'APK. A seguire, esamineremo il supporto di Android per gli APK crittografati e i contenitori di applicazioni sicuri, utilizzati per implementare una forma di DRM per le applicazioni a pagamento. Per finire, descriveremo il meccanismo di verifica dei package di Android e la sua implementazione più utilizzata: il servizio di verifica delle applicazioni di Google Play.

Formato dei package di applicazione Android

Le applicazioni Android sono distribuite e installate nella forma di package applicativi, solitamente chiamati *file APK* (*Application Package*). I file APK sono file contenitori che includono sia il codice dell'applicazione sia le risorse, nonché il file manifest dell'applicazione; possono inoltre contenere una firma del codice. Il formato APK è un'estensione del formato Java JAR (`http://bit.ly/11rmJtR`), che a sua volta è un'estensione del famoso formato di file ZIP. I file APK hanno generalmente l'estensione `.apk` e sono associati al tipo MIME *application/vnd.android.package-archive*.

Visto che i file APK non sono altro che file ZIP, possiamo esaminarne facilmente il contenuto estraendoli con una qualunque utility di decompressione che supporta il formato ZIP. Nel Listato 3.1 è mostrato il contenuto di un tipico file APK dopo la sua estrazione.

Listato 3.1 Contenuto di un tipico file APK.

```
apk/
|-- AndroidManifest.xml(1)
|-- classes.dex(2)
|-- resources.arsc(3)
|-- assets/(4)
|-- lib/(5)
|   |-- armeabi/
|   |   '-- libapp.so
|   '-- armeabi-v7a/
|       '-- libapp.so
|-- META-INF/(6)
|   |-- CERT.RSA
|   |-- CERT.SF
|   '-- MANIFEST.MF
'-- res/(7)
    |-- anim/
    |-- color/
    |-- drawable/
    |-- layout/
    |-- menu/
    |-- raw/
    '-- xml/
```

Ogni file APK include un file `AndroidManifest.xml` **(1)** che dichiara il nome del package, la versione, i componenti e altri metadati dell'applicazione. Il file `classes.dex` **(2)** contiene il codice eseguibile dell'applicazione ed è nel formato DEX nativo di Dalvik VM. `resources.arsc` **(3)** riunisce tutte le risorse

compilate dell'applicazione, quali stringhe e stili. La directory *assets* **(4)** è utilizzata per contenere i file degli asset non elaborati dell'applicazione, quali font e file musicali.

Le applicazioni che sfruttano le librerie native tramite JNI contengono una directory *lib* **(5)**, con sottodirectory per ogni architettura di piattaforma supportata. Le risorse referenziate direttamente dal codice Android, sia in via diretta con la classe `android.content.res.Resources` sia indirettamente tramite API di livello superiore, sono conservate nella directory *res* **(7)**, con sottodirectory separate per ogni tipo di risorsa (animazioni, immagini, definizioni di menu e così via). Come i file JAR, i file APK contengono anche una directory *META-INF* **(6)** che ospita il file manifest del package e le firme del codice. Il contenuto di questa directory è descritto nel prossimo paragrafo.

Firma del codice

Come abbiamo imparato nel Capitolo 2, Android usa la firma del codice APK, e in particolare il certificato di firma APK, per controllare a quali applicazioni concedere i permessi con il livello di protezione *signature*. Il certificato di firma APK è utilizzato anche per vari controlli durante il processo di installazione dell'applicazione, quindi prima di entrare nei dettagli dell'installazione APK è opportuno acquisire dimestichezza con la firma del codice in Android. In questo paragrafo sono disponibili i dettagli sulla firma del codice Java in generale, con le differenze rispetto all'implementazione di Android bene in evidenza.

Per iniziare, parliamo della firma del codice in generale. Perché mai qualcuno vorrebbe firmare il codice? I motivi sono sempre gli stessi: integrità e autenticità. Prima di eseguire programmi di terze parti, volete essere certi che non siano stati manomessi (integrità) e che siano effettivamente creati dall'entità che sostiene di metterli a disposizione (autenticità). Queste caratteristiche sono solitamente implementate da uno schema di firma digitale, che garantisce che solo l'entità in possesso della chiave di firma possa generare una firma del codice valida.

Il processo di verifica si accerta che il codice non sia stato manomesso e che la firma sia stata prodotta con la chiave prevista. Tuttavia, esiste un problema che non può essere risolto direttamente con la firma del codice: stabilire se il firmatario (ovvero l'autore del software) può essere considerato degno di fiducia. Il modo tradizionale per stabilire l'attendibilità consiste nel richiedere che il firmatario possieda un certificato digitale e lo alleghi al codice firmato. I verifier decidono se considerare attendibile il certificato in base a un modello di attendibilità (come PKI o la *web of trust*) oppure procedendo caso per caso.

Un altro problema che la firma del codice non tenta nemmeno di risolvere è stabilire se il codice firmato può essere eseguito in sicurezza. Come hanno dimostrato Flame e altri malware firmati a livello di codice (http://bit.ly/ZG3TOq), anche quello che sembra essere firmato da una terza parte attendibile non è necessariamente sicuro.

Firma del codice Java

La firma del codice Java viene eseguita a livello di file JAR e riutilizza ed estende i file manifest JAR per aggiungere una firma all'archivio JAR. Il file manifest JAR principale (`MANIFEST.MF`) contiene voci con il nome file e il valore digest di ogni file nell'archivio. Per esempio, nel Listato 3.2 è mostrato l'inizio del file manifest JAR di un tipico file APK (per tutti gli esempi di questo paragrafo vengono utilizzati file APK anziché JAR).

Listato 3.2 Estratto del file manifest JAR.

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

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

Name: res/menu/main.xml
SHA1-Digest: kG8WDil9ur0f+F2AxgcSSKDhjn0=

Name: ...
```

Implementazione

La firma del codice Java viene implementata aggiungendo un altro file manifest chiamato *file della firma* (con estensione `.SF`), che contiene i dati da firmare e la relativa firma digitale. La firma digitale è chiamata *file del blocco della firma* ed è salvata nell'archivio come file binario con una delle estensioni `.RSA`, `.DSA` o `.EC`, in base all'algoritmo di firma utilizzato. Come mostrato nel Listato 3.3, il file della firma è molto simile al manifest.

Listato 3.3 Estratto del file della firma JAR.

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: ZKXxNW/3Rg7JA1r0+RlbJIP6IMA=
Created-By: 1.7.0_51 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: zb0XjEhVBxE0z2ZC+B4OW25WBxo=(1)

Name: res/drawable-xhdpi/ic_launcher.png
SHA1-Digest: jTeE2Y5L3uBdQ2g40PB2n72L3dE=(2)

Name: res/menu/main.xml
SHA1-Digest: kSQDLtTE07cLhTH/cY54UjbbNBo=(3)

Name: ...
```

Il file della firma contiene il digest dell'intero file manifest (`SHA1-Digest-Manifest` **(1)**), nonché i digest per ogni voce in `MANIFEST.MF` (**(2)** e **(3)**). SHA-1 era l'algoritmo di digest predefinito fino a Java 6, mentre Java 7 e

versioni successive possono generare digest di file e manifest con gli algoritmi hash SHA-256 e SHA-512; in questo caso gli attributi digest diventano rispettivamente *SHA-256-Digest* e *SHA-512-Digest*. A partire dalla versione 4.3, Android supporta i digest SHA-256 e SHA-512.

I digest nel file della firma possono essere verificati facilmente utilizzando i comandi OpenSSL, come mostrato nel Listato 3.4.

Listato 3.4 Verifica dei digest nel file della firma JAR tramite OpenSSL.

```
$ openssl sha1 -binary MANIFEST.MF |openssl base641
zb0XjEhVBxE0z2ZC+B4OW25WBxo=
$ echo -en "Name: res/drawable-xhdpi/ic_launcher.png\r\nSHA1-Digest: \
K/0Rd/lt0qSlgDD/9DY7aCNlBvU=\r\n\r\n"|openssl sha1 -binary |openssl base642
jTeE2Y5L3uBdQ2g40PB2n72L3dE=
```

Il primo comando **(1)** recupera il digest SHA-1 dell'intero file manifest e lo codifica in Base64 per produrre il valore `SHA1-Digest-Manifest`. Il secondo comando **(2)** simula il calcolo del digest di una singola voce del manifest e dimostra il formato di canonicalizzazione degli attributi richiesto dalla specifica JAR.

La firma digitale vera e propria è nel formato binario PKCS#7 (o, più in generale, CMS) e include il valore della firma e il certificato di firma (`http://bit.ly/1nq17bc`; Housley, *RFC 5652 – Cryptographic Message Syntax (CMS)*, `http://tools.ietf.org/html/rfc5652`). I file del blocco della firma prodotti con l'algoritmo RSA sono salvati con l'estensione `.RSA`, mentre quelli generati con chiavi DSA o EC sono salvati con le estensioni `.DSA` o `.EC`. È possibile inoltre eseguire più firme, che generano più file `.SF` e `.RSA/DSA/EC` nella directory *META-INF* del file JAR.

Il formato CMS consente la firma e la crittografia ricorrendo ad algoritmi e parametri diversi; può inoltre essere esteso tramite attributi personalizzati firmati e non firmati. Una discussione approfondita va oltre gli scopi di questo capitolo (consultate RFC 5652 per i dettagli su CMS), ma nell'ambito della firma JAR una struttura CMS contiene fondamentalmente l'algoritmo di digest, il certificato di firma e il valore della firma. Le specifiche CMS consentono l'inclusione di dati firmati nella struttura CMS `SignedData` (una variazione del formato chiamata *firma collegata*), ma le firme JAR non li includono. Se i dati firmati non sono inclusi nella struttura CMS, la firma è detta *firma scollegata* e i verifier

devono avere una copia dei dati firmati originali per effettuare la verifica. Nel Listato 3.5 è mostrato un file del blocco della firma RSA sottoposto a parsing in ASN.1, con i dettagli del certificato tagliati (l'*Abstract Syntax Notation One*, o ASN.1, è una notazione standard che descrive regole e strutture per la codifica dei dati nelle telecomunicazioni e nelle reti di computer; è ampiamente utilizzata negli standard di crittografia per definire la struttura degli oggetti crittografici).

Listato 3.5 Contenuto del blocco della firma di un file JAR.

```
$ openssl asn1parse -i -inform DER -in CERT.RSA
    0:d=0  hl=4 l= 888 cons: SEQUENCE
    4:d=1  hl=2 l=   9 prim:  OBJECT            :pkcs7-signedData(1)
   15:d=1  hl=4 l= 873 cons:  cont [ 0 ]
   19:d=2  hl=4 l= 869 cons:   SEQUENCE
   23:d=3  hl=2 l=   1 prim:    INTEGER           :01(2)
   26:d=3  hl=2 l=  11 cons:    SET
   28:d=4  hl=2 l=   9 cons:     SEQUENCE
   30:d=5  hl=2 l=   5 prim:      OBJECT            :sha1(3)
   37:d=5  hl=2 l=   0 prim:      NULL
   39:d=3  hl=2 l=  11 cons:    SEQUENCE
   41:d=4  hl=2 l=   9 prim:     OBJECT            :pkcs7-data(4)
   52:d=3  hl=4 l= 607 cons:    cont [ 0 ](5)
   56:d=4  hl=4 l= 603 cons:     SEQUENCE
   60:d=5  hl=4 l= 452 cons:      SEQUENCE
   64:d=6  hl=2 l=   3 cons:       cont [ 0 ]
   66:d=7  hl=2 l=   1 prim:        INTEGER           :02
   69:d=6  hl=2 l=   1 prim:       INTEGER           :04
   72:d=6  hl=2 l=  13 cons:       SEQUENCE
   74:d=7  hl=2 l=   9 prim:        OBJECT            :sha1WithRSAEncryption
   85:d=7  hl=2 l=   0 prim:        NULL
   87:d=6  hl=2 l=  56 cons:       SEQUENCE
   89:d=7  hl=2 l=  11 cons:        SET
   91:d=8  hl=2 l=   9 cons:         SEQUENCE
   93:d=9  hl=2 l=   3 prim:          OBJECT            :countryName
   98:d=9  hl=2 l=   2 prim:          PRINTABLESTRING   :JP
  --altro codice--
  735:d=5  hl=2 l=   9 cons:      SEQUENCE
  737:d=6  hl=2 l=   5 prim:       OBJECT            :sha1(6)
  744:d=6  hl=2 l=   0 prim:       NULL
  746:d=5  hl=2 l=  13 cons:      SEQUENCE
  748:d=6  hl=2 l=   9 prim:       OBJECT            :rsaEncryption(7)
  759:d=6  hl=2 l=   0 prim:       NULL
  761:d=5  hl=3 l= 128 prim:      OCTET STRING      [HEX DUMP]:892744D30DCEDF74933007...(8)
```

Il blocco della firma contiene un identificatore di oggetto **(1)** che descrive il tipo di dati (oggetto ASN.1) che segue `SignedData` e i dati stessi. L'oggetto `SignedData` incluso contiene una versione **(2)** (1); un set di identificatori dell'algoritmo hash in uso **(3)** (solo uno per un singolo firmatario, SHA-1 in questo esempio); il tipo di dati firmati **(4)** (`pkcs7-data`, che significa semplicemente "dati binari arbitrari"); il set dei certificati di firma **(5)**; una o più (una per ogni firmatario) strutture `SignerInfo` che incapsulano il valore della firma (non mostrate per intero nel Listato 3.5). `SignerInfo` contiene una versione; un oggetto `SignerIdentifier`, che di solito

contiene il DN dell'autorità di certificazione e il numero di serie del certificato (non mostrato); l'algoritmo digest usato **(6)** (SHA-1, incluso in **(3)**); l'algoritmo di crittografia dei digest usato per generare il valore della firma **(7)**; il digest crittografato stesso (valore della firma) **(8)**.

Gli elementi più importanti della struttura `SignedData`, almeno nell'ambito delle firme JAR e APK, sono il set dei certificati di firma **(5)** e il valore della firma **(8)** (o i valori, in presenza di più firmatari).

Se estraiamo il contenuto di un file JAR, possiamo utilizzare il comando OpenSSL `smime` per verificare la firma specificando il file della firma come contenuto o dati firmati. Il comando `smime` visualizza i dati firmati e il risultato della verifica, come mostrato nel Listato 3.6.

Listato 3.6 Verifica del blocco della firma di un file JAR.

```
$ openssl smime -verify -in CERT.RSA -inform DER -content CERT.SF signing-cert.pem
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: ZKXxNW/3Rg7JA1r0+RlbJIP6IMA=
Created-By: 1.7.0_51 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: zb0XjEhVBxE0z2ZC+B4OW25WBxo=

Name: res/drawable-xhdpi/ic_launcher.png
SHA1-Digest: jTeE2Y5L3uBdQ2g40PB2n72L3dE=

--altro codice--
Verification successful
```

Firma del file JAR

Gli strumenti JDK ufficiali per la firma e la verifica di file JAR sono i comandi `jarsigner` e `keytool`. A partire da Java 5.0 `jarsigner` supporta anche il timestamping della firma da parte di un'autorità di timestamping (TSA), utile quando occorre stabilire se una firma è stata prodotta prima o dopo la scadenza del certificato di firma. Questa funzionalità non è tuttavia molto diffusa e non è supportata in Android.

Un file JAR viene firmato utilizzando il comando `jarsigner`, specificando un file keystore (Capitolo 5), l'alias della chiave da usare per la firma (i primi otto caratteri dell'alias diventano il nome di base per il file del blocco della firma, a meno che non sia specificata l'opzione `-sigfile`) e facoltativamente un algoritmo di firma. Osservate il punto **(1)** nel Listato 3.7 per un esempio di chiamata di `jarsigner`.

NOTA

A partire da Java 7, l'algoritmo predefinito è diventato SHA256withRSA, quindi dovete specificare esplicitamente se volete usare SHA-1 per la compatibilità con le versioni precedenti. Le firme basate su SHA-256 e SHA-512 sono supportate da Android 4.3.

Listato 3.7 Firma di un file APK e verifica della firma con il comando jarsigner.

```
$ jarsigner -keystore debug.keystore -sigalg SHA1withRSA test.apk androiddebugkey (1)
$ jarsigner -keystore debug.keystore -verify -verbose -certs test.apk (2)
--altro codice--

smk      965 Sat Mar 08 23:55:34 JST 2014 res/drawable-xxhdpi/ic_launcher.png

      X.509, CN=Android Debug, O=Android, C=US (androiddebugkey)(3)
      [certificate is valid from 6/18/11 7:31 PM to 6/10/41 7:31 PM]

smk   458072 Sun Mar 09 01:16:18 JST 2013 classes.dex

      X.509, CN=Android Debug, O=Android, C=US (androiddebugkey)(4)
      [certificate is valid from 6/18/11 7:31 PM to 6/10/41 7:31 PM]

         903 Sun Mar 09 01:16:18 JST 2014 META-INF/MANIFEST.MF
         956 Sun Mar 09 01:16:18 JST 2014 META-INF/CERT.SF
         776 Sun Mar 09 01:16:18 JST 2014 META-INF/CERT.RSA

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.
```

Lo strumento `jarsigner` può utilizzare tutti i tipi di keystore supportati dalla piattaforma, nonché i keystore non supportati nativamente e che richiedono un provider JCA dedicato, come quelli supportati da una smart card, da HSM o da un altro dispositivo hardware. Il tipo di store da usare per la firma viene specificato con l'opzione `-storetype`, il nome e la classe del provider con le opzioni `-providerName` e `-providerClass`. Le versioni più recenti dello strumento `signapk` specifico per Android (descritto nel paragrafo "Strumenti di firma del codice Android" in questo capitolo) supportano anche l'opzione `-providerClass`.

Verifica del file JAR

La verifica del file JAR viene eseguita con il comando `jarsigner` specificando l'opzione `-verify`. Il secondo comando `jarsigner` al punto **(2)** del Listato 3.7 verifica per prima cosa il blocco della firma e il certificato di firma, assicurando che il file della firma non sia stato manomesso. A seguire verifica che ogni digest nel file della firma (`CERT.SF`) corrisponda alla sezione relativa nel file manifest (`MANIFEST.MF`). Il numero di voci nel file della firma non deve corrispondere a quello nel file manifest. È possibile

aggiungere file a un JAR firmato senza invalidarne la firma: finché i file originali restano invariati, la verifica ha esito positivo.

Infine, `jarsigner` legge ogni voce del manifest e controlla che il digest del file corrisponda al contenuto effettivo del file. Se è stato specificato un keystore con l'opzione `-keystore` (come nell'esempio), `jarsigner` controlla anche se il certificato di firma è presente nel keystore specificato. A partire da Java 7 è disponibile una nuova opzione `-strict` che consente ulteriori convalide del certificato, tra cui un controllo della validità temporale e una verifica della catena di certificati. Gli errori di convalida sono trattati come avvisi e sono riportati nel codice in output del comando `jarsigner`.

Visualizzazione o estrazione delle informazioni sul firmatario

Come potete osservare nel Listato 3.7, per impostazione predefinita `jarsigner` visualizza i dettagli del certificato per ogni voce (**(3)** e **(4)**) anche se sono gli stessi per tutte le voci. Un metodo migliore per visualizzare le informazioni sul firmatario durante l'uso di Java 7 consiste nello specificare le opzioni `-verbose:summary` o `-verbose:grouped` oppure, in alternativa, nell'usare il comando `keytool` come illustrato nel Listato 3.8.

Listato 3.8 Visualizzazione delle informazioni sul firmatario di un file APK con il comando keytool.

```
$ keytool -list -printcert -jarfile test.apk
Signer #1:
Signature:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4dfc7e9a
Valid from: Sat Jun 18 19:31:54 JST 2011 until: Mon Jun 10 19:31:54 JST 2041
Certificate fingerprints:
         MD5:  E8:93:6E:43:99:61:C8:37:E1:30:36:14:CF:71:C2:32
         SHA1: 08:53:74:41:50:26:07:E7:8F:A5:5F:56:4B:11:62:52:06:54:83:BE
         Signature algorithm name: SHA1withRSA
         Version: 3
```

Una volta individuato il nome file del blocco della firma (elencando per esempio il contenuto dell'archivio), potete utilizzare OpenSSL con il comando `unzip` per estrarre facilmente il certificato di firma in un file, come mostrato nel Listato 3.9. Se la struttura `SignedData` include più di un certificato, saranno estratti tutti i certificati; in tal caso, dovrete eseguire il

parsing della struttura `SignedInfo` per individuare l'identificatore del certificato di firma effettivo.

Listato 3.9 Estrazione del certificato di firma APK con i comandi unzip e pkcs7 di OpenSSL.

```
$ unzip -q -c test.apk META-INF/CERT.RSA|openssl pkcs7 -inform DER -print_certs -out cert.pem
```

Firma del codice Android

La firma del codice Android si basa sulla firma dei JAR di Java, pertanto utilizza la crittografia a chiave pubblica e i certificati X.509 come molti schemi di firma del codice; tuttavia, le somiglianze finiscono qui.

In tutte le altre piattaforme che usano la firma del codice (come Java ME e Windows Phone), i certificati di firma del codice devono essere rilasciati da una CA considerata attendibile dalla piattaforma. Anche se esistono molte CA che rilasciano certificati di firma del codice, può essere piuttosto difficile ottenerne uno considerato attendibile da tutti i dispositivi target. Android risolve il problema in maniera semplice, ovvero disinteressandosi del contenuto o del firmatario del certificato di firma. Visto che non serve che siano rilasciati da una CA, praticamente tutti i certificati del codice usati in Android sono autofirmati. Inoltre, non dovete verificare la vostra identità in alcun modo: potete usare quello che volete come nome del soggetto (Google Play Store effettua alcuni controlli per eliminare i nomi comuni, ma il sistema operativo Android in sé non esegue verifiche). Android tratta i certificati di firma come blob binari e il fatto che siano nel formato X.509 è soltanto una conseguenza dell'uso del formato JAR.

Android non convalida i certificati con le modalità di PKI (vedete il Capitolo 6). In effetti, se un certificato non è autofirmato, il certificato della CA firmataria non deve essere presente o attendibile; Android installa senza problemi le app con un certificato di firma scaduto. Se avete un background PKI tradizionale, questa potrebbe sembrarvi un'eresia; dovete però ricordare che Android non usa PKI per la firma del codice, ma adotta solamente i suoi formati per la firma e i certificati.

Un'altra differenza tra la firma in Android e quella "standard" dei JAR riguarda la necessità di firmare tutte le voci dell'APK con lo stesso set di

certificati. Il formato JAR permette che ogni file sia firmato da un firmatario diverso e consente le voci non firmate. Questo ha senso nella sandbox e nel meccanismo di controllo di accesso di Java, progettato in origine per le applet, perché tale modello definisce un'*origine del codice* come una combinazione di certificato del firmatario e URL di origine del codice. Tuttavia, Android assegna i firmatari per APK (solitamente uno solo, ma sono supportati anche più firmatari) e non permette firmatari diversi per voci diverse del file APK.

Il modello di firma del codice di Android, unito all'interfaccia scadente della classe `java.util.jar.JarFile` (che non è una valida astrazione delle complessità del formato di firma CMS sottostante), rende piuttosto difficile verificare correttamente la firma dei file APK. Anche se Android si occupa sia di verificare l'integrità del file APK sia di garantire che tutte le voci del file APK siano state firmate con lo stesso set di certificati aggiungendo ulteriori controlli del certificato di firma alle routine di parsing del package, è evidente che il formato di file JAR non è la scelta migliore per la firma del codice Android.

Strumenti di firma del codice Android

Come dimostrato negli esempi del paragrafo "Firma del codice Java", potete usare i normali strumenti di firma del codice JDK per firmare o verificare i file APK. Oltre a questi strumenti, la directory *build/* di AOSP ne contiene uno specifico per Android chiamato `signapk`, che esegue praticamente la stessa operazione di `jarsigner` nella modalità di firma, ma con alcune differenze degne di nota. Se `jarsigner` richiede che le chiavi siano salvate in un file keystore compatibile, `signapk` accetta in input una chiave di firma separata (nel formato codificato DER PKCS#8, `http://bit.ly/1wLNenp`) e un file di certificato (nel formato codificato DER X.509). Il vantaggio del formato PKCS#8, ovvero il formato di codifica delle chiavi standard in Java, è l'inclusione di un identificatore di algoritmo esplicito che descrive il tipo di chiave privata codificata. Questa potrebbe includere il materiale della chiave, possibilmente

crittografato, oppure solo un riferimento, come un ID chiave, a una chiave memorizzata in un dispositivo hardware.

A partire da Android 4.4, `signapk` può produrre firme solamente con i meccanismi SHA1withRSA o SHA256withRSA (aggiunto alla piattaforma in Android 4.3). Al momento della scrittura di questo libro, la versione di `signapk` presente nel ramo principale di AOSP era stata estesa per supportare le firme ECDSA.

Per quanto le chiavi private non elaborate nel formato PKCS#8 siano rare, è possibile generare facilmente una coppia di chiavi di test e un certificato autofirmato utilizzando lo script `make_key` in *development/tools/*. Se disponete già di chiavi OpenSSL, dovrete prima convertirle nel formato PKCS#8 utilizzando un metodo simile al comando OpenSSL `pkcs8` mostrato nel Listato 3.10.

Listato 3.10 Conversione di una chiave OpenSSL nel formato PKCS#8.

```
$ echo "keypwd"|openssl pkcs8 -in mykey.pem -topk8 -outform DER -out mykey.pk8 -passout stdin
```

Una volta ottenute le chiavi necessarie, potete firmare un APK utilizzando `signapk`, come mostrato nel Listato 3.11.

Listato 3.11 Firma di un file APK con lo strumento signapk.

```
$ java -jar signapk.jar cert.cer key.pk8 test.apk test-signed.apk
```

Firma del codice dei file OTA

Oltre alla sua modalità predefinita di firma degli APK, lo strumento `signapk` dispone di una modalità "firma intero file" che può essere abilitata con l'opzione `-w`. In questa modalità, oltre a firmare ogni singola voce JAR, lo strumento genera una firma per l'intero archivio. La modalità non è supportata da `jarsigner` ed è specifica per Android.

Perché firmare l'intero archivio se abbiamo già firmato ogni file? Per supportare gli aggiornamenti OTA (*over-the-air*). I package OTA sono file ZIP in un formato simile ai file JAR contenenti file aggiornati e gli script per applicarli. I package includono una directory *META-INF/*, manifest, un blocco della firma e qualche file supplementare, come *META-INF/com/android/otacert* che contiene il certificato di firma di

aggiornamento (in formato PEM). Prima dell'avvio nel recovery per applicare gli aggiornamenti, Android verifica la firma del package e controlla se il certificato di firma è attendibile per la firma degli aggiornamenti. I certificati attendibili per OTA sono separati dal "normale" archivio di attendibilità del sistema (consultate il Capitolo 6) e risiedono in un file ZIP solitamente salvato come `/system/etc/security/otacerts.zip`. In un dispositivo di produzione, questo file tipicamente contiene un singolo file, di solito chiamato `releasekey.x509.pem`. Dopo il riavvio del dispositivo, il sistema operativo di recovery verifica ancora una volta la firma del package OTA prima di applicarlo, al fine di garantire che il file OTA non sia stato nel frattempo manomesso.

Se i file OTA sono come i file JAR e i file JAR non supportano la firma dell'intero file, dove va a finire la firma? Lo strumento Android `signapk` usa in maniera leggermente impropria il formato ZIP aggiungendo un commento stringa terminato da null nella sezione dei commenti del file ZIP, seguito dal blocco della firma binario e da un record finale di 6 byte contenente l'offset della firma e le dimensioni dell'intera sezione dei commenti. L'aggiunta del record di offset alla fine del file facilita la verifica del package attraverso una prima lettura e verifica del blocco della firma nella parte finale del file; il resto del file (che può essere nell'ordine delle centinaia di megabyte) viene letto solo se la firma è stata verificata.

Processo di installazione dei file APK

Esistono alcuni metodi per installare le applicazioni Android.

- Tramite il client di uno store di applicazioni (come Google Play Store): questa è la modalità con cui la maggior parte degli utenti installa le applicazioni.
- Direttamente sul dispositivo aprendo i file delle app scaricate (se l'opzione per le "origini sconosciute" è abilitata nelle impostazioni di sistema): questo metodo è solitamente detto *sideloading* di un'app.
- Da un computer connesso tramite USB con il comando `adb install` dell'SDK di Android, che a sua volta chiama l'utility a riga di comando `pm` con il parametro `install`. Questo metodo è usato principalmente dagli sviluppatori di applicazioni.
- Copiando direttamente un file APK in una delle directory delle applicazioni di sistema utilizzando la shell di Android. Visto che le directory delle applicazioni non sono accessibili nelle build di produzione, questo metodo può essere utilizzato solo su dispositivi che eseguono una build di sviluppo.

Quando un file APK viene copiato direttamente in una delle directory delle applicazioni, viene rilevato e installato automaticamente dal package manager, che monitora le variazioni in queste directory. Per tutti gli altri metodi di installazione, l'installer (che si tratti del client Google Play Store, dell'attività di installazione package predefinita del sistema, del comando `pm` o di altro) chiama uno dei metodi `installPackage()` del package manager di sistema, che quindi copia l'APK in una delle directory delle applicazioni e lo installa. Nei paragrafi che seguono vedremo le tappe principali del processo di installazione dei package Android, esaminando i passaggi più complessi, come la creazione di un contenitore crittografato e la verifica del package.

La funzionalità di gestione dei package Android è distribuita su diversi componenti di sistema che interagiscono durante l'installazione del package, come mostrato nella Figura 3.1. Le frecce continue nella figura

rappresentano le dipendenze tra i componenti, nonché le chiamate di funzione. Le frecce tratteggiate puntano a file o directory monitorati (per rilevare le variazioni) da un componente, ma che non sono modificati direttamente da tale componente.

Posizione di dati e package delle applicazioni

Nel Capitolo 1 abbiamo affermato che Android fa distinzione tra applicazioni installate dal sistema e applicazioni installate dall'utente. Le applicazioni di sistema si trovano nella partizione di sola lettura *system* (in basso a sinistra nella Figura 3.1) e non possono essere modificate o disinstallate sui dispositivi di produzione. Le applicazioni di sistema sono quindi considerate attendibili, ricevono più privilegi e sono sottoposte a controlli della firma meno rigorosi. La maggior parte delle applicazioni di sistema si trova nella directory */system/app/*, mentre */system/priv-app/* contiene le app privilegiate a cui si possono concedere permessi con livello di protezione *signatureOrSystem* (come spiegato nel Capitolo 2). La directory */system/vendor/app/* ospita le applicazioni specifiche per i vari vendor. Le applicazioni installate dall'utente si trovano nella partizione di lettura/scrittura *userdata* (in basso a destra nella Figura 3.1) e possono essere disinstallate o sostituite in qualsiasi momento. La maggior parte delle applicazioni installate dagli utenti viene inserita nella directory */data/app/*.

Le directory dati per le applicazioni di sistema e installate dall'utente sono create nella partizione *userdata* all'interno della directory */data/data/*. La partizione *userdata* ospita anche i file DEX ottimizzati per le applicazioni installate dall'utente (in */data/dalvik-cache/*), il database dei package di sistema (in `/data/system/packages.xml`) e altri database di sistema e file di impostazioni. Le rimanenti directory della partizione *userdata*, mostrate nella Figura 3.1, saranno descritte quando parleremo del processo di installazione dei file APK.

Componenti attivi

Dopo aver stabilito i ruoli delle partizioni *userdata* e *system*, vediamo i componenti attivi che hanno un ruolo nell'installazione dei package.

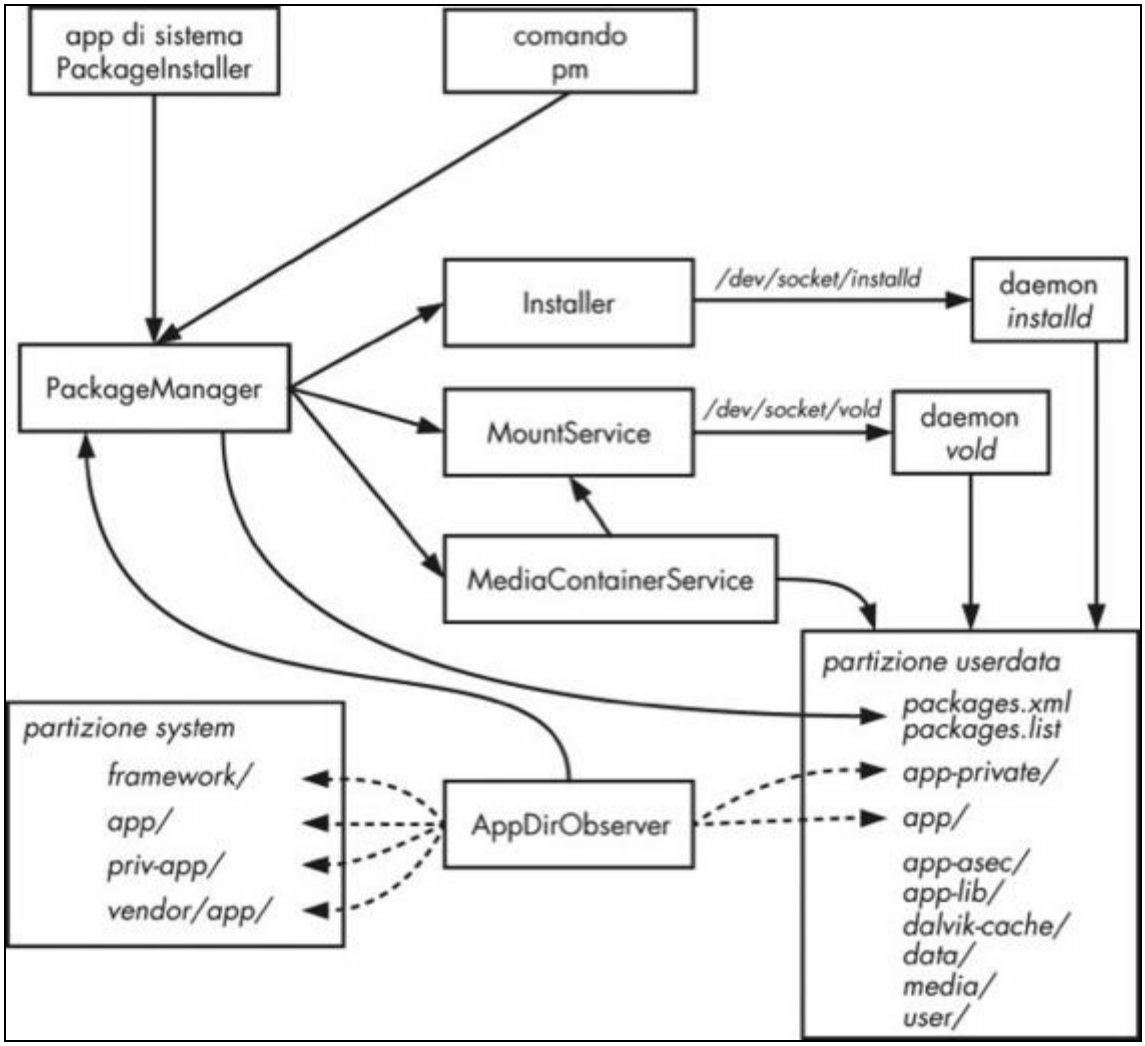

Figura 3.1 Componenti di gestione dei package.

Applicazione di sistema PackageInstaller

È il gestore predefinito dei file APK, che fornisce una GUI di base per la gestione dei package e, quando riceve l'URI di un file APK con l'azione di intent `VIEW` o `INSTALL_ACTION`, esegue il parsing del package e visualizza una schermata di conferma che mostra i permessi richiesti dall'applicazione (Figura 2.1). L'installazione con l'applicazione `PackageInstaller` è possibile solo se l'utente ha abilitato l'opzione per le origini sconosciute nelle impostazioni di protezione del dispositivo (Figura 3.2). Se le origini sconosciute non sono abilitate, `PackageInstaller` mostra una finestra che informa l'utente che l'installazione delle app ottenute da origini sconosciute è bloccata.

Che cos'è una "origine sconosciuta"? Il suggerimento a video la definisce un'origine di app diversa da Play Store, ma in realtà la definizione è più ampia. All'avvio, `PackageInstaller` recupera l'UID e il package dell'app che ha richiesto l'installazione APK e verifica se si tratta di un'app privilegiata (installata in */system/priv-app/*). Se l'app richiedente non è privilegiata, viene considerata come di origine sconosciuta. Se l'opzione relativa è selezionata e l'utente conferma l'installazione, `PackageInstaller` chiama `PackageManagerService`, che si occupa dell'installazione vera e propria.

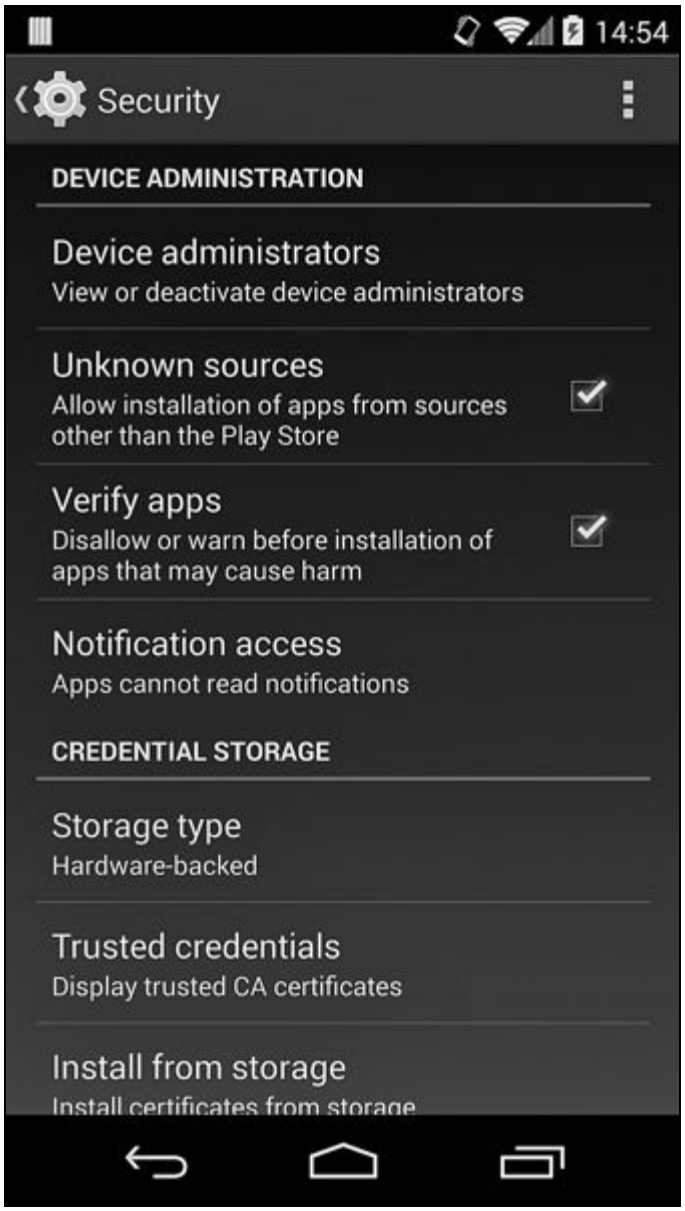

Figura 3.2 Impostazioni di protezione per l'installazione di applicazioni.

La GUI di `PackageInstaller` è visibile anche durante l'aggiornamento dei package sideloaded o durante la disinstallazione delle app dalla schermata

Applicazioni delle impostazioni di sistema.

Il comando pm

Il comando `pm` (presentato nel Capitolo 2) offre un'interfaccia a riga di comando per alcune funzioni per package manager di sistema. Può essere usato per installare o disinstallare i package se chiamato rispettivamente come `pm install` o `pm uninstall` dalla shell di Android. Inoltre, il client *Android Debug Bridge* (ADB) offre le scorciatoie `adb install/uninstall`.

A differenza di `PackageInstaller`, `pm install` non dipende dall'opzione di sistema per le origini sconosciute e non mostra una GUI; fornisce comunque varie opzioni utili per l'installazione dei package di test che non possono essere specificate con la GUI di `PackageInstaller`. Per avviare il processo di installazione, chiama la stessa API `PackageManager` usata dall'installer GUI.

PackageManagerService

`PackageManagerService` (`PackageManager` nella Figura 3.1) è l'oggetto centrale nell'infrastruttura di gestione dei package di Android. È responsabile del parsing dei file APK, dell'avvio dell'installazione delle applicazioni, dell'aggiornamento e della disinstallazione dei package, della manutenzione del database dei package e della gestione dei permessi.

`PackageManagerService` fornisce anche numerosi metodi `installPackage()` che possono eseguire l'installazione dei package con varie opzioni. Il più generico tra questi è `installPackageWithVerificationAndEncryption()`, che consente l'installazione di un file APK crittografato e la verifica del package tramite un agent specifico. La crittografia e la verifica delle app sono discusse più avanti nei paragrafi "Installazione di file APK crittografati" e "Verifica dei package".

NOTA

La classe di facciata dell'SDK di Android `android.content.pm.PackageManager` espone un sottoinsieme delle funzionalità di `PackageManagerService` alle applicazioni di terze parti.

Classe Installer

Anche se `PackageManagerService` è uno dei servizi di sistema con il maggior numero di privilegi in Android, è tuttora eseguito nel processo server di sistema (con UID `system`) ed è privo dei privilegi root. Tuttavia, poiché la creazione, l'eliminazione e la modifica del proprietario delle directory delle applicazioni richiedono capacità di superuser, `PackageManagerService` delega queste operazioni al daemon `installd` (descritto più avanti). La classe `Installer` si connette al daemon `installd` tramite il socket di dominio Unix */dev/socket/installd* e incapsula il protocollo orientato ai comandi `installd`.

Il daemon installd

`installd` è un daemon nativo con privilegi elevati che fornisce le funzionalità di gestione delle directory di applicazioni e utenti (per i dispositivi multiutente) al package manager di sistema. È usato anche per avviare il comando `dexopt`, che genera file DEX ottimizzati per i package appena installati.

Al daemon `installd` si accede tramite il socket locale `installd`, a cui possono accedere solo i processi in esecuzione con l'UID `system`. Il daemon `installd` non viene eseguito come root (sebbene si comportava in tal senso nelle precedenti versioni di Android), ma sfrutta le capability Linux `CAP_DAC_OVERRIDE` e `CAP_CHOWN` per impostare il proprietario e il group ID delle directory e dei file dell'applicazione creati sul proprietario e sul group ID dell'applicazione proprietaria. Per una descrizione delle capability di Linux, leggete il Capitolo 39 del libro di Michael Kerrisk *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* (No Starch Press, 2010).

MountService

`MountService` è responsabile del mounting della memoria esterna rimovibile, per esempio le schede SD, e dei *file OBB* (*Opaque Binary Blob*) utilizzati come file di espansione per le applicazioni. È inoltre usato

per dare il via alla crittografia del dispositivo (Capitolo 10) e per cambiare la password di crittografia.

`MountService` gestisce inoltre i *contenitori sicuri*, che contengono i file delle applicazioni che non devono essere accessibili alle applicazioni non di sistema. I contenitori sicuri sono crittografati e usati per implementare una forma di DRM chiamata *forward locking* (presentata più avanti nei paragrafi "Forward locking" e "Implementazione del forward locking in Android 4.1"). Il forward locking è utilizzato principalmente durante l'installazione di applicazioni a pagamento per garantire che i relativi file APK non possano essere facilmente copiati dal dispositivo e ridistribuiti.

Il daemon vold

`vold` è il daemon di gestione dei volumi di Android. Anche se `MountService` contiene la maggior parte delle API di sistema relative alla gestione dei volumi, il fatto che venga eseguito con utente `system` fa sì che i privilegi necessari per montare e smontare i volumi del disco non siano disponibili. Queste operazioni privilegiate sono implementate nel daemon `vold`, che viene eseguito come root.

`vold` dispone di un'interfaccia socket locale esposta dal socket di dominio Unix */dev/socket/vold*, accessibile unicamente a root e ai membri del gruppo `mount`. Visto che l'elenco di GID supplementari del processo `system_server` (che ospita `MountService`) include `mount` (GID 1009), `MountService` può accedere al socket dei comandi di `vold`. Oltre a montare e smontare i volumi, `vold` può creare e formattare i file system e gestire i contenitori sicuri.

MediaContainerService

`MediaContainerService` copia i file APK nella posizione di installazione finale o in un contenitore crittografato e consente a `PackageManagerService` di accedere ai file negli archivi rimovibili. I file APK ottenuti da una posizione remota (direttamente o tramite un market di applicazioni) vengono scaricati utilizzando il servizio `DownloadManager` di Android; i file scaricati sono

accessibili tramite l'interfaccia del content provider di `DownloadManager`. `PackageManager` concede al processo `MediaContainerService` l'accesso temporaneo a ogni APK scaricato. Se il file APK è crittografato, `MediaContainerService` lo decodifica (come spiegato più avanti nel paragrafo "Installazione di un file APK crittografato con verifica dell'integrità"). Se è stato richiesto un contenitore crittografato, `MediaContainerService` delega la sua creazione a `MountService` e copia la parte protetta dell'APK (codice e risorse) nel contenitore appena creato. I file che non devono essere protetti da un contenitore vengono copiati direttamente nel file system.

AppDirObserver

`AppDirObserver` è un componente che monitora la directory di un'applicazione per rilevare cambiamenti nei file APK e che chiama il metodo appropriato di `PackageManagerService` in base al tipo di evento; il monitoraggio dei file è implementato utilizzando la funzionalità `inotify` di Linux. (Per maggiori dettagli su `inotify`, leggete il Capitolo 19 del libro di Michael Kerrisk *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, No Starch Press, 2010.) Quando al sistema viene aggiunto un file APK, `AppDirObserver` avvia una scansione del package che provoca l'installazione o l'aggiornamento dell'applicazione. Se viene rimosso un file APK, `AppDirObserver` avvia il processo di disinstallazione, che rimuove le directory dell'app e la voce a essa relativa nel database dei package di sistema.

La Figura 3.1 mostra, per ragioni di spazio, un'unica istanza di `AppDirObserver`, ma tenete conto che esiste un'istanza dedicata per ogni directory seguita. Le directory monitorate nella partizione *system* sono */system/framework/* (che contiene il package delle risorse del framework `framework-res.apk`) */system/app/* e */system/priv-app/* (package di sistema) e la directory dei package dei vendor */system/vendor/app/*. Le directory monitorate nella partizione *userdata* sono */data/app/* e */data/app-private/*, che ospita gli APK con forward locking "vecchio stile"

(precedenti ad Android 4.1) e i file temporanei prodotti durante la decodifica dei file APK.

Installazione di un package locale

Ora che conosciamo il coinvolgimento dei componenti Android nell'installazione dei package possiamo esaminare il processo di installazione, partendo dal caso più semplice: l'installazione di un package locale non crittografato senza verifica né forward locking.

Parsing e verifica del package

L'apertura di un file APK locale avvia l'handler *application/vnd.android.package-archive*, generalmente `PackageInstallerActivity` dall'applicazione di sistema `PackageInstaller`. `PackageInstallerActivity` verifica per prima cosa che l'applicazione che ha richiesto l'installazione sia attendibile (ovvero non considerata come proveniente da una "origine sconosciuta"). Se non è attendibile e se `Settings.Global.INSTALL_NON_MARKET_APPS` è `false` (l'impostazione è `true` quando è selezionata la casella di controllo *Unknown sources*, Figura 3.2), `PackageInstaller` mostra una finestra di avviso e termina il processo di installazione.

Se l'installazione è consentita, `PackageInstallerActivity` esegue il parsing del file APK e raccoglie informazioni dal file `AndroidManifest.xml` e dalla firma del package. L'integrità del file APK viene verificata automaticamente durante l'estrazione dei certificati di firma per ognuna delle sue voci utilizzando `java.util.jar.JarFile` e le classi correlate. Questa implementazione è necessaria perché l'API della classe `JarFile` è priva di metodi espliciti per verificare la firma dell'intero file o di una particolare voce. Le applicazioni di sistema sono implicitamente attendibili, pertanto durante il parsing dei loro file APK viene verificata solo l'integrità del file `AndroidManifest.xml`. Invece, per i package che non sono parte dell'immagine di sistema, come le applicazioni installate dall'utente o gli aggiornamenti delle applicazioni di sistema, vengono verificate tutte le voci dell'APK.

Durante il parsing dell'APK viene inoltre calcolato il valore hash di `AndroidManifest.xml`, che viene trasferito alle fasi successive dell'installazione per verificare che il file APK non sia stato sostituito tra il momento in cui l'utente ha scelto *OK* nella finestra di installazione e il momento in cui è stato avviato il processo di copia dell'APK.

NOTA

È interessante notare che, durante l'installazione, l'integrità del file APK viene verificata utilizzando le classi della libreria Java standard, mentre in fase di esecuzione la macchina virtuale Dalvik carica i file APK utilizzando la sua implementazione nativa di un parser di file ZIP/JAR. Le piccole differenze nelle loro implementazioni sono state causa di numerosi bug di Android, in particolare del bug #8219321 (comunemente noto come *Android Master Key*), che consente di modificare un file APK firmato e far sì che sia considerato valido anche senza una nuova firma. Per affrontare il problema, nel ramo master di AOSP è stata aggiunta una classe `StrictJarFile` che utilizza la stessa implementazione del parsing di file ZIP usata da Dalvik. `StrictJarFile` è usato dal package manager di sistema durante il parsing dei file APK per garantire che sia Dalvik sia il package manager effettuino il parsing dei file APK in maniera identica. Questa nuova implementazione unificata dovrebbe essere incorporata nelle versioni future di Android.

Accettazione dei permessi e avvio del processo di installazione

Una volta eseguito il parsing dell'APK, `PackageInstallerActivity` visualizza informazioni sull'applicazione e sui permessi richiesti in una finestra simile a quella della Figura 2.1. Se l'utente conferma l'installazione, `PackageInstallerActivity` inoltra il file APK e il digest del suo manifest, insieme ai metadati di installazione quali URL del referrer, nome di package dell'installer e UID di origine, all'activity `InstallAppProgress`, che dà il via al processo di installazione vero e proprio. `InstallAppProgress` passa quindi l'URI dell'APK e i metadati di installazione al metodo `installPackageWithVerificationAndEncryption()` di `PackageManagerService`, avviando l'installazione. Attende quindi il completamento del processo e gestisce eventuali errori.

Per prima cosa, il metodo di installazione verifica che il chiamante abbia il permesso `INSTALL_PACKAGES`, che usa il livello di protezione *signature* ed è riservato alle applicazioni di sistema. Sui dispositivi multiutente il metodo verifica anche che l'utente chiamante sia autorizzato a installare

applicazioni. A seguire determina la posizione di installazione preferita, corrispondente all'archiviazione interna o esterna.

Copia nella directory dell'applicazione

Se il file APK non è crittografato e non è richiesta alcuna verifica, il passo successivo è la copia nella directory dell'applicazione (*/data/app/*). Per copiare il file, `PackageManagerService` crea un file temporaneo nella directory dell'applicazione (con il prefisso `vmdl` e l'estensione `.tmp`) e quindi delega la copia a `MediaContainerService`. Il file non viene copiato direttamente perché potrebbe dover essere decrittato o perché necessita di un contenitore crittografato se sarà soggetto a forward locking. `MediaContainerServices` incapsula questi task, pertanto `PackageManagerService` non deve preoccuparsi dell'implementazione sottostante.

Se il file APK viene copiato correttamente, le librerie native che contiene vengono estratte in una directory dedicata dell'app all'interno della directory delle librerie native del sistema (*/data/app-lib/*). Successivamente, il file APK temporaneo e la directory della libreria ricevono i loro nomi finali basati sul nome del package, per esempio *com.example.app-1.apk* per l'APK e */data/app-lib/com.example.app-1* per la directory delle librerie. Infine, i permessi del file APK vengono impostati a *0644* e ne viene impostato il contesto SELinux (consultate il Capitolo 12).

NOTA

Per impostazione predefinita, i file APK sono leggibili a tutti e qualunque altra applicazione può accedervi. Questa situazione facilita la condivisione di risorse per le app pubbliche e consente lo sviluppo di launcher di terze parti e altre applicazioni che devono mostrare un elenco di tutti i package installati. Tuttavia, questi permessi predefiniti consentono a chiunque di estrarre i file APK da un dispositivo, causando problemi legati alle applicazioni a pagamento distribuite tramite uno store. Il forward locking dei file APK permette di mantenere pubbliche le risorse ma di limitare l'accesso al codice e agli asset.

Scansione del package

Il passaggio successivo del processo di installazione è l'attivazione di una scansione del package con il metodo `scanPackageLI()` di `PackageManagerService`. Se il processo di installazione si interrompe prima della scansione del

nuovo file APK, alla fine sarà rilevato dall'istanza `AppDirObserver` che monitora la directory */data/app/* e attiva anch'essa una scansione del package.

Nel caso di una nuova installazione, il package manager crea una nuova struttura `PackageSettings` contenente il nome del package, il percorso del codice, un percorso separato per le risorse se il package è forward locked e un percorso per le librerie native. Assegna quindi un UID al nuovo package e lo salva nella struttura delle impostazioni. Quando la nuova app dispone di un UID può essere creata la sua directory dati.

Creazione di directory dati

`PackageManagerService` non ha privilegi sufficienti per creare e impostare la proprietà delle directory delle app, pertanto delega la creazione delle directory al daemon `installd` inviandogli il comando `install`, che richiede come parametri il nome del package, l'UID, il GID e il tag `seinfo` (usato da SELinux). Il daemon `installd` crea la directory dati del package (per esempio */data/data/com.example.app/* durante l'installazione del package *com.example.app*), la directory delle librerie native condivise (*/data/app-lib/com.example.app/*) e la directory delle librerie locali (*/data/data/com.example.app/lib/*). Imposta quindi i permessi delle directory del package su *0751* e crea collegamenti simbolici per le librerie native dell'app (se presenti) nella directory delle librerie locali. Infine, imposta il contesto SELinux della directory del package e cambia il suo proprietario impostando l'UID e il GID assegnati all'app.

Se il sistema prevede più utenti, il passaggio successivo è la creazione delle directory dati per ogni utente con l'invio del comando `mkuserdata` a `installd` (consultate il Capitolo 4). Una volta create tutte le directory necessarie, il controllo ritorna a `PackageManagerService`, che estrae eventuali librerie native nella directory delle librerie native dell'applicazione e crea collegamenti simbolici in */data/data/com.example.app/lib/*.

Generazione di Optimized DEX

Il prossimo passo è la generazione di Optimized DEX per il codice dell'applicazione. Anche questa operazione viene delegata a `installd` con l'invio del comando `dexopt`. Il daemon `installd` esegue il fork di un processo `dexopt`, che crea il file DEX ottimizzato nella directory */data/dalvik-cache/* (il processo di ottimizzazione è detto anche *sharpening*).

NOTA

Se il dispositivo utilizza l'*Android Runtime* (ART) sperimentale introdotto nella versione 4.4, invece di generare Optimized DEX, `installd` genera codice nativo utilizzando il comando `dex2oat`.

Struttura di file e directory

Una volta completati tutti i passaggi precedenti, i file e le directory dell'applicazione appaiono indicativamente come mostrato nel Listato 3.12 (sono stati omessi i timestamp e le dimensioni dei file).

Listato 3.12 File e directory creati dopo l'installazione di un'applicazione.

```
-rw-r--r-- system   system   ... /data/app/com.example.app-1.apk(1)
-rwxr-xr-x system   system   ... /data/app-lib/com.example.app-1/libapp.so(2)
-rw-r--r-- system   all_a215 ... /data/dalvik-cache/data@app@com.example.app-1.apk@classes.dex(3)
drwxr-x--x u0_a215  u0_a215  ... /data/data/com.example.app(4)
drwxrwx--x u0_a215  u0_a215  ... /data/data/com.example.app/databases(5)
drwxrwx--x u0_a215  u0_a215  ... /data/data/com.example.app/files
lrwxrwxrwx install  install  ... /data/data/com.example.app/lib -> /data/app-lib/com.example.app-
1(6)
drwxrwx--x u0_a215  u0_a215  ... /data/data/com.example.app/shared_prefs
```

Qui **(1)** indica il file APK, mentre **(2)** è il file delle librerie native estratto. Entrambi i file sono di proprietà di `system` e sono leggibili a tutti. Il file in **(3)** è il file Optimized DEX per il codice dell'applicazione. Il suo proprietario è impostato a `system` e il suo gruppo è impostato sul gruppo speciale `all_a215`, che include tutti gli utenti del dispositivo che hanno installato l'app. In questo modo possono condividere il medesimo file Optimized DEX, evitando la necessità di crearne una copia per ogni utente e occupare troppo spazio su disco su un dispositivo multiutente. La directory dei dati dell'applicazione **(4)** e le sue sottodirectory (come *databases/* **(5)**) sono di proprietà dell'utente Linux dedicato creato combinando l'ID dell'utente del dispositivo che ha installato l'applicazione (`u0`, l'unico utente sui dispositivi a utente singolo) e l'app ID (`a215`) per ottenere `u0_a215`. Le directory dati dell'app non sono leggibili o modificabili dagli altri utenti in conformità al modello di sicurezza della

sandbox di Android. La directory *lib/* **(6)** è semplicemente un collegamento simbolico alla directory delle librerie condivise dell'app in */data/app-lib/*.

Aggiunta del nuovo package a packages.xml

Il prossimo passo è l'aggiunta del package al database dei package di sistema. A tal fine viene generata e aggiunta a `packages.xml` una nuova voce simile a quella mostrata nel Listato 3.13.

Listato 3.13 Voce nel database dei package per un'applicazione appena installata.

```
<package name="com.google.android.apps.chrometophone"
         codePath="/data/app/com.google.android.apps.chrometophone-2.apk"
         nativeLibraryPath="/data/app-lib/com.google.android.apps.chrometophone-2"
         flags="572996"
         ft="142dfa0e588"
         it="142cbeac305"
         ut="142dfa0e8d7"
         version="16"
         userId="10088"
         installer="com.android.vending">(1)
    <sigs count="1">
        <cert index="7" key="30820252..." />
    </sigs>(2)
    <perms>
        <item name="android.permission.USE_CREDENTIALS" />
        <item name="com.google.android.apps.chrometophone.permission.C2D_MESSAGE" />
        <item name="android.permission.GET_ACCOUNTS" />
        <item name="android.permission.INTERNET" />
        <item name="android.permission.WAKE_LOCK" />
        <item name="com.google.android.c2dm.permission.RECEIVE" />
    </perms>(3)
    <signing-keyset identifier="2" />(4)
</package>
```

Qui l'elemento `<sigs>` **(2)** contiene i valori codificati DER dei certificati di firma del package (in genere solo uno) in formato stringa esadecimale, oppure un riferimento alla prima occorrenza del certificato nel caso di molteplici app firmate con la stessa chiave e lo stesso certificato. Gli elementi `<perms>` **(3)** contengono i permessi concessi all'applicazione, come descritto nel Capitolo 2. L'elemento `<signing-keyset>` **(4)** è una novità di Android 4.4 e contiene un riferimento al set di chiavi di firma dell'applicazione, che a sua volta contiene tutte le chiavi pubbliche (ma *non* i certificati) usate per firmare i file nell'APK. `PackageManagerService` raccoglie e memorizza le chiavi di firma per tutte le applicazioni in un elemento globale `<keyset-settings>`, ma i set di chiavi non sono verificati o comunque utilizzati come in Android 4.4.

Attributi dei package

L'elemento root `<package>` **(1)** (mostrato nel Listato 3.13) contiene gli attributi fondamentali di ogni package, come la posizione di installazione e la versione. Gli attributi principali dei package sono elencati nella Tabella 3.1. Le informazioni in ogni voce di package possono essere ottenute con il metodo `getPackageInfo(String packageName, int flags)` della classe SDK `android.content.pm.PackageManager`, che dovrebbe restituire un'istanza `PackageInfo` che incapsula gli attributi disponibili in ogni voce `packages.xml`, nonché le informazioni su componenti, permessi e funzionalità definiti nel manifest dell'applicazione.

Tabella 3.1 Attributi dei package.

Nome attributo	Descrizione
`name`	Nome del package.
`codePath`	Percorso completo della posizione del package.
`resourcePath`	Percorso completo della posizione delle parti disponibili al pubblico del package (package di risorse primario e manifest). Impostato solo per le app con forward locking.
`nativeLibraryPath`	Percorso completo della directory in cui sono memorizzate le librerie native.
`flags`	Flag associati all'applicazione.
`ft`	Timestamp del file APK (tempo Unix in millisecondi, ottenuto con `System.currentTimeMillis()`).
`it`	Il momento in cui l'app è stata installata per la prima volta (tempo Unix in millisecondi).
`ut`	Il momento in cui l'app è stata aggiornata per l'ultima volta (tempo Unix in millisecondi).
`version`	Il numero di versione del package, specificato dall'attributo `versionCode` nel manifest dell'app.
`userId`	L'UID del kernel assegnato all'applicazione.
`installer`	Il nome del package dell'applicazione che ha installato l'app.
`sharedUserId`	Lo user ID condiviso del package, specificato dall'attributo `sharedUserId` nel manifest dell'app.

Aggiornamento di componenti e permessi

Dopo aver creato la voce di `packages.xml`, `PackageManagerService` esamina tutti i componenti Android definiti nei manifest delle nuove applicazioni e li aggiunge al suo registro interno dei componenti in memoria. Dopodiché, i gruppi di permessi e i permessi dichiarati dall'app vengono esaminati e aggiunti al registro dei permessi.

NOTA

I permessi personalizzati definiti dalle applicazioni sono registrati utilizzando una strategia che assegna la "vittoria" alla prima applicazione: se entrambe le app A e B definiscono il permesso P e

A viene installata per prima, la definizione del permesso di A viene registrata, mentre la definizione del permesso di B viene ignorata (perché P è già registrato). Questo è possibile perché i nomi dei permessi non sono associati al package dell'app che li definisce, pertanto qualsiasi app può definire qualunque permesso. Questa strategia può causare una riduzione del livello di protezione: se la definizione del permesso di A usa un livello di protezione inferiore (per esempio *normal*) rispetto a quello della definizione di B (per esempio *signature*) e A viene installata per prima, l'accesso ai componenti di B protetti da P non richiede che i chiamanti siano firmati con la stessa chiave di B. Di conseguenza, se usate i permessi personalizzati per proteggere i componenti, assicuratevi di verificare che il permesso registrato abbia il livello di protezione previsto per l'app (consultate CommonsWare, CWAC-Security, https://github.com/commonsguy/cwac-security, per altre informazioni e per un progetto di esempio che mostra come eseguire il controllo).

Infine, le modifiche al database dei package (voce del package ed eventuali nuovi permessi) vengono salvate su disco e `PackageManagerService` invia `ACTION_PACKAGE_ADDED` per informare gli altri componenti della nuova applicazione aggiunta.

Aggiornamento di un package

Il processo di aggiornamento di un package segue la maggior parte dei passaggi visti per l'installazione, pertanto qui prenderemo in esame solo le differenze.

Verifica della firma

Il primo passo consiste nel verificare se il nuovo package è stato firmato dallo stesso set di firmatari del package esistente. Questa regola è definita *criterio della stessa origine*, o *Trust On First Use* (TOFU). Questo controllo della firma garantisce che l'aggiornamento sia prodotto dalla stessa entità dell'applicazione originale (supponendo che la chiave di firma non sia stata compromessa) e stabilisce una relazione di trust tra l'aggiornamento e l'applicazione esistente. Come vedremo nel paragrafo "Aggiornamento delle app non di sistema", l'aggiornamento eredita i dati dell'applicazione originale.

NOTA

Quando vengono confrontati, i certificati di firma non subiscono una convalida PKI (la validità temporale, l'attendibilità dell'autorità emittente, la revoca e così via non vengono verificate).

La verifica di uguaglianza dei certificati è eseguita dal metodo `PackageManagerService.compareSignatures()`, mostrato nel Listato 3.14.

Listato 3.14 Metodo di confronto delle firme dei package.

```
static int compareSignatures(Signature[] s1, Signature[] s2) {
    if (s1 == null) {
        return s2 == null
            ? PackageManager.SIGNATURE_NEITHER_SIGNED
            : PackageManager.SIGNATURE_FIRST_NOT_SIGNED;
    }
    if (s2 == null) {
        return PackageManager.SIGNATURE_SECOND_NOT_SIGNED;
    }
    HashSet<Signature> set1 = new HashSet<Signature>();
    for (Signature sig : s1) {
        set1.add(sig);
    }
    HashSet<Signature> set2 = new HashSet<Signature>();
    for (Signature sig : s2) {
        set2.add(sig);
    }
    // Verifica che s2 contenga tutte le firme in s1.
    if (set1.equals(set2)) {(1)
        return PackageManager.SIGNATURE_MATCH;
    }
    return PackageManager.SIGNATURE_NO_MATCH;
}
```

Qui la classe `Signature` agisce come una "rappresentazione opaca e immutabile di una firma associata al package di un'applicazione" (http://bit.ly/1rxKS76). In pratica, si tratta di un wrapper per il certificato di firma codificato DER associato a un file APK. Nel Listato 3.15 è riportato un estratto relativo ai metodi `equals()` e `hashCode()`.

Listato 3.15 Rappresentazione delle firme dei package.

```
public class Signature implements Parcelable {
    private final byte[] mSignature;
    private int mHashCode;
    private boolean mHaveHashCode;
    --altro codice--
    public Signature(byte[] signature) {
        mSignature = signature.clone();
    }

    public PublicKey getPublicKey() throws CertificateException {
        final CertificateFactory certFactory =
                CertificateFactory.getInstance("X.509");
        final ByteArrayInputStream bais = new ByteArrayInputStream(mSignature);
        final Certificate cert = certFactory.generateCertificate(bais);
        return cert.getPublicKey();
    }

    @Override
    public boolean equals(Object obj) {
        try {
            if (obj != null) {
                Signature other = (Signature)obj;
                return this == other
                    || Arrays.equals(mSignature, other.mSignature);(1)
            }
        } catch (ClassCastException e) {
        }
        return false;
    }

    @Override
    public int hashCode() {
        if (mHaveHashCode) {
```

```
            return mHashCode;
        }
        mHashCode = Arrays.hashCode(mSignature);(2)
        mHaveHashCode = true;
        return mHashCode;
    }
--altro codice--
}
```

Come potete osservare al punto **(1)**, due classi di firma sono considerate uguali se la codifica DER dei certificati X.509 sottostanti corrisponde esattamente, e il codice hash della classe `Signature` viene calcolato solo in base al certificato codificato **(2)**. Se i certificati di firma non corrispondono, i metodi `compareSignatures()` restituiscono il codice di errore `INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES`.

Questo confronto binario dei certificati ovviamente non è a conoscenza delle CA o delle date di scadenza: è per questo motivo che, dopo l'installazione di un'app (identificata da un nome di package univoco), gli aggiornamenti devono usare gli stessi certificati di firma (fatta eccezione per gli aggiornamenti delle app di sistema come spiegato nel paragrafo "Aggiornamento delle app di sistema").

Per quanto sia raro che le app Android usino più firme, a volte questo accade: se l'applicazione originale è stata firmata da più firmatari, eventuali aggiornamenti devono essere firmati dagli stessi firmatari, ognuno dei quali deve utilizzare il suo certificato di firma originale (**(1)** nel Listato 3.14). In pratica, se i certificati di firma di uno sviluppatore scadono o se questi non ha più accesso alla sua chiave di firma, egli non potrà più aggiornare l'app e dovrà rilasciarne una nuova. In questo modo, oltre a perdere gli utenti e le valutazioni già ottenuti, non avrà più accesso ai dati e alle impostazioni dell'app legacy.

La soluzione a questo problema è semplice, se non ideale: basta effettuare il backup della chiave di firma e non lasciare scadere il certificato. Il periodo di validità attualmente consigliato è di almeno 25 anni; Google Play Store richiede la validità almeno fino a ottobre 2033. Anche se tecnicamente il problema può essere risolto, è probabile che alla fine venga aggiunto alla piattaforma il supporto per la migrazione dei certificati.

Se il package manager stabilisce che l'aggiornamento è stato firmato con lo stesso certificato, si procede all'aggiornamento del package. Il processo è diverso per le app di sistema e quelle installate dall'utente, come spiegato nei prossimi paragrafi.

Aggiornamento delle app non di sistema

Le app non di sistema vengono aggiornate con una reinstallazione dell'app che mantiene la stessa directory dati. Per prima cosa occorre terminare qualunque processo del package da aggiornare; a seguire, si rimuove il package dalle strutture interne e dal database dei package, eliminando così tutti i componenti già registrati dall'app. `PackageManagerService` attiva quindi una scansione del package con il metodo `scanPackageLI()`: la scansione procede come per le nuove installazioni, ma aggiorna il codice, il percorso delle risorse, la versione e il timestamp del package. Il manifest del package viene sottoposto a scansione e i componenti definiti vengono registrati nel sistema. Successivamente, vengono concessi nuovamente i permessi per tutti i package in modo che corrispondano alle definizioni nel package aggiornato. Per finire, il database dei package aggiornato viene scritto su disco e viene inviato un broadcast di sistema `PACKAGE_REPLACED`.

Aggiornamento delle app di sistema

Come le app installate dall'utente, le app preinstallate (generalmente presenti in */system/app/*) possono essere aggiornate senza ricorrere a un aggiornamento completo del sistema, di solito tramite Google Play Store o un servizio simile di distribuzione delle app. Visto che la partizione *system* è montata in sola lettura, gli aggiornamenti vengono installati in */data/app/* e l'app originale resta invariata. Oltre alla voce `<package>`, l'app aggiornata dispone anche di una voce `<updated-package>` simile a quella mostrata nel Listato 3.16.

Listato 3.16 Voce nel database dei package per un package di sistema aggiornato.

```
<package name="com.google.android.keep"
         codePath="/data/app/com.google.android.keep-1.apk"(1)
         nativeLibraryPath="/data/app-lib/com.google.android.keep-1"
```

```
        flags="4767461"(2)
        ft="142ee64d980"
        it="14206f3e320"
        ut="142ee64dfcb"
        version="2101"
        userId="10053"(3)
        installer="com.android.vending">
    <sigs count="1">
        <cert index="2" />
    </sigs>
    <signing-keyset identifier="3" />
    <signing-keyset identifier="34" />
</package>
--altro codice--
<updated-package name="com.google.android.keep"
                 codePath="/system/app/Keep.apk"
                 nativeLibraryPath="/data/app-lib/Keep"
                 ft="ddc8dee8"
                 it="14206f3e320"
                 ut="ddc8dee8"
                 version="2051"
                 userId="10053">(4)
    <perms>
        <item name="android.permission.READ_EXTERNAL_STORAGE" />
        <item name="android.permission.USE_CREDENTIALS" />
        <item name="android.permission.WRITE_EXTERNAL_STORAGE" />
        --altro codice--
    </perms>
</updated-package>
```

L'attributo `codePath` dell'aggiornamento viene impostato sul percorso del nuovo APK in */data/app/* **(1)**, che eredita i permessi e l'UID dell'app originale (**(3)** e **(4)**) e viene contrassegnato come aggiornamento a un'app di sistema aggiungendo `FLAG_UPDATED_SYSTEM_APP` (0x80) al suo attributo `flags` **(2)**.

Le app di sistema possono essere aggiornate direttamente anche nell'applicazione *system*, di solito a seguito di un aggiornamento di sistema OTA; il tal caso l'APK di sistema aggiornato può essere firmato con un certificato diverso. La logica alla base prevede che, se l'installer dispone di privilegi sufficienti per la scrittura nella partizione *system*, di sicuro può anche cambiare il certificato di firma. L'UID, i file e i permessi vengono mantenuti. L'eccezione riguarda il caso in cui il package è parte di un utente condiviso (come spiegato nel Capitolo 2), dove la firma non può essere aggiornata perché la modifica potrebbe influire su altre app. Nel caso contrario, se una nuova app di sistema è firmata da un certificato diverso da quello dell'app non di sistema attualmente installata (con lo stesso nome di package), l'app non di sistema viene prima eliminata.

Installazione di file APK crittografati

Il supporto per l'installazione di file APK crittografati è stato aggiunto in Android 4.1 insieme al supporto per il forward locking utilizzando i contenitori ASEC. Entrambe le funzionalità sono state annunciate come *crittografia delle app*, ma le presenteremo separatamente partendo dal supporto per i file APK crittografati. Prima, però, vediamo come installare questi file.

I file APK crittografati possono essere installati usando il client Google Play Store oppure con il comando `pm` della shell di Android; tuttavia, `PackageInstaller` di sistema non li supporta. Dal momento che non possiamo controllare il flusso di installazione di Google Play Store, per installare un file APK crittografato dobbiamo utilizzare il comando `pm` oppure scrivere una nostra app installer. Scegliamo la via facile e usiamo il comando `pm`.

Creazione e installazione di un file APK crittografato

Il comando `adb install` copia l'APK in un file temporaneo sul dispositivo e avvia il processo di installazione. Il comando offre un comodo wrapper per i comandi `adb push` e `pm install`. `adb install` richiede tre nuovi parametri in Android 4.1 per supportare gli APK crittografati (Listato 3.17).

Figura 3.17 Opzioni del comando adb install.

```
adb install [-l] [-r] [-s] [--algo <algorithm name> --key <hex-encoded key>
--iv <hex-encoded iv>] <file>
```

I parametri `--algo`, `--key` e `--iv` consentono di specificare rispettivamente l'algoritmo di crittografia, la chiave e il vettore di inizializzazione (IV). Tuttavia, per usare questi nuovi parametri, dobbiamo prima creare un APK crittografato.

Un file APK può essere crittografato utilizzando i comandi OpenSSL `enc` mostrati nel Listato 3.18. Qui usiamo AES nella modalità CBC con una chiave a 128 bit e specifichiamo un IV identico alla chiave per semplificare le cose.

Listato 3.18 Crittografia di un file APK con OpenSSL.

```
$ openssl enc -aes-128-cbc -K 000102030405060708090A0B0C0D0E0F
-iv 000102030405060708090A0B0C0D0E0F -in my-app.apk -out my-app-enc.apk
```

A seguire installiamo il nostro APK crittografato passando la chiave dell'algoritmo di crittografia (nel formato stringa di trasformazione `javax.crypto.Cipher`, presentato nel Capitolo 5) e i byte IV al comando `adb install`, come illustrato nel Listato 3.19.

Listato 3.19 Installazione di un APK crittografato con adb install.

```
$ adb install --algo 'AES/CBC/PKCS5Padding' \
--key 000102030405060708090A0B0C0D0E0F \
--iv 000102030405060708090A0B0C0D0E0F my-app-enc.apk
        pkg: /data/local/tmp/my-app-enc.apk
Success
```

Come indicato dall'output `Success`, l'APK viene installato senza errori. Il file APK vero e proprio viene copiato in */data/app/*; un confronto del suo hash con il nostro APK crittografato rivela che si tratta di un file diverso. Il valore dell'hash è lo stesso del file APK originali (non crittografato), da cui possiamo dedurre che l'APK viene decriptato in fase di installazione utilizzando i parametri di crittografia forniti (algoritmo, chiave e IV).

Implementazione e parametri di crittografia

Vediamo come viene implementato tutto questo. Dopo aver trasferito l'APK al dispositivo, `adb install` chiama l'utility a riga di comando Android `pm` Android con il parametro `install` e il percorso del file APK copiato. Il componente responsabile dell'installazione delle app su Android è `PackageManagerService` e il comando `pm` non è altro che un comodo front-end per alcune delle sue funzionalità. Se avviato con il parametro `install`, `pm` chiama il metodo `installPackageWithVerificationAndEncryption()`, convertendo le sue opzioni nei parametri pertinenti. Il Listato 3.20 mostra la firma completa del metodo.

Listato 3.20 Firma del metodo PackageManagerService.installPackageWithVerificationAndEncryption().

```
public void installPackageWithVerificationAndEncryption(Uri packageURI,
        IPackageInstallObserver observer, int flags,
        String installerPackageName,
        VerificationParams verificationParams,
        ContainerEncryptionParams encryptionParams) {
--altro codice--
}
```

La maggior parte dei parametri del metodo è già stata presentata nel paragrafo “Processo di installazione dei file APK”; dobbiamo però ancora parlare delle classi `VerificationParams` e `ContainerEncryptionParams`. Come suggerito dal nome, la classe `VerificationParams` incapsula un parametro usato durante la verifica del package, di cui parleremo più avanti nel paragrafo “Verifica dei package”. La classe `ContainerEncryptionParams` contiene i parametri di crittografia, compresi i valori passati tramite le opzioni `--algo`, `--key` e `--iv` di `adb install`. Il Listato 3.21 mostra i suoi membri dati.

Listato 3.21 Membri dati di ContainerEncryptionParams.

```
public class ContainerEncryptionParams implements Parcelable {
    private final String mEncryptionAlgorithm;
    private final IvParameterSpec mEncryptionSpec;
    private final SecretKey mEncryptionKey;
    private final String mMacAlgorithm;
    private final AlgorithmParameterSpec mMacSpec;
    private final SecretKey mMacKey;
    private final byte[] mMacTag;
    private final long mAuthenticatedDataStart;
    private final long mEncryptedDataStart;
    private final long mDataEnd;
    --altro codice--
}
```

I parametri di `adb install` sopra corrispondono ai primi tre campi della classe. Sebbene non siano disponibili tramite il wrapper `adb install`, il comando `pm install` accetta anche i parametri `--macalgo`, `--mackey` e `--tag`, corrispondenti ai campi `mMacAlgorithm`, `mMacKey` e `mMacTag` della classe `ContainerEncryptionParams`. Per utilizzare questi parametri dobbiamo prima calcolare il valore MAC del file APK crittografato utilizzando il comando OpenSSL `dgst`, come mostrato nel Listato 3.22.

Listato 3.22 Calcolo del valore MAC di un file APK crittografato.

```
$ openssl dgst -hmac 'hmac_key_1' -sha1 -hex my-app-enc.apk
HMAC-SHA1(my-app-enc.apk)= 962ecdb4e99551f6c2cf72f641362d657164f55a
```

NOTA

Il comando `dgst` non consente di specificare la chiave HMAC nel formato esadecimale o Base64, quindi dobbiamo limitarci ai caratteri ASCII. Potrebbe non essere una buona idea per l’uso in produzione, quindi valutate la possibilità di usare una chiave reale e di calcolare il valore MAC in un altro modo (per esempio con un programma JCE).

Installazione di un file APK crittografato con verifica dell’integrità

Ora possiamo installare un APK crittografato e verificare la sua integrità aprendo la shell di Android con `adb shell` ed eseguendo il comando mostrato nel Listato 3.23.

Listato 3.23 Installazione di un APK crittografato con verifica dell'integrità usando pm install.

```
$ pm install -r --algo 'AES/CBC/PKCS5Padding' \
--key 000102030405060708090A0B0C0D0E0F \
--iv 000102030405060708090A0B0C0D0E0F \
--macalgo HmacSHA1 --mackey 686d61635f6b65795f31 \
--tag 962ecdb4e99551f6c2cf72f641362d657164f55a /sdcard/my-app-enc.apk
        pkg: /sdcard/kr-enc.apk
Success
```

L'integrità dell'app viene verificata confrontando il tag MAC specificato con il valore calcolato in base al contenuto effettivo del file; i contenuti vengono decriptati e l'APK decriptato viene copiato in */data/app/*. Per controllare l'effettiva esecuzione della verifica MAC dovete cambiare leggermente il valore del tag; così facendo otterrete un errore di installazione con codice `INSTALL_FAILED_INVALID_APK`.

Come abbiamo visto nei Listati 3.19 e 3.23, i file APK copiati in */data/app/* non sono crittografati e di conseguenza il processo di installazione è lo stesso visto per gli APK non crittografati, fatta eccezione per la decodifica del file e la verifica facoltativa dell'integrità. La decodifica e la verifica dell'integrità sono eseguite in maniera trasparente da `MediaContainerService` durante la copia dell'APK nella directory dell'applicazione. Se un'istanza di `ContainerEncryptionParams` viene passata al suo metodo `copyResource()`, i parametri di crittografia forniti vengono usati per istanziare le classi JCA `Cipher` e `Mac` (consultate il Capitolo 5) che possono eseguire la decodifica e la verifica dell'integrità.

NOTA

Il tag MAC e l'APK crittografato possono essere riuniti in un singolo file; in questo caso `MediaContainerService` usa i membri `mAuthenticatedDataStart`, `mEncryptedDataStart` e `mDataEnd` per estrarre i dati MAC e APK dal file.

Forward locking

Il forward locking è apparso quando sugli smartphone è iniziata la vendita di suonerie, sfondi e altri "articoli" digitali. Visto che in Android i file APK sono leggibili a tutti, è relativamente facile estrarre le app

persino da un dispositivo di produzione. Nel tentativo di bloccare le app a pagamento (e impedire che un utente le inoltri a un altro), ma senza perdere la flessibilità del sistema operativo, nelle precedenti versioni di Android è stato introdotto il forward locking (detto anche *protezione contro la copia*).

L'idea alla base del forward locking è la divisione dei package delle app in due parti: una leggibile a tutti e contenente le risorse e il manifest (in */data/app/*) e una leggibile solo dall'utente `system` e contenente il codice eseguibile (in */data/app-private/*). Il package del codice veniva protetto dai permessi del file system, che lo rendevano inaccessibile agli utenti sulla maggior parte dei dispositivi consumer, ma che ne consentivano l'estrazione dai dispositivi con accesso root; questo meccanismo primitivo di forward locking è stato rapidamente deprecato e sostituito da un servizio di licenze online chiamato Google Play Licensing.

Il problema di Google Play Licensing era che trasferiva l'implementazione della protezione delle app dal sistema operativo agli sviluppatori di app, con risultati variabili. L'implementazione del forward locking è stata riprogettata in Android 4.1 e oggi permette di salvare gli APK in un contenitore crittografato che richiede una chiave specifica del dispositivo per il mounting in fase di runtime. Scendiamo un po' nei dettagli.

Implementazione del forward locking di Android 4.1

Se l'uso dei contenitori di app crittografati come meccanismo di forward locking è stato introdotto in Android versione 4.1, i contenitori crittografati furono presentati in origine in Android 2.2. A quel tempo (metà del 2010), la maggior parte dei dispositivi Android aveva uno spazio di memoria interno limitato e una memoria esterna relativamente grande (qualche gigabyte), di solito sotto forma di scheda microSD. Per facilitare la condivisione dei file, la memoria esterna veniva formattata con il file system FAT, che però mancava dei permessi per i file. Di

conseguenza, i file sulla scheda SD potevano essere letti e scritti da qualunque applicazione.

Per impedire agli utenti di copiare le app a pagamento dalla scheda SD, Android 2.2 creava un file di immagine del file system crittografato e salvava l'APK al suo interno quando un utente decideva di spostare un'app nella memoria esterna. Il sistema creava quindi un punto di montaggio per l'immagine crittografata e lo montava usando il device-mapper di Linux. Android caricava i file di ogni app dal suo punto di mount in fase di runtime.

Android 4.1 ha esteso questa idea utilizzando il file system ext4, che consente i permessi sui file, per il contenitore. Un tipico punto di mount per un'app con forward locking oggi appare come nel Listato 3.24 (timestamp omessi).

Listato 3.24 Contenuto del punto di mount per un'app con forward locking.

```
# ls -l /mnt/asec/com.example.app-1
drwxr-xr-x system   system            lib
drwx------ root     root              lost+found
-rw-r----- system   u0_a96    1319057 pkg.apk
-rw-r--r-- system   system     526091 res.zip
```

Qui `res.zip` contiene le risorse dell'app e il file manifest ed è leggibile a tutti, mentre il file `pkg.apk` che contiene l'intero APK è leggibile solo dal sistema e dall'utente dedicato dell'app (`u0_a96`). I contenitori effettivi delle app sono salvati in */data/app-asec/* all'interno di file con estensione `.asec`.

Contenitori delle app crittografati

I contenitori di app crittografati sono chiamati *Android Secure External Cache*, o *contenitori ASEC.* La gestione dei contenitori ASEC (creazione, eliminazione, mounting e unmounting) è implementata nel daemon dei volumi di sistema (`vold`), mentre `MountService` offre un'interfaccia per le sue funzionalità ai servizi del framework. Possiamo utilizzare anche l'utility a riga di comando `vdc` per interagire con `vold` e gestire le app con forward locking dalla shell di Android (Listato 3.25).

Listato 3.25 Generazione di comandi di gestione ASEC con vdc.

```
# vdc asec list1
vdc asec list
111 0 com.example.app-1
```

```
111 0 org.foo.app-1
200 0 asec operation succeeded

# vdc asec path com.example.app-12
vdc asec path com.example.app-1
211 0 /mnt/asec/com.example.app-1

# vdc asec unmount org.example.app-13
200 0 asec operation succeeded

# vdc asec mount com.example.app-1 000102030405060708090a0b0c0d0e0f 10004
com.example.app-1 000102030405060708090a0b0c0d0e0f  1000
200 0 asec operation succeeded
```

Qui il comando `asec list` **(1)** elenca i namespace ID dei contenitori ASEC montati. I namespace ID si basano sul nome del package e hanno lo stesso formato dei nomi dei file APK per le applicazioni senza forward locking. Tutti gli altri comandi accettano un namespace ID come parametro.

Il comando `asec path` **(2)** mostra il punto di mount del contenitore ASEC specificato, mentre `asec unmount` lo smonta **(3)**. Oltre a un namespace ID, `asec mount` **(4)** richiede di specificare la chiave di crittografia e l'ID del proprietario del punto di mount (1000 corrisponde a `system`).

L'algoritmo di crittografia e la lunghezza della chiave del contenitore ASEC restano invariati rispetto all'implementazione originale di Android 2.2 con le app sulla scheda SD, Twofish con chiave a 128 bit memorizzata in */data/misc/systemkeys/*, come mostrato nel Listato 3.26.

Listato 3.26 Posizione e contenitore della chiave di crittografia di un contenitore ASEC.

```
# ls -l /data/misc/systemkeys
-rw------- system   system         16 AppsOnSD.sks
# od -t x1 /data/misc/systemkeys/AppsOnSD.sks
0000000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0000020
```

Il forward locking di un'applicazione viene attivato specificando l'opzione `-l` di `pm install` o specificando il flag `INSTALL_FORWARD_LOCK` durante la chiamata a uno dei metodi `installPackage()` di `PackageManager`.

Installazione di file APK con forward locking

Il processo di installazione degli APK con forward locking prevede due passaggi in più: la creazione e il mounting del contenitore sicuro e l'estrazione dei file di risorse pubblici dal file APK. Come per gli APK crittografati, questi passaggi sono incapsulati da `MediaContainerService` e sono

eseguiti durante la copia dell'APK nella directory dell'applicazione. `MediaContainerService` non dispone di privilegi sufficienti per creare e montare contenitori sicuri, pertanto delega la gestione dei contenitori al daemon `vold` chiamando i metodi di `MountService` appropriati (`createSecureContainer()`, `mountSecureContainer()` e così via).

App crittografate e Google Play

Poiché l'installazione di app, crittografate o meno, senza l'interazione dell'utente richiede i permessi di sistema, solo le applicazioni di sistema hanno la possibilità di installare le applicazioni. Il client Android di Google, Play Store, sfrutta sia le app crittografate sia il forward locking. La descrizione precisa del funzionamento del client Google Play richiederebbe una conoscenza dettagliata del protocollo sottostante (che non è aperto ed è in costante evoluzione); tuttavia, anche solo uno sguardo all'implementazione di un client Google Play Store recente può rivelare qualche informazione utile.

I server Google Play inviano numerosi metadati sull'app che si sta per scaricare e installare, quali URL di download, dimensioni del file APK, codice della versione e finestra per le operazioni di rimborso. Tra questi, gli `EncryptionParams` mostrati nel Listato 3.27 appaiono molto simili ai `ContainerEncryptionParams` del Listato 3.21.

Listato 3.27 EncryptionParams usati nel protocollo di Google Play Store.

```
class AndroidAppDelivery$EncryptionParams {
  --altro codice--
  private String encryptionKey;
  private String hmacKey;
  private int version;
}
```

L'algoritmo di crittografia e l'algoritmo HMAC delle applicazioni a pagamento scaricate da Google Play sono sempre impostati rispettivamente su *AES/CBC/PKCS5Padding* e *HMACSHA1*. IV e il tag MAC sono inseriti in un singolo blob con l'APK crittografato. Una volta letti e verificati tutti i parametri, questi vengono convertiti in un'istanza `ContainerEncryptionParams` e l'app viene installata con il metodo `PackageManager.installPackageWithVerification()`.

Il flag `INSTALL_FORWARD_LOCK` viene impostato durante l'installazione di un'app a pagamento per abilitare il forward locking. Il sistema operativo lo rileva e prosegue con il processo come descritto nei due paragrafi precedenti: le app gratuite vengono decriptate e i relativi APK finiscono in */data/app/*, mentre per le app a pagamento viene creato un contenitore crittografato in */data/app-asec/* e lo stesso viene montato in */mnt/asec/<package-name>*.

Qual è il livello di sicurezza, in pratica? Google Play sostiene che le app a pagamento sono sempre trasferite e memorizzate in forma crittografata, e lo stesso vale per il canale di distribuzione della vostra app, se decidete di implementarla con le funzionalità di crittografia delle app fornite da Android. Il contenuto del file APK deve prima o poi essere messo a disposizione del sistema operativo, quindi se avete accesso root a un dispositivo Android è ancora possibile estrarre un APK con forward locking o la chiave di crittografia del contenitore.

Verifica dei package

La verifica dei package è stata introdotta come funzionalità ufficiale di Android nella versione 4.2 con il nome *verifica dell'applicazione* ed è successivamente stata trasferita a tutte le versioni con sistema operativo Android 2.3 (e successivi) e Google Play Store. L'infrastruttura che consente la verifica dei package è integrata nel sistema operativo, ma Android non viene fornito con alcun verifier integrato.
L'implementazione più usata per la verifica dei package è quella integrata nel client Google Play Store, sostenuta dall'infrastruttura di analisi delle app di Google. È studiata per proteggere i dispositivi Android da quelle che Google definisce "applicazioni potenzialmente dannose" (backdoor, applicazioni di phishing, spyware e così via), comunemente note come *malware* (Google, *Android Practical Security from* the *Ground Up*, presentato al VirusBulletin 2013; recuperato da `http://bit.ly/1tDYzYf`).

Se la verifica dei package è attivata, i file APK vengono esaminati da un verifier prima dell'installazione e il sistema mostra un avviso (Figura 3.3) oppure blocca l'installazione se il verifier ritiene l'APK potenzialmente dannoso. La verifica è attiva per impostazione predefinita sui dispositivi supportati, ma richiede l'approvazione dell'utente al primo utilizzo, in quanto invia i dati dell'applicazione a Google. La verifica delle applicazioni può essere attivata o disattivata tramite l'opzione *Verify Apps* nella schermata *Security* delle impostazioni di sistema (Figura 3.2).

Nei paragrafi seguenti vedremo l'infrastruttura di verifica dei package di Android, quindi esamineremo brevemente l'implementazione di Google Play.

Supporto di Android per la verifica dei package

Come per la maggior parte delle funzioni che riguardano la gestione delle applicazioni, la verifica dei package è implementata in `PackageManagerService` ed è disponibile a partire da Android 4.0 (API livello 14). La verifica è eseguita da uno o più *agent di verifica* e dispone di un

verifier obbligatorio e di zero o più *sufficient verifier*. La verifica è considerata completa quando il verifier obbligatorio e almeno uno dei sufficient verifier restituiscono un risultato positivo. Un'applicazione può registrarsi come verifier obbligatorio dichiarando un broadcast receiver con un filtro intent corrispondente all'azione `PACKAGE_NEEDS_VERIFICATION` e al tipo MIME del file APK (`application/vnd.android.package-archive`), come mostrato nel Listato 3.28.

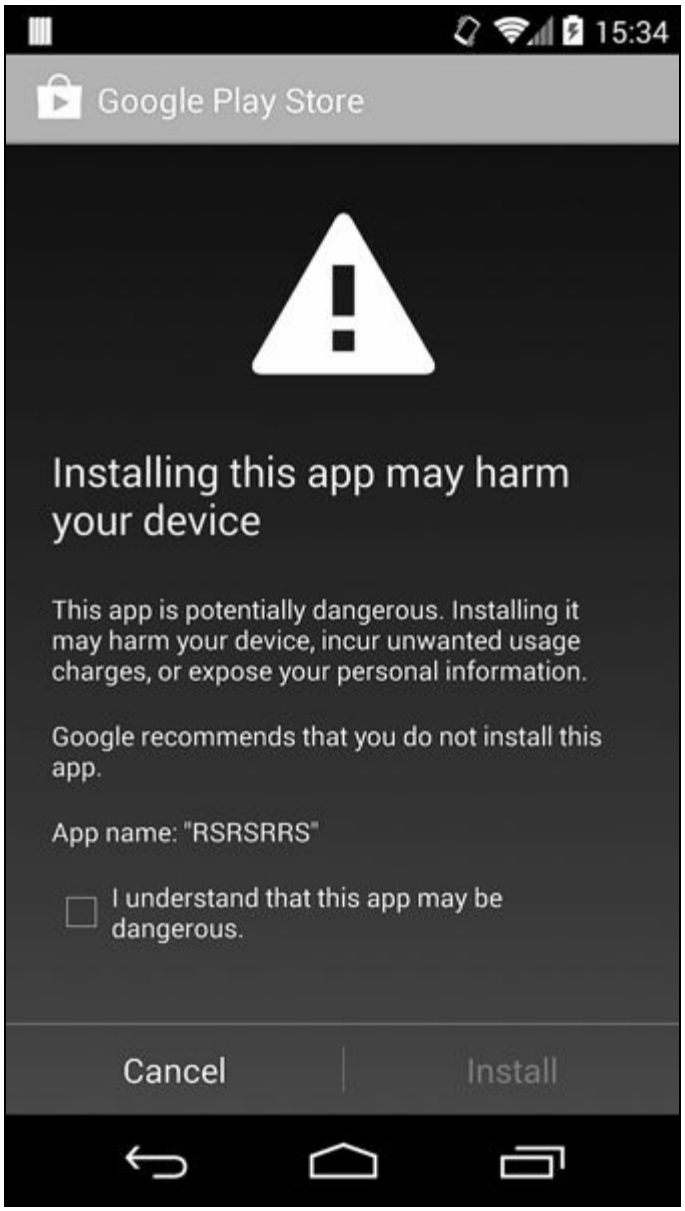

Figura 3.3 Finestra di avviso per la verifica delle applicazioni.

Listato 3.28 Dichiarazione di verifica obbligatoria in AndroidManifest.xml.

```
<receiver android:name=".MyPackageVerificationReceiver"
          android:permission="android.permission.BIND_PACKAGE_VERIFIER">
    <intent-filter>
        <action
             android:name="android.intent.action.PACKAGE_NEEDS_VERIFICATION" />
        <action android:name="android.intent.action.PACKAGE_VERIFIED" />
        <data android:mimeType="application/vnd.android.package-archive" />
```

```
    </intent-filter>
</receiver>
```

Inoltre, l'applicazione dichiarante deve disporre del permesso `PACKAGE_VERIFICATION_AGENT`: poiché si tratta di un permesso di firma riservato alle applicazioni di sistema (`signature|system`), solo le applicazioni di sistema possono divenire agent di verifica obbligatori.

Le applicazioni possono registrare sufficient verifier aggiungendo un tag `<package-verifier>` al loro manifest ed elencando il nome del package e la chiave pubblica del sufficient verifier negli attributi del tag (Listato 3.29).

Listato 3.28 Dichiarazione di sufficient verifier in AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.example.app">
        <package-verifier android:name="com.example.verifier"
                          android:publicKey="MIIB..." />
    <application ...>
     --altro codice--
    </application>
</manifest>
```

Durante l'installazione di un package, `PackageManagerService` esegue la verifica quando è installato un verifier obbligatorio e l'impostazione di sistema `Settings.Global.PACKAGE_VERIFIER_ENABLE` è `true`. La verifica viene abilitata aggiungendo l'APK a una coda di installazione in sospeso e inviando il broadcast `ACTION_PACKAGE_NEEDS_VERIFICATION` ai verifier registrati.

I broadcast contengono un verification ID univoco e vari metadati sul package da verificare. Gli agent di verifica rispondono chiamando il metodo `verifyPendingInstall()` e passando il verification ID e uno stato di verifica. La chiamata al metodo richiede il permesso `PACKAGE_VERIFICATION_AGENT`, che garantisce che le app non di sistema non possano partecipare alla verifica dei package. Ogni volta che viene chiamato `verifyPendingInstall()`, `PackageManagerService` controlla se è stata ricevuta una verifica sufficiente per l'installazione in sospeso; in questo caso rimuove l'installazione in sospeso dalla coda, invia il broadcast `PACKAGE_VERIFIED` e avvia il processo di installazione del package. Se il package viene rifiutato dagli agent di verifica, o se non si riceve una verifica sufficiente nel tempo previsto, l'installazione termina con l'errore `INSTALL_FAILED_VERIFICATION_FAILURE`.

Implementazione di Google Play

L'implementazione della verifica delle applicazioni di Google è integrata nel client Google Play Store. L'app Google Play Store si registra come agent di verifica obbligatorio e, se l'opzione di verifica delle app è attivata, riceve un broadcast ogni volta che sta per essere installata un'applicazione, sia tramite il client Google Play Store stesso, sia tramite l'applicazione `PackageInstaller` o `adb install`.

L'implementazione non è open source e ben pochi dettagli sono disponibili al pubblico, ma la pagina dell'help Android di Google "Protezione contro app dannose" afferma: "Quando verifichi le applicazioni, Google riceve informazioni di log, URL relativi all'applicazione e informazioni generiche sul dispositivo, come l'ID del dispositivo, la versione del sistema operativo e l'indirizzo IP" (`http://bit.ly/1rCIcsw`). Nel periodo in cui è stato scritto questo libro, il client Play Store inviava il valore hash SHA-256 del file APK, le dimensioni del file, il nome del package dell'app, i nomi delle sue risorse con i relativi hash SHA-256, gli hash SHA-256 dei file manifest e delle classi dell'app, il suo codice di versione e i certificati di firma, nonché alcuni metadati sull'applicazione di installazione e sugli URL di riferimento, se disponibili. Sulla base di queste informazioni, gli algoritmi di analisi APK di Google determinano se il file APK è potenzialmente dannoso e restituiscono al client Play Store un risultato contenente un codice di stato e un messaggio di errore da visualizzare qualora l'APK sia ritenuto potenzialmente dannoso. A sua volta, il client Play Store chiama il metodo `verifyPendingInstall()` di `PackageManagerService` con il codice di stato appropriato. L'installazione dell'applicazione viene accettata o rifiutata in base all'algoritmo descritto nel paragrafo precedente.

In pratica (almeno sui dispositivi "Google Experience"), il verifier di Google Play Store è solitamente l'unico agent di verifica, quindi l'installazione o il rifiuto del package dipende unicamente dalla risposta del servizio di verifica online di Google.

Riepilogo

I package delle applicazioni Android (file APK) sono un'estensione del formato di file JAR e contengono risorse, codice e un file manifest. I file APK sono firmati usando il formato di firma del codice dei file JAR, ma richiedono che tutti i file siano firmati con lo stesso set di certificati. Android usa il certificato del firmatario del codice per stabilire la medesima origine delle app e dei loro aggiornamenti e per stabilire relazioni di trust tra le app. I file APK vengono installati copiandoli nella directory */data/app/* e creando una directory dati dedicata per ogni applicazione in */data/data/*.

Android supporta i file APK crittografati e i contenitori di app protetti per le app con forward locking. Le app crittografate vengono automaticamente decriptate prima di essere copiate nella directory delle applicazioni. Le app con forward locking sono suddivise in una parte pubblicamente accessibile contenente le risorse e il manifest e in una parte privata con il codice e gli asset, quest'ultima salvata in un contenitore crittografato dedicato a cui può accedere direttamente solo il sistema operativo.

Facoltativamente Android può verificare le app prima di installarle consultando uno o più agent di verifica. Attualmente, l'agent di verifica più utilizzato è quello integrato nel client Google Play Store, che usa il servizio di verifica online delle app di Google per rilevare le applicazioni potenzialmente dannose.

Gestione degli utenti

Android in origine era destinato ai dispositivi personali, come gli smartphone, pertanto presumeva che per ogni dispositivo esistesse un solo utente. Visto l'aumento della popolarità dei tablet e di altri device condivisi, nella versione 4.2 è stato aggiunto il supporto multiutente, poi esteso nelle versioni successive.

In questo capitolo vedremo la gestione degli utenti che condividono dispositivi e dati in Android. Per iniziare parleremo dei tipi di utenti supportati da Android e della modalità di conservazione dei metadati utente, poi ci occuperemo del modo in cui Android condivide le applicazioni installate tra gli utenti pur isolando i dati dell'applicazione e mantenendoli privati per ogni utente. Infine, vedremo in che modo Android implementa la memoria esterna isolata.

Panoramica sul supporto multiutente

Il supporto multiutente di Android consente a più utenti di condividere un singolo device mettendo a disposizione di ogni utente un ambiente personale isolato. Ogni utente può quindi avere una propria schermata home, widget, app, account online e file che non sono accessibili agli altri utenti.

Gli utenti sono identificati da uno *user ID* univoco (da non confondersi con gli UID Linux) e solo il sistema può effettuare il cambio di utente.

Questa operazione viene normalmente attivata dalla selezione di un utente nella schermata di blocco di Android e, facoltativamente, dall'autenticazione con un pattern, un PIN, una password e così via (fate riferimento al Capitolo 10). Le applicazioni possono ottenere informazioni sull'utente corrente dall'API `UserManager`, ma in genere non è necessario effettuare modifiche al codice per supportare un ambiente multiutente. Le applicazioni che necessitano di cambiare il loro comportamento se utilizzate da un profilo con restrizioni rappresentano un'eccezione: queste applicazioni richiedono un codice aggiuntivo che verifichi quali restrizioni sono eventualmente imposte all'utente corrente (leggete il paragrafo "Profili con restrizioni" per i dettagli). Il supporto multiutente è integrato nella piattaforma Android core ed è quindi disponibile su tutti i dispositivi con sistema operativo Android 4.2 o versioni successive. Tuttavia, la configurazione predefinita della piattaforma prevede un solo utente e di conseguenza disabilita il supporto multiutente. Per abilitare il supporto di più utenti è necessario impostare la risorsa di sistema `config_multiuserMaximumUsers` a un valore maggiore di uno, generalmente aggiungendo un file di configurazione sostitutivo specifico per il dispositivo. Per esempio, su Nexus 7 (2013) la sostituzione viene inserita nel file `device/asus/flo/overlay/frameworks/base/core/res/res/values/config.xml` e l'impostazione `config_multiuserMaximumUsers` viene definita come mostrato nel Listato 4.1 per consentire un massimo di otto utenti.

Listato 4.1 Abilitazione del supporto multiutente con un file di sostituzione delle risorse.

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    --altro codice--
    <!-- Numero massimo di utenti supportati -->
    <integer name="config_multiuserMaximumUsers">8</integer>
    --altro codice--
</resources>
```

NOTA

La definizione di compatibilità di Android richiede che sui dispositivi che supportano la telefonia (come i cellulari) non sia abilitato il supporto multiutente, in quanto il comportamento delle API di telefonia sui device con più utenti attualmente non è definito (*Android 4.4 Compatibility Definition*, http://bit.ly/1GwI0CI). Di conseguenza, nelle build di produzione attuali, tutti i telefoni sono configurati come dispositivi monoutente.

Se viene abilitato il supporto multiutente, l'applicazione delle impostazioni di sistema mostra una voce *Users* che consente al proprietario del dispositivo (il primo utente creato, come spiegato nel prossimo paragrafo) di creare e gestire gli utenti. La schermata di gestione degli utenti è mostrata nella Figura 4.1.

Non appena vengono creati più utenti, la schermata di blocco mostra un widget utente che visualizza gli utenti correnti e permette di cambiare utente. La Figura 4.2 mostra l'aspetto della schermata di blocco su un dispositivo con otto utenti.

Tipi di utenti

Anche se Android è privo delle funzionalità complete di gestione degli utenti presenti nella maggior parte dei sistemi operativi multiutente, che generalmente permettono di aggiungere più amministratori e di definire gruppi di utenti, supporta comunque la configurazione di tipi di utenti con privilegi diversi. Ogni tipo di utente è descritto, insieme ai privilegi relativi, nei prossimi paragrafi.

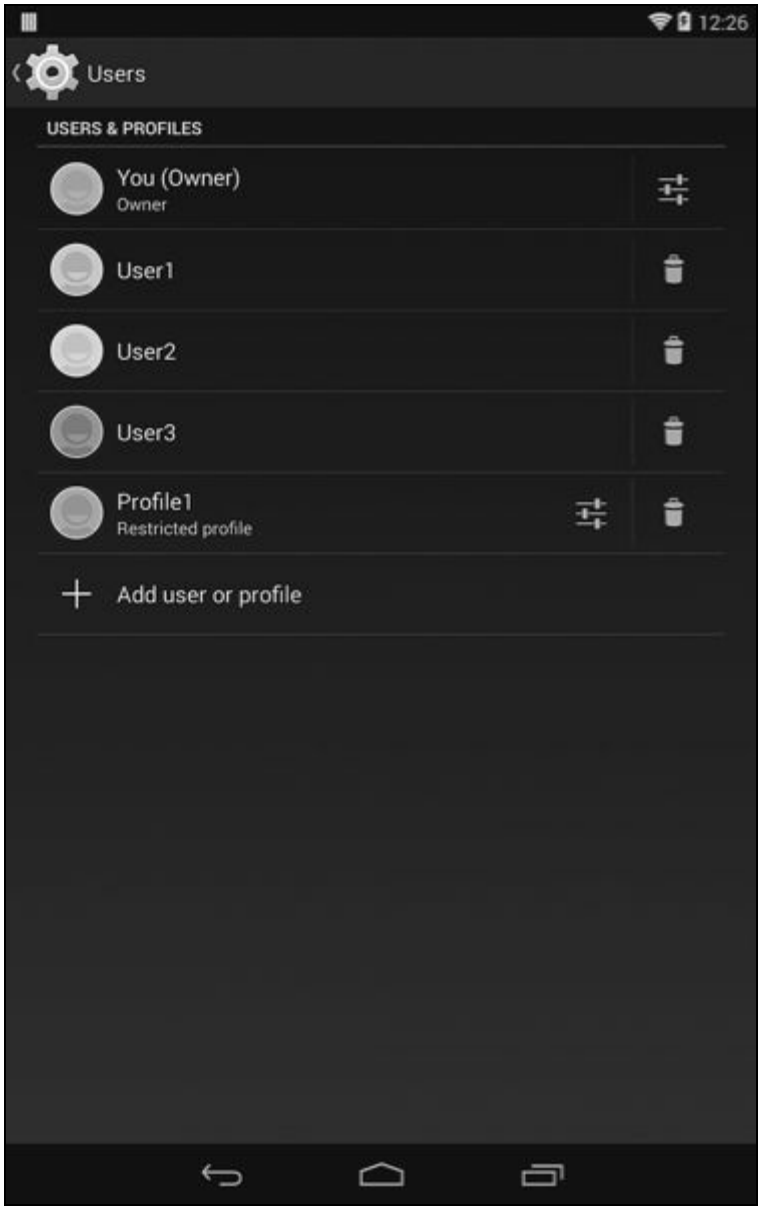

Figura 4.1 Schermata di gestione degli utenti.

Utente primario (proprietario)

L’*utente primario*, detto anche *proprietario* del dispositivo, è il primo utente creato su un dispositivo multiutente o l’unico su un device monoutente. Il proprietario viene creato per impostazione predefinita ed è sempre presente; a lui viene assegnato lo user ID 0. Sui dispositivi monoutente, dove l’utente primario è l’unico utente, Android si comporta in maniera simile alle versioni precedenti prive del supporto multiutente: le directory e gli UID assegnati alle applicazioni installate mantengono lo stesso formato e gli stessi permessi delle versioni precedenti (consultate i paragrafi “Gestione degli utenti” e “Condivisione delle applicazioni” per i dettagli).

All’utente primario vengono assegnati tutti i privilegi, pertanto può creare ed eliminare altri utenti e cambiare le impostazioni che influiscono su tutti gli altri, comprese quelle relative alla sicurezza del dispositivo, alla connettività di rete e alla gestione delle applicazioni. I privilegi di gestione degli utenti e del dispositivo vengono concessi all’utente primario mostrando le rispettive schermate delle impostazioni, che vengono invece nascoste agli altri utenti. Inoltre, i servizi di sistema sottostanti verificano l’identità dell’utente chiamante prima di eseguire operazioni che possono interessare tutti gli utenti e consentono l’esecuzione solo se la chiamata è stata effettuata dal proprietario del dispositivo.

Figura 4.2 Schermata di blocco con widget per il cambio utente.

A partire da Android versione 4.4, le seguenti schermate della sezione *Wireless and Networks* delle impostazioni di sistema sono visibili solo all'utente primario:

- *Cell Broadcasts*;
- *Manage Mobile Plan*;
- *Mobile Network*;
- *Tethering and Portable Hotspot*;
- *VPN*;
- *WiMAX* (se supportato dal dispositivo).

Anche le seguenti schermate della sezione *Security* sono riservate all'utente primario:

- *Device Encryption*;
- *SIM Card Lock*;
- *Unknown Sources* (che controlla il sideloading delle app, come spiegato nel Capitolo 3);
- *Verify Apps* (che controlla la verifica dei package, come spiegato nel Capitolo 3).

Utenti secondari

Fatta eccezione per i profili con restrizioni (di cui parliamo nel prossimo paragrafo), tutti gli utenti aggiunti sono *utenti secondari*. Ognuno riceve una directory utente dedicata (leggete "Gestione degli utenti" più avanti in questo capitolo), il proprio elenco di app installate e le directory dati private per ogni app installata.

Gli utenti secondari non possono aggiungere o gestire gli utenti; possono soltanto impostare il loro username dalla schermata *Users* (Figura 4.1). Inoltre, non possono eseguire le operazioni privilegiate, che sono riservate all'utente primario (come spiegato nei paragrafi precedenti). Possono comunque eseguire molte delle operazioni consentite all'utente primario, come installare e usare le applicazioni e cambiare l'aspetto e le impostazioni del sistema.

Per quanto siano soggetti a restrizioni, gli utenti secondari possono comunque influire sul comportamento del dispositivo e sugli altri utenti, per esempio possono aggiungere e connettersi a una nuova rete Wi-Fi. Poiché lo stato di connettività Wi-Fi è condiviso a livello di sistema, il passaggio a un utente diverso non provoca il reset della connessione wireless, quindi il nuovo utente risulterà connesso alla rete wireless scelta dall'utente precedente. Gli utenti secondari possono inoltre attivare e disattivare la modalità aereo e NFC, nonché cambiare le impostazioni globali dello schermo e dell'audio. Quello che è più importante notare è che, dal momento che i package delle applicazioni sono condivisi tra tutti gli utenti (come sarà spiegato nel paragrafo "Condivisione delle applicazioni"), se un utente secondario aggiorna un'applicazione che

aggiunge nuovi permessi, i permessi verranno concessi all'applicazione senza richiedere il consenso di altri utenti, e gli altri utenti non verranno informati del cambiamento a livello di permessi.

Profili con restrizioni

A differenza degli utenti secondari, i profili con restrizioni (aggiunti in Android 4.3) sono basati sull'utente primario e ne condividono le applicazioni, i suoi dati e i suoi account, ma con alcune restrizioni. Di conseguenza, l'utente primario deve impostare una password di blocco dello schermo per proteggere i suoi dati: se tale password non è configurata, quando l'utente primario crea un profilo con restrizioni Android lo invita a crearne una.

Restrizioni utente

Android definisce le restrizioni elencate di seguito predefinite per controllare ciò che possono fare gli utenti. Tutte le restrizioni sono `false` di default. L'elenco mostra tra parentesi il valore relativo agli utenti con restrizioni.

- `DISALLOW_CONFIG_BLUETOOTH`: specifica se un utente non può configurare Bluetooth. Impostazione predefinita: `false`.
- `DISALLOW_CONFIG_CREDENTIALS`: specifica se un utente non può configurare le credenziali utente. Se questa restrizione è `true`, i profili con restrizioni non possono aggiungere certificati di CA attendibili né importare chiavi private nell'archivio delle credenziali di sistema; leggete i Capitoli 6 e 7 per i dettagli. Impostazione predefinita: `false`.
- `DISALLOW_CONFIG_WIFI`: specifica se un utente non può cambiare i punti di accesso Wi-Fi. Impostazione predefinita: `false`.
- `DISALLOW_INSTALL_APPS`: specifica se un utente non può installare applicazioni. Impostazione predefinita: `false`.
- `DISALLOW_INSTALL_UNKNOWN_SOURCES`: specifica se un utente non può abilitare l'impostazione per le origini sconosciute (fate riferimento al Capitolo

3). Impostazione predefinita: `false`.

- `DISALLOW_MODIFY_ACCOUNTS`: specifica se un utente non può aggiungere e rimuovere account. Impostazione predefinita: `true`.
- `DISALLOW_REMOVE_USER`: specifica se un utente non può rimuovere utenti. Impostazione predefinita: `false`.
- `DISALLOW_SHARE_LOCATION`: specifica se un utente non può attivare o disattivare la condivisione della posizione. Impostazione predefinita: `true`.
- `DISALLOW_UNINSTALL_APPS`: specifica se un utente non può disinstallare applicazioni. Impostazione predefinita: `false`.
- `DISALLOW_USB_FILE_TRANSFER`: specifica se un utente non può trasferire file tramite USB. Impostazione predefinita: `false`.

Applicazione delle restrizioni

In fase di esecuzione, le applicazioni possono utilizzare il metodo `UserManager.getUserRestrictions()` per ottenere un `Bundle` (una classe contenitore universale che mappa chiavi stringa su vari tipi di valori) contenente le restrizioni imposte a un utente. Le restrizioni sono definite da coppie chiave-valore, dove la chiave è il nome della restrizione e il valore booleano specifica se è attiva. Le applicazioni possono utilizzare tale valore per disabilitare alcune funzionalità durante l'esecuzione in un profilo con restrizioni. Per esempio, l'app delle impostazioni di sistema controlla il valore della restrizione `DISALLOW_SHARE_LOCATION` quando visualizza le preferenze sulla posizione. Se il valore è `true`, l'impostazione della modalità per la posizione viene disabilitata. Un altro esempio riguarda `PackageManagerService`, che controlla le restrizioni `DISALLOW_INSTALL_APPS` e `DISALLOW_UNINSTALL_APPS` prima di installare o disinstallare le app e restituisce il codice di errore `INSTALL_FAILED_USER_RESTRICTED` se tali restrizioni sono impostate a `true` per l'utente chiamante.

L'utente primario può scegliere quali applicazioni saranno disponibili per un profilo con restrizioni. Quando viene creato un profilo con

restrizioni, tutte le applicazioni installate sono inizialmente disabilitate e il proprietario deve abilitare esplicitamente quelle che vuole mettere a disposizione di tale profilo (Figura 4.3).

Oltre alle restrizioni predefinite dal sistema operativo, le applicazioni possono definire restrizioni personalizzate creando un `BroadcastReceiver` che riceve l'intent `ACTION_GET_RESTRICTION_ENTRIES`. Android richiama questo intent per interrogare tutte le app, conoscere le restrizioni disponibili e creare automaticamente una UI che consente al proprietario del dispositivo di attivare o disattivare le restrizioni personalizzate dell'app.

Sempre durante il runtime, le applicazioni possono utilizzare il metodo `UserManager.getApplicationRestrictions()` per ottenere un `Bundle` contenente le restrizioni salvate come coppie chiave-valore. L'applicazione può quindi disabilitare o modificare alcune funzionalità in base alle restrizioni applicate. Il proprietario del dispositivo può attivare e disattivare le restrizioni di sistema e personalizzate nella stessa schermata delle impostazioni usata per gestire le applicazioni disponibili per un profilo con restrizioni.

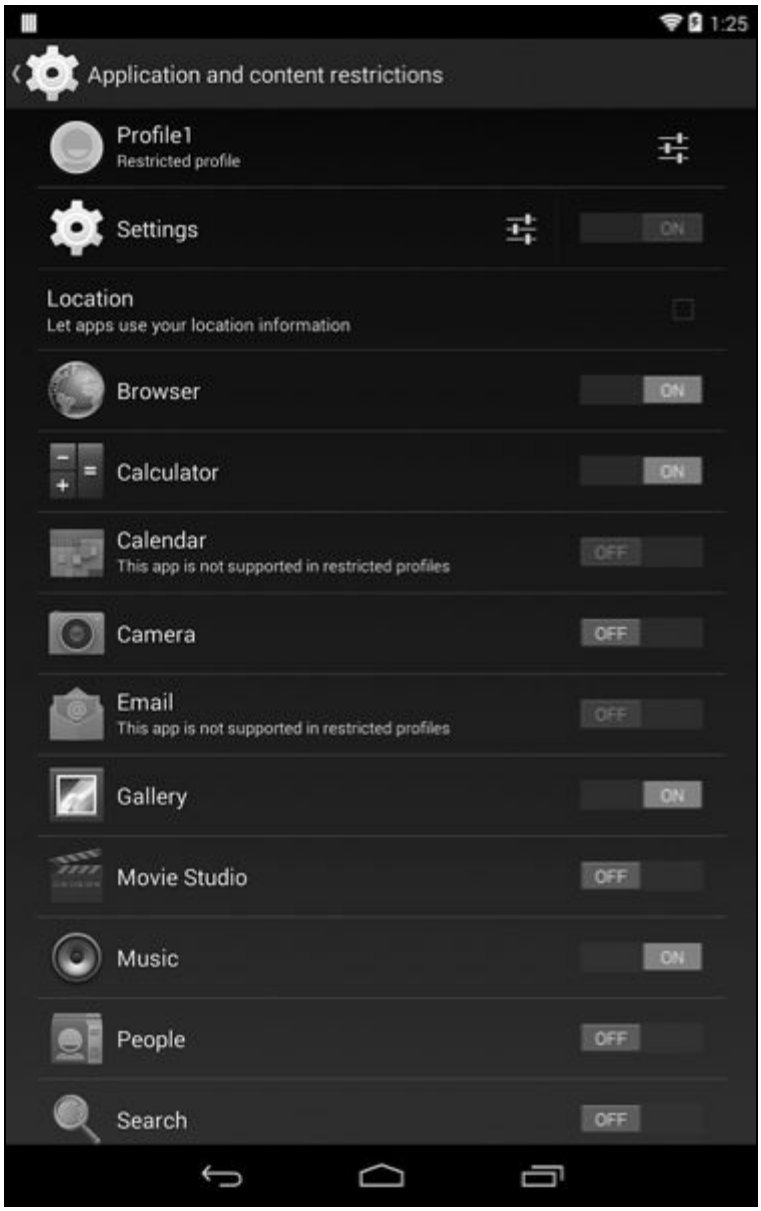

Figura 4.3 Schermata di gestione del profilo con restrizioni.

Per esempio, nella Figura 4.3 l'unica restrizione dell'applicazione supportata dall'app *Settings* (per stabilire se le app possono usare le informazioni sulla posizione) è mostrata sotto l'interruttore dell'applicazione principale.

Accesso agli account online

I profili con restrizioni possono inoltre accedere agli account online dell'utente primario tramite l'API `AccountManager` (consultate il Capitolo 8); questo accesso è però disabilitato per impostazione predefinita. Le applicazioni che devono accedere agli account durante l'esecuzione in un profilo con restrizioni devono dichiarare esplicitamente i tipi di account

di cui necessitano usando l'attributo `restrictedAccountType` del tag `<application>`, come mostrato nel Listato 4.2.

Listato 4.2 Consentire l'accesso agli account del proprietario da un profilo con restrizioni.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.app" ...>
    <application android:restrictedAccountType="com.google" ... >
        --altro codice--
    </application>
</manifest>
```

Le applicazioni che invece non vogliono esporre le informazioni degli account ai profili con restrizioni possono dichiararlo specificando il tipo di account (un asterisco consente la corrispondenza con tutti i tipi di account) come valore dell'attributo `requiredAccountType` del tag `<application>`. Se è specificato l'attributo `requiredAccountType`, Android disabilita automaticamente tali applicazioni per i profili con restrizioni. Per esempio, poiché l'applicazione *Calendario* di Android dichiara `android:requiredAccountType="*"` nel suo manifest, non può essere messa a disposizione dei profili con restrizioni ed è disabilitata nella schermata delle impostazioni di restrizione (Figura 4.3).

Utente guest

Android supporta un singolo utente guest, ma di default questa funzionalità è disabilitata. L'utente guest può essere abilitato chiamando il metodo `UserManager.setGuestEnabled()`, ma nelle attuali versioni di Android non sembra essere referenziato altrove che in `UserManager` e nelle classi correlate. I commenti nel codice indicano che l'utente guest potrebbe essere temporaneo, ma attualmente il suo scopo preciso non è chiaro. Sembra essere una reminiscenza di una funzionalità proposta ma poi rifiutata o mai implementata completamente.

Gestione degli utenti

Gli utenti di Android sono gestiti da `UserManagerService`, responsabile della lettura e della persistenza delle informazioni utente e della gestione dell'elenco di utenti attivi. La gestione degli utenti è strettamente correlata a quella dei package, pertanto `PackageManagerService` chiama `UserManagerService` per interrogare o modificare gli utenti durante l'installazione o la rimozione dei package. La classe `android.os.UserManager` offre una facciata per `UserManagerService` ed espone un sottoinsieme delle sue funzionalità alle applicazioni di terze parti. Le applicazioni possono ottenere il numero di utenti su un sistema, un numero di serie dell'utente, il nome e l'elenco di restrizioni per l'utente corrente, nonché l'elenco di restrizioni per un package, senza necessitare di permessi speciali. Tutte le altre operazioni utente, quali query, aggiunte, rimozioni o modifiche agli utenti, richiedono il permesso di firma del sistema `MANAGE_USERS`.

Strumenti a riga di comando

Le operazioni di gestione degli utenti possono essere eseguite anche nella shell di Android con il comando `pm`. Questi comandi possono essere eseguiti tramite la shell senza permessi root, perché l'utente `shell` (UID 2000) possiede il permesso `MANAGE_USERS`. Potete usare il comando `pm create-user` per creare un nuovo utente e `pm remove-user` per rimuoverlo. Il comando `pm get-max-users` restituisce il numero massimo di utenti supportati dal sistema operativo, mentre `pm list users` elenca tutti gli utenti. L'output del comando `pm list users` potrebbe essere simile a quello del Listato 4.3 su un dispositivo con cinque utenti. Il numeri tra parentesi graffe indicano lo user ID, il nome e i flag.

Listato 4.3 Recupero dell'elenco degli utenti con il comando pm list.

```
$ pm list users
Users:
        UserInfo{0:Owner:13}
        UserInfo{10:User1:10}
        UserInfo{11:User2:10}
        UserInfo{12:User3:10}
        UserInfo{13:Profile1:18}
```

Stati utente e broadcast correlati

`UserManagerService` invia diversi broadcast per informare gli altri componenti di eventi legati agli utenti. All'aggiunta di un utente invia il broadcast `USER_ADDED`, mentre quando viene rimosso un utente invia `USER_REMOVED`. Se viene cambiato lo username o l'icona del profilo, `UserManagerService` invia il broadcast `USER_INFO_CHANGED`. Il cambio di utente attiva i broadcast `USER_BACKGROUND`, `USER_FOREGROUND` e `USER_SWITCHED`, che contengono tutti lo user ID pertinente come extra. Anche se Android supporta un massimo di otto utenti, solo tre possono essere in esecuzione contemporaneamente. Un utente viene avviato quando viene attivato dal meccanismo di cambio utente nella schermata di blocco. Android blocca gli utenti inattivi utilizzando un algoritmo di cache LRU (*Least Recently Used*, utilizzati meno di recente) per assicurare che non vi siano più di tre utenti attivi. Quando un utente viene fermato, i suoi processi vengono terminati e non riceve più alcun broadcast. All'avvio o all'arresto degli utenti, il sistema invia i broadcast `USER_STARTING`, `USER_STARTED`, `USER_STOPPING` e `USER_STOPPED`. L'utente primario viene avviato automaticamente al boot del sistema e non viene mai bloccato. L'avvio, l'arresto e il cambio di utente, nonché la selezione di un utente specifico in un broadcast, richiedono il permesso `INTERACT_ACROSS_USERS`. Si tratta di un permesso di sistema con protezione signature e con il flag `development` impostato (consultate il Capitolo 2), quindi può essere concesso in maniera dinamica alle applicazioni non di sistema che lo dichiarano (usando il comando `pm grant`). Il permesso di firma `INTERACT_ACROSS_USERS_FULL` consente l'invio di broadcast a tutti gli utenti, la modifica dell'amministratore del dispositivo e altre operazioni privilegiate che influiscono su tutti gli utenti.

Metadati utente

Android archivia i dati utente nella directory */data/system/users/*, che ospita i metadati degli utenti in formato XML, e nelle directory utente. Su un dispositivo con cinque utenti, il contenuto può apparire come nel Listato 4.4 (timestamp omessi).

Listato 4.4 Contenuti di /data/system/users/.

```
# ls -lF /data/system/users
drwx------ system   system            0(1)
-rw------- system   system        230 0.xml(2)
drwx------ system   system            10
-rw------- system   system        245 10.xml
drwx------ system   system            11
-rw------- system   system        245 11.xml
drwx------ system   system            12
-rw------- system   system        245 12.xml
drwx------ system   system            13
-rw------- system   system        299 13.xml
-rw------- system   system        212 userlist.xml(3)
```

File dell'elenco utenti

Nel Listato 4.4 ogni utente possiede una directory dedicata, chiamata *directory di sistema utente*, il cui nome corrisponde allo user ID assegnato (**(1)** per l'utente primario), e un file XML contenente i metadati sull'utente, anche in questo caso con un nome file basato sullo user ID (**(2)** per l'utente primario). Il file `userlist.xml` **(3)** contiene i dati su tutti gli utenti creati su un sistema e può essere simile a quello mostrato nel Listato 4.5 su un sistema con cinque utenti.

Listato 4.5 Contenuto di userlist.xml.

```
<users nextSerialNumber="19" version="4">
    <user id="0" />
    <user id="10" />
    <user id="11" />
    <user id="12" />
    <user id="13" />
</users>
```

Il file è sostanzialmente un elenco di tag `<user>` contenenti l'ID assegnato a ogni utente. L'elemento root `<users>` dispone di un attributo `version` che specifica la versione corrente del file e un attributo `nextSerialNumber` contenente il numero di serie da assegnare all'utente successivo. All'utente primario viene sempre assegnato lo user ID 0.

Il fatto che gli UID assegnati alle applicazioni siano basati sullo user ID dell'utente proprietario garantisce che, sui dispositivi a singolo utente, gli UID assegnati alle applicazioni siano gli stessi utilizzati prima dell'introduzione del supporto multiutente (per ulteriori informazioni sugli UID delle applicazioni, leggete il paragrafo "Directory dati delle applicazioni"). Per gli utenti secondari e ai profili con restrizioni, gli ID vengono assegnati partendo dal numero 10.

File dei metadati utente

Gli attributi di ogni utente sono salvati in un file XML dedicato. Il Listato 4.6 mostra un esempio per un profilo con restrizioni.

Listato 4.6 Contenuto del file dei metadati utente.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<user id="13"
      serialNumber="18"
      flags="24"
      created="1394551856450"
      lastLoggedIn="1394551882324"
      icon="/data/system/users/13/photo.png">(1)
    <name>Profile1</name>(2)
    <restrictions no_modify_accounts="true" no_share_location="true" />(3)
</user>
```

Qui il tag `<name>` **(2)** contiene il nome dell'utente, mentre il tag `<restrictions>` **(3)** dispone di attributi per ogni restrizione abilitata (fate riferimento al paragrafo "Profili con restrizioni" per un elenco delle restrizioni predefinite). La Tabella 4.1 riepiloga gli attributi dell'elemento root `<user>` mostrati al punto **(1)** del Listato 4.6.

Tabella 4.1 Attributi dell'elemento <user>.

Nome	Formato	Descrizione
`id`	Integer	User ID.
`serialNumber`	Integer	Numero di serie dell'utente.
`flags`	Integer	Flag che indicano il tipo di utente.
`created`	Millisecondi da epoca Unix, come da `System.currentTimeMillis()`	Data/ora di creazione dell'utente.
`lastLoggedIn`	Millisecondi da epoca Unix, come da `System.currentTimeMillis()`	Data/ora dell'ultimo login.
`icon`	String	Percorso completo del file dell'icona utente.
`partial`	Booleano	Indica che l'utente è parzialmente inizializzato. Per gli utenti parziali potrebbero ancora non essere stati creati tutti i file e le directory.
`pinHash`	String esadecimale	Hash del PIN SHA1+MD5 con salt per restrizioni protette da

`pinHash`	String esadecimale	PIN.
`salt`	Long integer	Salt del PIN per restrizioni protette da PIN.
`failedAttempts`	Integer	Numero di tentativi di inserimento PIN non riusciti per restrizioni protette da PIN.
`lastAttemptMs`	Millisecondi da epoca Unix, come da `System.currentTimeMillis()`	Data/ora dell'ultimo tentativo di inserimento del PIN per restrizioni protette da PIN (in millisecondi da epoca Unix, come da `System.currentTimeMillis()`).

L'attributo `flags` è uno dei più importanti in quanto determina il tipo di utente. Attualmente per il tipo di utente sono usati sei bit del valore del flag; gli altri sono riservati. I flag definiti sono i seguenti:

- `FLAG_PRIMARY` (0x00000001): segnala l'utente primario.
- `FLAG_ADMIN` (0x00000002): segnala gli utenti amministratori. L'amministratore può creare ed eliminare utenti.
- `FLAG_GUEST` (0x00000004): segnala l'utente guest.
- `FLAG_RESTRICTED` (0x00000008): segnala gli utenti con restrizioni.
- `FLAG_INITIALIZED` (0x00000010): segnala un utente come totalmente inizializzato.

Per quanto siano possibili svariate configurazioni dei flag, la maggior parte di queste non rappresenta uno stato o un tipo di utente valido. In pratica, gli attributi per il proprietario sono impostati a 19 (0x13 o `FLAG_INITIALIZED|FLAG_ADMIN|FLAG_PRIMARY`); gli utenti secondari hanno i flag 16 (0x10 o `FLAG_INITIALIZED`), mentre i profili con restrizioni hanno i flag 24 (0x18 o `FLAG_INITIALIZED|FLAG_RESTRICTED`).

Directory di sistema utente

Ogni directory di sistema utente contiene impostazioni di sistema e dati specifici per l'utente, ma non i dati delle applicazioni. Come sarà spiegato nel prossimo paragrafo, ogni applicazione installata da un utente riceve una directory dati dedicata in */data*, proprio come sui dispositivi con un solo utente (consultate il Capitolo 3 per ulteriori informazioni sulle directory dati delle applicazioni). Per esempio, nel caso di un utente secondario con user ID 12, la directory di sistema utente è chiamata

/data/system/users/12/ e può contenere i file e le directory elencati nel Listato 4.7.

Listato 4.7 Contenuto di una directory utente.

```
- accounts.db(1)
- accounts.db-journal
- appwidgets.xml(2)
- device_policies.xml(3)
- gesture.key(4)
d inputmethod(5)
- package-restrictions.xml(6)
- password.key(7)
- photo.png(8)
- settings.db(9)
- settings.db-journal
- wallpaper(10)
- wallpaper_info.xml
```

Il file `accounts.db` **(1)** è un database SQLite che contiene i dettagli degli account online (la gestione degli account online è presentata nel Capitolo 8). Il file `appwidgets.xml` **(2)** contiene le informazioni sui widget aggiunti dall'utente alla sua schermata home. Il file `device_policies.xml` **(3)** descrive la policy corrente del dispositivo (i dettagli sono forniti nel Capitolo 9), mentre `gesture.key` **(4)** e `password.key` **(7)** contengono rispettivamente l'hash del pattern di blocco schermo o il PIN/password (consultate il Capitolo 10 per i dettagli sul formato).

La directory `inputmethod` **(5)** contiene informazioni sui metodi di input. Il file `photo.png` **(8)** conserva la fotografia o l'immagine del profilo dell'utente. Il file `settings.db` **(9)** contiene le impostazioni di sistema specifiche per l'utente, mentre `wallpaper` **(10)** è l'immagine di sfondo attualmente selezionata. Il file `package-restrictions.xml` **(6)** definisce le applicazioni installate dall'utente e ne conserva lo stato (la condivisione delle applicazioni e i dati dell'applicazione per utente sono affrontati nel prossimo paragrafo).

Gestione delle applicazioni per utente

Come già spiegato nel paragrafo "Panoramica sul supporto multiutente", insieme alle impostazioni e agli account dedicati ogni utente riceve una copia personale, e inaccessibile agli altri utenti, dei dati delle applicazioni. Android ottiene questo risultato assegnando un nuovo effective UID per utente a ogni applicazione e creando una directory dati dell'applicazione dedicata di proprietà dell'UID in questione. I dettagli di questa implementazione sono disponibili nei prossimi paragrafi.

Directory dati delle applicazioni

Come spiegato nel Capitolo 3, Android installa i package APK copiandoli nella directory */data/app/* e creando una directory dati dedicata per ogni applicazione in */data/data/*. Quando è abilitato il supporto multiutente, questo layout non viene modificato ma esteso per supportare altri utenti. I dati delle applicazioni per l'utente primario restano in */data/data/* per la compatibilità con le versioni precedenti.

Se sul sistema esistono altri utenti quando viene installata una nuova applicazione, `PackageManagerService` crea le directory dati delle applicazioni per ogni utente. Come la directory dati dell'utente primario, queste directory sono create con l'aiuto del daemon `installd` (utilizzando il comando `mkuserdata`), perché l'utente `system` non dispone di privilegi sufficienti per cambiare la proprietà della directory.

Le directory dati utente sono salvate in */data/user/* e denominate in base allo user ID. La directory del proprietario del dispositivo (*0/*) è un collegamento simbolico a */data/data/*, come mostrato nel Listato 4.8.

Listato 4.8 Contenuto di /data/user/ su un dispositivo multiutente.

```
# ls -l /data/user/
lrwxrwxrwx root     root          0 -> /data/data/
drwxrwx--x system   system        10
drwxrwx--x system   system        11
drwxrwx--x system   system        12
drwxrwx--x system   system        13
```

Il contenuto di ogni directory dati delle applicazioni è equivalente a quello di */data/data/*, ma le directory dell'applicazione per ogni istanza

utente della stessa applicazione sono di proprietà di utenti Linux diversi, come mostrato nel Listato 4.9.

Listato 4.9 Contenuto delle directory dati delle applicazioni per l'utente primario e per un utente secondario.

```
# ls -l /data/data/1
drwxr-x--x u0_a12   u0_a12               com.android.apps.tag
drwxr-x--x u0_a0    u0_a0                com.android.backupconfirm
drwxr-x--x bluetooth bluetooth           com.android.bluetooth
drwxr-x--x u0_a16   u0_a16               com.android.browser(2)
drwxr-x--x u0_a17   u0_a17               com.android.calculator2
drwxr-x--x u0_a18   u0_a18               com.android.calendar
--altro codice--
# ls -l /data/user/13/3
ls -l /data/user/13
drwxr-x--x u13_system u13_system         android
drwxr-x--x u13_a12  u13_a12              com.android.apps.tag
drwxr-x--x u13_a0   u13_a0               com.android.backupconfirm
drwxr-x--x u13_bluetooth u13_bluetooth   com.android.bluetooth
drwxr-x--x u13_a16  u13_a16              com.android.browser(4)
drwxr-x--x u13_a17  u13_a17              com.android.calculator2
drwxr-x--x u13_a18  u13_a18              com.android.calendar
--altro codice--
```

Questo listato mostra il contenuto delle directory dati delle app per l'utente primario **(1)** e l'utente secondario con user ID `13` **(3)**. Come potete notare, anche se entrambi gli utenti hanno directory dati per le stesse app, come l'app browser (**(2)** per il proprietario e **(4)** per l'utente secondario), queste directory sono di proprietà di utenti Linux diversi: `u0_a16` nel caso del proprietario e `u13_a16` nel caso dell'utente secondario. Se controlliamo l'UID di questi utenti con i comandi `su` e `id`, vediamo che `u0_a16` ha UID=10016, mentre `u13_a16` ha UID=1310016.

Il fatto che entrambi gli UID contengano il numero 10016 non è una coincidenza: la parte ripetuta è l'*app ID*, che corrisponde all'UID assegnato all'app durante la prima installazione su un dispositivo monoutente. Sui dispositivi multiutente l'app UID viene derivato dallo user ID e dall'app ID utilizzando questo codice:

```
uid = userId * 100000 + (appId % 100000)
```

Visto che lo user ID del proprietario è sempre 0, gli UID per le app del proprietario del dispositivo corrispondono sempre ai relativi app ID. Se la stessa applicazione viene eseguita nel contesto di utenti diversi, viene eseguita con gli UID assegnati all'istanza dell'applicazione di ogni utente. Per esempio, se l'applicazione browser viene eseguita contemporaneamente dal proprietario del dispositivo e da un utente secondario con user ID 13, vengono creati due processi separati in

esecuzione con gli utenti Linux `u0_a16` e `u13_a16` (UID 10016 per il proprietario **(1)** e UID 1310016 per l'utente secondario **(2)**), come mostrato nel Listato 4.10.

Listato 4.10 Informazioni sui processi per l'applicazione browser eseguita da utenti del dispositivo diversi.

```
USER      PID   PPID  VSIZE   RSS   WCHAN    PC         NAME
--altro codice--
u13_a16   1149  180   1020680 72928 ffffffff 4006a58c R com.android.browser(1)
--altro codice--
u0_a16    30500 180   1022796 73384 ffffffff 4006b73c S com.android.browser(2)
--altro codice--
```

Condivisione delle applicazioni

Anche se le applicazioni installate hanno una directory dati dedicata per ogni utente, i file APK sono condivisi tra tutti gli utenti. I file APK vengono copiati in */data/app/* e sono leggibili da tutti gli utenti; le librerie condivise usate dalle app sono copiate in */data/app-lib/*<nome pacchetto>*/* e sono sottoposte a symlinking con */data/user/*<user ID>*/*<nome pacchetto>*/lib/*; i file Optimized DEX per ogni app sono salvati in */data/dalvik-cache/* e sono anch'essi condivisi da tutti gli utenti. Di conseguenza, una volta installata un'applicazione, questa è accessibile a tutti gli utenti del dispositivo, per ognuno dei quali viene creata automaticamente una directory dati dell'app.

Android consente agli utenti di avere applicazioni diverse creando un file `package-restrictions.xml` (**(6)** nel Listato 4.7) nella directory di sistema di ogni utente, che viene utilizzato per registrare se un'app è abilitata per un utente specifico. Oltre allo stato di installazione dei package, questo file contiene informazioni sui componenti disabilitati di ogni applicazione, nonché un elenco delle applicazioni preferite da avviare quando gli intent di elaborazione possono essere gestiti da più applicazioni (per esempio l'apertura di un file di testo). Il contenuto di `package-restrictions.xml` per un utente secondario è simile a quello mostrato nel Listato 4.11.

Listato 4.11 Contenuto del file package-restrictions.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<package-restrictions>
    <pkg name="com.example.app" inst="false" stopped="true" nl="true" />(1)
    <pkg name="com.example.app2" stopped="true" nl="true" />(2)
    --altro codice--
    <pkg name="com.android.settings">
        <disabled-components>
```

```
            <item name="com.android.settings.CryptKeeper" />
        </disabled-components>
    </pkg>
    <preferred-activities />
</package-restrictions>
```

Nell'esempio il package `com.example.app` è disponibile sul sistema, ma non è installato per l'utente secondario, come indicato dall'aggiunta di `<pkg>` per l'app e dall'impostazione dell'attributo `inst` su `false` **(1)**. Sulla base di queste informazioni, `PackageManagerService` segnala il package `com.example.app` come non installato per tale utente, pertanto il package non viene visualizzato nel launcher o nell'elenco di app nelle impostazioni.

Le applicazioni possono essere installate, ma comunque segnalate come arrestate, come mostrato nel punto **(2)**. Qui il package `com.example.app2` è installato ma segnalato come arrestato con l'impostazione dell'attributo `stopped` su `true`. Android dispone di uno stato speciale per le applicazioni che non sono mai state avviate, che viene reso persistente con l'attributo `nl` del tag `<pkg>`. Il proprietario del dispositivo può bloccare un package per un determinato utente; in questo caso l'attributo `blocked` è impostato su `true` (non mostrato nella Figura 4.4).

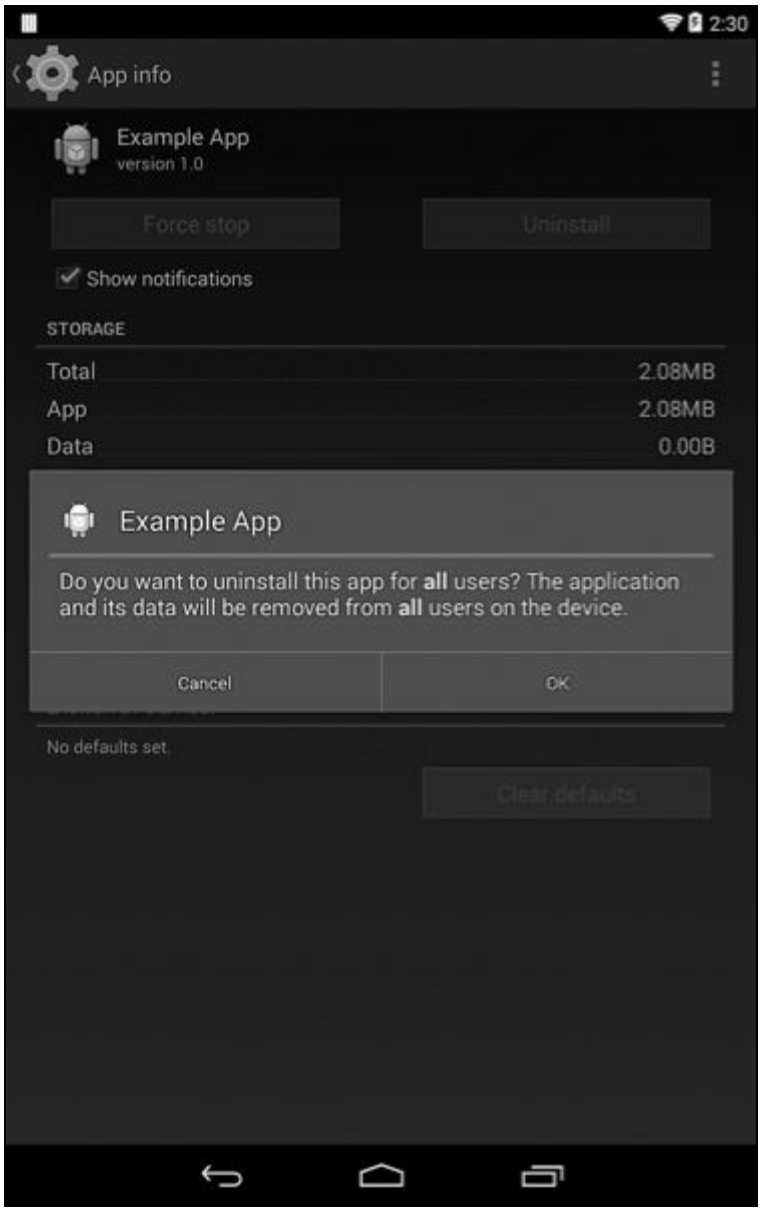

Figura 4.4 Avviso mostrato quando il proprietario del dispositivo tenta di disinstallare un'app per tutti gli utenti.

Quando un utente del dispositivo installa un'applicazione, viene aggiunto un tag `<pkg>` con `inst="false"` ai file `package-restrictions.xml` di tutti gli utenti. Quando un altro utente installa la medesima applicazione, l'attributo `inst` viene rimosso e l'applicazione viene considerata come installata per tale utente (a seconda della modalità di avvio del processo di installazione, il file APK in */data/app/* potrebbe essere sostituito come avviene in un aggiornamento dell'applicazione).

Gli utenti con restrizioni non possono installare le applicazioni, ma viene comunque applicata la stessa procedura quando il proprietario del dispositivo abilita un'app per un utente con restrizioni: l'applicazione viene installata chiamando il metodo

`PackageManagerService.installExistingPackageAsUser()`, che imposta il flag di installazione per il package e aggiorna `package-restrictions.xml` di conseguenza.

Quando un utente disinstalla un package, i relativi dati dell'app vengono eliminati e il flag interno di installazione del package viene impostato su `false`. Questo stato viene reso persistente impostando `inst="false"` nel tag del package rimosso all'interno del file `package-restrictions.xml` dell'utente. Il file APK e la directory delle librerie native vengono rimossi solo quando l'ultimo utente per cui è installata l'app la disinstalla. Tuttavia, il proprietario può vedere tutte le app installate nel sistema dalla scheda *All* della schermata *Apps Settings*, comprese quelle che non ha installato personalmente, e può quindi disinstallare tali app per tutti gli utenti. L'azione *Uninstall for all users* è nascosta all'interno del menu, quindi non può essere selezionata per sbaglio, e produce l'avviso mostrato nella Figura 4.4. Se l'utente seleziona *OK*, vengono rimosse le directory dell'app per tutti gli utenti e il file APK viene eliminato dal dispositivo.

Lo schema di condivisione delle app implementato sui dispositivi Android multiutente è compatibile con le versioni precedenti e permette di risparmiare spazio sul device evitando di copiare i file APK per tutti gli utenti. Tuttavia, presenta un importante svantaggio: un'applicazione può essere aggiornata da qualunque utente, anche se in origine è stata installata da un altro.

Questo schema di solito non è un problema, perché l'istanza dell'app di ogni utente dispone di una directory dati separata, ma a volte capita che l'aggiornamento aggiunga nuovi permessi. Visto che Android concede i permessi in fase di installazione, se un utente aggiorna un'app e accetta un nuovo permesso che agisce sulla privacy dell'utente (per esempio `READ_CONTACTS`), tale permesso viene applicato a tutti gli utenti che usano l'app. Gli altri utenti non vengono informati che all'app è stato concesso un nuovo permesso e potrebbero non accorgersi della modifica, a meno che non esaminino manualmente i dettagli dell'app nelle impostazioni di sistema. Android mostra un avviso che informa gli

utenti di questo quando abilitano per la prima volta il supporto multiutente, ma non invia notifiche successive in merito alle singole app.

Memoria esterna

Android ha incluso il supporto per la memoria esterna sin dalle sue prime versioni pubbliche. Poiché le prime generazioni dei dispositivi Android implementavano la memoria esterna semplicemente montando una scheda SD rimovibile con file system FAT, la memoria esterna è spesso definita come "la scheda SD". Detto questo, la definizione di memoria esterna è più ampia: in Android è un file system che non fa distinzione tra maiuscole e minuscole, dotato di modalità e classi di permessi POSIX invariabili (http://bit.ly/1fnGUvI). L'implementazione sottostante non è importante, purché soddisfi questa definizione.

Implementazioni della memoria esterna

I dispositivi più recenti tendono a implementare la memoria esterna con l'emulazione; alcuni non dispongono nemmeno dello slot per schede SD. Per esempio, l'ultimo dispositivo Google Nexus a disporre di uno slot per schede SD è stato Nexus One, rilasciato nel gennaio 2010; tutti i dispositivi Nexus rilasciati successivamente a Nexus S (che usa una partizione dedicata per la memoria esterna) implementano la memoria esterna tramite l'emulazione. Sui dispositivi privi di scheda SD, la memoria esterna viene implementata sia montando direttamente una partizione in formato FAT, residente sullo stesso dispositivo della memoria primaria, sia utilizzando un daemon helper per l'emulazione.

A partire da Android versione 4.4, le app possono gestire le loro directory specifiche per package (*Android/data/com.example.app/* per un'app con package `com.example.app`) sulla memoria esterna senza richiedere il permesso `WRITE_EXTERNAL_STORAGE`, che concede l'accesso a tutti i dati sulla memoria esterna, comprese le immagini della fotocamera, i video e altri elementi multimediali. In questo caso si parla di *permessi sintetizzati*: la relativa implementazione AOSP è basata su un daemon FUSE che esegue il wrapping della memoria non elaborata del dispositivo e gestisce

l'accesso ai file e i permessi in base a una specifica modalità di emulazione dei permessi.

NOTA

Filesystem in Userspace, o FUSE, è una funzione di Linux che permette l'implementazione di un file system totalmente funzionante in un programma dello userspace (http://fuse.sourceforge.net/). Questo risultato si ottiene utilizzando un modulo kernel FUSE generico che instrada tutte le chiamate di sistema *Virtual Filesystem* (VFS) per il file system target alla sua implementazione nello userspace. Il modulo kernel e l'implementazione dello userspace comunicano tramite un descrittore di file speciale ottenuto aprendo */dev/fuse*.

A partire da Android versione 4.4, le applicazioni possono accedere a più dispositivi di memoria esterna, ma sono in grado di scrivere file arbitrari solo sulla *memoria esterna primaria* (se possiedono il permesso `WRITE_EXTERNAL_STORAGE`) e hanno accesso limitato agli altri dispositivi di memoria esterni, definiti *memoria esterna secondaria*. La nostra descrizione si concentra principalmente sulla memoria esterna primaria, visto che è più legata al supporto multiutente.

Memoria esterna multiutente

Per supportare il modello di sicurezza di Android in un ambiente multiutente, il *Compatibility Definition Document* (CDD) impone numerosi requisiti per la memoria esterna. Il più importante di questi afferma che ogni istanza utente su un dispositivo Android deve disporre di directory separate e isolate sulla memoria esterna (*Android 4.4 Compatibility Definition*, http://bit.ly/1GwI0CI).

Purtroppo, l'implementazione di questo requisito è problematica, perché la memoria esterna è per tradizione leggibile da tutti e implementata con il file system FAT, che non supporta i permessi. L'implementazione di Google della memoria esterna multiutente sfrutta tre funzioni del kernel Linux per offrire una memoria esterna per utente compatibile con le versioni precedenti: i namespace di mount, i mount bind e i subtree condivisi.

Funzionalità di mounting Linux avanzate

Come gli altri sistemi Unix, Linux gestisce tutti i file dai dispositivi di memoria come se fossero parte di una singola struttura (o tree) di directory. Ogni file system viene collegato a un subtree specifico con il mounting in una directory specificata, detta *punto di mount*. Per tradizione, la struttura di directory è condivisa da tutti i processi e ogni processo vede la medesima gerarchia di directory.

Linux 2.4.19 e versioni successive hanno aggiunto il supporto per i namespace di mount per processo, che consente a ogni processo di avere un proprio set di punti di mount e di conseguenza di usare una gerarchia di directory diversa da quella degli altri processi (Michael Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook,* No Starch Press, 2010, p. 261). L'elenco corrente dei mount per ogni processo può essere letto dal file virtuale */proc/PID/mounts*, dove *PID* è il process ID. Un processo Linux sottoposto a fork può richiedere un namespace di mount separato specificando il flag `CLONE_NEWNS` per le chiamate di sistema Linux `clone()` e `unshare()` (*ibid*., pp. 598 e 603). In questo caso, il namespace del processo padre è definito *namespace padre*.

Un *mount bind* consente di montare una directory o un file in un altro percorso della struttura di directory, affinché lo stesso file o directory sia visibile da più posizioni. Un mount bind viene creato specificando il flag `MS_BIND` per la chiamata di sistema `mount()` o passando il parametro `--bind` al comando `mount`.

Infine, i *subtree condivisi*, introdotti per la prima volta in Linux 2.6.15, consentono di controllare la propagazione dei mount del file system nei namespace di mount (*Shared Subtrees*, http://bit.ly/1nq203x). I subtree condivisi permettono a un processo di avere il proprio namespace e di accedere comunque ai file system montati dopo il suo avvio. I subtree condivisi mettono a disposizione quattro diversi tipi di mount; Android usa i mount condiviso e slave. Un *mount condiviso* creato in un namespace padre si propaga a tutti i namespace figlio ed è quindi visibile a tutti i processi clonati da un namespace. Un *mount slave* dispone di un mount master condiviso e si propaga ai nuovi mount. La propagazione è

tuttavia unidirezionale: i mount del master si propagano agli slave, ma i mount degli slave non si propagano al master. Questo schema consente a un processo di mantenere i suoi mount invisibili agli altri processi, pur continuando a vedere i mount di sistema condivisi. I mount condivisi vengono creati passando il flag `MS_SHARED` alla chiamata di sistema `mount()`, mentre la creazione di mount slave richiede il passaggio del flag `MS_SLAVE`.

Implementazione in Android

A partire da Android 4.4, il mounting diretto della memoria esterna non è più supportato, ma viene emulato con il daemon `sdcard` di FUSE, anche quando il dispositivo sottostante è una scheda SD fisica. La nostra descrizione sarà basata su una configurazione tipica per i dispositivi privi di scheda SD fisica, ovvero con una directory nella memoria interna. La documentazione ufficiale contiene maggiori dettagli su altre possibili configurazioni (consultate *External Storage > Typical Configuration Examples*, http://bit.ly/1w8DK6t).

Su un dispositivo in cui la memoria esterna primaria è sostenuta dalla memoria interna, il daemon FUSE `sdcard` usa la directory */data/media/* come origine e crea un file system emulato in */mnt/shell/emulated*. Il Listato 4.12 mostra la dichiarazione del servizio `sdcard` nel file specifico del dispositivo `init.rc` per il caso descritto **(7)**.

Listato 4.12 Dichiarazione del servizio sdcard per la memoria esterna emulata.

```
--altro codice--
on init
    mkdir /mnt/shell/emulated 0700 shell shell(1)
    mkdir /storage/emulated 0555 root root(2)

    export EXTERNAL_STORAGE /storage/emulated/legacy(3)
    export EMULATED_STORAGE_SOURCE /mnt/shell/emulated(4)
    export EMULATED_STORAGE_TARGET /storage/emulated(5)

    # Supporta i percorsi legacy
    symlink /storage/emulated/legacy /sdcard(6)
    symlink /storage/emulated/legacy /mnt/sdcard
    symlink /storage/emulated/legacy /storage/sdcard0
    symlink /mnt/shell/emulated/0 /storage/emulated/legacy
# daemon sdcard virtuale in esecuzione come media_rw (1023)
service sdcard /system/bin/sdcard -u 1023 -g 1023 -l /data/media /mnt/shell/emulated(7)
    class late_start
--altro codice--
```

Qui le opzioni `-u` e `-g` specificano l'utente e il gruppo con cui eseguire il daemon, mentre `-l` specifica il layout usato per la memoria emulata (come

spiegato in seguito nel paragrafo). Come potete osservare al punto **(1)**, la directory */mnt/shell/emulated/* (disponibile tramite la variabile di ambiente `EMULATED_STORAGE_SOURCE` **(4)**) è di proprietà dell'utente `shell`, che è l'unico a potervi accedere. Il contenuto potrebbe essere simile a quello del Listato 4.13 su un dispositivo con cinque utenti.

Listato 4.13 Contenuti di /mnt/shell/emulated/.

```
 # ls -l /mnt/shell/emulated/
drwxrwx--x root     sdcard_r          0
drwxrwx--x root     sdcard_r          10
drwxrwx--x root     sdcard_r          11
drwxrwx--x root     sdcard_r          12
drwxrwx--x root     sdcard_r          13
drwxrwx--x root     sdcard_r          legacy
drwxrwx--x root     sdcard_r          obb
```

Come per le directory dati delle app, ogni utente riceve una directory dati di memoria esterna dedicata che prende il nome dal suo user ID. Android usa una combinazione di namespace di mount e mount bind per far sì che la directory dati di memoria esterna di ogni utente sia disponibile solo per le applicazioni avviate dall'utente, senza mostrare loro le directory dati degli altri utenti. Visto che tutte le applicazioni vengono sottoposte a fork esternamente al processo *zygote* (rileggete il Capitolo 2), la configurazione della memoria esterna viene implementata in due fasi, la prima comune a tutti i processi e la seconda specifica per ogni processo. Per prima cosa, i punti di mount condivisi da tutti i processi delle app sottoposte a fork vengono configurati nel processo univoco *zygote*. Successivamente, i punti di mount dedicati, visibili solo al processo in questione, sono configurati come parte della specializzazione del processo per ogni app.

Vediamo per prima cosa la parte condivisa nel processo *zygote*. Nel Listato 4.14 è mostrato un frammento della funzione `initZygote()` (disponibile in `dalvik/vm/Init.cpp`) che evidenzia la configurazione dei punti di mount.

Listato 4.14 Configurazione del punto di mount in zygote.

```
static bool initZygote()
{
   setpgid(0,0);

   if (unshare(CLONE_NEWNS) == -1) {(1)
        return -1;
   }
```

```
    // Segnala rootfs come slave affinché le modifiche rispetto
    // al namespace predefinito siano applicate solo ai figli.
    if (mount("rootfs", "/", NULL, (MS_SLAVE | MS_REC), NULL) == -1) {(2)
        return -1;
    }

    const char* target_base = getenv("EMULATED_STORAGE_TARGET");
    if (target_base != NULL) {
        if (mount("tmpfs", target_base, "tmpfs", MS_NOSUID | MS_NODEV,(3)
                "uid=0,gid=1028,mode=0751") == -1) {
            return -1;
        }
    }
    --altro codice--
    return true;
}
```

Qui *zygote* passa il flag `CLONE_NEWNS` alla chiamata di sistema `unshare()` **(1)** per creare un nuovo namespace di mount privato che sarà condiviso da tutti i suoi figli (processi dell'app). Segnala quindi il file system root (montato in */*) come slave passando il flag `MS_SLAVE` alla chiamata di sistema `mount()` **(2)**. In questo modo le modifiche rispetto al namespace di mount predefinito, come il mounting di contenitori crittografati o memorie rimovibili, si propagheranno solo ai suoi figli; nello stesso tempo, ci si assicura che ogni mount creato dai figli non si propaghi al namespace predefinito. Per finire, *zygote* crea il punto di mount in memoria `EMULATED_STORAGE_TARGET` (solitamente */storage/emulated/*) creando un file system `tmpfs` **(3)**, che i figli usano per il mount bind della memoria esterna nei loro namespace privati.

Il Listato 4.15 mostra la configurazione del punto di mount specifico per il processo trovata in `dalvik/vm/native/dalvik_system_Zygote.cpp`, che viene eseguita durante il fork di ogni processo dell'app da *zygote*. La gestione degli errori, la registrazione e alcune dichiarazioni di variabili sono state omesse.

Listato 4.15 Configurazione della memoria esterna per i processi delle app.

```
static int mountEmulatedStorage(uid_t uid, u4 mountMode) {
    userid_t userid = multiuser_get_user_id(uid);(1)

    // Crea un secondo namespace di mount privato per il processo
    if (unshare(CLONE_NEWNS) == -1) {(2)
        return -1;
    }

    // Crea i mount bind per esporre la memoria esterna
    if (mountMode == MOUNT_EXTERNAL_MULTIUSER
            || mountMode == MOUNT_EXTERNAL_MULTIUSER_ALL) {
        // Questi percorsi devono essere già stati creati da init.rc
        const char* source = getenv("EMULATED_STORAGE_SOURCE");(3)
        const char* target = getenv("EMULATED_STORAGE_TARGET");(4)
        const char* legacy = getenv("EXTERNAL_STORAGE");(5)
        if (source == NULL || target == NULL || legacy == NULL) {
```

```
            return -1;
        }
        --altro codice--
        // /mnt/shell/emulated/0
        snprintf(source_user, PATH_MAX, "%s/%d", source, userid);(6)
        // /storage/emulated/0
        snprintf(target_user, PATH_MAX, "%s/%d", target, userid);(7)
        --altro codice--
        if (mountMode == MOUNT_EXTERNAL_MULTIUSER_ALL) {
            // Monta l'intera struttura della memoria esterna per tutti gli utenti
            if (mount(source, target, NULL, MS_BIND, NULL) == -1) {
                return -1;
            }
        } else {
            // Monta solo la memoria esterna specifica dell'utente
            if (mount(source_user, target_user, NULL, MS_BIND, NULL) == -1) {(8)
                return -1;
            }
        }
        --altro codice--
        // Infine, monta il percorso specifico dell'utente per gli utenti legacy
        if (mount(target_user, legacy, NULL, MS_BIND | MS_REC, NULL) == -1) {(9)
            return -1;
        }

    } else {
        return -1;
    }

    return 0;
}
```

La funzione `mountEmulatedStorage()` ottiene per prima cosa lo user ID corrente dall'UID del processo **(1)**, quindi usa la chiamata di sistema `unshare()` per creare un nuovo namespace di mount per il processo passando il flag `CLONE_NEWNS` **(2)**. La funzione ottiene quindi i valori delle variabili di ambiente `EMULATED_STORAGE_SOURCE` **(3)**, `EMULATED_STORAGE_TARGET` **(4)** ed `EXTERNAL_STORAGE` **(5)**, che vengono tutte inizializzate nel file `init.rc` specifico del dispositivo (fate riferimento a **(3)**, **(4)** e **(5)** nel Listato 4.12). A seguire prepara i percorsi delle directory di origine **(6)** e destinazione **(7)** del mounting in base ai valori di `EMULATED_STORAGE_SOURCE`, `EMULATED_STORAGE_TARGET` e allo user ID corrente.

Se non esistono, vengono create le directory, quindi il metodo `bind` monta la directory di origine (per esempio */mnt/shell/emulated/0* per l'utente proprietario) nel percorso di destinazione (per esempio */storage/emulated/0* per l'utente proprietario) **(8)**. In questo modo la memoria esterna è accessibile dalla shell di Android (avviata con il comando `adb shell`), molto utilizzata per lo sviluppo e il debug delle applicazioni.

L'ultimo passaggio è il mount bind ricorsivo della directory di destinazione nella directory legacy prefissata (*/storage/emulated/legacy/*) **(9)**. Questa è soggetta a symlinking con */sdcard/* nel file `init.rc` specifico del dispositivo (**(6)** nel Listato 4.12) per la compatibilità con le versioni precedenti delle app che utilizzano questo percorso direttamente nel codice (solitamente ottenuto con l'API `android.os.Environment.getExternalStorageDirectory()`).

Una volta eseguite tutte le operazioni, il nuovo processo creato per l'app può vedere solamente la memoria esterna assegnata all'utente che lo ha avviato. Per verificarlo possiamo osservare l'elenco dei mount per due processi dell'app eseguiti da utenti diversi mostrato nel Listato 4.16.

Listato 4.16 Elenco dei punti di mount per processi avviati da utenti diversi.

```
# cat /proc/7382/mounts
--altro codice--
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(1)
/dev/fuse /storage/emulated/0 fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(2)
/dev/fuse /storage/emulated/legacy fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(3)

# cat /proc/7538/mounts
--altro codice--
/dev/fuse /mnt/shell/emulated fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(4)
/dev/fuse /storage/emulated/10 fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(5)
/dev/fuse /storage/emulated/legacy fuse rw,nosuid,nodev,relatime,user_id=1023,
group_id=1023,default_permissions,allow_other 0 0(6)
```

Il processo avviato dall'utente proprietario con PID `7382` dispone di un punto di mount `/storage/emulated/0` **(2)**, che è un mount bind di */mnt/shell/emulated/0/*, mentre il processo `7538` (avviato da un utente secondario) presenta un punto di mount `/storage/emulated/10` **(5)**, che è un mount bind di */mnt/shell/emulated/10/*.

Visto che nessun processo dispone di un punto di mount per la directory di memoria esterna dell'altro processo, ogni processo può vedere e modificare soltanto i suoi file. Entrambi i processi hanno un punto di mount `/storage/emulated/legacy` (**(3)** e **(6)**), ma associato a directory diverse (rispettivamente */storage/emulated/0/* e */mnt/shell/emulated/10/*), pertanto ogni processo vedere contenuti diversi. Entrambi i processi possono vedere `/mnt/shell/emulated/` (**(1)** e **(4)**), ma visto che la directory è

accessibile unicamente all'utente `shell` (permessi 0700), i processi dell'app non possono vederne il contenuto.

Permessi della memoria esterna

Per emulare il file system FAT usato in origine per la memoria esterna, il daemon FUSE `sdcard` assegna un proprietario, un gruppo e alcuni permessi di accesso a ogni file o directory nella memoria esterna. Inoltre, i permessi non sono modificabili e i symlink e gli hardlink non sono supportati. Il proprietario e i permessi assegnati sono determinati dalla modalità di derivazione dei permessi usata dal daemon `sdcard`. Nella modalità legacy (specificata con l'opzione `-l`), compatibile con le versioni precedenti di Android e tuttora predefinita in Android 4.4, la maggior parte dei file e delle directory appartiene all'utente `root` e il loro gruppo è impostato su `sdcard_r`. Le applicazioni che ricevono il permesso `READ_EXTERNAL_STORAGE` dispongono di `sdcard_r` come uno dei loro gruppi supplementari, pertanto possono leggere la maggior parte dei file sulla memoria esterna anche se in origine sono stati creati da un'applicazione diversa. Il Listato 4.17 mostra il proprietario e i permessi dei file e delle directory nella radice (`root`) della memoria esterna.

Listato 4.17 Proprietario e permessi dei file nella memoria esterna.

```
# ls -l /sdcard/
drwxrwx--- root     sdcard_r          Alarms
drwxrwx--x root     sdcard_r          Android
drwxrwx--- root     sdcard_r          DCIM
--altro codice--
-rw-rw---- root     sdcard_r        5 text.txt
```

Nelle precedenti versioni di Android, tutti i file e le directory sulla memoria esterna ricevevano lo stesso proprietario e gli stessi permessi; Android 4.4 tratta invece in maniera diversa le directory di file esterne specifiche dell'applicazione (*Android/data/*<nome pacchetto>*/*; il percorso esatto è restituito dal metodo `Context.getExternalFilesDir()`). Le applicazioni non devono avere il permesso `WRITE_EXTERNAL_STORAGE` per leggere e scrivere file in questa directory, perché è di proprietà dell'applicazione che la crea.

Detto questo, anche in Android 4.4 la directory dei file esterna dell'applicazione è accessibile unicamente dalle applicazioni con i permessi `READ_EXTERNAL_STORAGE` o `WRITE_EXTERNAL_STORAGE` perché il gruppo della directory è impostato su `sdcard_r`, come mostrato nel Listato 4.18.

Listato 4.18 Proprietario e permessi della directory file esterna di un'app.

```
$ ls -l Android/data/
drwxrwx--- u10_a16  sdcard_r          com.android.browser
```

Android 4.4 supporta una modalità di derivazione dei permessi più flessibile basata sulla struttura delle directory, che può essere specificata passando l'opzione `-d` al daemon `sdcard`. Questa modalità di derivazione imposta gruppi dedicati alle directory *Pictures/* e *Music/* (`sdcard_pics` **(1)** e `sdcard_av` **(2)**, come mostrato nel Listato 4.19), che consentono un controllo preciso dei file a cui possono accedere le applicazioni. Quando è stato scritto questo libro Android non supportava questo controllo di accesso preciso, ma potete implementarlo facilmente definendo permessi aggiuntivi mappati ai gruppi `sdcard_pics` e `sdcard_av`. Nella modalità con permessi basati sulla struttura delle directory, le directory utente sono ospitate in *Android/user/* **(3)**.

NOTA

Anche se questa nuova modalità di deriva dei permessi è supportata in Android 4.4, al momento della stesura di questo libro i dispositivi Nexus usavano ancora la modalità legacy per i permessi.

Listato 4.19 Proprietari e permessi delle directory nella nuova modalità di derivazione dei permessi.

```
rwxrwx--x root:sdcard_rw     /
rwxrwx--- root:sdcard_pics   /Pictures(1)
rwxrwx--- root:sdcard_av     /Music(2)

rwxrwx--x root:sdcard_rw     /Android
rwxrwx--x root:sdcard_rw     /Android/data
rwxrwx--- u0_a12:sdcard_rw   /Android/data/com.example.app
rwxrwx--x root:sdcard_rw     /Android/obb/
rwxrwx--- u0_a12:sdcard_rw   /Android/obb/com.example.app

rwxrwx--- root:sdcard_all    /Android/user(3)
rwxrwx--x root:sdcard_rw     /Android/user/10
rwxrwx--- u10_a12:sdcard_rw  /Android/user/10/Android/data/com.example.app
```

Altre funzionalità multiutente

Oltre alle directory delle app dedicate, alla memoria esterna e alle impostazioni, esistono altre funzioni di Android che supportano una configurazione multiutente del dispositivo. Per esempio, nella versione 4.4, l'archivio delle credenziali di Android (che consente la gestione sicura delle chiavi di crittografia) permette a ogni utente di disporre del proprio archivio chiavi (l'archiviazione delle credenziali è descritta nei dettagli nel Capitolo 7).

Inoltre, il database degli account online di Android, accessibile dall'API `AccountManager`, è stato esteso per consentire agli utenti secondari di disporre dei propri account, nonché per permettere ai profili con restrizioni di condividere alcuni account dell'utente primario (se l'app che necessita dell'accesso all'account supporta questa funzione). Il supporto degli account online e l'API `AccountManager` sono presentati nel Capitolo 8. Infine, Android consente di impostare policy di amministrazione del dispositivo diverse per ogni utente. Nella versione 4.4 supporta inoltre la configurazione di VPN per utente, che instradano solamente il traffico di un singolo utente e non sono accessibili agli altri utenti. L'amministrazione del dispositivo, le VPN e altre funzionalità per aziende sono presentate nel Capitolo 9.

Riepilogo

Android permette a più utenti di condividere un dispositivo mettendo a disposizione una memoria interna ed esterna dedicata per ogni utente. Il supporto multiutente segue il modello di sicurezza stabilito; alle applicazioni di ogni utente viene assegnato uno UID univoco e le stesse vengono eseguite in processi dedicati che non possono accedere ai dati degli altri utenti. L'isolamento degli utenti è ottenuto combinando uno schema di assegnazione degli UID, che tiene conto dello user ID, con le regole di mounting della memoria, che consentono a ogni utente di vedere solo il proprio spazio di memoria.

Quando è stato scritto questo libro, il supporto multiutente era disponibile solo sui dispositivi privi del supporto per la telefonia (generalmente i tablet), in quanto il comportamento della telefonia in un ambiente multiutente non era definito. La maggior parte delle funzioni di Android, come la gestione del database degli account, l'archiviazione delle credenziali, le policy del dispositivo e il supporto VPN, è utilizzabile nella modalità multiutente e permette a ogni utente di utilizzare una propria configurazione.

Provider di crittografia

In questo capitolo viene presentata l'architettura dei provider di crittografia di Android e vengono descritti i provider predefiniti e gli algoritmi da loro supportati. Android è basato su *Java Cryptography Architecture* (JCA), di cui illustreremo in breve la struttura partendo dal framework CSP (*Cryptographic Service Provider*). A seguire parleremo delle classi e delle interfacce JCA principali, nonché delle primitive di crittografia che implementano (ogni primitiva di crittografia sarà descritta solo brevemente, perché una discussione approfondita va oltre l'ambito di questo libro e richiede una certa dimestichezza con la crittografia di base). A seguire presenteremo i provider JCA di Android e le librerie di crittografia, nonché gli algoritmi supportati da ogni provider. Infine, vedremo come utilizzare algoritmi di crittografia supplementari installando un provider JCA personalizzato.

Architettura dei provider JCA

JCA offre un framework estensibile per i provider di crittografia, nonché un set di API per le più importanti primitive di crittografia in uso oggi (cifratura a blocchi, digest del messaggio, firme digitali e così via). Questa architettura mira a essere indipendente dall'implementazione ed estensibile. Le applicazioni che usano le API JCA standard devono solamente indicare l'algoritmo di crittografia da utilizzare, nella maggior parte dei casi senza dipendere da una specifica implementazione del provider. Il supporto per i nuovi algoritmi di crittografia può essere aggiunto registrando un provider aggiuntivo che implementi gli algoritmi richiesti. Inoltre, i servizi di crittografia offerti da provider diversi sono generalmente interoperabili (con alcune restrizioni qualora le chiavi siano protette dall'hardware o se il materiale delle chiavi non è per altre ragioni disponibile direttamente), e le applicazioni possono combinare e abbinare liberamente i servizi di diversi provider in base alle necessità. Scendiamo ora nei dettagli dell'architettura di JCA.

Provider del servizio di crittografia

JCA suddivide la funzionalità di crittografia in numerosi servizi di crittografia astratti chiamati *engine* e definisce le API per ogni servizio sotto forma di *classe engine*. Per esempio, le firme digitali sono rappresentate dalla classe engine `Signature`, mentre la crittografia viene modellata con la classe `Cipher`. Un elenco completo di classi engine è disponibile nel prossimo paragrafo.

Nel contesto di JCA, un *provider del servizio di crittografia* (detto anche CSP o semplicemente *provider*) è un package (o un set di package) che fornisce un'implementazione concreta di alcuni servizi di crittografia. Ogni provider annuncia i servizi e gli algoritmi che implementa, permettendo al framework JCA di mantenere un registro degli algoritmi supportati e dei relativi provider. Questo registro mantiene un ordine di preferenza per i provider; di conseguenza, se un particolare algoritmo è

offerto da più provider, all'applicazione richiedente viene restituito quello con l'ordine di preferenza più alto. Un'eccezione alla regola viene fatta per le classi engine che supportano la *selezione ritardata del provider* (`Cipher`, `KeyAgreement`, `Mac` e `Signature`). Con la selezione ritardata del provider, il provider non viene scelto durante la creazione della classe engine, ma durante l'inizializzazione della classe stessa per una specifica operazione di crittografia. L'inizializzazione richiede un'istanza di `Key`, che il sistema usa per trovare un provider che accetti l'oggetto `Key` specificato. La selezione ritardata del provider è utile quando si usano chiavi archiviate nell'hardware, perché il sistema non può trovare il provider hardware solo in base al nome dell'algoritmo. Ad ogni modo, le istanze concrete di `Key` passate ai metodi di inizializzazione di solito contengono informazioni sufficienti per determinare il provider sottostante.

NOTA

Le versioni attuali di Android non supportano la selezione ritardata del provider, ma sono in corso interventi correlati nella diramazione master, pertanto è probabile che la funzione sarà supportata in una versione futura.

Vediamo un esempio che usa la configurazione dei provider mostrata nella Figura 5.1.

Se un'applicazione richiede un'implementazione dell'algoritmo digest SHA-256 senza specificare un provider (come mostrato nel Listato 5.1), il framework dei provider restituisce l'implementazione trovata in *ProviderB* (numero 2 nell'elenco della Figura 5.1), non quella di *ProviderC*, che supporta anch'esso SHA-256 ma che ha il numero 3 nell'elenco della Figura 5.1.

Listato 5.1 Richiesta di un'implementazione SHA-256 senza specificare un provider.

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
```

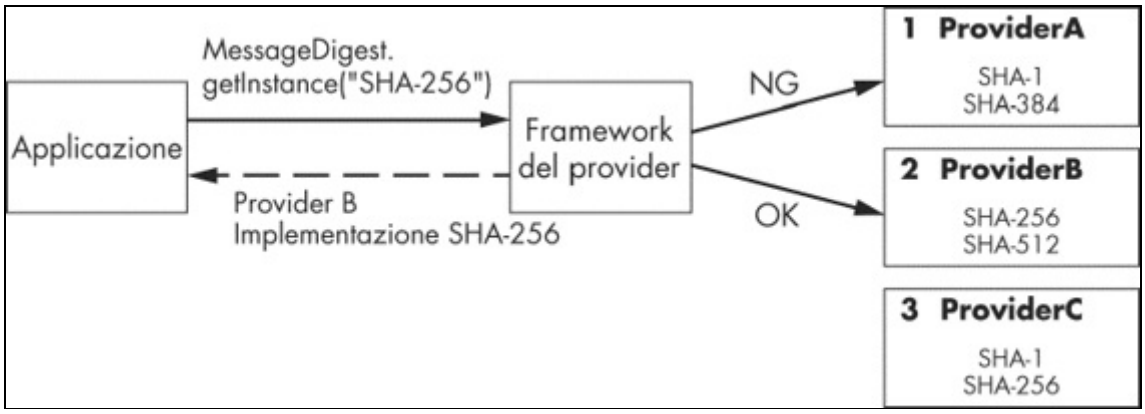

Figura 5.1 Scelta dell'implementazione dell'algoritmo JCA quando non è specificato il provider.

D'altro canto, se l'applicazione richiede esplicitamente *ProviderC* (come mostrato nel Listato 5.2), sarà restituita l'implementazione relativa anche se *ProviderB* ha un ordine di preferenza superiore.

Listato 5.2 Richiesta di un'implementazione SHA-256 da un provider specifico.

```
MessageDigest md = MessageDigest.getInstance("SHA-256", "ProviderC");
```

In generale, le applicazioni non dovrebbero richiedere esplicitamente un provider, a meno che non includano il provider richiesto come parte di esse o siano in grado di gestire il fallback se il provider preferito non è disponibile.

Implementazione dei provider

Il framework JCA garantisce l'indipendenza dall'implementazione richiedendo che tutte le implementazioni di un particolare algoritmo o servizio di crittografia siano conformi a un'interfaccia comune. Per ogni classe engine che rappresenta un particolare servizio di crittografia, il framework definisce una classe astratta *Service Provider Interface* (SPI) corrispondente. I provider che offrono un particolare servizio di crittografia implementano e annunciano la classe SPI. Per esempio, un provider che implementa un dato algoritmo di crittografia deve avere un'implementazione della classe `CipherSpi` corrispondente alla classe engine `Cipher`. Quando un'applicazione chiama il metodo factory `Cipher.getInstance()`, il framework JCA individua il provider appropriato usando il processo descritto nel paragrafo "Provider del servizio di crittografia" e restituisce

un'istanza di `Cipher` che instrada tutte le chiamate ai suoi metodi alla sottoclasse `CipherSpi` implementata nel provider selezionato.

Oltre alle classi di implementazione SPI, ogni provider dispone di una sottoclasse della classe astratta `java.security.Provider` che definisce il nome e la versione del provider e soprattutto un elenco degli algoritmi supportati e delle classi di implementazione SPI corrispondenti. Il framework del provider JCA usa questa classe `Provider` per creare il registro dei provider, che viene interrogato durante la ricerca delle implementazioni dell'algoritmo da restituire ai client.

Registrazione statica dei provider

Affinché un provider sia visibile al framework JCA è necessario registrarlo utilizzando un metodo statico o dinamico. La registrazione statica richiede la modifica del file delle proprietà di sicurezza del sistema con l'aggiunta di una voce per il provider. In Android questo file delle proprietà è chiamato `security.properties` ed è presente solo all'interno della libreria di sistema `core.jar`; non può quindi essere modificato e pertanto la registrazione statica dei provider non è supportata. Ce ne occupiamo comunque per ragioni di completezza.

La voce di un provider nel file delle proprietà di sicurezza ha il formato mostrato nel Listato 5.3.

Listato 5.3 Registrazione statica di un provider JCA.

```
security.provider.n=ProviderClassName
```

Qui `n` è l'ordine di preferenza del provider utilizzato durante la ricerca degli algoritmi richiesti (quando non viene specificato il nome del provider). L'ordine è a base 1, quindi 1 è il provider preferito, seguito da 2 e così via. `ProviderClassName` è il nome dell'implementazione della classe `java.security.Provider` descritta nel paragrafo "Implementazione dei provider".

Registrazione dinamica dei provider

I provider vengono registrati dinamicamente (ovvero in fase di esecuzione) con i metodi `addProvider()` e `insertProviderAt()` della classe

`java.security.Security`. Questi metodi restituiscono la posizione effettiva in cui è stato aggiunto il provider, o –1 se il provider non è stato aggiunto perché era già installato. I provider possono anche essere rimossi dinamicamente chiamando il metodo `removeProvider()`.

La classe `Security` gestisce l'elenco dei `Provider` di sicurezza e agisce come registro dei provider (come spiegato nei paragrafi precedenti). In Java SE i programmi richiedono permessi speciali per registrare i provider e modificare il registro, in quanto inserendo un nuovo provider in cima all'elenco è possibile sostituire l'implementazione della sicurezza di sistema. In Android le modifiche al registro sono limitate al processo dell'app corrente e non possono influire sul sistema o sulle altre applicazioni. Non sono quindi richiesti permessi speciali per registrare un provider JCA.

Le modifiche dinamiche al registro dei provider vengono generalmente inserite in un blocco statico per garantire che siano eseguite prima del codice dell'applicazione. Nel Listato 5.4 è mostrato un esempio di sostituzione del provider predefinito (priorità più alta) con uno personalizzato.

Listato 5.4 Inserimento dinamico di un provider JCA personalizzato.

```
static {
    Security.insertProviderAt(new MyProvider(), 1);
}
```

NOTA

Se la classe viene caricata più volte (per esempio da diversi class loader), il blocco statico potrebbe essere eseguito più volte. Per risolvere questo problema potete verificare se il provider è già disponibile oppure utilizzare una classe holder, che viene caricata una volta sola.

Classi engine JCA

Una classe engine fornisce l'interfaccia per un tipo specifico di servizio di crittografia. Gli engine JCA offrono uno dei seguenti servizi.

- Operazioni di crittografia (codifica/decodifica, firma/verifica, hash e così via).
- Generazione o conversione di materiale di crittografia (chiavi e parametri degli algoritmi).
- Gestione e conservazione degli oggetti di crittografia, quali chiavi e certificati digitali.

Recupero dell'istanza di una classe engine

Oltre a fornire un'interfaccia unificata per le operazioni di crittografia, le classi engine separano il codice client dall'implementazione sottostante; per questo motivo non è possibile istanziarle direttamente, ma occorre utilizzare il metodo factory statico `getInstance()` per richiedere indirettamente un'implementazione. Il metodo `getInstance()` di solito presenta una delle firme mostrate nel Listato 5.5.

Listato 5.5 Firme del metodo factory della classe engine JCA.

```
static EngineClassName getInstance(String algorithm)(1)
    throws NoSuchAlgorithmException
static EngineClassName getInstance(String algorithm, String provider)(2)
    throws NoSuchAlgorithmException, NoSuchProviderException
static EngineClassName getInstance(String algorithm, Provider provider)(3)
    throws NoSuchAlgorithmException
```

Di solito è sufficiente utilizzare la firma al punto **(1)** e specificare soltanto il nome dell'algoritmo. Le firme in **(2)** e **(3)** consentono di richiedere un'implementazione da un provider specifico. Tutte le varianti generano una `NoSuchAlgorithmException` se non è disponibile un'implementazione dell'algoritmo richiesto, mentre **(2)** genera `NoSuchProviderException` se non è registrato un provider con il nome specificato.

Nomi degli algoritmi

Il parametro stringa `algorithm` accettato da tutti i metodi factory crea un mapping con una trasformazione o un algoritmo di crittografia particolare, oppure specifica una strategia di implementazione per oggetti di livello più alto che gestiscono le raccolte di certificati o chiavi. Solitamente il mapping è diretto: per esempio, SHA-256 viene mappato a un'implementazione dell'algoritmo di hashing SHA-256, mentre *AES* richiede un'implementazione dell'algoritmo di crittografia AES. Alcuni nomi di algoritmi sono strutturati e specificano più parametri dell'implementazione richiesta. Per esempio, *SHA256withRSA* specifica un'implementazione della firma che usa SHA-256 per l'hashing del messaggio firmato e RSA per eseguire l'operazione di firma. Gli algoritmi possono avere anche degli alias, e più nomi di algoritmo possono essere mappati alla stessa implementazione.

I nomi degli algoritmi non fanno distinzione tra maiuscole e minuscole. I nomi degli algoritmi standard supportati da ogni classe engine JCA sono definiti nella *JCA Standard Algorithm Name Documentation*, a volte chiamata semplicemente *Standard Names* (http://bit.ly/1F25gHU). I provider possono inoltre definire i propri nomi di algoritmi e alias (fate riferimento alla documentazione di ogni provider per i dettagli). Potete usare il codice del Listato 5.6 per elencare tutti i provider, i nomi degli algoritmi dei servizi di crittografia offerti da ciascun provider e le classi di implementazione a cui sono mappati.

Listato 5.6 Recupero dell'elenco di tutti i provider JCA e degli algoritmi supportati.

```
Provider[] providers = Security.getProviders();
for (Provider p : providers) {
     System.out.printf("%s/%s/%f\n", p.getName(), p.getInfo(), p.getVersion());
     Set<Service> services = p.getServices();
     for (Service s : services) {
         System.out.printf("\t%s/%s/%s\n", s.getType(),
              s.getAlgorithm(), s.getClassName());
    }
}
```

Nei paragrafi seguenti presenteremo le classi engine principali e mostreremo il formato che usano per il nome dell'algoritmo.

SecureRandom

La classe `SecureRandom` rappresenta un *generatore di numeri casuali* (RNG, *Random Number Generator*) di crittografia. Anche se non lo userete molto spesso, è utilizzato internamente dalla maggior parte dei provider di crittografia per generare le chiavi e altro materiale di crittografia. La tipica implementazione del software è di solito quella di uno *pseudo-generatore di numeri casuali crittograficamente sicuro* (CSPRNG, *Cryptographically Secure Pseudo Random Number Generator*), che produce una sequenza di numeri approssimata alle proprietà dei veri numeri casuali sulla base di un valore iniziale detto *seed*. La qualità dei numeri casuali prodotti da un generatore CSPRNG dipende in gran parte dal seed, pertanto questo viene scelto con cura, in genere in base all'output di un vero RNG.

In Android, le implementazioni dei generatori CSPRNG ottengono il seed leggendo i byte di seed del file del dispositivo */dev/urandom* Linux standard, che non è altro che un'interfaccia al CSPRNG del kernel. Il CSPRNG del kernel stesso può essere in uno stato piuttosto prevedibile subito dopo l'avvio, pertanto Android ne salva periodicamente lo stato (4096 byte in Android 4.4) nel file `/data/system/entropy.dat`. Il contenuto del file viene riscritto in */dev/urandom* all'avvio per ripristinare lo stato precedente del CSPRNG; l'operazione viene svolta dal servizio di sistema `EntropyMixer`.

A differenza della maggior parte delle classi engine, `SecureRandom` dispone di costruttori pubblici utilizzabili per creare un'istanza. Il metodo consigliato per ottenere un'istanza dal seed corretto in Android prevede l'uso del costruttore predefinito, privo di argomenti (**(1)** nel Listato 5.7). Se utilizzate il metodo factory `getInstance()` dovrete passare *SHA1PRNG* come nome di algoritmo, visto che è l'unico universalmente supportato per `SecureRandom`. SHA1PRNG non è esattamente uno standard di crittografia, pertanto le implementazioni di provider diversi possono comportarsi in maniera differente. Per far sì che `SecureRandom` generi byte casuali dovete passare un array di byte al suo metodo `nextBytes()` (**(2)** nel Listato 5.7). Verrà così generato un numero di byte corrispondente alla lunghezza dell'array (`16` nel Listato 5.7) e tali byte saranno salvati nell'array.

Listato 5.7 Uso di SecureRandom per generare byte casuali.

```
SecureRandom sr = new SecureRandom();(1)
byte[] output = new byte[16];
sr.nextBytes(output);(2)
```

Il seeding manuale di `SecureRandom` è sconsigliato, perché un seeding errato del generatore CSPRNG di sistema può causare la produzione di una sequenza di byte prevedibile, in grado di mettere a repentaglio le operazioni di livello più alto che richiedono un input casuale. Ad ogni modo, se per qualche motivo dovete effettuare il seeding manuale di `SecureRandom` (per esempio se l'implementazione di seeding del sistema predefinito è notoriamente difettosa), potete farlo usando il costruttore `SecureRandom(byte[] seed)` o chiamando il metodo `setSeed()`. Durante il seeding manuale, assicuratevi che il seed usato sia sufficientemente casuale, per esempio leggendolo da */dev/urandom*.

Inoltre, a seconda dell'implementazione sottostante, la chiamata a `setSeed()` potrebbe non sostituire lo stato CSPRNG interno, ma solo aggiungersi a esso; in questo caso, le due istanze `SecureRandom` ottenute con lo stesso valore di seed potrebbero non produrre la stessa sequenza numerica. Di conseguenza, è preferibile non usare `SecureRandom` quando sono richiesti valori deterministici; in questi casi va usata una primitiva di crittografia studiata per produrre un output deterministico da un dato input, per esempio un algoritmo hash o una funzione di derivazione delle chiavi.

MessageDigest

La classe `MessageDigest` rappresenta le funzionalità di un digest del messaggio di crittografia, detto anche *funzione hash*. Un digest del messaggio crittografico accetta una sequenza di byte di lunghezza arbitraria e genera una sequenza di byte di lunghezza fissa detta *digest* o *hash*. Una buona funzione hash garantisce che anche un minimo cambiamento nell'input provochi un output completamente diverso e che sia davvero difficile trovare due input diversi che producono lo stesso valore hash (*resistenza alla collisione*) o generare un input con un

determinato hash (*resistenza al pre-imaging*). Un'altra proprietà importante delle funzioni hash è la resistenza al secondo pre-imaging. Per resistere a questo tipo di attacchi, una funzione hash dovrebbe rendere difficile trovare un secondo input m_2 che produca lo stesso valore hash di un input m_1.

Il Listato 5.8 mostra come usare la classe `MessageDigest`.

Listato 5.8 Uso di MessageDigest per l'hashing dei dati.

```
MessageDigest md = MessageDigest.getInstance("SHA-256");(1)
byte[] data = getMessage();
byte[] digest = md.digest(data);(2)
```

Un'istanza di `MessageDigest` viene creata passando il nome dell'algoritmo hash al metodo factory `getInstance()` **(1)**. L'input può essere fornito in blocchi utilizzando uno dei metodi `update()` e chiamando poi uno dei metodi `digest()` per ottenere il valore hash calcolato. In alternativa, se la dimensione dei dati di input è fissa e relativamente breve, l'hash può essere creato in un solo passaggio utilizzando il metodo `digest(byte[] input)` **(2)**, come avviene nel Listato 5.8.

Signature

La classe `Signature` offre un'interfaccia comune per gli algoritmi di firma digitale basati sulla crittografia asimmetrica. Un algoritmo di firma digitale accetta un messaggio arbitrario e una chiave privata e genera una stringa di byte di lunghezza fissa detta *firma* (o *signature*). Le firme digitali di solito applicano un algoritmo digest al messaggio di input, codificano il valore hash calcolato e poi usano un'operazione della chiave privata per produrre la firma. Quest'ultima può quindi essere verificata utilizzando la chiave pubblica corrispondente, eseguendo l'operazione inversa, calcolando il valore hash del messaggio firmato e confrontandolo con quello codificato nella firma. L'esito positivo della verifica garantisce l'integrità del messaggio firmato e la sua autenticità (sempre che la chiave privata di firma sia rimasta realmente privata).

Le istanze di `Signature` vengono create con il metodo factory standard `getInstance()`. Il nome dell'algoritmo usato è generalmente nella forma

<digest>with<encryption>, dove *<digest>* è il nome dell'algoritmo hash usato da `MessageDigest` (per esempio SHA256), mentre *<encryption>* è un algoritmo di crittografia asimmetrica (come RSA o DSA). Per esempio, una `Signature` *SHA512withRSA* usa prima l'algoritmo hash SHA-512 per produrre un valore digest e quindi crittografa il digest codificato con una chiave privata RSA per produrre la firma. Per gli algoritmi di firma che usano una funzione di generazione della maschera come RSA-PSS, il nome dell'algoritmo assume la forma *<digest>with<encryption>and<mgf>* (per esempio *SHA256withRSAandMGF1*).

Il Listato 5.9 mostra come usare la classe `Signature` per generare e verificare una firma di crittografia.

Listato 5.9 Generazione e verifica di una firma con la classe Signature.

```
PrivateKey privKey = getPrivateKey();
PublicKey pubKey = getPublicKey();
byte[] data = "sign me".getBytes("ASCII");

Signature sig = Signature.getInstance("SHA256withRSA");
sig.initSign(privKey);(1)
sig.update(data);(2)
byte[] signature = sig.sign();(3)

sig.initVerify(pubKey);(4)
sig.update(data);
boolean valid = sig.verify(signature);(5)
```

Dopo aver ottenuto un'istanza, l'oggetto `Signature` viene inizializzato sia per la firma, passando una chiave privata al metodo `initSign()` (**(1)** nel Listato 5.9), sia per la verifica, passando una chiave pubblica o un certificato al metodo `initVerify()` **(4)** per la verifica.

La procedura di firma è simile al calcolo di un hash con `MessageDigest`: i dati da firmare sono inviati in blocchi a uno dei metodi `update()` **(2)** o in gruppo al metodo `sign()` **(3)**, che restituisce il valore della firma. Per verificare una firma, i dati firmati vengono passati a uno dei metodi `update()`. Infine, la firma viene passata al metodo `verify()` **(5)**, che restituisce `true` se la firma è valida.

Cipher

La classe `Cipher` offre un'interfaccia comune per le operazioni di crittografia e decodifica. La crittografia è il processo che prevede l'uso di un algoritmo (chiamato *cifratura*) e di una chiave per trasformare i dati (detti *testo normale* o *messaggio di testo normale*) in una forma all'apparenza casuale (detta *testo cifrato*). L'operazione inversa, chiamata *decodifica*, trasforma il testo cifrato nel testo normale originale.

I due tipi di crittografia maggiormente diffusi oggi sono la *crittografia simmetrica* e la *crittografia asimmetrica*. La crittografia simmetrica, o *a chiave segreta*, usa la stessa chiave per crittografare e decodificare i dati. La crittografia asimmetrica usa una coppia di chiavi, una *chiave pubblica* e una *chiave privata*. I dati crittografati con una delle chiavi possono essere decodificati solo con l'altra chiave della coppia. La classe `Cipher` supporta la crittografia sia simmetrica sia asimmetrica.

In base all'input del processo, la cifratura può essere *a blocchi* o *a flussi*. La cifratura a blocchi utilizza appunto blocchi di dati di dimensione fissa. Se l'input non può essere diviso in un numero intero di blocchi, l'ultimo blocco viene *riempito mediante padding* aggiungendo il numero di byte necessario per raggiungere la dimensione del blocco. Sia l'operazione sia i byte aggiunti sono chiamati *padding*. Il padding viene rimosso nel processo di decodifica e non è incluso nel testo normale decodificato. Se è specificato un algoritmo di padding, la classe `Cipher` può aggiungere e rimuovere automaticamente il padding. La cifratura a flussi elabora invece i dati in input un byte (o addirittura un bit) alla volta e non necessita del padding.

Modalità di funzionamento della cifratura a blocchi

La cifratura a blocchi adotta diverse strategie per l'elaborazione dei blocchi di input al fine di generare il testo cifrato finale (o il testo normale in fase di decodifica). Queste strategie sono dette *modalità di funzionamento*, *modalità di cifratura* o semplicemente *modalità*. La strategia di elaborazione più semplice prevede di dividere il testo normale in blocchi (con il padding, se necessario), applicare la cifratura a ogni blocco e quindi concatenare i blocchi crittografati per produrre il testo

cifrato. Questa modalità è detta *Electronic Code Book* (ECB) e, per quanto semplice e facile da usare, ha lo svantaggio di generare blocchi di testo cifrato identici da blocchi di testo normale uguali. La struttura del testo normale è quindi rispecchiata nel testo cifrato e questo compromette la riservatezza del messaggio e facilita la crittoanalisi. Il problema è spesso illustrato ricorrendo al famigerato "ECB Penguin" citato alla voce di Wikipedia sulle modalità di cifratura a blocchi (`http://bit.ly/1CoRiM0`). La nostra versione Android è presentata nella Figura 5.2 (il robot Android è riprodotto o modificato dall'opera creata e condivisa da Google ed è usato alle condizioni descritte nella licenza di attribuzione Creative Commons 3.0). Qui **(1)** è l'immagine originale, **(2)** è l'immagine crittografata nella modalità ECB e **(3)** è la stessa immagine crittografata nella modalità CBC. Come potete vedere, il pattern dell'immagine originale è distinguibile in **(2)**, mentre in **(3)** appare come un disturbo casuale.

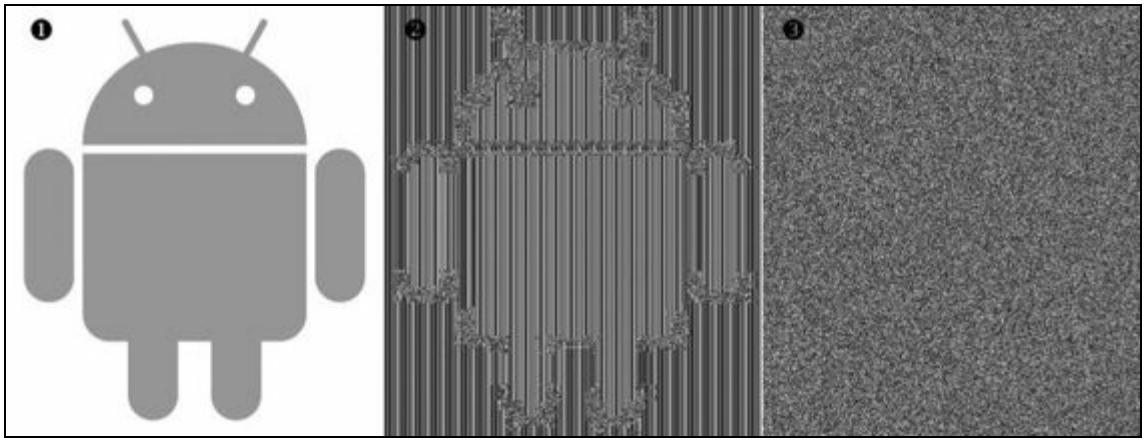

Figura 5.2 Pattern di testo cifrato prodotti da modalità di cifratura diverse.

Le *modalità di feedback* aggiungono la casualità al testo cifrato combinando il precedente blocco crittografato con il blocco di testo normale corrente prima della crittografia. Per produrre il primo blocco cifrato, combinano il primo blocco di testo normale con una stringa di byte, della stessa dimensione del blocco, non presente nel testo normale originale: questa stringa è detta *vettore di inizializzazione* (IV). Quando è configurata per usare una modalità di feedback, la classe `Cipher` può utilizzare un IV specificato dal client o generarne uno automaticamente.

Le modalità di feedback più usate sono chiamate *Cipher-Block Chaining* (CBC), *Cipher Feedback* (CFB) e *Output Feedback* (OFB).

Un altro modo per aggiungere la casualità al testo cifrato, adottato dalla modalità *Counter* (CTR), prevede di crittografare i valori successivi di una sequenza contatore per produrre una nuova chiave per ogni blocco di testo normale da crittografare. In questo modo la cifratura del blocco sottostante diventa una cifratura a flussi e non è richiesto alcun padding.

Le modalità di cifratura più nuove, come *Galois/Counter Mode* (GCM), oltre a diffondere pattern nel testo normale originale, autenticano il testo cifrato assicurando che non possa essere manomesso. Forniscono così la *crittografia autenticata* (AE, *Authenticated Encryption*) o la *crittografia autenticata con dati associati* (AEAD, *Authenticated Encryption with Associated Data*; D. McGrew, *An Interface and Algorithms for Authenticated Encryption*, http://www.ietf.org/rfc/rfc5116.txt). Le API `Cipher` sono state estese per supportare la crittografia autenticata in Java SE 7; queste estensioni sono disponibili a partire da Android 4.4, che dispone di un'API della libreria di runtime compatibile con Java 7. Le cifrature AE concatenano l'output del tag di autenticazione prodotto dall'operazione di crittografia con il testo cifrato prodotto dall'operazione per realizzare l'output finale. Nell'API `Cipher` di Java il tag è incluso (o verificato in fase di decodifica) implicitamente dopo la chiamata di `doFinal()`, pertanto non dovreste utilizzare l'output `update()` finché non siete certi della convalida del tag implicito.

Recupero di un'istanza di Cipher

Dopo aver visto i parametri principali di una cifratura possiamo finalmente parlare di come creare istanze di `Cipher`. Come per le altre classi engine, gli oggetti `Cipher` sono creati con il metodo factory `getInstance()`, che richiede non solo un nome di algoritmo, ma l'indicazione completa della *trasformazione* crittografica che la cifratura richiesta deve eseguire.

Il Listato 5.10 mostra come creare un'istanza di `Cipher` passando una stringa di trasformazione a `getInstance()`.

Listato 5.10 Creazione di un'istanza di Cipher.

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Una trasformazione deve specificare l'algoritmo di crittografia, la modalità di cifratura e il padding. La stringa di trasformazione passata a `getInstance()` è nel formato *algoritmo/modalità/padding*. Per esempio, la stringa di trasformazione usata nel Listato 5.10 consente di creare un'istanza di `Cipher` che utilizza AES come algoritmo di crittografia, CBC come modalità di cifratura e PKCS#5 come padding.

NOTA

Il termine PKCS compare più volte nella nostra descrizione delle classi engine e dei provider JCA. È l'acronimo di *Public Key Cryptography Standard* e fa riferimento a un gruppo di standard di crittografia sviluppati e pubblicati in origine da RSA Security, Inc. all'inizio degli anni Novanta. Molti di questi si sono evoluti in standard Internet pubblici e sono ora pubblicati e gestiti come RFC (*Request for Comments*, documenti formali che descrivono gli standard Internet), ma vi si fa ancora riferimento con il nome originale. Gli standard degni di nota comprendono PKCS#1, che definisce gli algoritmi di base per la crittografia RSA e le relative firme; PKCS#5, che definisce la crittografia basata su password; PKCS#7, che definisce la crittografia dei messaggi e la firma con un PKI ed è divenuto la base di S/MIME; PKCS#12, che definisce un contenitore per chiavi e certificati. Un elenco completo è disponibile sul sito web di EMC (`http://bit.ly/1zcOaH0`).

Un'istanza di `Cipher` può essere creata anche passando solo il nome dell'algoritmo, ma in questo caso l'implementazione restituita userebbe le impostazioni predefinite (specifiche per il provider) per la modalità di cifratura e il padding. Oltre a non essere una soluzione portabile tra i provider, può influire severamente sulla sicurezza del sistema, per esempio se in fase di runtime viene usata una modalità di cifratura meno sicura del previsto (come ECB). Questa "scorciatoia" è pertanto un difetto di progettazione del framework del provider JCA e non dovrebbe mai essere utilizzata.

Uso di un'istanza di Cipher

Dopo aver ottenuto un'istanza di `Cipher` è necessario inizializzarla prima di crittografare o decodificare i dati. `Cipher` viene inizializzata passando una costante integer che denota la modalità di funzionamento (`ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE` o `UNWRAP_MODE`), una chiave o un certificato e, facoltativamente, i parametri dell'algoritmo, a uno dei metodi `init()`

corrispondenti. `ENCRYPT_MODE` e `DECRYPT_MODE` sono usati per crittografare e decodificare dati arbitrari, mentre `WRAP_MODE` e `UNWRAP_MODE` sono modalità specializzate usate per crittografare (*wrapping*) e decodificare (*unwrapping*) il materiale della chiave di un oggetto `Key` usando un'altra chiave.

Il Listato 5.11 mostra come usare la classe `Cipher` per crittografare e decodificare i dati.

Listato 5.11 Uso della classe Cipher per crittografare e decodificare i dati.

```
SecureRandom sr = new SecureRandom();
SecretKey key = getSecretKey();
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");(1)

byte[] iv = new byte[cipher.getBlockSize()];
sr.nextBytes(iv);
IvParameterSpec ivParams = new IvParameterSpec(iv);(2)
cipher.init(Cipher.ENCRYPT_MODE, key, ivParams);(3)
byte[] plaintext = "encrypt me".getBytes("UTF-8");
ByteArrayOutputStream baos = new ByteArrayOutputStream();
byte[] output = cipher.update(plaintext);(4)
if (output != null) {
    baos.write(output);
}
output = cipher.doFinal();(5)
baos.write(output);
byte[] ciphertext = baos.toByteArray();

cipher.init(Cipher.DECRYPT_MODE, key, ivParams);(6)
baos = new ByteArrayOutputStream();
output = cipher.update(ciphertext);(7)
if (output != null) {
    baos.write(output);
}
output = cipher.doFinal();(8)
baos.write(output);
byte[] decryptedPlaintext = baos.toByteArray();(9)
```

In questo esempio creiamo un'istanza di `Cipher` che usa `AES` nella modalità `CBC` e il padding PKCS#5 **(1)**, generiamo un IV casuale e lo inseriamo in un oggetto `IvParameterSpec` **(2)**, quindi inizializziamo `Cipher` per la crittografia passando `ENCRYPT_MODE`, la chiave di crittografia e l'IV al metodo `init()` **(3)**. È quindi possibile crittografare i dati passando blocchi di dati al metodo `update()` **(4)**, che restituisce risultati immediati (o `null` se i dati di input sono troppo corti per generare un nuovo blocco), e ottenere l'ultimo blocco chiamando il metodo `doFinal()` **(5)**. Il testo cifrato finale viene ottenuto concatenando i risultati intermedi con il blocco finale.

Per la decodifica inizializziamo `cipher` in `DECRYPT_MODE` **(6)**, passando la stessa chiave e l'IV utilizzati per la crittografia. Chiamiamo poi `update()` **(7)**,

questa volta usando il testo cifrato come input, e infine chiamiamo `doFinal()` **(8)** per ottenere l'ultimo blocco del testo normale. Il testo normale finale viene ottenuto concatenando i risultati intermedi con il blocco finale **(9)**.

Mac

La classe `Mac` offre un'interfaccia comune per gli algoritmi *Message Authentication Code* (MAC, appunto). È usata per verificare l'integrità dei messaggi trasmessi su un canale non affidabile. Gli algoritmi MAC utilizzano una chiave segreta per calcolare un valore, il *MAC* (detto anche *tag*), che può essere usato per autenticare il messaggio e verificarne l'integrità. La stessa chiave è impiegata per eseguire la verifica, pertanto è necessario condividerla tra le parti comunicanti. Un MAC è spesso combinato con una cifratura per garantire sia la riservatezza sia l'integrità.

Listato 5.12 Uso della classe Mac per generare un codice di autenticazione del messaggio.

```
KeyGenerator keygen = KeyGenerator.getInstance("HmacSha256");
SecretKey key = keygen.generateKey();
Mac mac = Mac.getInstance("HmacSha256");(1)
mac.init(key);(2)
byte[] message = "MAC me".getBytes("UTF-8");
byte[] tag = mac.doFinal(message);(3)
```

Un'istanza di `Mac` si ottiene con il metodo factory `getInstance()` **(1)** (come mostrato nel Listato 5.12) richiedendo un'implementazione dell'algoritmo MAC HMAC che usi SHA-256 come funzione hash (consultate H. Krawczyk, M. Bellare e R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, http://tools.ietf.org/html/rfc2104). Viene quindi inizializzata **(2)** con un'istanza di `SecretKey`, che può essere generata con un `KeyGenerator` (leggete il paragrafo "KeyGenerator" più avanti nel capitolo), derivato da una password o istanziato direttamente dai byte della chiave non elaborata. Per le implementazioni di MAC basate su funzioni hash (come HMAC SHA-256 in questo esempio), il tipo di chiave non è importante, ma le implementazioni che usano una cifratura simmetrica possono richiedere il passaggio di un tipo di chiave corrispondente. Possiamo ora passare il messaggio in blocchi usando uno dei metodi `update()` e chiamare `doFinal()` per ottenere il valore MAC finale, oppure eseguire l'operazione in

un solo passaggio trasferendo i byte del messaggio direttamente a `doFinal()` **(3)**.

Key

L'interfaccia `Key` rappresenta le chiavi *opache* nel framework JCA. Le chiavi opache possono essere usate nelle operazioni di crittografia, ma di solito non consentono l'accesso al *materiale della chiave* sottostante (byte della chiave non elaborata). Questo ci permette di usare le stesse interfacce e classi JCA sia con le implementazioni software degli algoritmi di crittografia che salvano il materiale della chiave in memoria, sia con le implementazioni hardware in cui il materiale della chiave può risiedere in un token hardware (smart card, HSM – *Hardware Security Module* – e così via) e non è direttamente accessibile.

L'interfaccia `Key` definisce solo tre metodi.

- `String getAlgorithm()`: restituisce il nome dell'algoritmo di crittografia (simmetrico o asimmetrico) con cui può essere usata la chiave. Alcuni esempi sono AES e RSA.
- `byte[] getEncoded()`: restituisce una forma codificata standard della chiave utilizzabile durante la trasmissione della chiave ad altri sistemi. Può essere crittografata per le chiavi private. Per le implementazioni hardware che non consentono l'esportazione del materiale della chiave, questo metodo in genere restituisce `null`.
- `String getFormat()`: restituisce il formato della chiave codificata, solitamente RAW per le chiavi che non sono codificate in qualche particolare formato. Altri formati definiti in JCA sono X.509 e PKCS#8.

Potete ottenere un'istanza di `Key`:

- generando le chiavi con un `KeyGenerator` o un `KeyPairGenerator`;
- effettuando una conversione da una rappresentazione codificata con un `KeyFactory`;
- recuperando una chiave memorizzata da un `KeyStore`.

Nei prossimi paragrafi vedremo come creare e accedere ai diversi tipi di `Key`.

SecretKey e PBEKey

L'interfaccia `SecretKey` rappresenta le chiavi usate negli algoritmi simmetrici; è un'interfaccia marker e non aggiunge alcun metodo a quelli dell'interfaccia padre `Key`. Presenta una sola implementazione istanziabile direttamente, ovvero `SecretKeySpec`: si tratta sia di un'implementazione della chiave sia di una specifica della chiave (come descritto nel paragrafo "KeySpec" più avanti) e consente di istanziare `SecretKey` in base al materiale della chiave non elaborata.

La sottointerfaccia `PBEKey` rappresenta le chiavi derivate usando *Password Based Encryption* (PBE; fate riferimento a B. Kaliski, *PKCS #5: Password-Based Cryptography Specification, Version 2.0*, http://www.ietf.org/rfc/rfc2898.txt). PBE definisce algoritmi che traggono chiavi di crittografia forti da password e passphrase, che in genere hanno una bassa entropia e non possono essere usate direttamente come chiavi. PBE si basa su due idee fondamentali: l'uso di un *salt* per la protezione dagli attacchi con dizionario basati su tabelle (*salting*) e l'uso di un conteggio delle iterazioni elevato per rendere troppo onerosa la deduzione della chiave (*key stretching*). Il salt e il conteggio delle iterazioni sono usati come parametri dell'algoritmo PBE e devono pertanto essere conservati per generare la medesima chiave da una password specifica. Per questo le implementazioni di `PBEKey` sono tenute a implementare `getSalt()` e `getIterationCount()` insieme a `getPassword()`.

PublicKey, PrivateKey e KeyPair

Le chiavi pubbliche e private per gli algoritmi di crittografia asimmetrica sono modellate con le interfacce `PublicKey` e `PrivateKey`; si tratta di interfacce marker che non aggiungono alcun nuovo metodo. JCA definisce classi specializzate per gli algoritmi asimmetrici concreti che conservano i parametri delle chiavi corrispondenti, come `RSAPublicKey` e

`RSAPrivateCrtKey`. L'interfaccia `KeyPair` è semplicemente un contenitore per una chiave pubblica e una chiave privata.

KeySpec

Come spiegato nel precedente paragrafo "Key", l'interfaccia JCA `Key` rappresenta chiavi opache. `KeySpec`, invece, modella una *specifica della chiave*, vale a dire una rappresentazione *trasparente* della chiave che consente di accedere ai singoli parametri.

Nella pratica, la maggior parte delle interfacce `Key` e `KeySpec` per gli algoritmi concreti si sovrappone in maniera considerevole, perché per implementare gli algoritmi di crittografia è necessario che i parametri della chiave siano accessibili. Per esempio, sia `RSAPrivateKey` sia `RSAPrivateKeySpec` definiscono i metodi `getModulus()` e `getPrivateExponent()`. La differenza è importante solo quando l'algoritmo è implementato nell'hardware, caso in cui `KeySpec` contiene solamente un riferimento alla chiave gestita dall'hardware e non i parametri effettivi della chiave. L'elemento `Key` corrispondente contiene un handle per la chiave gestita dall'hardware e può essere usato per eseguire operazioni di crittografia, ma non contiene il materiale della chiave. Per esempio, una `RSAPrivateKey` archiviata in un dispositivo hardware restituisce `null` quando viene chiamato il suo metodo `getPrivateExponent()`.

Le implementazioni di `KeySpec` possono contenere una rappresentazione della chiave codificata e in questo caso sono indipendenti dall'algoritmo. Per esempio, `PKCS8EncodedKeySpec` può contenere una chiave RSA o una chiave DSA nel formato PKCS#8 con codifica DER (http://bit.ly/1nHGVys). D'altro canto, un elemento `KeySpec` specifico per un algoritmo contiene tutti i parametri della chiave sotto forma di campi. Per esempio, `RSAPrivateKeySpec` contiene il modulo e l'esponente privato di una chiave RSA, che possono essere recuperati utilizzando rispettivamente i metodi `getModulus()` e `getPrivateExponent()`. Indipendentemente dal tipo, gli elementi `KeySpec` vengono convertiti in oggetti `Key` utilizzando un `KeyFactory`.

KeyFactory

Un `KeyFactory` incapsula una routine di conversione necessaria per trasformare una rappresentazione trasparente di una chiave pubblica o privata (ovvero una sottoclasse di `KeySpec`) in un oggetto chiave opaco (alcune sottoclassi di `Key`, come `RSAPrivateKey`, espongono tutto il materiale della chiave e di conseguenza sono tecnicamente opache), ovvero in una sottoclasse di `Key`, utilizzabile per eseguire un'operazione di crittografia, o viceversa. Un `KeyFactory` che trasforma una chiave codificata in genere effettua il parsing dei dati della chiave codificata e memorizza ogni parametro della chiave nel campo corrispondente della classe concreta `Key`. Per esempio, per il parsing di una chiave pubblica RSA con codifica X.509, è possibile utilizzare il codice riportato di seguito (Listato 5.13).

Listato 5.13 Uso di KeyFactory per convertire una chiave codificata con X.509 in un oggetto RSAPublicKey.

```
KeyFactory kf = KeyFactory.getInstance("RSA");(1)
byte[] encodedKey = readRsaPublicKey();
X509EncodedKeySpec keySpec = new X509EncodedKeySpec(encodedKey);(2)
RSAPublicKey pubKey = (RSAPublicKey) kf.generatePublic(keySpec);(3)
```

Qui viene creato un `KeyFactory RSA` passando `RSA` a `KeyFactory.getInstance()` **(1)**. Viene poi letta la chiave RSA codificata e i byte della chiave codificata sono utilizzati per istanziare un `X509EncodedKeySpec` **(2)**; infine, si passa `KeySpec` al metodo factory `generatePublic()` **(3)** per ottenere un'istanza di `RSAPublicKey`.

Un `KeyFactory` può inoltre trasformare un `KeySpec` specifico per un algoritmo, per esempio `RSAPrivateKeySpec`, in un'istanza corrispondente di `Key` (in questo esempio `RSAPrivateKey`), ma in questo caso copia semplicemente i parametri della chiave (o l'handle della chiave) da una classe all'altra. La chiamata del metodo `KeyFactory.getKeySpec()` trasforma un oggetto `Key` in `KeySpec`, ma questa modalità d'uso non è molto comune perché la rappresentazione codificata della chiave si può ottenere semplicemente chiamando `getEncoded()` direttamente sull'oggetto chiave; inoltre, i `KeySpec` specifici degli algoritmi in genere non forniscono molte più informazioni di una classe concreta `Key`.

Un'altra funzione di `KeyFactory` è la trasformazione di un'istanza di `Key` da un provider diverso in un oggetto chiave corrispondente e compatibile

con il provider attuale. L'operazione è detta *key translation* (o conversione della chiave) e viene eseguita con il metodo `translateKey(Key key)`.

SecretKeyFactory

`SecretKeyFactory` è molto simile a `KeyFactory`, ma agisce unicamente sulle chiavi segrete (simmetriche). Potete usarlo per trasformare una specifica di chiave simmetrica in un oggetto `Key` e viceversa. In pratica, però, se avete accesso al materiale di una chiave simmetrica, è molto più facile usarlo per istanziare direttamente un `SecretKeySpec`, che è anche un oggetto `Key`: è per questo motivo che non viene usato molto spesso in questo modo.

Un caso di utilizzo molto più comune prevede la generazione di una chiave simmetrica da una password fornita dall'utente tramite PBE (Listato 5.14).

Listato 5.14 Generazione di una chiave segreta da una password con SecretKeyFactory.

```
byte[] salt = generateSalt();
int iterationCount = 1000;
int keyLength = 256;
KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt,
                                 iterationCount, keyLength);(1)
SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");(2)
SecretKey key = skf.generateSecret(keySpec);(3)
```

In questo caso `PBEKeySpec` viene inizializzato con la password, un salt generato in maniera casuale, il conteggio delle iterazioni e la lunghezza desiderata per la chiave **(1)**. Viene quindi ottenuto un factory `SecretKey` che implementa un algoritmo di derivazione della chiave PBE (in questo caso PBKDF2) con una chiamata a `getInstance()` **(2)**. Con il passaggio di `PBEKeySpec` a `generateSecret()` si esegue l'algoritmo di derivazione della chiave, che restituisce un'istanza di `SecretKey` **(3)** utilizzabile per la crittografia o la decodifica.

KeyPairGenerator

La classe `KeyPairGenerator` genera coppie di chiavi pubbliche e private. `KeyPairGenerator` viene istanziato passando il nome di un algoritmo asimmetrico al metodo factory `getInstance()` (**(1)** nel Listato 5.15).

Listato 5.15 Inizializzazione di KeyPairGenerator con i parametri specifici dell'algoritmo.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("ECDH");(1)
ECGenParameterSpec ecParamSpec = new ECGenParameterSpec("secp256r1");(2)
kpg.initialize(ecParamSpec);(3)
KeyPair keyPair = kpg.generateKeyPair();(4)
```

Esistono due tecniche per inizializzare un `KeyPairGenerator`: specificare la dimensione desiderata per la chiave oppure specificare i parametri dell'algoritmo. In entrambi i casi, è possibile passare anche un'istanza di `SecureRandom` da usare per la generazione della chiave. Se viene specificata solo la dimensione della chiave, per la generazione della chiave saranno usati i parametri predefiniti (se disponibili). Per specificare altri parametri è necessario istanziare e configurare un'istanza di `AlgorithmParameterSpec` appropriata per l'algoritmo asimmetrico in uso e passarla quindi al metodo `initialize()`, come mostrato nel Listato 5.15. In questo esempio, l'`ECGenParameterSpec` inizializzato in **(2)** è un `AlgorithmParameterSpec` che consente di specificare il nome della curva usata per generare le chiavi di crittografia *Elliptic Curve* (EC). Dopo il passaggio al metodo `initialize()` in **(3)**, la successiva chiamata di `generateKeyPair()` in **(4)** userà la curva specificata (`secp256r1`) per generare la coppia di chiavi.

NOTA

Anche se i vari standard definiscono curve denominate, la specifica Oracle JCA non definisce esplicitamente i nomi delle curve ellittiche. Poiché non esiste alcuno standard JCA ufficiale, i nomi delle curve supportati da Android variano in base alla versione della piattaforma.

KeyGenerator

`KeyGenerator` è molto simile alla classe `KeyPairGenerator`, ma genera chiavi simmetriche. Anche se la maggior parte delle chiavi simmetriche può essere generata richiedendo una sequenza di byte casuali a `SecureRandom`, le implementazioni di `KeyGenerator` effettuano controlli aggiuntivi sulle chiavi deboli e impostano i byte di parità della chiave ove appropriato (per DES e gli algoritmi derivati). Inoltre, poiché possono sfruttare l'hardware di crittografia disponibile, è sempre preferibile usare `KeyGenerator` anziché la generazione manuale delle chiavi.

Il Listato 5.16 mostra come generare una chiave AES usando `KeyGenerator`.

Listato 5.16 Generazione di una chiave AES con KeyGenerator.

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");(1)
kg.init(256);(2)
SecretKey key = keygen.generateKey();(3)
```

Per generare una chiave con `KeyGenerator`, create un'istanza **(1)**, specificate la dimensione desiderata per la chiave con `init()` **(2)** e infine chiamate `generateKey()` **(3)** per generare la chiave.

KeyAgreement

La classe `KeyAgreement` rappresenta un *protocollo di chiave concordata* che consente a due o più parti di generare una chiave condivisa senza scambiare informazioni segrete. Anche se esistono diversi protocolli di chiave concordata, quelli più diffusi oggi sono basati sullo *scambio di chiavi Diffie-Hellman* (DH), sia nella versione originale basata sulla crittografia a logaritmo discreto (nota come DH, http://bit.ly/1wawvKt), sia nella variante più recente basata sulla crittografia a chiave ellittica (ECDH, http://bit.ly/1wawvKt).

Entrambe le varianti del protocollo sono modellate in JCA utilizzando la classe `KeyAgreement` e possono essere applicate nello stesso modo; l'unica differenza riguarda le chiavi. Per entrambe le varianti, ogni entità facente parte della comunicazione deve avere una coppia di chiavi; entrambe le coppie devono essere generate dagli stessi parametri della chiave (modulo primo e generatore di base per DH e generalmente la stessa curva denominata ben definita per ECDH). Le parti devono quindi solamente scambiarsi le chiavi pubbliche ed eseguire l'algoritmo della chiave concordata per arrivare a un segreto comune.

Il Listato 5.17 mostra come usare la classe `KeyAgreement` per generare un segreto condiviso con ECDH.

Listato 5.17 Uso di KeyAgreement per generare un segreto condiviso.

```
PrivateKey myPrivKey = getPrivateKey();
PublicKey remotePubKey = getRemotePubKey();
KeyAgreement keyAgreement = KeyAgreement.getInstance("ECDH");(1)
keyAgreement.init(myPrivKey);(2)
```

```
keyAgreement.doPhase(remotePubKey, true);(3)
byte[] secret = keyAgreement.generateSecret();(4)
```

Un'istanza di `KeyAgreement` viene creata passando il nome dell'algoritmo, `ECDH`, al metodo factory `getInstance()` **(1)**; l'accordo viene quindi inizializzato passando la chiave privata al metodo `init()` **(2)**. A seguire, viene chiamato il metodo `doPhase()` $N - 1$ volte, dove N è il numero di entità che fanno parte della comunicazione; a ogni parte viene passata la chiave pubblica come primo parametro, mentre il secondo parametro viene impostato a `true` durante l'esecuzione dell'ultima fase dell'accordo **(3)** (per le due parti, come in questo esempio, il metodo `doPhase()` deve essere chiamato una volta sola). Infine, la chiamata al metodo `generateSecret()` **(4)** produce il segreto condiviso.

Il Listato 5.17 mostra il flusso delle chiamate per una sola delle parti (*A*); l'altra parte (*B*) deve eseguire la stessa sequenza con la propria chiave privata per inizializzare l'accordo, passando la chiave pubblica di *A* a `doPhase()`.

Anche se il valore (o parte del valore) restituito da `generateSecret()` può essere usato direttamente come chiave simmetrica, il metodo preferito consiste nell'utilizzare tale valore come input di una *funzione di derivazione della chiave* (KDF, *Key-Derivation Function*) e nell'utilizzare l'output di KDF come chiave. L'uso diretto del segreto condiviso generato può portare a una perdita di entropia, tale da limitare il numero di chiavi che possono essere prodotte con una singola operazione di chiave concordata DH. D'altro canto, l'uso di KDF diffonde qualunque struttura presente nel segreto (come il padding) e consente di generare più chiavi derivate con la combinazione in un salt.

`KeyAgreement` dispone di un altro metodo `generateSecret()` che accetta un nome di algoritmo come parametro e restituisce un'istanza di `SecretKey` utilizzabile per inizializzare direttamente `Cipher`. Se l'istanza di `KeyAgreement` è stata creata con una stringa di algoritmo contenente una specifica KDF (per esempio *ECDHwithSHA1KDF*), questo metodo applicherà KDF al segreto condiviso prima di restituire un elemento `SecretKey`. Se non è stato specificato un KDF, la maggior parte delle implementazioni tronca il

segreto condiviso per ottenere il materiale della chiave per l'elemento `SecretKey` restituito.

KeyStore

JCA usa il termine *keystore* per fare riferimento a un database di chiavi e certificati. Un keystore gestisce più oggetti di crittografia, definiti *voci* (o *entry*), ognuna delle quali è associata a un *alias* stringa. La classe `KeyStore` offre un'interfaccia ben definita a un keystore che definisce tre tipi di voci.

- `PrivateKeyEntry`: una chiave privata con una catena di certificati associata. Per un'implementazione software, il materiale della chiave privata è solitamente crittografato e protetto da una passphrase fornita dall'utente.
- `SecretKeyEntry`: una chiave segreta (simmetrica). Non tutte le implementazioni di `KeyStore` supportano la memorizzazione di chiavi segrete.
- `TrustedCertificateEntry`: un certificato a chiave pubblica di un'altra entità. Spesso contiene i certificati delle CA utilizzabili per stabilire relazioni di trust. Un keystore che contiene solamente `TrustedCertificateEntry` è detto *truststore*.

Tipi di KeyStore

Un'implementazione di `KeyStore` non deve necessariamente essere persistente, ma spesso lo è. Le varie implementazioni sono identificate da un *tipo di keystore* che definisce la memoria e il formato dati del keystore, nonché i metodi usati per proteggere le chiavi memorizzate. Il tipo di `KeyStore` predefinito è configurato con la proprietà di sistema `keystore.type`.

L'implementazione di `KeyStore` predefinita per la maggior parte di provider JCA prevede solitamente un tipo di keystore che salva i suoi dati in un file. Il formato di file può essere proprietario o basato su uno

standard pubblico. I formati proprietari comprendono il formato Java SE originale JKS e la sua versione con sicurezza avanzata JCEKS, nonché il formato *Bouncy Castle KeyStore* (BKS) predefinito in Android.

Keystore su file PKCS#12

Lo standard pubblico più diffuso che consente il bundling delle chiavi private e dei certificati associati in un file è *Personal Information Exchange Syntax Standard*, comunemente detto PKCS#12. È un successore dello standard *Personal Information Exchange Syntax* (PFX), pertanto i termini PKCS#12 e PFX sono spesso intercambiabili e i file PKCS#12 sono solitamente detti file PFX.

PKCS#12 è un formato di contenitore che può contenere più oggetti incorporati quali chiavi private, certificati e persino CRL. Analogamente ai precedenti standard PKCS su cui è basato PKCS#12, il contenuto del contenitore è definito in ASN.1 ed è essenzialmente una sequenza di strutture nidificate.

NOTA

Abstract Syntax Notation One, o ASN.1, è una notazione standard che descrive regole e strutture per la codifica dei dati nelle telecomunicazioni e nelle reti di computer; è ampiamente utilizzata negli standard di crittografia per definire la struttura degli oggetti crittografici.

Le strutture del contenitore interne sono dette `SafeBag`; vengono definite bag diverse per i certificati (`CertBag`), le chiavi private (`KeyBag`) e le chiavi private crittografate (`PKCS8ShroudedKeyBag`). L'integrità dell'intero file è protetta da un MAC che usa una chiave derivata da una *password di integrità*; la voce della chiave privata di ogni individuo è crittografata con una chiave derivata da una *password di privacy*. Nella pratica, le due password sono generalmente uguali. PKCS#12 può inoltre usare le chiavi pubbliche per proteggere la privacy e l'integrità dei contenuti dell'archivio, anche se questa modalità d'uso è poco comune.

Un tipico file PKCS#12 contenente la chiave della password crittografata di un utente e un certificato associato può assumere la struttura mostrata nella Figura 5.3 (dove alcune strutture wrapper sono state rimosse per favorire la chiarezza).

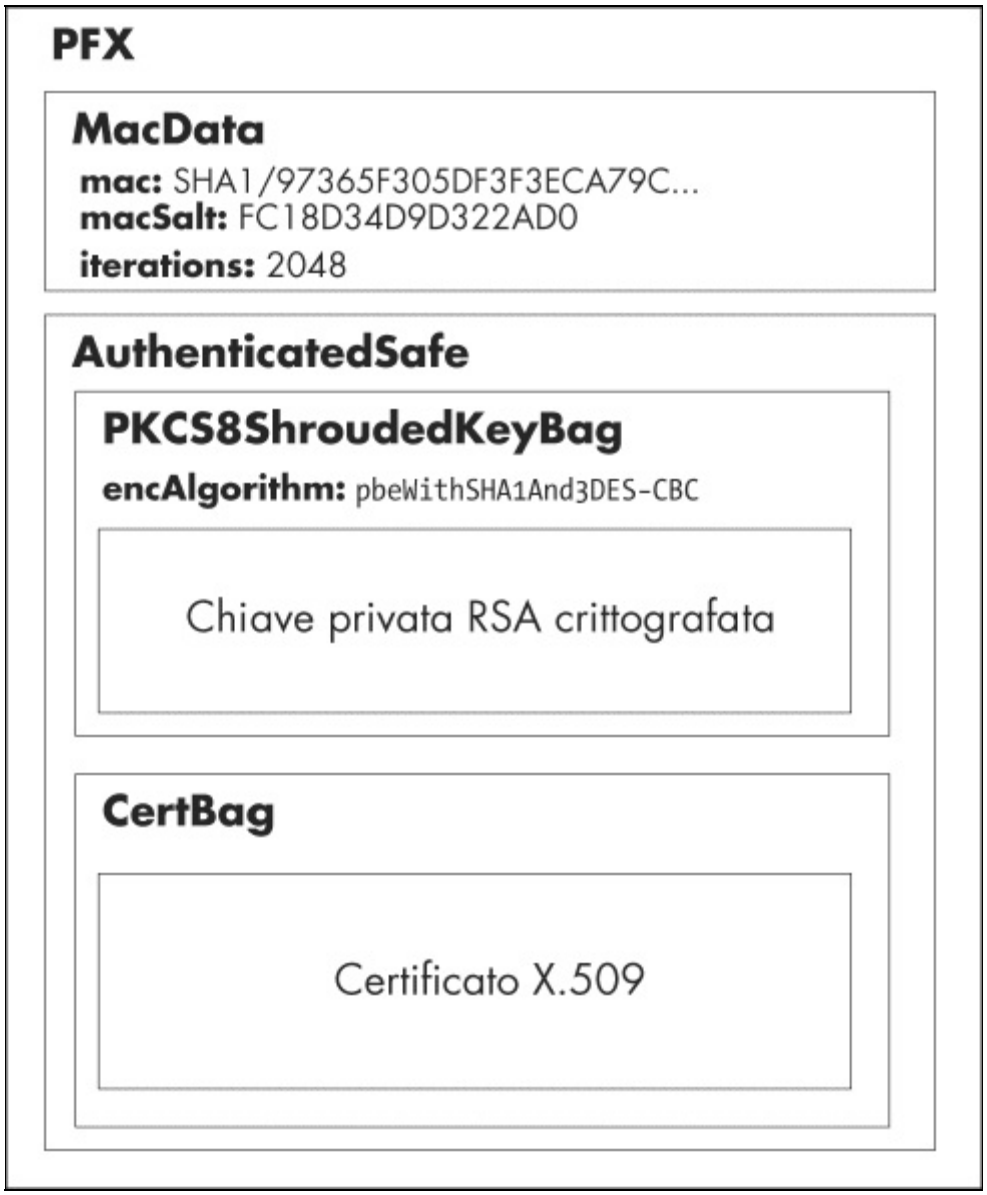

Figura 5.3 Struttura di un file PKCS#12 con una chiave privata e un certificato associato.

Listato 5.18 Uso della classe KeyStore per estrarre una chiave privata e un certificato da un file PKCS#12.

```
KeyStore keyStore = KeyStore.getInstance("PKCS12");(1)
InputStream in = new FileInputStream("mykey.pfx");
keyStore.load(in, "password".toCharArray());(2)
KeyStore.PrivateKeyEntry keyEntry =
      (KeyStore.PrivateKeyEntry)keyStore.getEntry("mykey", null);(3)
X509Certificate cert = (X509Certificate) keyEntry.getCertificate();(4)
RSAPrivateKey privKey = (RSAPrivateKey) keyEntry.getPrivateKey();(5)
```

La classe `KeyStore` può essere usata per accedere al contenuto di un file PKCS#12 specificando `PKCS12` come tipo di keystore durante la creazione di un'istanza (**(1)** nel Listato 5.18). Per caricare e sottoporre a parsing il file PKCS#12, chiamiamo il metodo `load()` **(2)** passando un `InputStream` da cui leggere il file e la password di integrità del file. Una volta caricato il file possiamo ottenere una voce della chiave privata chiamando il metodo

`getEntry()` e passando l'alias della chiave **(3)** e, facoltativamente, un'istanza di `KeyStore.PasswordProtection` inizializzata con la password per la voce richiesta, se è diversa dalla password di integrità del file. Se l'alias è sconosciuto, è possibile ottenere un elenco di tutti gli alias con il metodo `aliases()`. Una volta ottenuto `PrivateKeyEntry`, possiamo accedere al certificato della chiave pubblica **(4)** o alla chiave privata **(5)**. Le nuove voci possono essere aggiunte con il metodo `setEntry()` ed eliminate con il metodo `deleteEntry()`. Le modifiche al contenuto di `KeyStore` possono essere rese persistenti su disco chiamando il metodo `store()`, che accetta come parametri un `OutputStream` (in cui sono scritti i byte del keystore) e una password di integrità (usata per derivare MAC e le chiavi di crittografia).

Un'implementazione di `KeyStore` non deve usare un singolo file per salvare gli oggetti certificato e chiave; può usare più file, un database o qualunque altro meccanismo di archiviazione. In effetti, le chiavi possono anche non essere salvate sul sistema host, ma su un dispositivo hardware separato come una smart card o un *modulo di protezione hardware* (HSM, *Hardware Security Module*). Le implementazioni di `KeyStore` specifiche per Android, che forniscono un'interfaccia al trust store del sistema e all'archiviazione delle credenziali, sono presentate nei Capitoli 6 e 7.

CertificateFactory e CertPath

`CertificateFactory` agisce come un parser di certificati e CRL e può creare catene di certificati da un elenco di certificati. Può leggere un flusso contenente certificati o CRL codificati e generare in output un insieme (o una singola istanza) di oggetti `java.security.cert.Certificate` e `java.security.cert.CRL`. Solitamente è disponibile solo un'implementazione *X.*509 che esegue il parsing dei CRL e dei certificati X.509.

Il Listato 5.19 mostra come eseguire il parsing di un file di certificato con `CertificateFactory`.

Listato 5.19 Parsing di un file di certificato X.509 con CertificateFactory.

```
CertificateFactory cf = CertificateFactory.getInstance("X.509");(1)
InputStream in = new FileInputStream("certificate.cer");
```

```
X509Certificate cert = (X509Certificate) cf.generateCertificate(in);(2)
```

Per creare un elemento `CertificateFactory`, passiamo `X.509` come tipo di factory a `getInstance()` **(1)**, quindi chiamiamo `generateCertificate()` passando un `InputStream` per la lettura **(2)**. Visto che si tratta di un factory X.509, l'oggetto ottenuto può essere tranquillamente sottoposto a cast in `java.security.cert.X509Certificate`. Se il file letto include più certificati che formano una catena di certificati, è possibile ottenere un oggetto `CertPath` chiamando il metodo `generateCertPath()`.

CertPathValidator e CertPathBuilder

La classe `CertPathValidator` incapsula un algoritmo di convalida della catena di certificati definito dallo standard *Public-Key Infrastructure (X.509)* o *PKIX* (fate riferimento a D. Cooper *et al., Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, maggio 2008, http://tools.ietf.org/html/rfc5280). PKIX e la convalida della catena di certificati sono presentati nei dettagli nel Capitolo 6; il Listato 5.20 mostra solamente come usare `CertificateFactory` e `CertPathValidator` per creare e convalidare una catena di certificati.

Listato 5.20 Creazione e convalida di una catena di certificati con CertPathValidator.

```
CertPathValidator certPathValidator = CertPathValidator.getInstance("PKIX");(1)
CertificateFactory cf = CertificateFactory.getInstance("X.509");

X509Certificate[] chain = getCertChain();
CertPath certPath = cf.generateCertPath(Arrays.asList(chain));(2)
Set<TrustAnchor> trustAnchors = getTrustAnchors();
PKIXParameters result = new PKIXParameters(trustAnchors);(3)
PKIXCertPathValidatorResult result = (PKIXCertPathValidatorResult)
            certPathValidator.validate(certPath, pkixParams);(4)
```

Come potete osservare, recuperiamo per prima cosa un'istanza di `CertPathValidator` passando `PKIX` al metodo `getInstance()` **(1)**, poi creiamo una catena di certificati con il metodo `generateCertPath()`di `CertificateFactory` **(2)**. Se l'elenco di certificati passato non forma una catena valida, questo metodo genera una `CertificateException`. Se non sono già disponibili tutti i certificati necessari per formare una catena, possiamo usare un `CertPathBuilder` inizializzato con un `CertStore` per trovare i certificati necessari e generare un `CertPath` (non mostrato).

Una volta ottenuto un `CertPath`, inizializziamo la classe `PKIXParameters` con un set di *trust anchor* (di solito sono certificati CA attendibili; fate riferimento al Capitolo 6 per i dettagli) **(3)** e quindi chiamiamo `CertPathValidator.validate()` **(4)** passando il `CertPath` creato in **(2)** e l'istanza di `PKIXParameters`. Se la convalida ha esito positivo, `validate()` restituisce un'istanza di `PKIXCertPathValidatorResult`; in caso contrario, genera una `CertPathValidatorException` contenente informazioni dettagliate sulle cause dell'esito negativo.

Provider JCA di Android

I provider di crittografia di Android sono basati su JCA e seguono la sua architettura con alcune piccole eccezioni. Se i componenti Android di basso livello usano le librerie di crittografia native (come OpenSSL), i componenti del sistema e le applicazioni di terze parti usano l'API di crittografia principale, appunto JCA.

Android dispone di tre provider JCA fondamentali che includono le implementazioni delle classi engine viste nel paragrafo precedente e due provider *Java Secure Socket Extension* (JSSE) che implementano funzionalità SSL (JSSE è descritto nei dettagli nel Capitolo 6).

Vediamo ora i provider JCA fondamentali di Android.

Provider Crypto di Harmony

L'implementazione della libreria di runtime Java di Android deriva dal vecchio progetto Apache Harmony (`http://harmony.apache.org/`), che includeva anche un provider JCA limitato chiamato Crypto, ideato per fornire implementazioni dei servizi di crittografia di base quali la generazione di numeri casuali, l'hashing e le firme digitali. Crypto è ancora incluso in Android per la compatibilità con le versioni precedenti, ma presenta la priorità più bassa tra tutti i provider JCA; per questo, le implementazioni delle classi engine di Crypto non vengono restituite tranne in caso di richiesta esplicita. La Tabella 5.1 mostra le classi engine e gli algoritmi supportati da Crypto.

NOTA

Anche se gli algoritmi elencati nella Tabella 5.1 sono ancora disponibili in Android 4.4, tutti tranne SHA1PRNG sono stati rimossi dalla diramazione master di Android e potrebbero non essere disponibili nelle versioni future.

Tabella 5.1 Algoritmi supportati dal provider Crypto in Android 4.4.4.

Nome della classe engine	Algoritmi supportati
`KeyFactory`	DSA
`MessageDigest`	SHA-1
`SecureRandom`	SHA1PRNG
`Signature`	SHA1withDSA

Provider Bouncy Castle di Android

Prima di Android versione 4.0, l'unico provider JCA con funzionalità complete in Android era Bouncy Castle. Il provider è parte delle API Bouncy Castle Crypto, un set di implementazioni Java open source di protocolli e algoritmi di crittografia (`https://www.bouncycastle.org/java.html`).

Android include una versione modificata del provider Bouncy Castle, derivata dalla versione mainstream applicando un set di patch specifiche per Android. Queste patch sono mantenute nella struttura di origine di Android e vengono aggiornate a ogni nuova release del provider mainstream Bouncy Castle. Le differenze principali rispetto alla versione mainstream sono riepilogate di seguito.

- Gli algoritmi non sicuri come MD2 e RC2 sono stati rimossi.
- Le implementazioni basate su Java di MD5 e la famiglia SHA di algoritmi digest sono state sostituite con un'implementazione nativa.
- Alcuni algoritmi PBE sono stati rimossi (per esempio *PBEwithHmacSHA256*).
- Il supporto per l'accesso ai certificati archiviati in LDAP è stato rimosso.
- È stato aggiunto il supporto per le blacklist dei certificati (le blacklist sono affrontate nel Capitolo 6).
- Sono stati eseguiti vari interventi di ottimizzazione delle prestazioni.
- Il nome del package è cambiato in `com.android.org.bouncycastle` per evitare conflitti con le app in Bouncy Castle (a partire da Android 3.0).

NOTA

Gli algoritmi, le modalità e i parametri dell'algoritmo non supportati dall'implementazione di riferimento (RI, *Reference Implementation*) di Java sono stati rimossi (RIPEMD, SHA-224, GOST3411, Twofish, CMAC, El Gamal, RSA-PSS, ECMQV e così via).

Le classi engine e gli algoritmi supportati dal provider Bouncy Castle di Android a partire dalla versione 4.4.4 (basata su Bouncy Castle 1.49) sono elencati nella Tabella 5.2.

Tabella 5.2 Algoritmi supportati dal provider Bouncy Castle in Android 4.4.4.

Nome della classe engine	Algoritmi supportati
`CertPathBuilder`	PKIX
`CertPathValidator`	PKIX

CertStore	Collection
CertificateFactory	X.509
Cipher	AES AESWRAP ARC4 BLOWFISH DES DESEDE DESEDEWRAP PBEWITHMD5AND128BITAES-CBC-OPENSSL PBEWITHMD5AND192BITAES-CBC-OPENSSL PBEWITHMD5AND256BITAES-CBC-OPENSSL PBEWITHMD5ANDDES PBEWITHMD5ANDRC2 PBEWITHSHA1ANDDES PBEWITHSHA1ANDRC2 PBEWITHSHA256AND128BITAES-CBC-BC PBEWITHSHA256AND192BITAES-CBC-BC PBEWITHSHA256AND256BITAES-CBC-BC PBEWITHSHAAND128BITAES-CBC-BC PBEWITHSHAAND128BITRC2-CBC PBEWITHSHAAND128BITRC4 PBEWITHSHAAND192BITAES-CBC-BC PBEWITHSHAAND2-KEYTRIPLEDES-CBC PBEWITHSHAAND256BITAES-CBC-BC PBEWITHSHAAND3-KEYTRIPLEDES-CBC PBEWITHSHAAND40BITRC2-CBC PBEWITHSHAAND40BITRC4 PBEWITHSHAANDTWOFISH-CBC RSA
KeyAgreement	DH ECDH
KeyFactory	DH DSA EC RSA
KeyGenerator	AES ARC4 BLOWFISH DES DESEDE HMACMD5 HMACSHA1 HMACSHA256 HMACSHA384 HMACSHA512
KeyPairGenerator	DH DSA EC RSA
KeyStore	BKS (predefinito) BouncyCastle PKCS12
Mac	HMACMD5 HMACSHA1 HMACSHA256 HMACSHA384 HMACSHA512 PBEWITHHMACSHA PBEWITHHMACSHA1

MessageDigest	MD5 SHA-1 SHA-256 SHA-384 SHA-512
SecretKeyFactory	DES DESEDE PBEWITHHMACSHA1 PBEWITHMD5AND128BITAES-CBC-OPENSSL PBEWITHMD5AND192BITAES-CBC-OPENSSL PBEWITHMD5AND256BITAES-CBC-OPENSSL PBEWITHMD5ANDDES PBEWITHMD5ANDRC2 PBEWITHSHA1ANDDES PBEWITHSHA1ANDRC2 PBEWITHSHA256AND128BITAES-CBC-BC PBEWITHSHA256AND192BITAES-CBC-BC PBEWITHSHA256AND256BITAES-CBC-BC PBEWITHSHAAND128BITAES-CBC-BC PBEWITHSHAAND128BITRC2-CBC PBEWITHSHAAND128BITRC4 PBEWITHSHAAND192BITAES-CBC-BC PBEWITHSHAAND2-KEYTRIPLEDES-CBC PBEWITHSHAAND256BITAES-CBC-BC PBEWITHSHAAND3-KEYTRIPLEDES-CBC PBEWITHSHAAND40BITRC2-CBC PBEWITHSHAAND40BITRC4 PBEWITHSHAANDTWOFISH-CBC PBKDF2WithHmacSHA1 PBKDF2WithHmacSHA1And8BIT
Signature	ECDSA MD5WITHRSA NONEWITHDSA NONEwithECDSA SHA1WITHRSA SHA1withDSA SHA256WITHECDSA SHA256WITHRSA SHA384WITHECDSA SHA384WITHRSA SHA512WITHECDSA SHA512WITHRSA

Provider AndroidOpenSSL

Come affermato nel paragrafo “Provider Bouncy Castle di Android”, gli algoritmi hash nel provider Bouncy Castle di Android sono stati sostituiti con codice nativo per ragioni di prestazioni. Per migliorare ulteriormente le performance di crittografia, il numero di classi engine e algoritmi supportati nel provider nativo AndroidOpenSSL aumenta costantemente a ogni rilascio sin dalla versione 4.0. In origine, AndroidOpenSSL era usato solo per implementare i socket SSL, ma a partire da Android 4.4 copre la maggior parte delle funzionalità offerte da

Bouncy Castle. Dal momento che è il provider preferito (con la priorità più alta, ovvero 1), le classi engine che non richiedono esplicitamente Bouncy Castle ottengono un'implementazione del provider AndroidOpenSSL. Come suggerito dal nome, la sua funzionalità di crittografia è fornita dalla libreria OpenSSL. L'implementazione del provider usa JNI per collegare il codice nativo di OpenSSL alle classi Java SPI richieste per implementare un provider JCA. La parte più imponente dell'implementazione si trova nella classe Java `NativeCrypto`, chiamata dalla maggior parte delle classi SPI. AndroidOpenSSL è parte della libreria *libcore* di Android, che implementa la sezione core della libreria di runtime Java di Android. A partire da Android 4.4, AndroidOpenSSL è stato dissociato da *libcore* affinché possa essere compilato come libreria autonoma e incluso nelle applicazioni che desiderano un'implementazione della crittografia stabile e indipendente dalla versione della piattaforma. Il provider autonomo è chiamato Conscrypt e risiede nel package `org.conscrypt`, che prende il nome `com.android.org.conscrypt` quando è generato come parte della piattaforma Android.

Le classi engine e gli algoritmi supportati dal provider AndroidOpenSSL a partire dalla versione 4.4.4 sono elencati nella Tabella 5.3.

OpenSSL

OpenSSL è un toolkit di crittografia open source che implementa i protocolli SSL e TLS ed è ampiamente utilizzato come libreria di crittografia a scopo generico (https://www.bouncycastle.org/java.html). È incluso in Android come libreria di sistema ed è usato per implementare il provider JCA AndroidOpenSSL, presentato nel paragrafo "Provider AndroidOpenSSL" e altri componenti del sistema.

Tabella 5.3 Algoritmi supportati dal provider AndroidOpenSSL in Android 4.4.4.

Nome della classe engine	Algoritmi supportati
`CertificateFactory`	X509
	AES/CBC/NoPadding AES/CBC/PKCS5Padding AES/CFB/NoPadding

`Cipher`	AES/CTR/NoPadding AES/ECB/NoPadding AES/ECB/PKCS5Padding AES/OFB/NoPadding ARC4 DESEDE/CBC/NoPadding DESEDE/CBC/PKCS5Padding DESEDE/CFB/NoPadding DESEDE/ECB/NoPadding DESEDE/ECB/PKCS5Padding DESEDE/OFB/NoPadding RSA/ECB/NoPadding RSA/ECB/PKCS1Padding
`KeyAgreement`	ECDH
`KeyFactory`	DSA EC RSA
`KeyPairGenerator`	DSA EC RSA
`Mac`	HmacMD5 HmacSHA1 HmacSHA256 HmacSHA384 HmacSHA512
`MessageDigest`	MD5 SHA-1 SHA-256 SHA-384 SHA-512
`SecureRandom`	SHA1PRNG
`Signature`	ECDSA MD5WithRSA NONEwithRSA SHA1WithRSA SHA1withDSA SHA256WithRSA SHA256withECDSA SHA384WithRSA SHA384withECDSA SHA512WithRSA SHA512withECDSA

Le diverse release di Android usano versioni OpenSSL differenti (generalmente l'ultima versione stabile, ovvero la 1.0.1e in Android 4.4), con un set in evoluzione di patch applicate. Android non offre quindi un'API OpenSSL pubblica stabile e le applicazioni che devono usare OpenSSL sono tenute a includere la libreria e non a creare un link alla versione del sistema. L'unica API di crittografia pubblica è quella di JCA, che mette a disposizione un'interfaccia stabile dissociata dall'implementazione sottostante.

Uso di un provider personalizzato

Anche se i provider predefiniti di Android coprono le primitive di crittografia più usate, non supportano alcuni degli algoritmi più "esotici" e nemmeno alcuni degli standard più nuovi. Come già affermato nella descrizione dell'architettura JCA, le applicazioni Android possono registrare provider personalizzati per uso proprio, senza agire sui provider a livello di sistema.

Uno dei provider JCA più diffusi e ricchi di funzionalità è Bouncy Castle, su cui si basa anche uno dei provider predefiniti di Android. Tuttavia, come già affermato nel paragrafo "Provider Bouncy Castle di Android", nella versione fornita con Android sono stati rimossi diversi algoritmi. Se dovete utilizzare uno di questi, potete semplicemente provare a eseguire il bundle dell'intera libreria Bouncy Castle nella vostra applicazione; potreste però provocare dei conflitti di caricamento delle classi, specialmente nelle versioni di Android precedenti alla 3.0, in cui il nome del package Bouncy Castle di sistema non è stato modificato. Per evitare il problema, potete cambiare il package root della libreria con uno strumento come jarjar (Chris Nokleberg, *Jar Jar Links*, `https://code.google.com/p/jarjar/`), oppure utilizzare Spongy Castle (Roberto Tyley, *Spongy Castle*, `http://rtyley.github.io/spongycastle/`).

Spongy Castle

Spongy Castle è una versione repackaged di Bouncy Castle, in cui tutti i nomi dei package sono modificati da `org.bouncycastle.*` a `org.spongycastle.*` per evitare conflitti di caricamento delle classi; anche il nome del provider *BC* diventa *SC*. I nomi delle classi non subiscono modifiche e anche l'API è la stessa di Bouncy Castle. Per utilizzare Spongy Castle dovete semplicemente registrarlo con il framework JCA usando `Security.addProvider()` o `Security.insertProviderAt()`. Potete quindi richiedere gli algoritmi non implementati dai provider predefiniti di Android passando il nome dell'algoritmo al rispettivo metodo `getInstance()`.

Per richiedere esplicitamente un'implementazione di Spongy Castle, passate la stringa `SC` come nome del provider. Se avete eseguito il bundling della libreria Spongy Castle con la vostra app, potete usare direttamente anche l'API di crittografia leggera di Bouncy Castle (spesso più flessibile) senza addentrarvi nelle classi engine JCA. Inoltre, alcune operazioni di crittografia, come la firma di un certificato X.509 o la creazione di un messaggio S/MIME, non dispongono di API JCA corrispondenti e possono essere eseguite soltanto con le API Bouncy Castle di livello inferiore.

Il Listato 5.21 mostra come registrare il provider Spongy Castle e richiedere un'implementazione RSA-PSS (definito in origine in PKCS#1; fate riferimento a J. Jonsson e B. Kaliski, *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*, `http://tools.ietf.org/html/rfc3447`) di `Signature`, che non è supportata da alcun provider JCA predefinito di Android.

Listato 5.21 Registrazione e uso del provider Spongy Castle.

```
static {
    Security.insertProviderAt(
        new org.spongycastle.jce.provider.BouncyCastleProvider(), 1);
}
Signature sig = Signature.getInstance("SHA1withRSA/PSS", "SC");
```

Riepilogo

Android implementa la *Java Cryptography Architecture* (JCA) ed è fornito con numerosi provider di crittografia. JCA definisce le interfacce per gli algoritmi di crittografia comuni sotto forma di classi engine. I provider di crittografia forniscono le implementazioni di queste classi engine e spesso consentono ai client di richiedere un'implementazione dell'algoritmo per nome, senza dover conoscere l'implementazione sottostante. I due provider JCA principali di Android sono Bouncy Castle e AndroidOpenSSL. Bouncy Castle è implementato con codice Java puro, mentre AndroidOpenSSL usa il codice nativo e garantisce prestazioni superiori. A partire da Android 4.4 AndroidOpenSSL è il provider JCA privilegiato.

Sicurezza di rete e PKI

Nel capitolo precedente abbiamo affermato che Android include vari provider di crittografia che implementano la maggior parte delle moderne primitive di crittografia: hashing, crittografia simmetrica e asimmetrica e codici di autenticazione del messaggio. Queste primitive possono essere combinate per implementare la comunicazione sicura, ma anche un piccolo errore può causare gravi vulnerabilità; per questo, il metodo preferito per implementare la comunicazione sicura è l'uso di protocolli standard studiati per proteggere la privacy e l'integrità dei dati trasferiti in una rete.

I protocolli sicuri più diffusi sono *Secure Sockets Layer* (SSL) e *Transport Layer Security* (TLS). Android supporta questi protocolli fornendo un'implementazione dello standard *Java Secure Socket Extension* (JSSE). In questo capitolo esamineremo brevemente l'architettura JSSE e illustreremo alcuni dettagli sull'implementazione JSSE di Android. La nostra descrizione dello stack SSL di Android è incentrata sulla convalida dei certificati e sulla gestione dei trust anchor, che sono strettamente integrati nella piattaforma e rappresentano una delle più grandi differenze rispetto alle altre implementazioni di JSSE.

NOTA

Anche se TLS e SSL sono protocolli tecnicamente diversi, di solito usiamo il termine più comune *SSL* per fare riferimento a entrambi; distingueremo tra SSL e TLS solo nella descrizione delle differenze tra i protocolli.

Panoramica su PKI e SSL

TLS e SSL (il suo predecessore) sono protocolli di comunicazione point-to-point sicuri, ideati per fornire l'autenticazione (opzionale), la riservatezza dei messaggi e l'integrità dei messaggi a due entità che comunicano su TCP/IP. Usano una combinazione di crittografia simmetrica e asimmetrica per implementare la riservatezza e l'integrità dei messaggi e fanno affidamento sui certificati a chiave pubblica per implementare l'autenticazione (fate riferimento a T. Dierks ed E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, agosto 2008, http://tools.ietf.org/html/rfc5246, e A. Freier, P. Karlton e P. Kocher, *The Secure Sockets Layer (SSL) Protocol Version 3.0*, agosto 2011, http://tools.ietf.org/html/rfc6101).

Per avviare un canale SSL sicuro, un client contatta un server e invia la versione del protocollo SSL da esso supportata, insieme a un elenco di suite di cifratura suggerite. Una *suite di cifratura* è un set di algoritmi e dimensioni di chiavi utilizzato per l'autenticazione, la chiave concordata, la crittografia e l'integrità. Per stabilire un canale sicuro, il server e il client negoziano una suite di cifratura supportata da entrambi e quindi verificano reciprocamente l'identità dell'altro in base ai certificati. Infine, le entità concordano un algoritmo di crittografia simmetrica e calcolano una chiave simmetrica condivisa da usare per la crittografia delle comunicazioni successive. In genere, viene verificata solo l'identità del server (*autenticazione del server*) e non quella del client; il protocollo SSL supporta anche la verifica dell'identità del client (*autenticazione del client*), ma questa viene usata molto raramente.

NOTA

Anche se nelle specifiche SSL sono definite suite di cifratura anonime (o non autenticate), come *TLS_DH_anon_WITH_AES_128_CBC_SHA*, queste sono vulnerabili agli attacchi *man-in-the-middle* (MITM) e vengono pertanto impiegate solo quando SSL è usato come parte di un protocollo più complesso che dispone di altri mezzi per garantire l'autenticazione.

Certificati a chiave pubblica

Come affermato nel paragrafo precedente, SSL fa affidamento sui certificati a chiave pubblica per implementare l'autenticazione. Un certificato a chiave pubblica è un costrutto che associa un'identità a una chiave pubblica. Per i *certificati X.509*, usati nella comunicazione SSL, l'identità è un set di attributi che in genere include un nome comune (CN), un'organizzazione e una posizione che insieme formano il nome distinto (DN) dell'entità. Altri attributi importanti dei certificati X.509 sono il DN dell'autorità emittente, il periodo di validità e un set di estensioni, che possono essere attributi supplementari dell'entità o possono riguardare il certificato stesso (per esempio l'uso previsto per la chiave).

Il binding viene creato applicando una firma digitale sulla chiave pubblica dell'entità e su tutti gli attributi aggiuntivi per produrre un certificato digitale. La chiave di firma usata può essere la chiave privata propria dell'entità certificata, caso in cui il certificato è detto *autofirmato*, oppure può appartenere a una terza parte affidabile detta *autorità di certificazione* (CA).

Il contenuto di un tipico certificato del server X.509 viene sottoposto a parsing dal comando OpenSSL `x509` come mostrato nel Listato 6.1. Questo particolare certificato esegue il binding del DN `C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.googlecode.com` **(2)** e di un set di nomi DNS alternativi **(4)** con la chiave RSA a 2048 bit del server **(3)** ed è firmato con la chiave privata della CA `Google Internet Authority G2` **(1)**.

Listato 6.1 Contenuto del certificato X.509 sottoposto a parsing da OpenSSL.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            09:49:24:fd:15:cf:1f:2e
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=Google Inc, CN=Google Internet Authority G2(1)
        Validity
            Not Before: Oct  9 10:33:36 2013 GMT
            Not After : Oct  9 10:33:36 2014 GMT
        Subject: C=US, ST=California, L=Mountain View, O=Google Inc, CN=*.googlecode.com (2)
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)(3)
                Modulus:
                    00:9b:58:02:90:d6:50:03:0a:7c:79:06:99:5b:7a:
                    --altro codice--
                Exponent: 65537 (0x10001)
        X509v3 extensions:
```

```
        X509v3 Extended Key Usage:
            TLS Web Server Authentication, TLS Web Client Authentication
        X509v3 Subject Alternative Name:
            DNS:*.googlecode.com, DNS:*.cloud.google.com, DNS:*.code.google.com,(4)
            --altro codice--
        Authority Information Access:
            CA Issuers - URI:http://pki.google.com/GIAG2.crt
            OCSP - URI:http://clients1.google.com/ocsp

        X509v3 Subject Key Identifier:
            65:10:15:1B:C4:26:13:DA:50:3F:84:4E:44:1A:C5:13:B0:98:4F:7B
        X509v3 Basic Constraints: critical
            CA:FALSE
        X509v3 Authority Key Identifier:
            keyid:4A:DD:06:16:1B:BC:F6:68:B5:76:F5:81:B6:BB:62:1A:BA:5A:81:2F
        X509v3 Certificate Policies:
            Policy: 1.3.6.1.4.1.11129.2.5.1
        X509v3 CRL Distribution Points:
            Full Name:
              URI:http://pki.google.com/GIAG2.crl
Signature Algorithm: sha1WithRSAEncryption
     3f:38:94:1b:f5:0a:49:e7:6f:9b:7b:90:de:b8:05:f8:41:32:
     --altro codice--
```

Trust diretto e CA private

Se un client SSL comunica con un numero limitato di server, è possibile preconfigurarlo con un set di certificati del server ritenuti attendibili – detti *trust anchor* – e decidere se per considerare attendibile un'entità remota è sufficiente controllare se il suo certificato è in tale set. Questo modello consente un controllo preciso sui client attendibili, ma rende difficile la rotazione o l'aggiornamento delle chiavi del server, operazioni che richiedono l'emissione di un nuovo certificato autofirmato.

Questo problema può essere risolto usando una *CA privata* e configurando i client e i server per usarla come unico trust anchor. In questo modello le entità SSL non verificano il certificato di una particolare entità, ma considerano attendibile qualunque certificato emesso dalla CA privata. Questa scelta permette aggiornamenti trasparenti delle chiavi e dei certificati, senza che sia necessario aggiornare i client e i server SSL finché il certificato CA è valido. Lo svantaggio è che questo modello a CA singola crea un singolo "punto di errore": se la chiave della CA viene compromessa, chiunque abbia avuto accesso a essa può generare certificati fraudolenti che tutti i client riterranno attendibili (come vedremo più avanti, questo problema non riguarda solo le CA private). Il recupero della sicurezza a seguito di

questa situazione richiede l'aggiornamento di tutti i client e la sostituzione del certificato della CA.

Un altro problema di questo modello è l'impossibilità di usarlo per i client che non conoscono in anticipo i server a cui devono connettersi (di solito i client Internet generici quali browser web, applicazioni e-mail e client di messaggistica o VoIP). Questi client generici vengono solitamente configurati con un set di trust anchor che include autorità emittenti note che chiameremo *CA pubbliche*. Anche se esistono linee guida e requisiti specifici, il processo di selezione delle CA pubbliche da includere come trust anchor predefiniti varia notevolmente tra i browser e i sistemi operativi. Per esempio, per includere un certificato CA come trust anchor nei suoi prodotti, Mozilla richiede che la CA disponga di un documento *Certificate Policy and Certification Practice Statement* (CP/CPS) pubblico, che applichi l'autenticazione multifattore per gli account degli operatori e che il certificato CA non emetta direttamente certificati per le entità finali (http://mzl.la/1lvYZxY). Altri vendor hanno requisiti meno rigorosi; le versioni attuali della maggior parte dei sistemi operativi e dei browser è fornita con oltre 100 certificati CA inclusi come trust anchor.

Infrastruttura a chiave pubblica

Quando i certificati vengono emessi dalle CA pubbliche, viene eseguita una sorta di verifica dell'identità prima dell'emissione del certificato vero e proprio. Il processo di verifica varia notevolmente in base alle CA e ai tipi di certificati emessi: si va dall'accettazione di una conferma automatica tramite indirizzo e-mail (per i certificati server economici) alla richiesta di molteplici documenti di registrazione aziendali e statali (per i certificati a convalida estesa, o EV).

Le CA pubbliche dipendono da più persone, sistemi, procedure e policy per eseguire la verifica delle entità e per creare, gestire e distribuire i certificati. Il set di entità e sistemi è definito *infrastruttura a chiave pubblica* (PKI, *Public Key Infrastructure*). Le PKI possono essere infinitamente complesse, ma nel contesto della comunicazione sicura, e

di SSL in particolare, gli elementi più importanti sono i certificati della CA, che agiscono come trust anchor e sono utilizzati per la convalida dell'identità delle parti che comunicano. Di conseguenza, la gestione dei trust anchor sarà uno dei punti fondamentali nella nostra descrizione dell'implementazione di SSL e PKI in Android. Nella Figura 6.1 è mostrata una rappresentazione semplificata di una tipica PKI.

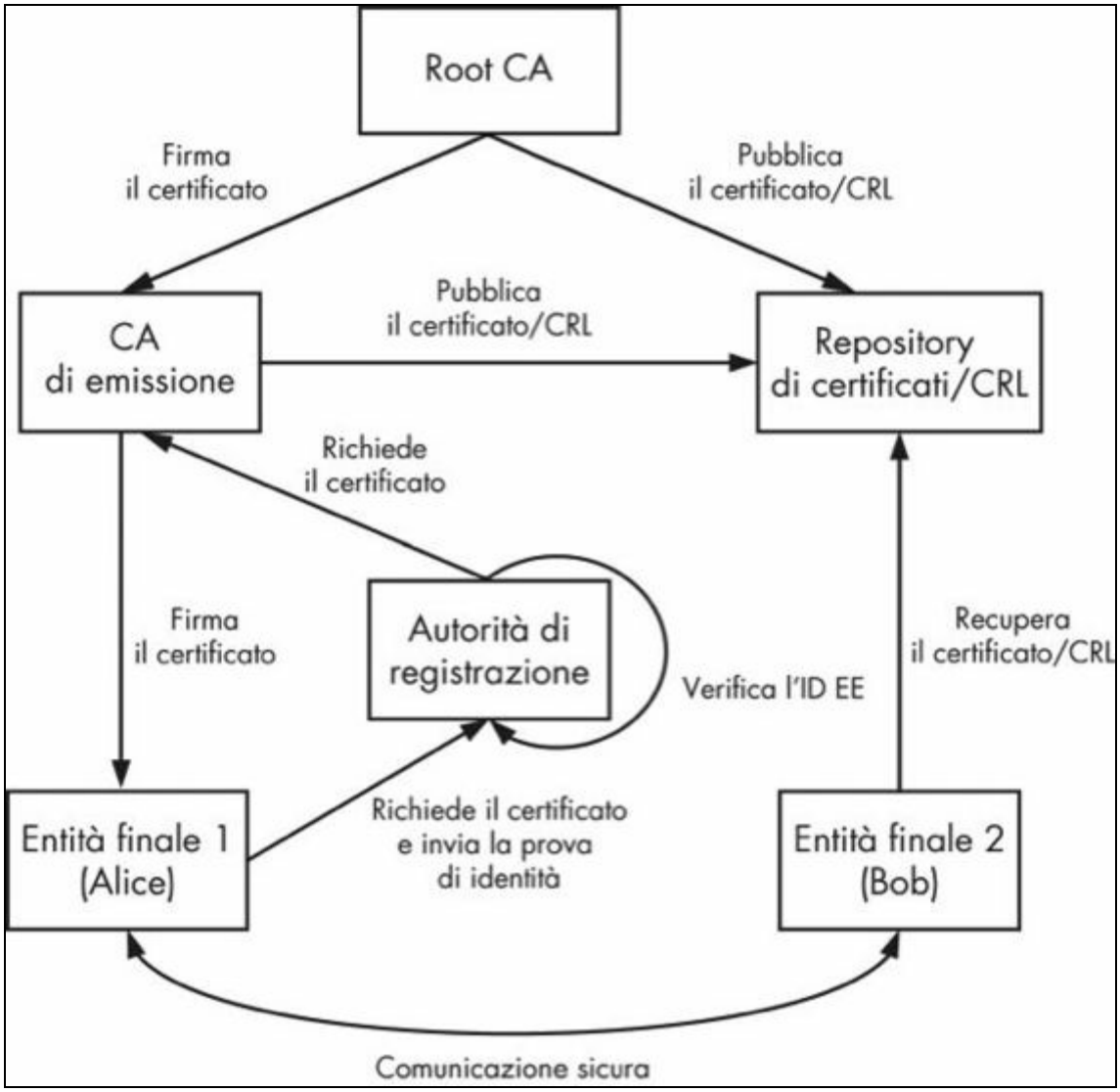

Figura 6.1 Entità PKI.

Qui, una persona o un server che possiede un certificato è detto *entità finale* (EE, *End Entity*). Per ottenere un certificato, un'entità finale invia una richiesta di certificato a un'autorità di registrazione (RA, *Registration Authority*). La RA ottiene una prova dell'identità dall'EE e ne verifica l'identità in base ai requisiti delle policy della CA. Dopo che la RA ha stabilito l'identità dell'EE, verifica che corrisponda al contenuto della richiesta di certificato e, in questo caso, inoltra la richiesta alla CA di

emissione. Una CA di emissione firma la richiesta di certificato dell'EE per generare i certificati EE e mantiene le informazioni sulla revoca (di cui parliamo nel prossimo paragrafo) dei certificati generati. D'altro canto, una CA root non firma direttamente i certificati EE, ma firma solamente i certificati per le CA di emissione e le informazioni di revoca riguardanti tali CA. La CA root è usata molto raramente e di solito è mantenuta offline per aumentare la sicurezza delle sue chiavi.

Nella PKI della Figura 6.1, un certificato EE è associato a due certificati CA: il certificato della CA di emissione, che lo ha firmato, e il certificato della CA root, che ha firmato il certificato della CA di emissione. I tre certificati formano una catena (detta anche *percorso di certificazione*). La catena inizia con il certificato dell'EE e termina con il certificato del CA root. Perché un certificato EE sia ritenuto attendibile, il suo percorso di certificazione deve portare a un certificato che il sistema ritiene implicitamente attendibile (un trust anchor). Anche se è possibile usare i certificati intermedi come trust anchor, questo ruolo è solitamente svolto dai certificati delle CA root.

Revoca dei certificati

Oltre a rilasciare certificati, le CA possono contrassegnare un certificato come non valido revocandolo. La *revoca* prevede l'aggiunta del numero di serie del certificato e di un motivo per la revoca a un elenco di revoche di certificati (CRL, *Certificate Revocation List*) che la CA firma e pubblica periodicamente. Le entità che convalidano un certificato possono quindi verificare se il certificato è stato revocato cercando il suo numero di serie (univoco all'interno di una CA specifica) nel CRL attuale della CA di emissione. Nel Listato 6.2 è mostrato il contenuto di un file CRL di esempio, rilasciato dalla Google Internet Authority G2. In questo esempio, i certificati con numero di serie `40BF8571DD53E3BB` **(1)** e `0A9F21196A442E45` **(2)** sono stati revocati.

Listato 6.2 Contenuto del file CRL.

```
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm: sha1WithRSAEncryption
```

```
        Issuer: /C=US/O=Google Inc/CN=Google Internet Authority G2
        Last Update: Jan 13 01:00:02 2014 GMT
        Next Update: Jan 23 01:00:02 2014 GMT
        CRL extensions:
            X509v3 Authority Key Identifier:
                keyid:4A:DD:06:16:1B:BC:F6:68:B5:76:F5:81:B6:BB:62:1A:BA:5A:81:2F
            X509v3 CRL Number:
                219
Revoked Certificates:
    Serial Number: 40BF8571DD53E3BB(1)
        Revocation Date: Sep 10 15:19:22 2013 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Affiliation Changed
--altro codice--
    Serial Number: 0A9F21196A442E45(2)
        Revocation Date: Jun 12 17:42:06 2013 GMT
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Superseded
    Signature Algorithm: sha1WithRSAEncryption
         40:f6:05:7d:...
```

Lo stato di revoca può essere verificato anche senza recuperare l'elenco completo di tutti i certificati revocati utilizzando l'*Online Certificate Status Protocol* (OCSP) (fate riferimento a S. Santesson *et al.*, *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*, giugno 2013, http://tools.ietf.org/html/rfc6960). Gli URI di CRL e OCSP sono spesso inclusi come estensioni nei certificati affinché le entità che svolgono la verifica non debbano conoscerne anticipatamente la posizione. Tutte le CA pubbliche mantengono le informazioni di revoca, ma in pratica molti client SSL non controllano del tutto la revoca o consentono la connessione (magari con un avviso) anche se il certificato dell'entità remota è stato revocato. Questo comportamento indulgente dei client SSL è dovuto soprattutto all'overhead associato al recupero delle informazioni di revoca attuali e alla volontà di garantire la connettività. Anche se i CRL delta (CRL che contengono solo le differenze, o *delta*, rispetto alla precedente versione del CRL) e il caching locale alleviano per certi versi il problema, i CRL per le CA più importanti sono generalmente enormi e devono essere scaricati prima di stabilire una connessione SSL, aumentando la latenza visibile all'utente. OCSP migliora la situazione ma richiede tuttora una connessione a un server diverso, aumentando ulteriormente la latenza.

In ogni caso, le informazioni di revoca potrebbero non essere disponibili a causa di un problema di rete o di configurazione nell'infrastruttura di una CA. Per una CA principale, un problema nel

database di revoca potrebbe disabilitare numerosi siti protetti, comportando perdite finanziarie per i relativi operatori. Infine, nessuno ama gli errori di connessione; quando si trovano di fronte un errore di revoca, molto utenti si limitano a scegliere un client SSL meno rigoroso che semplicemente “funzioni”.

Introduzione a JSSE

In questo paragrafo vedremo brevemente l'architettura e i componenti principali di JSSE (una descrizione completa è disponibile all'indirizzo http://bit.ly/1CoSPlj).

L'API JSSE risiede nei package `javax.net` e `javax.net.ssl` e fornisce classi che rappresentano le seguenti funzionalità.

- Socket client e server SSL.
- Un engine per la produzione e il consumo di flussi SSL (`SSLEngine`).
- Factory per la creazione di socket.
- Una classe contestuale per socket sicuri (`SSLContext`) che crea engine e factory per socket sicuri.
- Gestori di chiavi e trust basati su PKI e factory per la loro creazione.
- Una classe per i collegamenti URL HTTPS (`HttpsURLConnection`, *HTTP over TLS*, descritto in *RFC 2818*; vedi E. Rescorla, *HTTP Over TLS*, maggio 2000, http://tools.ietf.org/html/rfc2818).

Come i provider del servizio di crittografia JCA, un provider JSSE fornisce implementazioni delle classi engine definite nell'API. Queste classi di implementazione sono responsabili della creazione dei socket sottostanti, nonché dei gestori delle chiavi e dei trust richiesti per stabilire una connessione; tuttavia, gli utenti dell'API JSSE non interagiscono mai direttamente con esse, ma solo con le rispettive classi engine. Riesaminiamo brevemente le classi e le interfacce principali dell'API di JSSE, valutandone le relazioni reciproche.

Socket sicuri

JSSE supporta sia l'I/O di blocco basato su flussi che usa i socket sia l'I/O non di blocco basato su canali NIO (*New I/O*). La classe centrale per la comunicazione basata su flussi è `javax.net.ssl.SSLSocket`, creata da `SSLSocketFactory` o chiamando il metodo `accept()` della classe `SSLServerSocket`. A loro volta, le istanze di `SSLSocketFactory` e `SSLServerSocketFactory` sono create

chiamando i metodi factory appropriati della classe `SSLContext`. I factory dei socket SSL incapsulano i dettagli per la creazione e la configurazione dei socket SSL, comprendendo le chiavi di autenticazione, le strategie di convalida dei certificati dei peer e le suite di cifratura abilitate. Questi dettagli sono in genere comuni a tutti i socket SSL usati da un'applicazione e vengono configurati durante l'inizializzazione dell'elemento `SSLContext` dell'applicazione. Vengono poi passati a tutti i factory di socket SSL creati dall'istanza `SSLContext` condivisa. Se non è stato configurato esplicitamente un `SSLContext`, vengono usati i valori predefiniti di sistema per tutti i parametri SSL.

L'I/O SSL non di blocco è implementato nella classe `javax.net.ssl.SSLEngine`, che incapsula una macchina a stati SSL e opera sui buffer dei byte forniti dai suoi client. Anche se `SSLSocket` nasconde gran parte della complessità di SSL, per garantire una flessibilità superiore `SSLEngine` lascia gli incarichi di I/O e threading all'applicazione chiamante. Per questo, i clienti `SSLEngine` devono essere in grado di comprendere il protocollo SSL. Le istanze di `SSLEngine` vengono create direttamente da `SSLContext` e ne ereditano la configurazione SSL, analogamente ai factory dei socket SSL.

Autenticazione dei peer

L'autenticazione dei peer è parte integrante del protocollo SSL e fa affidamento sulla disponibilità di un set di trust anchor e chiavi di autenticazione. In JSSE, la configurazione dell'autenticazione dei peer è fornita con l'aiuto delle classi engine `KeyStore`, `KeyManagerFactory` e `TrustManagerFactory`. Un `KeyStore` rappresenta un mezzo di archiviazione delle chiavi e dei certificati di crittografia, utilizzabile per archiviare sia i certificati dei trust anchor sia le chiavi dell'entità finale con i certificati associati. `KeyManagerFactory` e `TrustManagerFactory` creano rispettivamente `KeyManager` e `TrustManager` sulla base dell'algoritmo di autenticazione specificato. Anche se sono possibili implementazioni basate su strategie di autenticazione diverse, in pratica SSL usa solamente una PKI basata su X.509 (PKIX) per l'autenticazione e l'unico algoritmo supportato da queste classi

factory è PKIX (con alias per `X.509`; fate riferimento a D. Cooper *et al., Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, maggio 2008, http://tools.ietf.org/html/rfc5280).

Un `SSLContext` può essere inizializzato con un set di istanze di `KeyManager` e `TrustManager` chiamando il metodo riportato di seguito. Tutti i parametri sono opzionali; quando è specificato `null`, viene utilizzato il valore predefinito del sistema (Listato 6.3).

Listato 6.3 Metodo di inizializzazione di SSLContext.

```
void init(KeyManager[] km, TrustManager[] tm, SecureRandom random);
```

Un `TrustManager` determina se le credenziali di autenticazione dei peer presentate devono essere considerate attendibili: in questo caso la connessione viene stabilita, altrimenti viene terminata. Nel contesto di PKIX, in pratica, viene convalidata la catena di certificati del certificato peer presentato sulla base dei trust anchor configurati. Questo comportamento è rispecchiato anche nell'uso di JSSE dell'interfaccia `X509TrustManager` (Listato 6.4).

Listato 6.4 Metodi dell'interfaccia X509TrustManager.

```
void checkClientTrusted(X509Certificate[] chain, String authType);
void checkServerTrusted(X509Certificate[] chain, String authType);
X509Certificate[] getAcceptedIssuers();
```

La convalida della catena di certificati viene eseguita usando l'implementazione dell'API di sistema Java Certification Path (o API CertPath, http://bit.ly/1stwwZd) responsabile della creazione e della convalida delle catene di certificati. Anche se l'API presenta un'interfaccia in parte indipendente dall'algoritmo, in pratica è strettamente correlata a PKIX e implementa gli algoritmi di creazione e convalida della catena definiti negli standard PKIX. L'implementazione predefinita di `TrustManagerFactory` in PKIX può creare un'istanza di `X509TrustManager` che preconfigura le classi dell'API CertPath sottostante con i trust anchor archiviati in un oggetto `KeyStore`.

L'oggetto `KeyStore` viene tipicamente inizializzato da un file keystore di sistema definito *trust store*. Se è richiesta una configurazione più precisa, è possibile usare un'istanza di `CertPathTrustManagerParameters` contenente

parametri dettagliati dell'API CertPath per inizializzare anche `TrustManagerFactory`. Quando l'implementazione di `X509TrustManager` di sistema non può essere configurata come richiesto usando le API fornite, è possibile creare un'istanza personalizzata implementando direttamente l'interfaccia, magari delegando i casi di base all'implementazione predefinita.

Un `KeyManager` determina quali credenziali di autenticazione inviare all'host remoto; nel contesto di PKIX, equivale a scegliere il certificato di autenticazione client da inviare a un server SSL. Il `KeyManagerFactory` predefinito può creare un'istanza di `KeyManager` che usa un `KeyStore` per cercare le chiavi di autenticazione client e i certificati correlati. Proprio come per `TrustManager`, le interfacce concrete `X509KeyManager` (Listato 6.5) e `X509ExtendedKeyManager` (che consente la selezione della chiave specifica per la connessione) sono specifiche di PKIX e selezionando un certificato client in base all'elenco di autorità emittenti attendibili fornito dal server. Se l'implementazione sostenuta dal `KeyStore` predefinito non è abbastanza flessibile, è possibile fornire un'implementazione personalizzata estendendo la classe astratta `X509ExtendedKeyManager`.

Listato 6.5 Interfaccia X509KeyManager.

```
String chooseClientAlias(String[] keyType, Principal[] issuers, Socket socket);
String chooseServerAlias(String keyType, Principal[] issuers, Socket socket);
X509Certificate[] getCertificateChain(String alias);
String[] getClientAliases(String keyType, Principal[] issuers);
PrivateKey getPrivateKey(String alias);
String[] getServerAliases(String keyType, Principal[] issuers);
```

Oltre al supporto per i socket SSL di tipo *raw*, JSSE fornisce il supporto per HTTPS con la classe `HttpsURLConnection`, che utilizza l'elemento `SSLSocketFactory` predefinito per creare socket sicuri durante l'apertura di una connessione a un server web. Se è richiesta una configurazione SSL aggiuntiva, per esempio specificando chiavi di autenticazione o trust anchor privati dell'app, l'elemento `SSLSocketFactory` può essere sostituito per tutte le istanze di `HttpsURLConnection` chiamando il metodo statico `setDefaultSSLSocketFactory()`. In alternativa, è possibile configurare il socket factory per una particolare istanza chiamando il suo metodo `setSSLSocketFactory()`.

Verifica del nome host

Anche se SSL verifica l'identità del server controllandone il certificato, il protocollo non impone alcuna verifica del nome host e, quando si usano socket SSL di tipo raw, il soggetto del certificato non viene abbinato al nome host del server. Lo standard HTTPS, invece, impone tale controllo e `HttpsURLConnection` lo esegue internamente. L'algoritmo predefinito per la verifica del nome host può essere sostituito assegnando un'istanza di `HostnameVerifier` alla classe o procedendo istanza per istanza. Il callback `verify()` che deve implementare è mostrato nel Listato 6.6. La classe `SSLSession` usata nel callback incapsula i dettagli sulla connessione SSL corrente, compresi il protocollo e la suite di cifratura selezionati, le catene di certificati locale e peer, il nome host del peer e il numero di porta per la connessione.

Listato 6.6 Callback di verifica del nome host HostnameVerifier.

```
boolean verify(String hostname, SSLSession session);
```

Abbiamo già presentato le classi e le interfacce principali dell'API di JSSE, descrivendone le relazioni reciproche mostrate nella Figura 6.2.

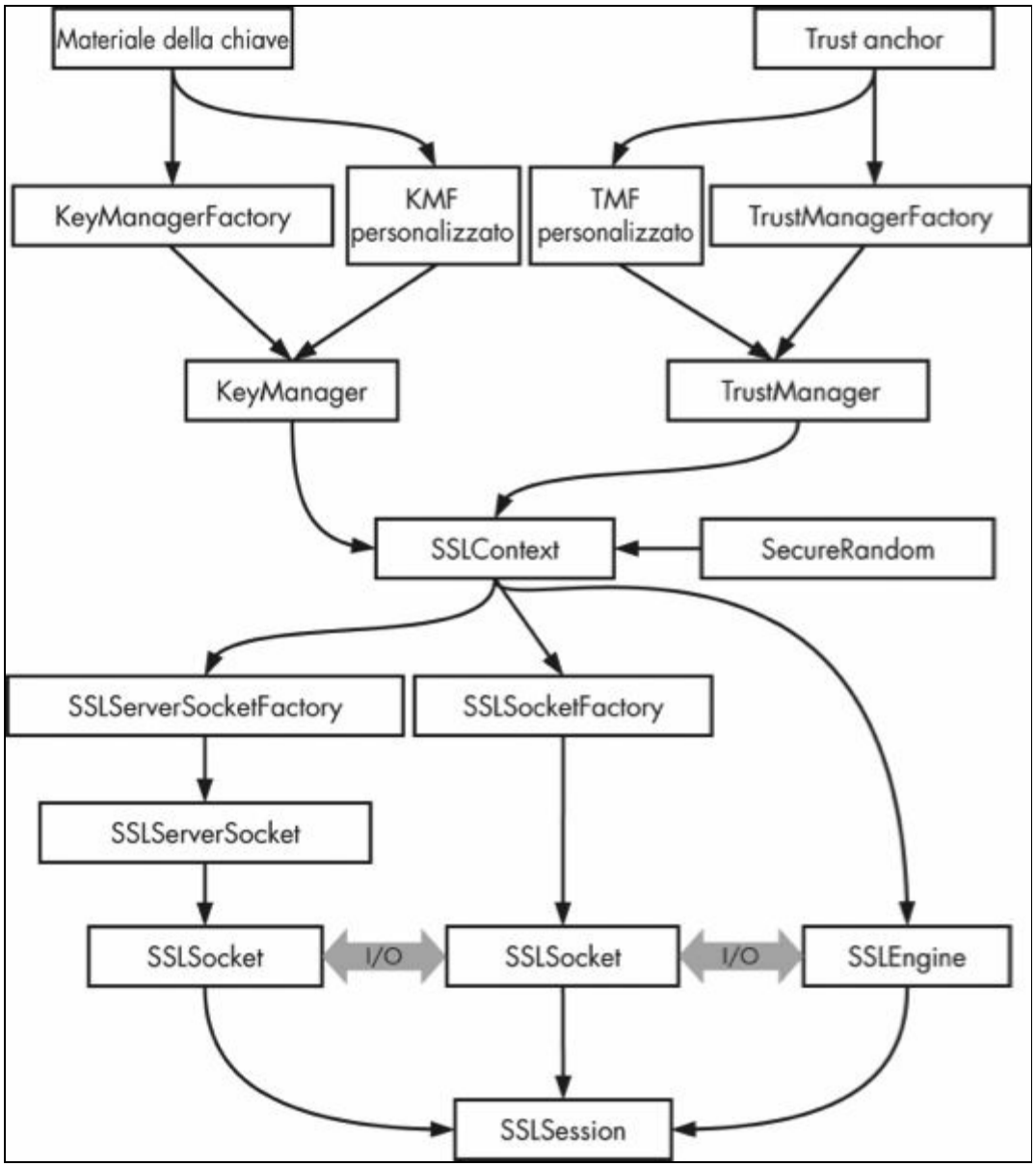

Figura 6.2 Classi JSSE e relative relazioni.

Implementazione JSSE di Android

Android è fornito con due provider JSSE: HarmonyJSSE, basato su Java, e AndroidOpenSSL, implementato perlopiù in codice nativo e associato all'API Java pubblica tramite JNI. HarmonyJSSE si basa sui socket Java e sulle classi JCA per implementare SSL, mentre AndroidOpenSSL implementa la maggior parte delle sue funzionalità con le chiamate alla libreria OpenSSL. Come spiegato nel Capitolo 5, AndroidOpenSSL è il provider JCA preferito in Android e fornisce le implementazioni predefinite `SSLSocketFactory` e `SSLServerSocketFactory` restituite rispettivamente da `SSLSocketFactory.getDefault()` e `SSLServerSocketFactory.getDefault()`.

Entrambi i provider JSSE sono parte della libreria Java fondamentale (presente in `core.jar` e `libjavacore.so`), mentre la parte nativa del provider AndroidOpenSSL viene compilata in `libjavacrypto.so`. HarmonyJSSE fornisce solo il supporto per SSLv3.0 e TLSv1.0, mentre AndroidOpenSSL supporta anche TLSv1.1 e TLSv1.2. Anche se l'implementazione del socket SSL è diversa, entrambi i provider condividono il codice per `TrustManager` e `KeyManager`.

NOTA

Il provider HarmonyJSSE è ancora disponibile in Android 4.4, ma è considerato deprecato e non viene mantenuto in maniera attiva. È possibile che venga rimosso dalle versioni future di Android.

Oltre alle versioni attuali del protocollo TLS, il provider basato su OpenSSL supporta l'estensione TLS *Server Name Indication* (SNI, definita in *RFC 3546*, S. Blake-Wilson *et al.*, *Transport Layer Security (TLS) Extensions*, giugno 2003, http://tools.ietf.org/html/rfc3546), che consente ai client SSL di specificare il nome host previsto durante la connessione ai server che ospitano più host virtuali. SNI è usato per impostazione predefinita quando si stabilisce una connessione con la classe `HttpsURLConnection` in Android 3.0 e versioni successive (la versione 2.3 offre un supporto SNI parziale). Tuttavia, SNI non è supportato durante l'uso della libreria client HTTP Apache fornita con Android (nel package `org.apache.http`).

Prima di Android 4.2, lo stack HTTP nella libreria Java core di Android, che comprende `HttpsURLConnection`, era basata sul codice Apache Harmony. In Android 4.2 e versioni successive, l'implementazione originale è stata sostituita dalla libreria client HTTP & SPDY di Square, *OkHttp* (http://square.github.io/okhttp/).

Gestione e convalida dei certificati

Le implementazioni JSSE di Android si conformano per la maggior parte alla specifica dell'API JSSE, ma presentano alcune differenze degne di nota. La più importante riguarda la gestione del trust store di sistema da parte di Android. Nelle implementazioni JSSE di Java SE, il trust store di sistema è un singolo file keystore (generalmente chiamato *cacerts*) la cui posizione può essere impostata con la proprietà di sistema `javax.net.ssl.trustStore`; Android segue tuttavia una strategia diversa. Le versioni recenti di Android forniscono anche moderne funzionalità di convalida dei certificati, quali blacklisting e pinning, che non sono specificate nel documento originale dell'architettura JSSE. Nei prossimi paragrafi vedremo l'implementazione del trust store e le funzionalità di convalida avanzata dei certificati di Android.

Trust store di sistema

Come spiegato nel paragrafo "Autenticazione dei peer", le implementazioni di JSSE usano un trust store per autenticare i peer di connessione. Anche se SSL supporta connessioni non autenticate che utilizzano unicamente la crittografia, in pratica i client SSL raw eseguono l'autenticazione del server, obbligatoria per HTTPS. Se non è stato fornito esplicitamente un trust store per applicazione, JSSE usa quello di sistema per eseguire l'autenticazione dei peer SSL. Il trust store di sistema è particolarmente importante per i client Internet generici come i browser, che di solito non ne gestiscono uno proprio sui dispositivi mobili (le versioni desktop dei client Mozilla mantengono store privati di certificati e credenziali, ma questo non avviene su Android). I trust store

di sistema sono quindi fondamentali per la sicurezza di tutte le applicazioni che usano JSSE, pertanto è utile vederne l'implementazione nei dettagli.

Fino ad Android 4.0, il trust store del sistema operativo era inserito direttamente nel sistema e gli utenti non avevano alcun controllo su di esso. I certificati da inserire nello store venivano scelti solamente dal produttore del dispositivo o dal gestore telefonico. L'unico modo per apportare cambiamenti era eseguire il root del device, il repackage del file dei certificati attendibili e la sostituzione dell'originale, una procedura chiaramente poco pratica e che ostacola l'uso di Android nelle PKI aziendali. Sulla scia del compromesso accettato dalle CA più importanti, sono stati sviluppati strumenti che permettono di cambiare i certificati considerati attendibili dal sistema, ma anche in questo caso è necessario uno smartphone di root. Fortunatamente Android 4.0 ha reso molto più flessibile la gestione del trust store e ha concesso agli utenti il controllo sui trust.

Trust store di sistema Android 4.x

Prima di Android 4.0, il trust store di sistema era costituito da un singolo file, chiamato `/system/etc/security/cacerts.bks`, che era un file keystore nativo di Bouncy Castle (uno dei provider di crittografia usati in Android, illustrato nel Capitolo 5). Questo file conteneva tutti i certificati CA considerati attendibili da Android ed era usato sia dalle app di sistema, quali client e-mail e browser, sia dalle app di terze parti. Il file risiedeva nella partizione di sola lettura *system*, pertanto non poteva essere modificato nemmeno dalle applicazioni di sistema.

Android 4.0 ha introdotto una nuova classe `TrustedCertificateStore`, molto più flessibile, che consente di gestire i trust anchor predefiniti e di aggiungerne di nuovi. La classe legge ancora i certificati considerati attendibili dal sistema da */system/etc/security/*, ma aggiunge due nuove posizioni modificabili per archiviare i certificati CA in */data/misc/keychain/*: le directory *cacerts-added/* e *cacerts-removed/*. Il Listato 6.7 mostra il loro contenuto.

Listato 6.7 Contenuto delle directory cacerts-added/ e cacerts-removed/.

```
# ls -l /data/misc/keychain
drwxr-xr-x system   system            cacerts-added
drwxr-xr-x system   system            cacerts-removed
-rw-r--r-- system   system          81 pubkey_blacklist.txt
-rw-r--r-- system   system           7 serial_blacklist.txt
# ls -l /data/misc/keychain/cacerts-added
-rw-r--r-- system   system        653 30ef493b.0 (1)
# ls -l /data/misc/keychain/cacerts-removed
-rw-r--r-- system   system       1060 00673b5b.0 (2)
```

Ogni file in queste directory contiene un certificato CA. I nomi dei file possono apparire familiari: sono basati sugli hash MD5 dei nomi dei soggetti CA (calcolati con la funzione OpenSSL `X509_NAME_hash_old()`), proprio come sono usati in *mod_ssl* e in altri software di crittografia implementati con OpenSSL. In questo modo è facile individuare facilmente i certificati senza esaminare l'intero archivio, trasformando direttamente il DN in un nome file. Vanno osservati anche i permessi delle directory: *0775 system system* garantisce che solo l'utente `system` possa aggiungere o rimuovere certificati, ma che tutti gli altri possano leggerli. Come previsto, l'aggiunta di certificati CA trusted viene implementata archiviando il certificato nella directory *cacerts-added/* utilizzando il nome file appropriato. Il certificato salvato nel file `30ef493b.0` (**(1)** nel Listato 6.7) sarà visualizzato anche nella scheda *User* dell'applicazione di sistema *Trusted credentials* (*Settings* > *Security* > *Trusted credentials*). Vediamo ora come disabilitare i certificati considerati attendibili dal sistema operativo. Dal momento che i certificati CA preinstallati sono ancora archiviati in */system/etc/security/* (montato in sola lettura), una CA viene segnalata come non attendibile inserendo una copia del suo certificato nella directory *cacerts-removed/*. Per riabilitarlo basta ovviamente rimuovere il file. In questo caso specifico, `00673b5b.0` (**(2)** nel Listato 6.7) è *thawte Primary Root CA*, mostrata come disabilitata nella scheda *System* (Figura 6.3).

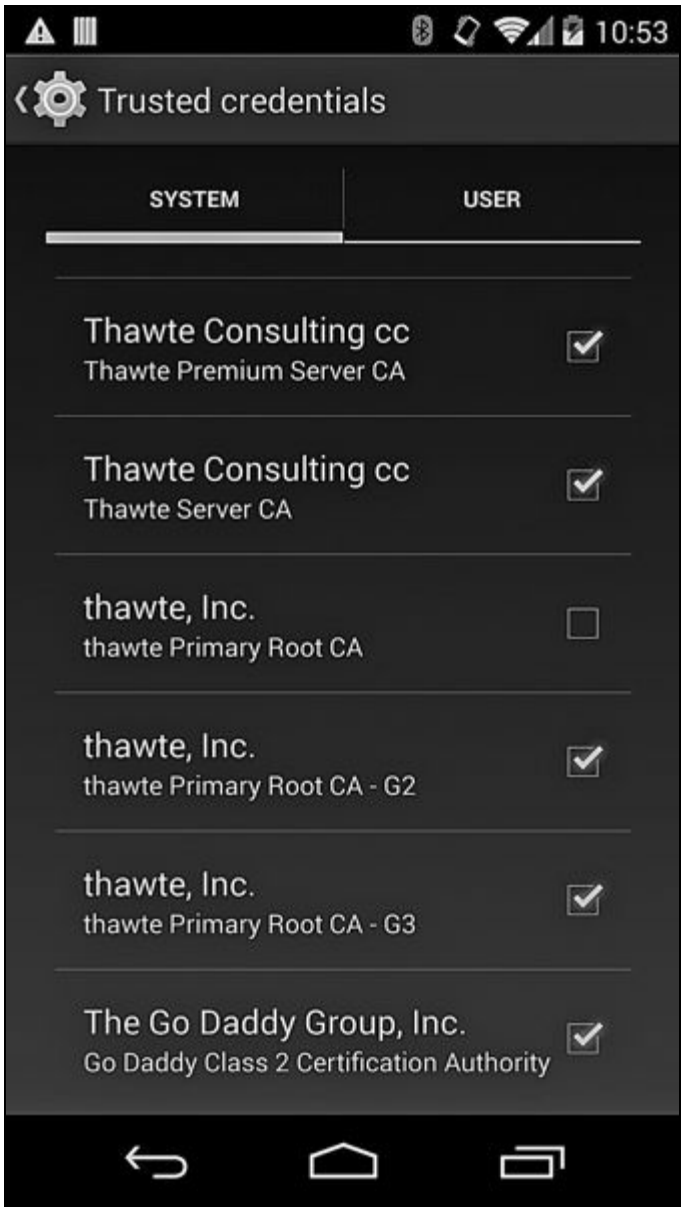

Figura 6.3 Certificato CA preinstallato contrassegnato come non attendibile.

Uso del trust store di sistema

`TrustedCertificateStore` non è parte dell'SDK di Android, ma dispone di un wrapper (`TrustedCertificateKeyStoreSpi`) accessibile tramite l'API `KeyStore` JCA standard e utilizzabile dalle applicazioni (Listato 6.8).

Listato 6.8 Visualizzazione di un elenco dei certificati attendibili con AndroidCAStore.

```
KeyStore ks = KeyStore.getInstance("AndroidCAStore");(1)
ks.load(null, null);(2)
Enumeration<String> aliases = ks.aliases();(3)
while (aliases.hasMoreElements()) {
    String alias = aliases.nextElement();
    Log.d(TAG, "Certificate alias: " +  alias);
    X09Certificate cert = (X509Certificate) ks.getCertificate(alias);(4)
    Log.d(TAG, "Subject DN: " + cert.getSubjectDN().getName());
    Log.d(TAG, "Issuer DN: " + cert.getIssuerDN().getName());
}
```

Per ottenere un elenco dei certificati attendibili attuali si procede come segue.

1. Create un'istanza di `KeyStore` specificando *AndroidCAStore* come parametro `type` **(1)**.
2. Chiamate il suo metodo `load()` e passate `null` per entrambi i parametri **(2)**.
3. Ottenete un elenco di alias dei certificati con il metodo `aliases()` **(3)**.
4. Passate ogni alias al metodo `getCertificate()` per ottenere l'oggetto certificato vero e proprio **(4)**.

Esaminando l'output di questo codice noterete che gli alias dei certificati iniziano con il prefisso *user:* (per i certificati installati dall'utente) o *system:* (per quelli preinstallati) seguito dal valore hash del soggetto.

L'implementazione AndroidCAStore di `KeyStore` consente di accedere facilmente ai certificati considerati attendibili dal sistema operativo; tuttavia, un'applicazione è probabilmente più interessata a capire se considerare attendibile un particolare certificato del server piuttosto che a conoscere i trust anchor effettivi. Android facilita questa operazione integrando `TrustedCertificateKeyStoreSpi` nella sua implementazione di JSSE. Il `TrustManagerFactory` predefinito lo usa per ottenere un elenco di trust anchor e quindi convalida automaticamente i certificati del server confrontandoli con i certificati ritenuti attualmente attendibili dal sistema. Il codice di livello più alto che usa `HttpsURLConnection` o `HttpClient` (entrambi basati su JSSE) dovrebbe quindi funzionare senza doversi preoccupare di creare e inizializzare un `SSLSocketFactory` personalizzato.

Per installare il nostro certificato CA personale (per esempio quello di una CA aziendale privata) nel trust store di sistema, dobbiamo convertirlo nel formato DER (binario) e copiarlo sul dispositivo. Nelle versioni precedenti ad Android 4.4.1, il file del certificato deve essere copiato nella radice della memoria esterna usando un'estensione `.crt` o `.cer`. Android 4.4.1 e versioni successive usano il framework di accesso alla memoria, introdotto in Android 4.4, e consentono di scegliere un file

di certificato da qualunque back-end di memoria a cui il dispositivo può accedere, compresi i provider cloud integrati come Google Drive. Possiamo quindi importare il certificato usando l'app di sistema *Settings*, selezionando *Settings* > *Personal* > *Security* > *Credential storage* > *Install from storage*. Viene così visualizzato un elenco di file di certificato disponibili, e toccando un nome file viene mostrata la finestra di dialogo di importazione (Figura 6.4).

Il certificato importato sarà visualizzato nella scheda *User* della schermata *Trusted credentials* (Figura 6.5). Potete visualizzare i dettagli del certificato toccando la relativa voce nell'elenco, nonché rimuoverlo scorrendo fino in fondo la schermata dei dettagli e toccando il pulsante *Remove*.

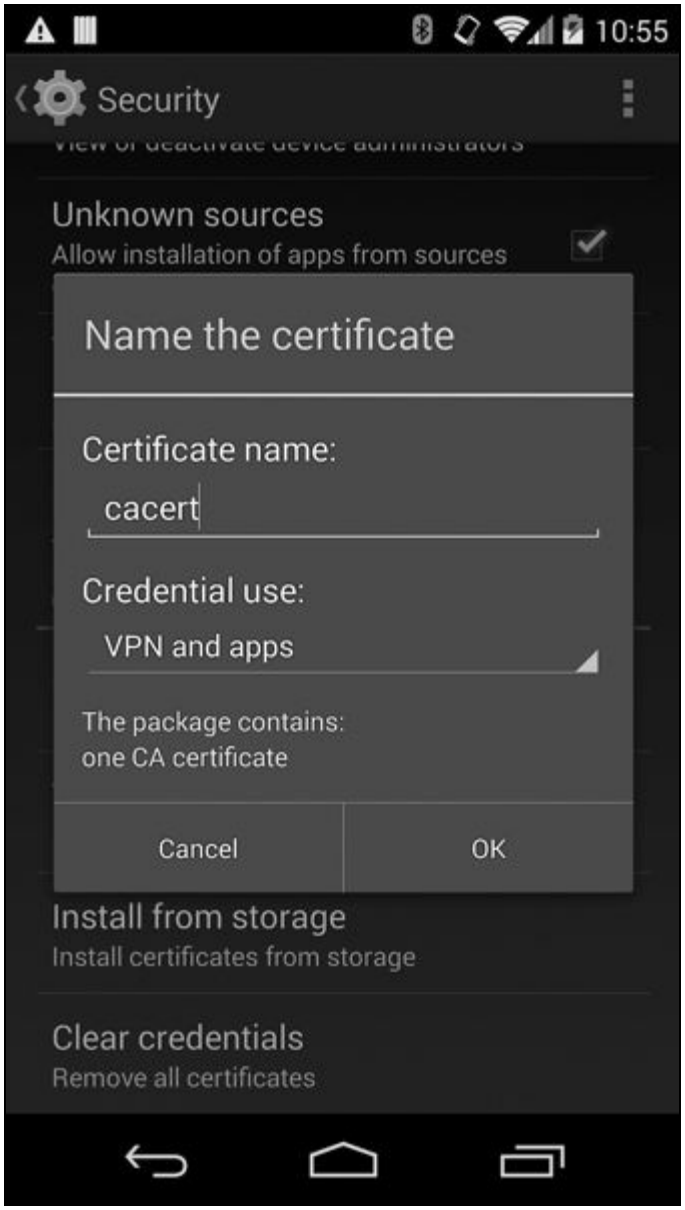

Figura 6.4 Finestra di dialogo di importazione di un certificato CA.

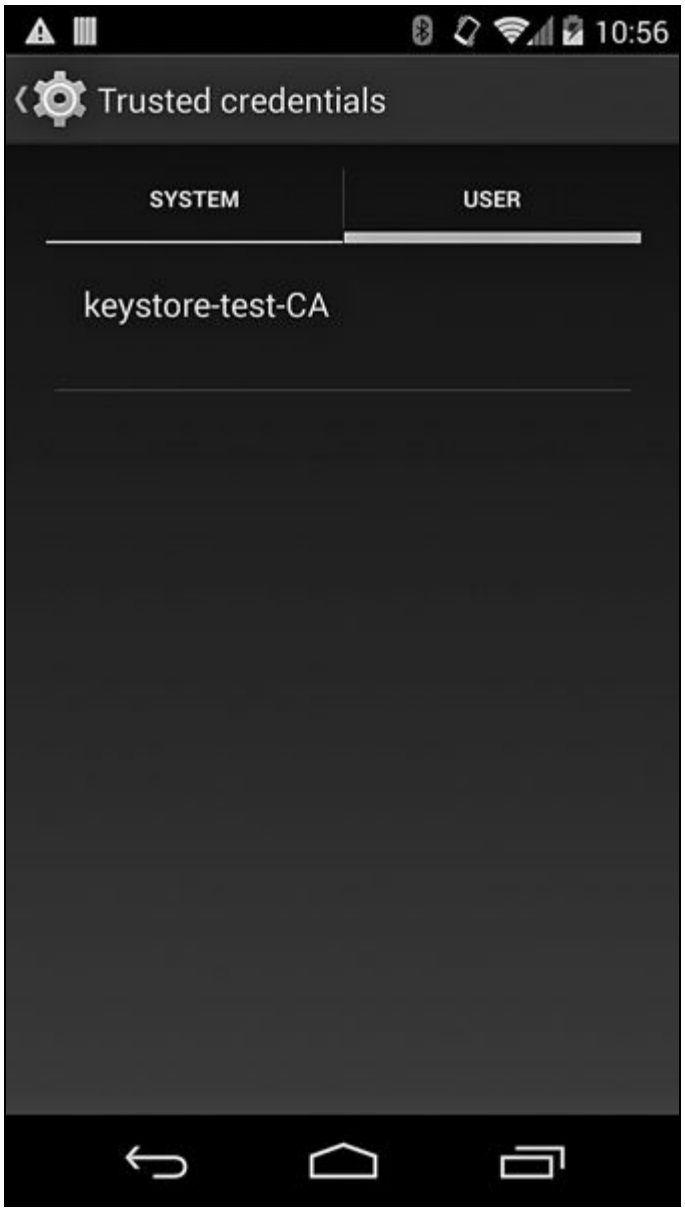

Figura 6.5 Certificati CA importati dall'utente.

NOTA

Se il certificato viene importato correttamente, il file del certificato nel file di memoria esterna sarà eliminato nelle versioni precedenti ad Android 4.4.1.

A partire da Android 4.4, il sistema visualizza una notifica che informa l'utente che l'attività di rete potrebbe essere monitorata qualora vi siano certificati attendibili installati dall'utente. Il monitoraggio della connessione SSL può essere effettuato usando un server proxy di intercettazione che restituisce i certificati generati automaticamente per i siti a cui l'utente cerca di accedere. Finché questi certificati sono rilasciati da una CA considerata attendibile da Android (come quella installata manualmente nel trust store), la maggior parte delle applicazioni non nota la differenza tra una connessione all'host originale e una connessione al

proxy di intercettazione (tranne in caso di pinning dell'host target; consultate il paragrafo "Pinning dei certificati" per i dettagli). In *Quick Settings* e accanto alla preferenza *Security* nelle impostazioni di sistema compare un'icona di avviso che, se selezionata, visualizza il messaggio mostrato nella Figura 6.6.

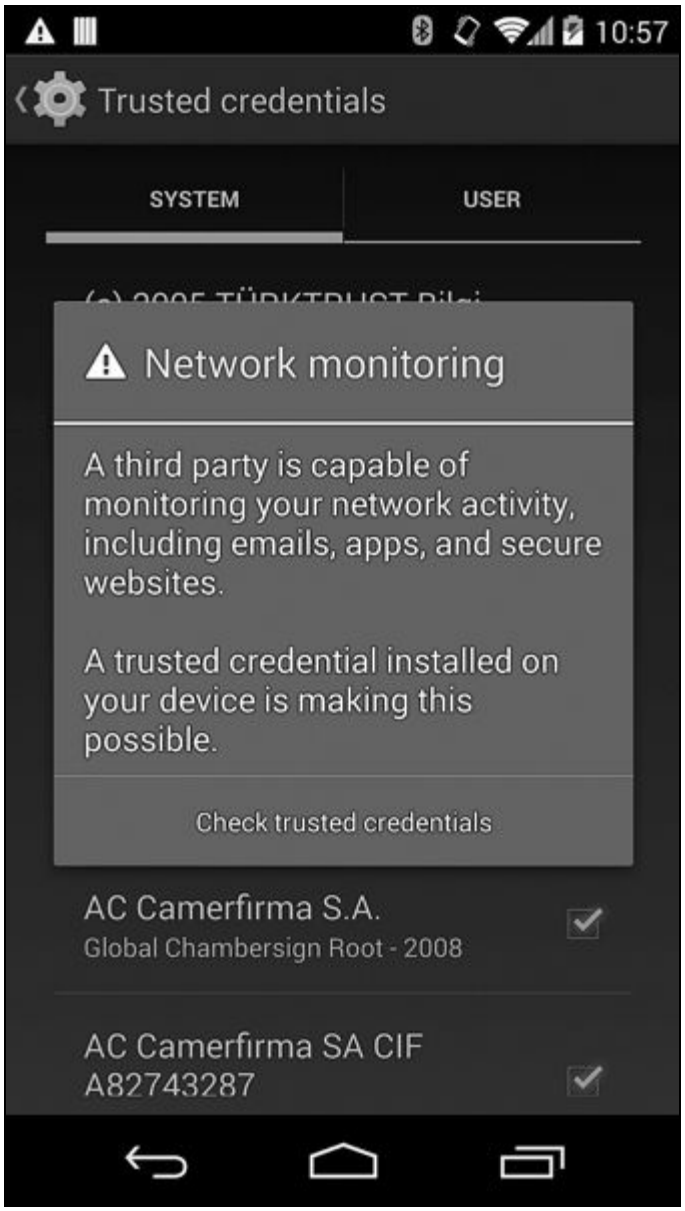

Figura 6.6 Avviso per il monitoraggio della rete in Android 4.4.

API del trust store di sistema

Le applicazioni di terze parti possono chiedere all'utente di importare un certificato necessario nel trust store di sistema utilizzando l'API `KeyChain`, introdotta in Android 4.0 (l'API `KeyChain` è descritta nel Capitolo 7). A partire da Android 4.4, le applicazioni di amministrazione del

dispositivo possono installare certificati CA nel trust store di sistema, senza informare l'utente, se possiedono il permesso di sistema *MANAGE_CA_CERTIFICATES*. L'amministrazione del dispositivo e le API correlate sono presentate nel Capitolo 9.

Dopo aver importato un certificato CA nel trust store di sistema, possiamo utilizzarlo per convalidare i certificati usando l'API JSSE `TrustManager` come mostrato nel Listato 6.9.

Listato 6.9 Inizializzazione di un TrustManager con i trust anchor di sistema e convalida di un certificato.

```
// Catena di certificati che include il certificato dell'entità finale (server)
// ed eventuali autorità emittenti intermedie.
X509Certificate[] chain = { endEntityCert };
TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");(1)
tmf.init((KeyStore) null);(2)
TrustManager[] tms = tmf.getTrustManagers();
X509TrustManager xtm = (X509TrustManager) tms[0];(3)
Log.d(TAG, "checking chain with " + xtm.getClass().getName());
xtm.checkServerTrusted(chain, "RSA");(4)
Log.d(TAG, "chain is valid");
```

In questo esempio recuperiamo per prima cosa la PKIX di sistema (con alias per `X509`) `TrustManagerFactory` (**(1)** nel Listato 6.9), la inizializziamo utilizzando il trust store di sistema, passando `null` al suo metodo `init(KeyStore)` **(2)** e otteniamo la prima implementazione di `TrustManager` per l'algoritmo specificato (solitamente ce n'è solo una, ma controllate nel codice di produzione), dopodiché ne eseguiamo il cast all'interfaccia di convalida `X509TrustManager` specifica per l'algoritmo **(3)**. Infine, passiamo la catena di certificati e l'algoritmo di scambio delle chiavi usato (*RSA*, *DHE_DSS* e così via) al metodo `checkServerTrusted()` **(4)**. Se è possibile creare una catena che porta a un certificato CA attendibile, la convalida riesce e il metodo restituisce un risultato. Se invece uno dei certificati della catena è scaduto o non è valido, oppure se la catena non porta a un trust anchor di sistema, il metodo genera una `jva.security.cert.CertificateException` (o una delle sue sottoclassi). Le connessioni stabilite con `SSLSocket` e `HttpsURLConnection` effettuano automaticamente una convalida simile.

La cosa funziona, ma questo codice presenta un problema: non controlla la revoca. Il `TrustManager` predefinito di Android disattiva esplicitamente la revoca durante la convalida della catena di certificati; pertanto, anche se un certificato ha un'estensione *CRL Distribution Point*

(CDP) che punta a un CRL valido, o se l'URI del responder OCSP è stato incluso nell'estensione *Authority Information Access* (AIA) e il certificato è stato effettivamente revocato, viene tuttora convalidato in Android. Quello che manca è quindi il *controllo della revoca online*, vale a dire la capacità di recuperare, archiviare e aggiornare dinamicamente le informazioni di revoca in base alle necessità, sfruttando le informazioni disponibili nelle estensioni dei certificati.

Blacklisting dei certificati

Invece di usare i controlli della revoca online, Android fa affidamento sul blacklisting dei certificati delle CA e delle entità finali, di cui ci occupiamo in questo paragrafo. Il *blacklisting dei certificati* è il blocco esplicito di alcuni certificati da parte dei verifier, indipendentemente dal loro stato nel repository di PKI. Il blacklisting non è parte della filosofia PKI originale e non è definito negli standard correlati. Per quale motivo è allora necessario a livello pratico?

In un mondo perfetto, una PKI funzionante si occupa di rilasciare, distribuire e revocare i certificati in base alle necessità. Tutto ciò di cui un sistema ha bisogno per verificare le identità di macchine e utenti in precedenza sconosciuti sono alcuni certificati di trust anchor: qualunque certificato di entità finale sarà rilasciato da una delle CA attendibili o da una delle relative CA di rilascio subordinate (sub-CA). Nella pratica esistono però diversi problemi, per la maggior parte legati alla gestione delle chiavi compromesse. I certificati delle entità finali hanno un periodo di validità relativamente breve (di solito un anno), che limita il tempo per cui una chiave compromessa può essere sfruttata; tuttavia, i certificati delle CA hanno una validità estesa (di solito almeno 20 anni) e, visto che le CA sono considerate attendibili in via implicita, la compromissione di una chiave potrebbe rimanere sconosciuta per qualche tempo. Le violazioni recenti delle CA di primo livello hanno dimostrato che la compromissione delle chiavi CA non è un problema teorico e che le conseguenze della violazione di una CA possono essere devastanti.

Gestione delle compromissioni di chiavi CA

Il più grande problema di PKI è il fatto che la revoca dei certificati root non è effettivamente supportata. La maggior parte dei sistemi operativi e dei browser è fornita con un set preconfigurato di certificati CA attendibili (ve ne sono a dozzine!). Quando un certificato CA viene compromesso, esistono due modalità di gestione principali: chiedere agli utenti di rimuoverlo dal trust store o rilasciare un aggiornamento di emergenza che rimuova il certificato in questione. Chiaramente è irrealistico lasciare agli utenti la soluzione del problema, quindi resta la seconda soluzione.

Windows modifica i trust anchor del sistema operativo distribuendo patch con Windows Update, mentre i vendor dei browser rilasciano semplicemente una nuova versione con patch. Tuttavia, anche se un aggiornamento rimuove un certificato CA dal trust store di sistema, un utente può ancora installarlo nuovamente, specialmente se gli viene posto un ultimatum del tipo “o fai così, o non potrai accedere a questo sito”.

Per garantire che i trust anchor rimossi non vengano reintegrati, gli hash delle relative chiavi pubbliche vengono aggiunti a una blacklist e il sistema operativo o il browser li rifiuta anche se sono presenti nel trust store dell’utente. Questa metodologia provoca efficacemente la revoca dei certificati CA (ovviamente nell’ambito del sistema operativo o del browser) e risolve il problema legato all’incapacità di PKI di gestire i trust anchor compromessi. Tuttavia, la soluzione non è ideale perché anche un aggiornamento di emergenza richiede tempo per la preparazione e poiché, dopo il rilascio, non è detto che tutti gli utenti effettuino l’aggiornamento immediatamente (fortunatamente le compromissioni delle CA sono rare e ben pubblicizzate, quindi per ora questa soluzione sembra funzionare). Sono state proposte anche altre strategie, ma la maggior parte di queste non è molto diffusa; alcune delle soluzioni suggerite sono descritte nel paragrafo “Soluzioni radicali” di questo capitolo.

Gestione delle compromissioni di chiavi EE

Se le violazioni delle CA sono piuttosto rare, le compromissioni della chiave dell'entità finale (EE) sono invece più frequenti: si verificano infatti quotidianamente, per esempio a causa della violazione di un server, del furto di un portatile o dello smarrimento di una smart card. Per fortuna, i sistemi PKI moderni sono progettati in tal senso e le CA possono revocare i certificati e pubblicare le informazioni sulla revoca sotto forma di CRL, nonché fornire lo stato di revoca online utilizzando OCSP.

Purtroppo questa soluzione non è perfetta nel mondo reale: il controllo della revoca richiede infatti l'accesso di rete a un computer diverso da quello a cui si sta tentando di connettersi, operazione che presenta un alto tasso di fallimento. Per attenuare il problema, la maggior parte dei browser prova a recuperare le informazioni di revoca aggiornate, ma se tale intervento non riesce per un qualsiasi motivo, l'errore viene semplicemente ignorato (si parla di *soft-fail*), o al massimo viene visualizzata un'indicazione del fatto che le informazioni sulla revoca non sono disponibili.

NOTA

Per affrontare questa problematica, Google Chrome disabilita del tutto i controlli online della revoca (consultate il documento di Adam Langley, *Revocation checking and Chrome's CRL*, febbraio 2012, https://www.imperialviolet.org/2012/02/05/crlsets.html) e ricorre al suo meccanismo di aggiornamento per inviare proattivamente le informazioni di revoca ai browser senza richiedere l'aggiornamento o il riavvio dell'applicazione (i controlli online della revoca possono comunque essere abilitati impostando l'opzione `EnableOnlineRevocationChecks` su `true`; l'impostazione predefinita è `false`). Chrome può quindi disporre di una cache locale aggiornata delle informazioni di revoca, che accelera e rende più affidabile la convalida dei certificati. Questa cache può essere considerata come una blacklist (Chrome la definisce *set CRL*), questa volta basata sulle informazioni pubblicate da ogni CA. Il fatto che sia il fornitore del browser a gestire i dati di revoca per conto dell'utente è piuttosto insolito e non tutti pensano che sia una buona idea, ma finora ha funzionato bene.

Un'alternativa all'invio diretto delle informazioni di revoca come parte degli aggiornamenti del browser è lo *stapling OCSP*, precedentemente noto come *estensione Certificate Status Request* di TLS (fate riferimento a D. Eastlake 3rd, *Transport Layer Security (TLS) Extensions: Extension Definitions*, sezione 8, gennaio 2011, http://tools.ietf.org/html/rfc6066#section-8). Invece di richiedere ai client di generare una richiesta OCSP per il certificato del server, la risposta pertinente è inclusa (*to staple* significa

pinzare, graffare) nell'handshake SSL tramite la risposta dell'estensione Certificate Status Request. Visto che la risposta è firmata dalla CA, il client può considerarla attendibile come se l'avesse recuperata direttamente dal server OCSP della CA. Se il server non include una risposta OCSP nell'handshake SSL, è il client a doverne recuperare una autonomamente. Lo stapling OCSP è supportato da tutti i server HTTP principali, ma il supporto dei browser è tuttora disomogeneo, soprattutto nelle versioni mobile in cui la latenza rappresenta un problema.

Blacklisting dei certificati Android

Come avete appreso nel paragrafo "Trust store di sistema Android 4.x", Android 4.0 ha aggiunto una finestra di gestione e un'API SDK che consentono di inserire e rimuovere trust anchor dal trust store di sistema. Questa scelta non ha tuttavia risolto il problema più grave di PKI: a parte il caso in cui l'utente disabilita manualmente un trust anchor compromesso, è ancora richiesto un aggiornamento del sistema operativo per rimuovere un certificato CA compromesso. Inoltre, visto che Android non esegue i controlli della revoca online durante la convalida delle catene di certificati, non c'è modo di rilevare i certificati delle entità finali compromessi, nemmeno se sono stati revocati.

Per risolvere questo problema, Android 4.1 ha introdotto le *blacklist dei certificati*, che possono essere modificate senza richiedere un aggiornamento del sistema operativo. Ora sono disponibili due blacklist di sistema:

- una blacklist per gli hash a chiave pubblica (per la gestione delle CA compromesse);
- una blacklist per i numeri di serie (per la gestione dei certificati EE compromessi)

Il componente di convalida della catena di certificati esamina questi due elenchi durante la verifica dei siti web o dei certificati utente. Scendiamo un po' nei dettagli dell'implementazione.

Android usa un content provider per memorizzare le impostazioni del sistema operativo in un database di sistema; alcune di queste impostazioni possono essere modificate dalle app di terze parti che possiedono i permessi necessari, altre sono riservate al sistema e possono essere cambiate solo dall'app di sistema *Settings* o da un'altra applicazione di sistema. Le impostazioni riservate al sistema sono dette *impostazioni sicure*. Android 4.1 aggiunge due nuove impostazioni sicure nei seguenti URI:

- `content://settings/secure/pubkey_blacklist`
- `content://settings/secure/serial_blacklist`

Come suggerito dal nome, il primo conserva gli hash a chiave pubblica delle CA compromesse, il secondo un elenco dei numeri di serie dei certificati EE. Inoltre, il server di sistema avvia un componente `CertBlacklister` che registra se stesso come `ContentObserver` per i due URI delle blacklist. Ogni volta che viene scritto un nuovo valore in una delle impostazioni sicure della blacklist, `CertBlacklister` riceve una notifica e scrive il valore in un file su disco. I file comprendono un elenco delimitato da virgole di hash a chiave pubblica con codifica hex o di numeri di serie dei certificati. I file sono i seguenti:

- Blacklist dei certificati: `/data/misc/keychain/pubkey_blacklist.txt`
- Blacklist dei numeri di serie: `/data/misc/keychain/serial_blacklist.txt`

Per quale motivo occorre scrivere i file su disco se sono già disponibili nel database delle impostazioni? Semplicemente perché il componente che usa effettivamente le blacklist è una classe API CertPath Java standard che non sa nulla di Android e dei suoi database di sistema. La classe di convalida per il percorso dei certificati, `PKIXCertPathValidatorSpi`, è parte del provider JCA Bouncy Castle modificato per gestire le blacklist dei certificati, che sono una funzionalità specifica di Android e non sono definite nell'API CertPath standard. L'algoritmo di convalida del certificato PKIX implementato dalla classe è piuttosto complesso, ma l'aggiunta di Android 4.1 è molto semplice.

- Durante la verifica di un certificato EE (leaf), viene verificato se il suo numero di serie è nella blacklist dei numeri di serie. In questo caso, viene restituito lo stesso errore (eccezione) generato quando il certificato è stato revocato.
- Durante la verifica di un certificato CA, viene verificato se l'hash della sua chiave pubblica è nella blacklist delle chiavi pubbliche. In questo caso, viene restituito lo stesso errore generato quando il certificato è stato revocato.

NOTA

L'uso del numero di serie non qualificato per l'indicizzazione dei certificati EE nella blacklist potrebbe causare problemi se due o più certificati di CA diverse usano lo stesso numero di serie. In questo caso, il blacklisting di un solo certificato provocherà l'effettivo inserimento nella blacklist di tutti gli altri certificati con lo stesso numero di serie. In pratica, però, la maggior parte delle CA pubbliche usa numeri di serie lunghi e generati in maniera casuale, quindi la probabilità di conflitti è piuttosto bassa.

Il componente di convalida del percorso del certificato è usato nell'intero sistema, pertanto le blacklist agiscono sulle applicazioni che usano le classi client HTTP, sul browser nativo di Android e su `WebView`. Come abbiamo già detto, la modifica delle blacklist richiede permessi di sistema, quindi solo le app del sistema core possono apportare cambiamenti. Nell'origine AOSP non vi sono app che chiamano queste API, ma un buon candidato per la gestione delle blacklist è il componente dei servizi Google disponibile sui device "Google Experience" (vale a dire i dispositivi su cui è preinstallato il client Play Store). Questo componente gestisce gli account Google e l'accesso ai servizi Google, mettendo a disposizione notifiche push tramite *Google Client Messaging* (GCM). GCM consente le notifiche push in tempo reale avviate dal server, che con ogni probabilità saranno usate per attivare gli aggiornamenti delle blacklist dei certificati.

Riesame del modello di trust PKI

Android ha compiuto diversi passi per aumentare la flessibilità del suo trust store, consentendo le modifiche su richiesta sia dei trust anchor sia delle blacklist dei certificati senza richiedere l'aggiornamento del sistema.

Anche se il blacklisting dei certificati rende Android più resiliente ad alcuni attacchi e vulnerabilità legati a PKI, non risolve tutti i problemi relativi all'uso di certificati rilasciati da CA pubbliche. Vediamo quindi alcuni di questi problemi e le relative soluzioni proposte. La discussione su PKI e SSL si concluderà poi con una descrizione dell'implementazione Android di una di queste soluzioni: il pinning dei certificati.

Problemi di trust nella PKI odierna

Nell'improbabile caso che non ne abbiate mai sentito parlare, negli ultimi anni l'affidabilità del modello di CA pubbliche esistente è stata gravemente compromessa. Il problema era nell'aria da anni, ma le recenti violazioni della sicurezza delle CA di alto profilo lo hanno portato in primo piano. Gli hacker hanno rilasciato certificati per un'ampia varietà di siti, tra cui i server Windows Update e Gmail; anche se non sono stati usati tutti (o perlomeno non tutti sono stati rilevati) in attacchi reali, gli incidenti hanno dimostrato l'elevata dipendenza dai certificati della tecnologia Internet attuale.

I certificati fraudolenti possono essere usati per qualunque operazione, dall'installazione di malware allo spionaggio delle comunicazioni nella Rete, inducendo gli utenti a pensare che il canale in uso sia sicuro o che l'eseguibile installato sia affidabile. Purtroppo, una maggiore sicurezza delle CA non rappresenta una soluzione, perché le CA principali hanno emesso volontariamente centinaia di certificati per nomi non qualificati come *localhost*, *webmail* ed *exchange* (http://bit.ly/1nq3drm). I certificati rilasciati a nomi host non qualificati possono essere utilizzati per lanciare un attacco MITM contro i client che accedono a server interni usando nomi non qualificati, facilitando così l'ascolto del traffico aziendale interno. Naturalmente, esiste anche la questione della creazione di certificati imposti, in cui un'agenzia governativa può imporre a una CA di rilasciare un falso certificato da utilizzare per l'intercettazione del traffico protetto.

Chiaramente, il sistema PKI attuale, ampiamente basato su un set preselezionato di CA affidabili (i cui certificati sono preinstallati come trust anchor), è problematico, ma quali sono i problemi veri e propri? Esistono diverse opinioni su questo argomento, ma per cominciare esistono troppe CA pubbliche. Il progetto SSL Observatory di Electronic Frontier Foundation ha dimostrato che i browser principali considerano attendibili più di 650 CA pubbliche (Electronic Frontier Foundation, *The EFF SSL Observatory*, https://www.eff.org/observatory). Le versioni di Android più recenti sono fornite con oltre 100 certificati CA truster e, fino alla versione 4.0, l'unico modo per rimuovere un certificato trusted era tramite un aggiornamento del sistema operativo avviato da un vendor.

Inoltre, generalmente non vi sono restrizioni tecniche sui certificati che le CA possono rilasciare. Come dimostrato dagli attacchi Comodo e DigiNotar, nonché dall'incidente alla CA intermedia ANNSI (*Agence nationale de la sécurité des systèmes d'information*), chiunque può rilasciare un certificato per `*.google.com` (i vincoli sui nomi non si applicano alle CA root e in realtà non funzionano nemmeno per le CA pubbliche). Inoltre, poiché le CA non pubblicizzano i certificati che hanno rilasciato, non c'è modo per gli operatori dei siti (in questo caso Google) di sapere se qualcuno rilascia un nuovo certificato, magari fraudolento, per uno dei loro siti e intraprendere le azioni appropriate (gli standard di trasparenza dei certificati mirano a risolvere questo problema; consultate B. Laurie, A. Langley ed E. Kasper, *Certificate Transparency*, giugno 2013, http://tools.ietf.org/html/rfc6962). In breve, con il sistema attuale, se uno dei trust anchor predefiniti viene compromesso, un hacker può rilasciare un certificato per qualunque sito senza che il proprietario del sito o gli utenti dello stesso se ne accorgano.

Soluzioni radicali

Le soluzioni proposte sono sia radicali (come l'eliminazione dell'intera idea delle PKI, sostituendole con qualcosa di nuovo e migliore come DNSSEC), sia moderate (usare l'infrastruttura corrente senza considerare implicitamente attendibili le CA), sia evolutive (mantenere la

compatibilità con il sistema attuale ma estenderlo in modo da limitare i danni della compromissione di una CA).

Purtroppo, DNSSEC non è ancora distribuito a livello universale, nonostante i domini TLD delle chiavi siano già stati firmati; inoltre, è per natura gerarchico (con i domini di primo livello per nazione controllati dai rispettivi paesi) e più rigido di PKI, quindi non è particolarmente adatto. Tuttavia, sono in corso numerose ricerche atte a migliorare la situazione corrente di PKI e a definire altre soluzioni radicali percorribili.

Per quanto riguarda le proposte moderate, il modello proposto è SSH (a volte detto *Trust on First Use*, o TOFU), in cui nessun sito o CA è considerato inizialmente attendibile e spetta agli utenti decidere di quali siti fidarsi durante il primo accesso. A differenza di SSH, però, il numero di siti a cui si accede direttamente o indirettamente (tramite CDN, contenuti incorporati e così via) è virtualmente illimitato, pertanto il trusting gestito dagli utenti risulta piuttosto irrealizzabile.

Convergence e trust agility

Simile, ma decisamente più pratico, è Convergence (http://convergence.io), un sistema basato sull'idea della *trust agility*, ovvero della capacità di scegliere facilmente di chi fidarsi e di poter rivedere tale decisione in qualsiasi momento. Abolisce pertanto il set di trust anchor preselezionato dal browser (o dal sistema operativo) e riconosce che non è possibile affidarsi agli utenti affinché prendano in maniera indipendente decisioni sull'affidabilità di tutti i siti che visitano. Le decisioni sul trust sono delegate a un set di notary che possono garantire per un sito confermando che il certificato ricevuto da un sito è quello che hanno esaminato in precedenza. Se più notary segnalano che lo stesso certificato è corretto, gli utenti possono essere ragionevolmente certi della sua autenticità e di conseguenza della sua affidabilità.

Convergence non è uno standard formale, ma ne è stata rilasciata un'implementazione funzionante, comprendente un plug-in Firefox (client) e un software notary lato server. Anche se il sistema è promettente, il numero di notary disponibili è attualmente limitato e

Google ha dichiarato pubblicamente che non lo aggiungerà a Chrome. Inoltre, al momento non può essere implementato come estensione del browser, perché Chrome non consente alle estensioni di terze parti di sostituire il modulo di convalida dei certificati predefinito.

Pinning dei certificati

Tutto questo ci porta alle attuali soluzioni evolutive, che sono state distribuite a un'ampia base di utenti sfruttando il browser Chrome. Una è il blacklisting dei certificati, di cui abbiamo già parlato; l'altra è il pinning dei certificati.

Il *pinning dei certificati* (o, per la precisione, il *pinning della chiave pubblica*) sceglie la via contraria alla metodologia delle blacklist: inserisce infatti in una whitelist le chiavi considerate attendibili per la firma dei certificati di un sito specifico. Il pinning è stato introdotto nella versione 13 di Google Chrome per limitare le CA che possono rilasciare certificati per le proprietà di Google e viene implementato mantenendo un elenco delle chiavi pubbliche considerate attendibili per il rilascio di certificati per uno specifico nome DNS. L'elenco viene consultato nella fase di convalida della catena di certificati di un host e, se la catena non include almeno una delle chiavi contenute nella whitelist, la convalida ha esito negativo. In pratica, il browser mantiene un elenco di hash SHA-1 del campo `SubjectPublicKeyInfo` (SPKI) dei certificati trusted. Il pinning delle chiavi pubbliche, anziché dei certificati veri e propri, consente di aggiornare i certificati host senza interrompere la convalida e richiedere l'aggiornamento delle informazioni di pinning.

Tuttavia, un elenco di pinning inserito direttamente nel codice non è scalabile; pertanto, sono stati proposti un paio di nuovi standard Internet per risolvere questo problema di scalabilità: *Public Key Pinning Extension for HTTP* (PKPE; C. Evans, C. Palmer e R. Sleevi, *Public Key Pinning Extension for HTTP*, 7 agosto 2014, http://tools.ietf.org/html/draft-ietf-websec-key-pinning-20) di Google e *Trust Assertions for Certificate Keys* (TACK, M. Marlinspike, *Trust Assertions for Certificate Keys*, 7 gennaio 2013, http://tack.io/draft.html) di Moxie Marlinspike. Il primo è più semplice

e propone un nuovo header HTTP (`Public-Key-Pin`, o *PKP*) contenente le informazioni di pinning sul certificato di un host. Il valore dell'header può includere hash a chiave pubblica, la durata del pinning e un flag che specifica se il pinning è applicabile anche ai sottodomini dell'host corrente. Le informazioni di pinning (o *pin*) vengono archiviate nella cache dal browser e utilizzate fino alla scadenza per prendere decisioni di trust. I pin devono essere forniti su una connessione sicura (SSL) e la prima connessione che include un header PKP è considerata implicitamente attendibile (o, facoltativamente, può essere convalidata rispetto ai pin integrati nel client). Il protocollo supporta anche un endpoint per la segnalazione delle convalide non riuscite tramite la direttiva `report-uri` e consente una modalità detta di *non-enforcing* (specificata con l'header `Public-Key-Pins-Report-Only`), in cui vengono segnalati gli errori di convalida pur consentendo comunque le connessioni. In questo modo è possibile segnalare agli amministratori degli host i possibili attacchi MITM contro i loro siti, affinché prendano le contromisure appropriate.

La proposta TACK, d'altro canto, è più complessa e definisce una nuova estensione TLS (detta anche TACK) che firma le informazioni di pinning con una *chiave TACK*. Le connessioni TLS a un nome host con pinning richiedono che il server presenti un "TACK" contenente la chiave con pinning e una firma corrispondente sulla chiave pubblica del server TLS. Di conseguenza, sia lo scambio di informazioni sul pinning sia la convalida sono eseguiti al livello TLS. PKPE usa invece il livello HTTP (sopra TLS) per inviare le informazioni di pinning ai client, ma richiede che la convalida sia eseguita al livello TLS, interrompendo la connessione se la convalida rispetto ai pin ha esito negativo.

Ora che sappiamo come funziona il pinning, vediamo come è implementato in Android.

Pinning dei certificati in Android

Il pinning è uno dei numerosi miglioramenti alla sicurezza introdotti in Android 4.2. Il sistema operativo non è fornito con alcun pin predefinito,

ma li legge da un file nella directory */data/misc/keychain/* (dove sono archiviati i certificati aggiunti dall'utente e le blacklist). Il file è chiamato semplicemente `pins` ed è nel seguente formato (Listato 6.10).

Listato 6.10 Formato del file di sistema pins.

```
hostname=enforcing|SPKI SHA512 hash, SPKI SHA512 hash,...
```

Qui `enforcing` è `true` o `false` ed è seguito da un elenco di hash `SPKI SHA512` separati da virgole. Non esiste alcun periodo di validità, pertanto i pin sono validi fino all'eliminazione. Il file è usato non solo dal browser, ma a livello di sistema in virtù dell'integrazione del pinning in *libcore*. In pratica, significa che l'implementazione di sistema predefinita (nonché unica) di `X509TrustManager` (`TrustManagerImpl`) consulta l'elenco dei pin durante la convalida delle catene di certificati.

Preparatevi però a un colpo di scena: il metodo standard `checkServerTrusted()` non consulta l'elenco dei pin. Di conseguenza, qualunque libreria legacy che non conosce il pinning dei certificati continuerà a funzionare esattamente come prima, indipendentemente dal contenuto dell'elenco dei pin. Questa scelta è stata probabilmente fatta per ragioni di compatibilità, ma è necessario tenerla ben presente: l'uso di Android 4.2 o versioni successive non garantisce automaticamente il vantaggio dei pin dei certificati a livello di sistema. La funzionalità del pinning viene esposta a librerie e app di terze parti con la nuova classe SDK `X509TrustManagerExtensions`. La classe ha un unico metodo, `checkServerTrusted()` (la firma completa è mostrata nel Listato 6.11), che restituisce una catena convalidata in caso di riuscita e che genera una `CertificateException` se la convalida non riesce.

Listato 6.11 Metodo di convalida dei certificati X509TrustManagerExtensions.

```
List<X509Certificate> checkServerTrusted(X509Certificate[] chain, String authType, String host)
```

L'ultimo parametro, `host`, è usato dall'implementazione sottostante (`TrustManagerImpl`) per cercare i pin corrispondenti nell'elenco dei pin: se ne viene trovato uno, le chiavi pubbliche nella catena vengono convalidate rispetto agli hash nella voce di pinning per tale host. Se non vi sono

corrispondenze, la convalida ha esito negativo e viene generata una `CertificateException`.

Ma allora quale parte del sistema usa la nuova funzionalità del pinning? Le implementazioni dell'engine SSL predefinito (JSSE provider), nello specifico l'handshake client (`ClientHandshakeImpl`), e del socket SSL (`OpenSSLSocketImpl`) verificano il loro `X509TrustManager` sottostante e, qualora supporti il pinning, eseguono un'ulteriore convalida rispetto all'elenco dei pin. Se la convalida non riesce, la connessione non viene stabilita e la convalida dei pin viene implementata al livello TLS come richiesto dagli standard visti nei paragrafi precedenti.

Il file `pins` non viene scritto direttamente dal sistema operativo. I suoi aggiornamenti sono attivati da un broadcast (`android.intent.action.UPDATE_PINS`) che contiene i nuovi pin nei suoi extra; gli extra contengono il percorso del nuovo file `pins`, la sua nuova versione (salvata in */data/misc/keychain/metadata/version/*), un hash dei pin correnti e una firma *SHA512withRSA* sopra tutto il resto. Il receiver del broadcast (`CertPinInstallReceiver`) verifica la versione, l'hash e la firma e, se sono validi, sostituisce il file `pins` corrente con il nuovo contenuto (la stessa procedura è usata per aggiornare l'elenco dei numeri SMS premium). La firma dei nuovi pin garantisce che possano essere aggiornati solo da chi controlla la chiave di firma privata. La chiave pubblica corrispondente usata per la convalida è archiviata come impostazione di sistema protetta nella chiave *config_update_certificate* (solitamente nella tabella protetta di `/data/data/com.android.providers.settings/databases/settings.db`). Quando è stato scritto questo libro, il file `pins` sui dispositivi Nexus conteneva oltre 40 voci di pinning, la maggior parte relativa alle proprietà di Google come i server Gmail, YouTube e Play Store.

Riepilogo

Android si basa su API Java standard come JSSE e CertPath per implementare le connessioni SSL e i meccanismi di autenticazione richiesti. La maggior parte delle funzionalità socket sicure è fornita dall'implementazione JSSE ampiamente nativa e basata su OpenSSL, mentre la convalida dei certificati e la gestione del trust store sono implementate in Java. Android fornisce un trust store di sistema condiviso che può essere gestito da *Settings* o dall'API `KeyStore`. Tutte le applicazioni che usano SSL o le API di convalida dei certificati ereditano i trust anchor di sistema, a meno che non venga specificato esplicitamente un trust store specifico per l'app. La convalida dei certificati in Android non usa il controllo online della revoca, ma si affida alla blacklist dei certificati di sistema per rilevare i certificati CA o EE compromessi. Infine, le versioni recenti di Android supportano il pinning dei certificati a livello di sistema per vincolare il set di certificati che possono rilasciare un certificato del server per un particolare host.

Archiviazione delle credenziali

Nel capitolo precedente abbiamo introdotto PKI e le sfide relative alla gestione dei trust e dell'affidabilità. Anche se PKI viene usata prevalentemente per autenticare l'entità a cui vi connettete (*autenticazione del server*), è usata anche per autenticare voi stessi presso queste entità (*autenticazione del client*). L'autenticazione del client è utilizzata soprattutto negli ambienti aziendali, dove è impiegata per ogni operazione, dal logon desktop all'accesso remoto ai server aziendali. L'autenticazione del client basata su PKI richiede al client di dimostrare di essere in possesso di una chiave di autenticazione (in genere una chiave privata RSA) eseguendo particolari operazioni di crittografia che il server può verificare in maniera indipendente. Di conseguenza, la sicurezza dell'autenticazione client si affida quasi notevolmente alla protezione delle chiavi di autenticazione dall'uso non autorizzato.

La maggior parte dei sistemi operativi offre un servizio di sistema utilizzabile dalle applicazioni per archiviare e accedere in sicurezza alle chiavi di autenticazione senza dover implementare autonomamente la protezione delle chiavi. Android dispone di un servizio simile dalla versione 1.6, servizio che è stato notevolmente migliorato a partire da Android 4.0.

L'archivio (o *store*) delle credenziali di Android può essere usato per memorizzare le credenziali delle funzionalità integrate, quali la connettività Wi-Fi e VPN, nonché delle app di terze parti. Le app possono accedere all'archivio delle credenziali tramite le API SDK standard e possono usarlo per gestire in sicurezza le loro chiavi. Le versioni recenti di Android dispongono di un archivio hardware delle chiavi che fornisce una protezione avanzata. Questo capitolo esamina

l’architettura e l’implementazione dell’archivio delle credenziali di Android e presenta le API pubbliche che fornisce.

Credenziali EAP per VPN e Wi-Fi

Le *reti private virtuali* (VPN) sono il mezzo preferito per offrire l'accesso remoto ai servizi aziendali privati. Parleremo nei dettagli delle VPN e delle tecnologie correlate nel Capitolo 9; in ogni caso, una VPN consente a un client remoto di unirsi a una rete privata creando un tunnel crittografato tra il client stesso e un endpoint pubblico del tunnel. Le implementazioni delle VPN cambiano in base all'uso della tecnologia di tunneling, ma tutte devono autenticare il client prima di stabilire una connessione sicura. Anche se alcune VPN usano una chiave condivisa o una password per l'autenticazione, le soluzioni aziendali spesso fanno affidamento all'autenticazione del client basata su PKI.

Extensible Authentication Protocol (EAP) è un framework di autenticazione utilizzato spesso nelle rete wireless e nelle connessioni *point-to-point* (P2P). EAP è descritto nei dettagli nel Capitolo 9. Analogamente alle VPN, EAP può usare molti metodi di autenticazione, ma quello preferito negli ambienti aziendali è l'*EAP-Transport Layer Security* (EAP-TLS), soprattutto quando è già stata distribuita una PKI aziendale.

Certificati e chiavi di autenticazione

Nel caso delle VPN basate sia su EAP-TLS sia su PKI, i client possiedono una chiave di autenticazione e viene loro rilasciato un certificato corrispondente, spesso dall'autorità di certificazione (CA) dell'azienda. Le chiavi sono a volte salvate in un dispositivo portatile resistente alle manomissioni, per esempio una smart card o un token USB: in questo modo la sicurezza è notevolmente superiore perché le chiavi non possono essere esportate o estratte dal dispositivo e di conseguenza l'autenticazione richiede sia il possesso fisico del token sia la conoscenza della passphrase o del PIN associati.

Quando la policy di sicurezza consente l'uso di chiavi di autenticazione che non sono protette da un dispositivo hardware, le chiavi e i certificati

associati sono solitamente salvati nel formato di file standard PKCS#12. Le chiavi private salvate nei file PKCS#12 sono crittografate con una chiave simmetrica derivata da una password fornita dall'utente, quindi la loro estrazione richiede di conoscere la password. Alcune applicazioni usano i file PKCS#12 come contenitori sicuri ed estraggono chiavi e certificati in memoria solo quando richiesto; di solito, però sono importati in un archivio delle credenziali di sistema o dell'applicazione prima dell'uso. Questa è la modalità di funzionamento anche di Android.

L'implementazione per l'utente dell'importazione delle credenziali in Android è piuttosto semplice: per importare una chiave di autenticazione e i relativi certificati gli utenti copiano i loro file PKCS#12 (e, se necessario, i certificati CA correlati) sulla memoria esterna del dispositivo (spesso una scheda SD) e selezionano *Install from storage* nella schermata *Security* delle impostazioni di sistema. Android cerca i file corrispondenti (con estensione `.pfx` o `.p12`) nella directory root della memoria esterna e mostra una finestra di importazione (Figura 7.1). Se viene fornita la password corretta, le chiavi vengono estratte dal file PKCS#12 e importate nell'archivio delle credenziali di sistema.

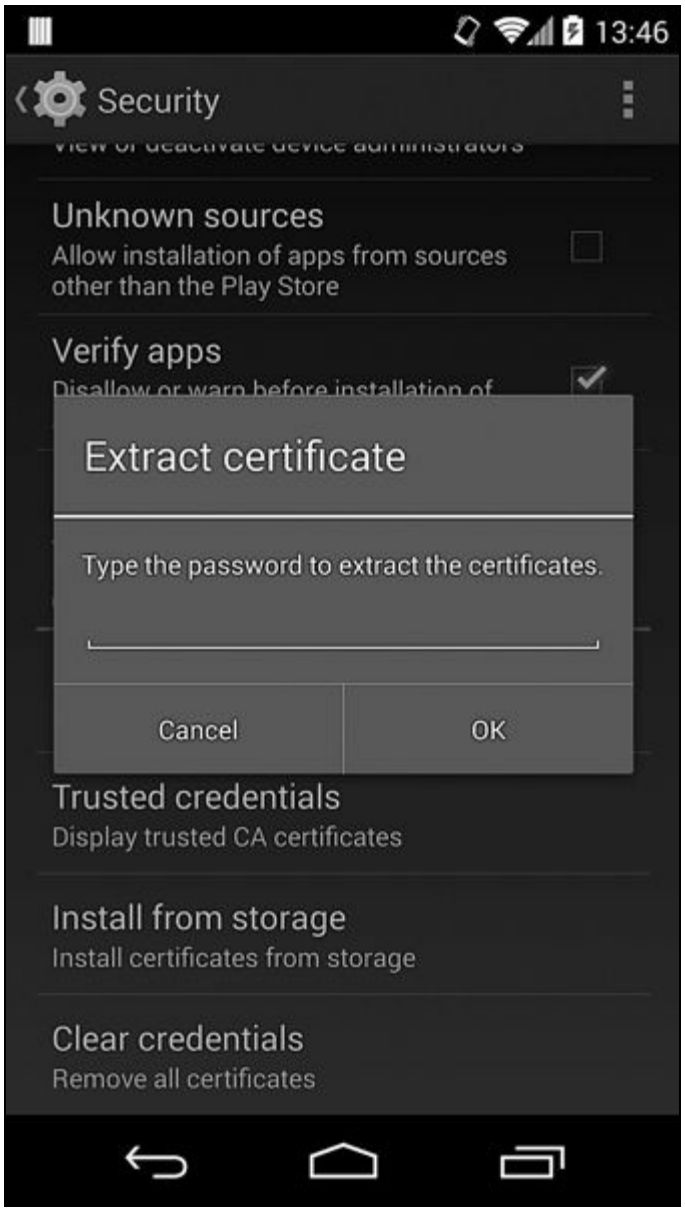

Figura 7.1 Finestra di dialogo della password per il file PKCS#12.

Archivio delle credenziali di sistema

L'archivio delle credenziali è un servizio di sistema che crittografa le credenziali importate prima di salvarle su disco. La chiave di crittografia viene derivata da una password fornita dall'utente: una password dedicata alla protezione dell'archivio nelle versioni precedenti alla 4.0 oppure un pattern di sblocco, un PIN o una password nelle versioni di Android successive alla 4.0. Inoltre, il servizio di sistema dell'archivio regola l'accesso alle credenziali archiviate e garantisce che vi possano accedere solo le app a cui è stato esplicitamente permesso.

L’archivio delle credenziali originale è stato introdotto in Android 1.6 e serviva unicamente per conservare le credenziali EAP per VPN e Wi-Fi. Alle chiavi e ai certificati archiviati poteva accedere solo il sistema e non le app di terze parti; inoltre, per importare le credenziali era possibile utilizzare soltanto la finestra delle impostazioni di sistema descritta nel paragrafo precedente, visto che non erano disponibili API pubbliche per la gestione dell’archivio.

Le API per l’accesso all’archivio delle credenziali di sistema è stato introdotto per la prima volta in Android 4.0. Successivamente l’archivio delle credenziali di sistema è stato esteso per supportare l’archiviazione hardware delle credenziali e per offrire, oltre alle chiavi di sistema condivise, anche le chiavi private delle app. La Tabella 7.1 riepiloga i principali miglioramenti dell’archivio delle credenziali aggiunti in ogni versione di Android. Questi miglioramenti, e le API correlate, saranno presentati nei paragrafi successivi.

Tabella 7.1 Miglioramenti progressivi delle funzionalità dell’archivio delle credenziali.

Versione di Android	Livello API	Modifiche all’archivio delle credenziali
1.6	4	Aggiunta dell’archivio delle credenziali per VPN e Wi-Fi.
4.0	14	Aggiunta dell’API pubblica per l’archivio delle credenziali (API `KeyChain`).
4.1	16	Aggiunta della capacità di generare e usare le chiavi senza esportarle. Introduzione del modulo HAL keymaster e del supporto iniziale per l’archiviazione delle chiavi RSA sull’hardware.
4.3	18	Aggiunta del supporto per la generazione e l’accesso alle chiavi private delle app con il provider JCA *AndroidKeyStore* e delle API per verificare se il dispositivo supporta l’archiviazione hardware delle chiavi RSA.
4.4	19	Aggiunta del supporto ECDSA e DSA al provider JCA *AndroidKeyStore*.

Implementazioni dell'archivio delle credenziali

Ora sappiamo che Android può crittografare le credenziali importate e gestire l'accesso alle stesse; vediamo come viene implementato questo comportamento.

Servizio keystore

La gestione dell'archivio delle credenziali in Android era originariamente implementata da un singolo daemon nativo chiamato `keystore`. Inizialmente il suo compito era quello di conservare blob arbitrari in forma crittografata e di verificare la password dell'archivio delle credenziali; a seguito dell'evoluzione di Android è stato poi esteso con nuove funzionalità. Offriva ai suoi client un'interfaccia basata su socket locali e ogni client aveva il compito di gestire il proprio stato e le proprie connessioni ai socket. Il daemon `keystore` è stato sostituito da un servizio Binder centralizzato in Android 4.3 per offrire una migliore integrazione con gli altri servizi del framework e per facilitare l'estensione. Vediamone il funzionamento. Il servizio `keystore` è definito in `init.rc`, come mostrato nel Listato 7.1.

Listato 7.1 Definizione del servizio keystore in init.rc.

```
service keystore /system/bin/keystore /data/misc/keystore
    class main
    user keystore
    group keystore drmrpc
```

Come potete vedere, `keystore` viene eseguito da un utente `keystore` dedicato e memorizza i suoi file in */data/misc/keystore/*. Esaminiamo per prima cosa questa directory. Se state usando un dispositivo monoutente, per esempio uno smartphone, troverete una sola directory *user_0/* nella directory *keystore/* (Listato 7.2, con timestamp omessi), mentre sui dispositivi multiutente dovreste trovare una directory per ogni utente di Android.

Listato 7.2 Esempio di contenuto della directory keystore su un dispositivo monoutente.

```
# ls -la /data/misc/keystore/user_0
-rw------- keystore keystore       84 .masterkey
-rw------- keystore keystore      980 1000_CACERT_cacert
-rw------- keystore keystore      756 1000_USRCERT_test
-rw------- keystore keystore      884 1000_USRPKEY_test
-rw------- keystore keystore      724 10019_USRCERT_myKey
-rw------- keystore keystore      724 10019_USRCERT_myKey1
```

In questo esempio, ogni nome file è costituito dall'UID dell'app che lo ha creato (`1000` è `system`), dal tipo di voce (certificato CA, certificato utente o chiave privata) e dal nome della chiave (alias); gli elementi sono collegati con un underscore. A partire da Android 4.3 sono supportate anche le chiavi di sistema e private delle app, e l'UID riflette lo user ID di Android, oltre che l'app ID. Sui dispositivi multiutente lo user ID è `UID / 100000`, come già spiegato nel Capitolo 4.

Oltre ai key blob di sistema o delle app, esiste anche un singolo file single `.masterkey` di cui parleremo tra poco. Quando un'app che possiede chiavi gestite dall'archivio viene disinstallata per un utente, vengono eliminate solo le chiavi create da tale utente. Se un'app viene invece completamente rimossa dal sistema, le sue chiavi vengono eliminate per tutti gli utenti. Poiché l'accesso alle chiavi è legato all'app ID, questa funzione impedisce a un'app che sembra avere lo stesso UID di accedere alle chiavi di un'app disinstallata. Anche il reset del keystore, che elimina sia i file delle chiavi sia la chiave master, agisce solo sull'utente corrente.

Nell'implementazione software predefinita, questi file dispongono dei contenuti elencati di seguito (i contenuti possono essere diversi per le implementazioni hardware; invece del materiale della chiave crittografato, spesso viene conservato solo un riferimento agli oggetti chiave gestiti dall'hardware).

- La chiave master (memorizzata in `.masterkey`) è crittografata con una chiave AES a 128 bit derivata dalla password di sblocco dello schermo applicando la funzione di derivazione della chiave PBKDF2 con 8192 iterazioni e un salt di 128 bit generato in maniera casuale. Il salt è memorizzato nell'header info del file `.masterkey`.
- Tutti gli altri file contengono key blob: un *key blob* (oggetto binario di grandi dimensioni) contiene una chiave serializzata e facoltativamente crittografata insieme ad alcuni dati che descrivono

la chiave (metadati). Ogni key blob del keystore contiene un header metadata, il vettore di inizializzazione (IV) usato per la crittografia e una concatenazione del valore hash MD5 dei dati con i dati stessi, crittografata con la chiave master AES a 128 bit nella modalità CBC. In maniera concisa: `metadata || Enc(MD5(data) || data)`.

In pratica, questa architettura implica che il keystore Android è piuttosto sicuro per una soluzione software: anche se accedete a un dispositivo rooted e siete riusciti a estrarre i key blob, vi serve ancora la password del keystore per recuperare la chiave master. Anche provando password diverse nel tentativo di decodificare la chiave master avreste bisogno di almeno 8192 iterazioni, proibitivamente costose. Inoltre, poiché la funzione di derivazione riceve un seed casuale di 128 bit, le tabelle delle password precalcolate sono inutilizzabili. Tuttavia, il meccanismo di integrità basato su MD5 utilizzato non impiega un algoritmo MAC (*Message Authentication Code*) standard come HMAC, ed è semplicemente un residuo dell'implementazione originale, conservato per ragioni di compatibilità con le versioni precedenti, ma che sarà probabilmente sostituito in una versione futura.

Tipi e versioni di key blob

A partire da Android 4.1 ai key blob sono stati aggiunti due campi: `version` e `type`. La versione corrente (per Android 4.4) è 2; i key blob vengono aggiornati automaticamente all'ultima versione quando un'applicazione vi accede per la prima volta. Attualmente sono definiti i seguenti tipi di chiave:

- `TYPE_ANY`
- `TYPE_GENERIC`
- `TYPE_MASTER_KEY`
- `TYPE_KEY_PAIR`

`TYPE_ANY` è un tipo di metachiave che corrisponde a qualunque tipo di chiave. `TYPE_GENERIC` è usato per i key blob salvati utilizzando l'interfaccia

get/put originale, che memorizza dati binari arbitrari, mentre `TYPE_MASTER_KEY` è usato naturalmente solo per la chiave master del keystore. Il tipo `TYPE_KEY_PAIR` è usato per i key blob creati utilizzando le operazioni `generate_keypair` e `import_keypair`, introdotte in Android 4.1. Queste operazioni saranno descritte nel paragrafo "Implementazione del modulo keymaster e del servizio keystore".

Android 4.3 è la prima versione a usare i campi `flags` dei key blob, che consentono di distinguere i key blob crittografati (predefiniti) da quelli non crittografati. I key blob protetti da un'implementazione hardware (disponibile su alcuni dispositivi) sono salvati senza una crittografia supplementare.

Restrizioni di accesso

I key blob sono di proprietà dell'utente `keystore`, pertanto su un dispositivo normale (non rooted) è necessario passare per il servizio `keystore` per accedervi. Il servizio applica le seguenti restrizioni di accesso.

- L'utente `root` non può bloccare o sbloccare il keystore, ma può accedere alla chiavi di sistema.
- L'utente `system` può eseguire la maggior parte delle operazioni di gestione del keystore (inizializzazione, reset e così via) e memorizzare le chiavi; tuttavia, non può usare o recuperare le chiavi di altri utenti.
- Gli utenti non di sistema possono inserire, eliminare e accedere alle chiavi, ma possono vedere solo le loro chiavi.

Ora che conosciamo lo scopo di `keystore`, esaminiamone l'implementazione.

Implementazione del modulo keymaster e del servizio keystore

Anche se l'implementazione originale basata su daemon includeva la gestione e la crittografia dei key blob in un singolo binario, Android 4.1 ha introdotto un nuovo modulo di sistema *Hardware Abstraction Layer*

(HAL) *keymaster* responsabile della generazione di chiavi asimmetriche e della firma/verifica dei dati senza che sia necessario esportare prima le chiavi.

Il modulo *keymaster* è pensato per separare il servizio `keystore` dall'implementazione delle operazioni sulla chiave asimmetrica sottostante e per facilitare l'integrazione delle implementazioni hardware specifiche per ogni dispositivo. Un'implementazione tipica userebbe una libreria fornita dal vendor per comunicare con l'hardware abilitato alla crittografia e fornirebbe una libreria HAL che faccia da "collante", a cui il daemon `keystore` può collegarsi.

Android è fornito inoltre con un modulo predefinito *softkeymaster* che esegue tutte le operazioni sulle chiavi nel solo software (usando la libreria OpenSSL di sistema). Questo modulo è usato sull'emulatore ed è incluso nei dispositivi privi dell'hardware di crittografia dedicato. La dimensione delle chiavi generate era inizialmente fissa a 2048 bit; erano inoltre supportate solo le chiavi RSA. Android 4.4 ha aggiunto il supporto per la scelta delle dimensioni della chiave, nonché degli algoritmi *Digital Signature Algorithm* (DSA) ed *Elliptic Curve DSA* (ECDSA) e delle rispettive chiavi.

Oggi il modulo *softkeymaster* predefinito supporta chiavi RSA e DSA con dimensioni comprese tra 512 e 8192 bit. Se la dimensione della chiave non è specificata esplicitamente, vengono usati 1024 bit per le chiavi DSA e 2048 bit per le chiavi RSA. Per le chiavi EC, la dimensione è associata a una curva standard basata sul rispettivo campo delle dimensioni. Per esempio, se viene specificato 384 come dimensione della chiave, viene utilizzata la curva *secp384r1* per generare le chiavi. Attualmente sono supportate le seguenti curve standard: *prime192v1*, *secp224r1*, *prime256v1*, *secp384r1* e *secp521r1*. Le chiavi per ognuno degli algoritmi supportati possono essere importate dopo la conversione nel formato standard PKCS#8.

L'interfaccia del modulo HAL è inserita in *hardware/keymaster.h* e definisce le operazioni elencate di seguito.

- `generate_keypair`

- `import_keypair`
- `sign_data`
- `verify_data`
- `get_keypair_public`
- `delete_keypair`
- `delete_all`

Tutte le operazioni sulle chiavi asimmetriche esposte dal servizio `keystore` sono implementate chiamando il modulo di sistema *keymaster*. Pertanto, se il modulo HAL *keymaster* è supportato da un dispositivo di crittografia hardware, tutte le API e i comandi di livello superiore che usano l'interfaccia di `keystore` ricorreranno automaticamente alla crittografia hardware. A parte le operazioni sulle chiavi asimmetriche, tutte le altre operazioni sull'archivio delle credenziali sono implementate dal servizio di sistema `keystore` e non dipendono dai moduli HAL. Il servizio registra se stesso nel `ServiceManager` di Android con il nome *android.security.keystore* e viene avviato in fase di boot. A differenza della maggior parte dei servizi Android, è implementato in C++ e la sua implementazione risiede in *system/security/keystore/*.

Implementazione con supporto hardware di Nexus 4

Per offrire un punto di vista sull'intera idea del "supporto hardware", vediamone brevemente l'implementazione in Nexus 4. Nexus 4 si basa sul SoC (*System on a Chip*) Qualcomm Snapdragon S4 Pro APQ8064. Analogamente ai più recenti SoC ARM, è abilitato per TrustZone e viene utilizzato per l'implementazione di *Qualcomm Secure Execution Environment* (QSEE).

La tecnologia TrustZone di ARM offre due processori virtuali con controllo di accesso basato sull'hardware, che consentono di partizionare un sistema SoC in due "ambienti" virtuali: l'*ambiente Secure* per il sottosistema di protezione e l'*ambiente Normal* per tutto il resto. Le applicazioni in esecuzione nell'ambiente Secure sono dette *applicazioni*

trusted e sono accessibili dalle applicazioni dell'ambiente Normal (in cui vengono eseguiti il sistema operativo Android e le app) solo attraverso un'interfaccia limitata esposta esplicitamente. Nella Figura 7.2 è mostrata la tipica configurazione software di un sistema TrustZone.

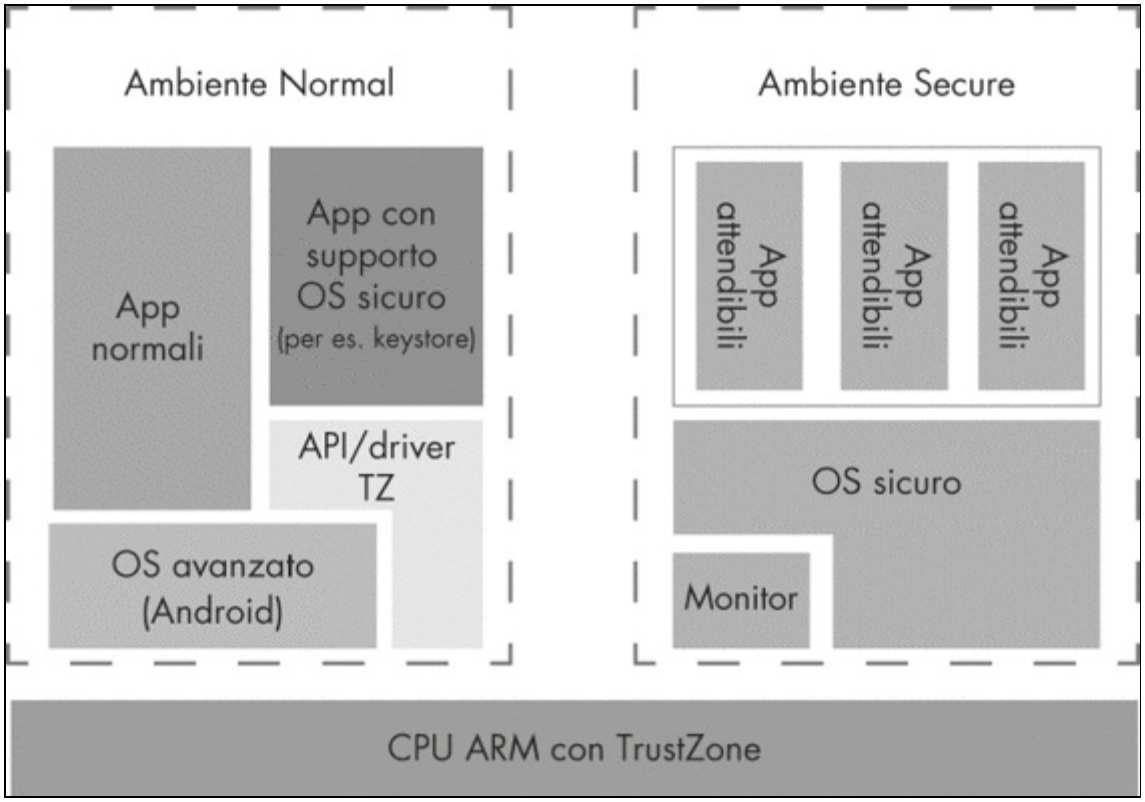

Figura 7.2 Architettura software di TrustZone.

Come al solito, i dettagli sull'implementazione sono piuttosto scarsi, ma su Nexus 4 l'unico modo per interagire con le applicazioni trusted è attraverso l'interfaccia controllata fornita dal dispositivo */dev/qseecom*. Le applicazioni Android che vogliono interagire con QSEE caricano la libreria proprietaria *libQSEEComAPI.so* e ne usano le funzioni per inviare comandi a QSEE.

Come con la maggior parte degli altri SEE, l'API di comunicazione *QSEECom* è di livello piuttosto basso e fondamentalmente consente solo lo scambio di blob opachi (tipicamente comandi e risposte), il contenuto dei quali dipende interamente dall'app sicura con cui state comunicando. Nel caso del *keymaster* di Nexus 4, i comandi usati sono `GENERATE_KEYPAIR`, `IMPORT_KEYPAIR`, `SIGN_DATA` e `VERIFY_DATA`. L'implementazione di *keymaster* crea semplicemente le strutture dei comandi, le invia tramite l'API *QSEECom*

ed effettua il parsing delle risposte. Non contiene tuttavia alcun codice di crittografia.

Un dettaglio interessante riguarda l'app trusted `keystore` di QSEE (che potrebbe non essere un'app dedicata, ma essere parte di un'applicazione attendibile generica), che non restituisce semplici riferimenti alle chiavi protette, ma usa key blob crittografati proprietari. In questo modello, l'unica cosa effettivamente protetta dall'hardware è una qualche forma di *Key-Encryption Key* (KEK) master; le chiavi generate dall'utente sono protette solo indirettamente grazie alla crittografia con la chiave KEK.

Questo metodo permette un numero praticamente illimitato di chiavi protette, ma presenta uno svantaggio: in caso di compromissione della chiave KEK, tutti i key blob memorizzati esternamente risultano anch'essi compromessi. Naturalmente, l'implementazione effettiva potrebbe generare una chiave KEK dedicata per ogni key blob creato, oppure la chiave potrebbe essere "fusa" nell'hardware; in entrambi i casi non sono disponibili dettagli sull'implementazione interna. Detto questo, i key blob del *keymaster* Qualcomm sono definiti in codice AOSP (Listato 7.3) e la definizione suggerisce che gli esponenti privati sono crittografati con AES **(1)**, probabilmente nella modalità CBC, con l'aggiunta di HMAC-SHA256 **(2)** per verificare l'integrità dei dati crittografati.

Listato 7.3 Definizione del blob keymaster QSEE (per Nexus 4).

```
#define KM_MAGIC_NUM     (0x4B4D4B42)    /* "KMKB" Key Master Key Blob in hex */
#define KM_KEY_SIZE_MAX  (512)           /* 4096 bits */
#define KM_IV_LENGTH     (16)          (1)/* AES128 CBC IV */
#define KM_HMAC_LENGTH   (32)          (2)/* SHA2 will be used for HMAC  */

struct  qcom_km_key_blob {
  uint32_t magic_num;
  uint32_t version_num;
  uint8_t  modulus[KM_KEY_SIZE_MAX];(3)
  uint32_t modulus_size;
  uint8_t  public_exponent[KM_KEY_SIZE_MAX];(4)
  uint32_t public_exponent_size;
  uint8_t  iv[KM_IV_LENGTH];(5)
  uint8_t  encrypted_private_exponent[KM_KEY_SIZE_MAX];(6)
  uint32_t encrypted_private_exponent_size;
  uint8_t  hmac[KM_HMAC_LENGTH];(7)
};
```

Come potete vedere nel Listato 7.3, il key blob QSEE contiene il modulo della chiave **(3)**, l'esponente pubblico **(4)**, l'IV **(5)** usato per la crittografia dell'esponente privato, l'esponente privato stesso **(6)** e il valore HMAC **(7)**.

Poiché il QSEE usato in Nexus 4 è implementato tramite le funzioni TrustZone del processore, in questo caso "l'hardware" dell'archivio delle credenziali hardware è semplicemente il SoC ARM. Sono possibili altre implementazioni? A livello teorico, un'implementazione hardware del *keymaster* non deve essere basata su TrustZone: può essere usato qualunque dispositivo dedicato che può generare e memorizzare chiavi in sicurezza e i candidati consueti sono i *Secure Element* (SE) e i *Trusted Platform Module* (TPM). SE e gli altri dispositivi a prova di manomissione sono descritti nel Capitolo 11; tuttavia, ad oggi, nessun dispositivo Android mainstream dispone di TPM dedicati, e i recenti dispositivi di punta hanno iniziato a essere forniti con SE incorporati. Di conseguenza, le implementazioni che usano hardware dedicato saranno difficilmente presenti nei dispositivi mainstream.

NOTA

Naturalmente, tutti i dispositivi mobili dispongono di qualche tipo di *Universal Integrated Circuit Card* (UICC), colloquialmente nota come carta SIM, che di solito può generare e archiviare le chiavi; tuttavia, Android ancora non possiede un'API standard di accesso alle schede UICC, anche se spesso il firmware del vendor ne include una. Per ora, quindi, in via teorica è possibile implementare un modulo *keymaster* basato su UICC, ma questo funzionerebbe solo sulle build Android personalizzate, e spetterebbe ai gestori di rete includere il supporto nelle loro UICC.

Integrazione nel framework

Anche se la gestione sicura delle credenziali è la funzione chiave dell'archivio delle credenziali di Android, il suo scopo principale è fornire questo servizio senza interruzioni al resto del sistema. Vediamo brevemente come si integra con il resto di Android prima di presentare le API pubbliche a disposizione delle app di terze parti.

`keystore` è un servizio Binder standard, pertanto per usarlo i client devono solamente ottenere un riferimento a esso da `ServiceManager`. Il framework Android offre la classe nascosta singola `android.security.KeyStore`, responsabile del recupero di un riferimento al servizio `keystore` e di fungere da proxy per l'interfaccia `IKeystoreService` che espone. La maggior parte delle applicazioni di sistema, come l'importer di file PKCS#12 (Figura 7.1), e

le implementazioni di alcune API pubbliche usano la classe proxy `KeyStore` per comunicare con `keystore`.

Nel caso delle librerie di basso livello che non sono parte del framework Android, come quelle native e le classi JCA nella libreria Java core, l'integrazione con l'archivio delle credenziali di sistema è fornita indirettamente tramite un engine OpenSSL chiamato *engine del keystore Android*.

Un engine OpenSSL è un modulo di crittografia pluggable implementato come libreria condivisa dinamica; l'engine *keystore* è un modulo di questo tipo che implementa tutte le sue operazioni chiamando il modulo HAL *keymaster* di sistema. Supporta solamente il caricamento e la firma con le chiavi private RSA, DSA o EC, ma tale supporto è sufficiente per implementare l'autenticazione basata su chiavi (come l'autenticazione client SSL). L'engine *keystore* consente al codice nativo che usa le API OpenSSL di impiegare le chiavi private salvate nell'archivio delle credenziali di sistema senza che sia necessario modificare il codice. Dispone inoltre di un wrapper Java (`OpenSSLEngine`), utilizzato per implementare l'accesso alle chiavi private gestite dal keystore nel framework JCA.

API pubbliche

Le applicazioni di sistema possono accedere all'interfaccia AIDL del daemon `keystore` direttamente o tramite la classe proxy `android.security.KeyStore`; tuttavia, queste interfacce sono eccessivamente legate all'implementazione per essere parte dell'API pubblica. Android offre astrazioni di livello superiore per le app di terze parti con l'API `KeyChain` e il provider JCA *AndroidKeyStoreProvider*. L'uso di queste API e i dettagli dell'implementazione sono presentati nei paragrafi successivi.

API KeyChain

Android offre un archivio delle credenziali a livello di sistema sin dalla versione 1.6, dove tuttavia era utilizzabile soltanto dai client EAP VPN e Wi-Fi predefiniti. Era possibile installare una coppia chiave privata/certificato utilizzando l'app *Settings*, ma le chiavi installate non erano accessibili alle applicazioni di terze parti.

Android 4.0 ha introdotto le API SDK per la gestione dei certificati trusted e l'archiviazione sicura delle credenziali con la classe `KeyChain`; questa funzionalità è stata estesa in Android 4.3 per supportare le nuove funzionalità basate sull'hardware. L'uso e l'implementazione sono descritti nei prossimi paragrafi.

Classe KeyChain

La classe `KeyChain` è piuttosto semplice: offre infatti sei metodi statici pubblici sufficienti per la maggior parte delle attività legate a chiavi e certificati. Vediamo quindi come installare una coppia chiave privata/certificato e come utilizzarla per accedere alla chiave privata gestita dall'archivio delle credenziali.

L'API `KeyChain` consente di installare una coppia chiave privata/certificato inserita in un file PKCS#12. Il metodo factory `KeyChain.createInstallIntent()` è una via d'accesso a questa funzionalità; non richiede parametri e restituisce un intent di sistema che può eseguire il parsing e l'installazione

di chiavi e certificati (in effetti è lo stesso intent usato internamente dall'app di sistema *Settings*).

Installazione di un file PKCS#12

Per installare un file PKCS#12 è necessario leggerlo in un array di byte, archiviarlo nella chiave `EXTRA_PKCS12` degli extra dell'intent e avviare l'activity associata (Listato 7.4).

Listato 7.4 Installazione di un file PKCS#12 con l'API KeyChain.

```
Intent intent = KeyChain.createInstallIntent();
byte[] p12 = readFile("keystore-test.pfx");
intent.putExtra(KeyChain.EXTRA_PKCS12, p12);
startActivity(intent);
```

Dovrebbe così esservi richiesta la password PKCS#12 per estrarre ed eseguire il parsing della chiave e del certificato. Se la password è corretta, viene richiesto un nome di certificato, come mostrato nella Figura 7.3. Se PKCS#12 dispone di un nome descrittivo, per impostazione predefinita sarà visualizzato quest'ultimo, altrimenti vedrete una lunga stringa hash esadecimale.

La stringa inserita qui è l'alias della chiave o del certificato che potrete usare in seguito per cercare e accedere alle chiavi tramite l'API `KeyChain`. Dovrebbe quindi esservi chiesto di impostare un PIN o una password della schermata di blocco per proteggere l'archivio delle credenziali (se non ne avete già impostato uno).

Uso di una chiave privata

Per usare una chiave privata nell'archivio delle credenziali di sistema dovete ottenere un riferimento alla chiave usando il suo alias e richiedere all'utente il permesso di accedere alla chiave. Se non avete mai eseguito prima l'accesso a una chiave e non ne conoscete l'alias, dovete prima chiamare `KeyChain.choosePrivateKeyAlias()` e fornire un'implementazione di callback che riceva l'alias selezionato, come mostrato nel Listato 7.5.

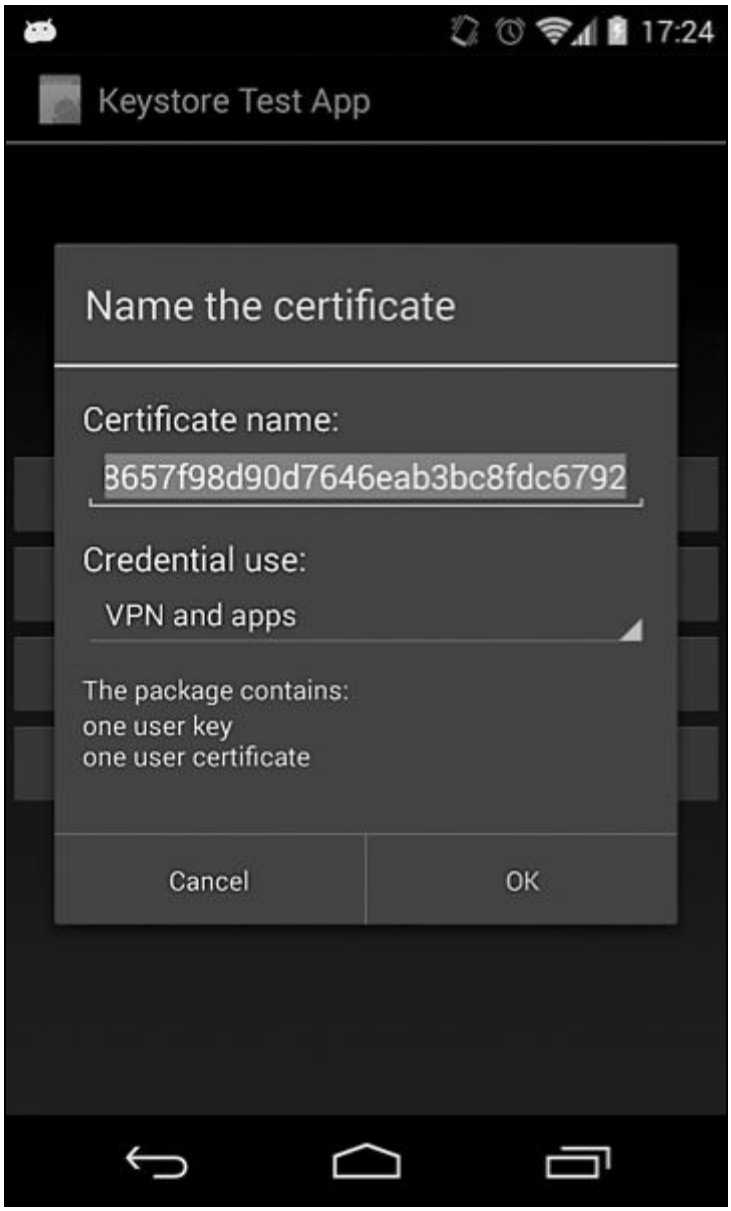

Figura 7.3 Finestra di dialogo di importazione di chiave privata e certificato.

Listato 7.5 Uso di una chiave privata nell'archivio delle credenziali di sistema.

```
public class KeystoreTest extends Activity implements OnClickListener, KeyChainAliasCallback {
    @Override
    public void onClick(View v) {
        KeyChain.choosePrivateKeyAlias((1)this, (2)(KeyChainAliasCallback)this,
            (3)new String[] { "RSA" }, (4)null, (5)null, (6)-1, (7)null);
    }
    @Override
    public void alias(final String alias) {(8)
        Log.d(TAG, "Thread: " + Thread.currentThread().getName());
        Log.d(TAG, "selected alias: " + alias);
    }
}
```

Il primo parametro **(1)** è il contesto corrente; il secondo **(2)** è il callback da richiamare; il terzo e il quarto specificano le chiavi accettabili **(3)** (`RSA` o `null` per qualsiasi chiave) e le autorità di certificazione accettabili **(4)** per il certificato corrispondente alla chiave privata. I due parametri

successivi sono l'host **(5)** e il numero di porta **(6)** del server che richiede un certificato, mentre l'ultimo **(7)** è l'alias da preselezionare nella finestra di selezione della chiave. Nell'esempio non abbiamo specificato nulla (`null` o `-1`), a parte il tipo di chiave, per poter scegliere tra tutti i certificati disponibili. Ricordate che l'alias `alias()` **(8)** non sarà chiamato sul thread principale, quindi non provate a usarlo per manipolare direttamente l'interfaccia (viene chiamato su un thread di Binder).

L'uso della chiave richiede l'autorizzazione dell'utente, pertanto Android deve visualizzare una finestra di selezione della chiave (Figura 7.4) che serve anche per concedere l'accesso alla chiave selezionata. Quando l'utente ha concesso l'accesso alla chiave, l'app può cercare la chiave direttamente senza passare per la finestra di selezione.

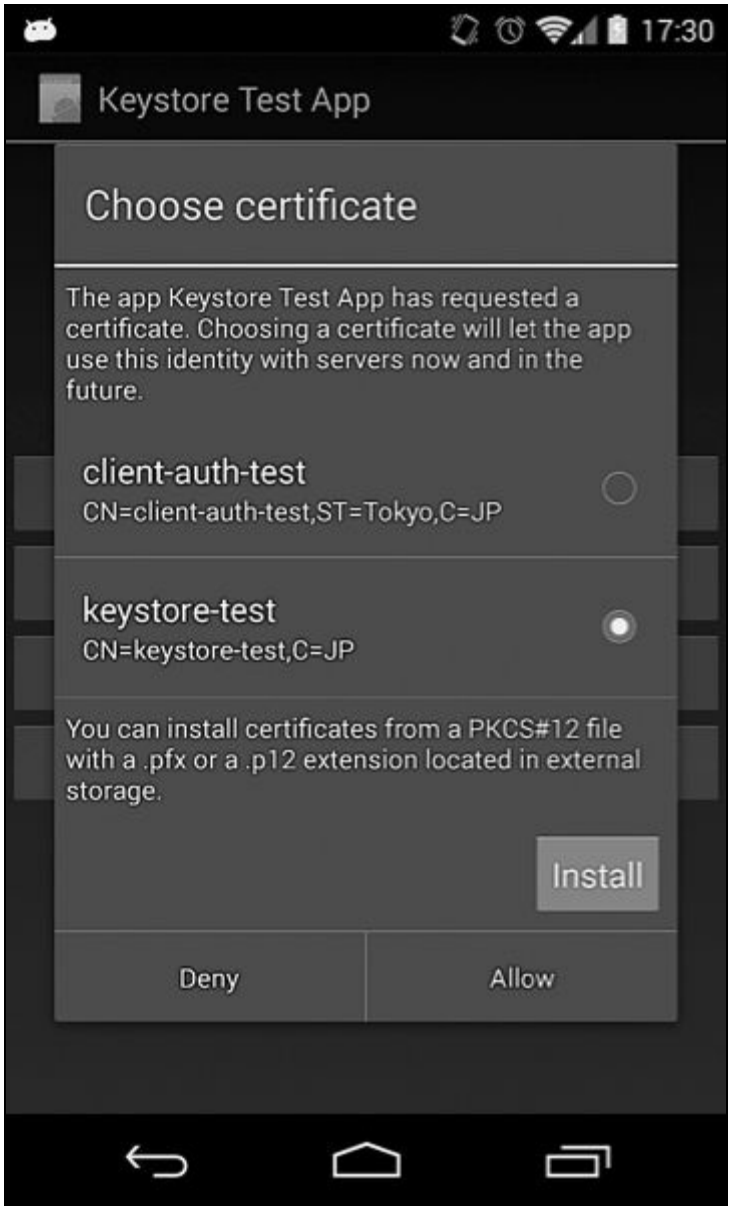

Figura 7.4 Finestra di selezione della chiave.

Il Listato 7.6 mostra come usare l'API `KeyChain` per ottenere un riferimento a una chiave privata gestita dal keystore di sistema.

Listato 7.6 Recupero di un'istanza della chiave e della sua catena di certificati.

```
PrivateKey pk = KeyChain.getPrivateKey(context, alias);(1)
X509Certificate[] chain = KeyChain.getCertificateChain(context, alias);(2)
```

Per ottenere un riferimento a una chiave privata, è necessario chiamare il metodo `KeyChain.getPrivateKey()` **(1)**, passando il nome alias della chiave ottenuto nel passo precedente. Se provate a chiamare questo metodo sul thread principale riceverete un'eccezione, quindi assicuratevi di chiamarlo da un thread in background come quello creato dalla classe utility `AsyncTask`. Il metodo `getCertificateChain()` **(2)** restituisce la catena di certificati associata alla chiave privata (Listato 7.6). Se non esiste una chiave o un certificato con l'alias specificato, il metodo `getPrivateKey()` e `getCertificateChain()` restituiscono `null`.

Installazione di un certificato CA

L'installazione di un certificato CA non è molto diversa dall'installazione di un file PKCS#12. Dovete caricare il certificato in un array di byte e passarlo come extra all'intent di installazione nella chiave `EXTRA_CERTIFICATE`, come mostrato nel Listato 7.7.

Listato 7.7 Installazione di un certificato CA con l'API KeyChain.

```
Intent intent = KeyChain.createInstallIntent();
intent.putExtra(KeyChain.EXTRA_CERTIFICATE, cert);
startActivity(intent);
```

Android esegue il parsing del certificato e, se l'estensione *Basic Constraints* è impostata su `CA:TRUE`, lo considera come un certificato CA e lo importa nel trust store dell'utente. È necessaria l'autenticazione per importare il certificato, ma purtroppo la finestra di importazione (Figura 7.5) non mostra né il DN del certificato né il suo valore hash. L'utente non può quindi sapere che cosa sta importando fino alla fine dell'operazione. Ben poche persone si preoccupano di verificare la validità di un certificato, quindi questa potrebbe essere una possibile minaccia alla sicurezza, visto che le applicazioni dannose potrebbero indurre gli utenti a installare certificati falsi.

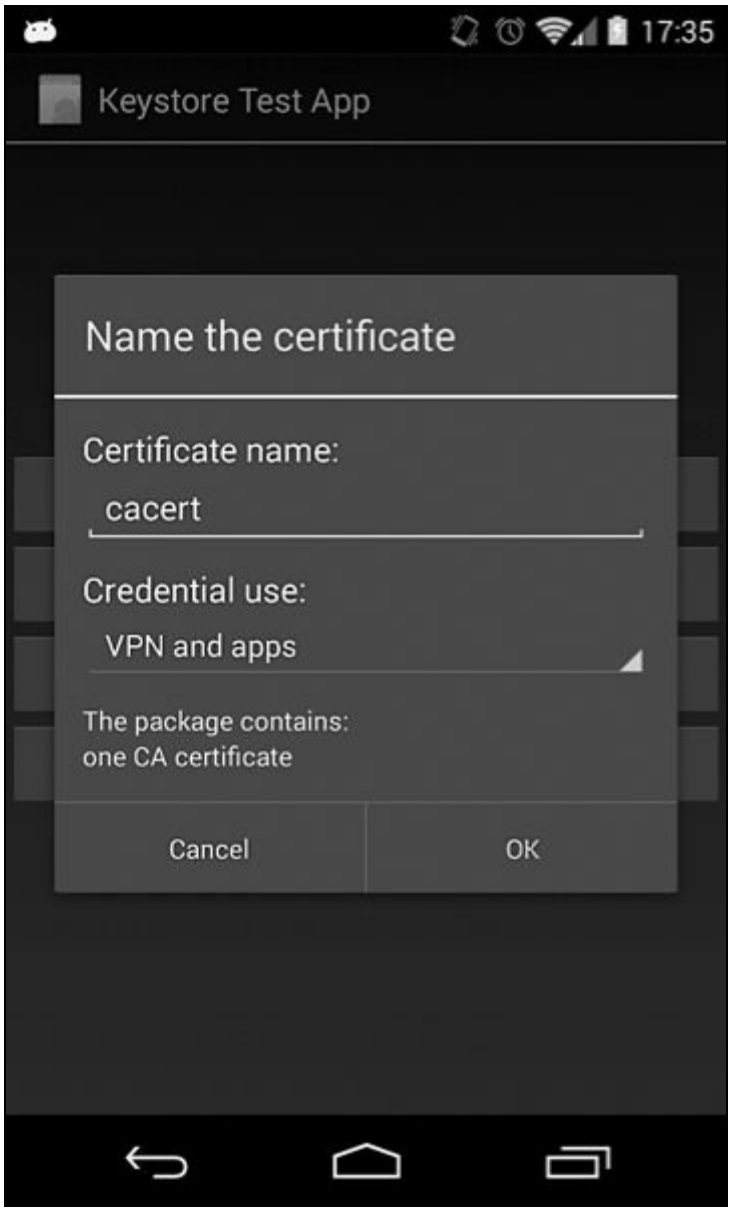

Figura 7.5 Finestra di dialogo di importazione di un certificato CA.

Una volta importato, il certificato appare nella scheda *User* della schermata *Trusted credentials* (*Settings* > *Security* > *Trusted credentials*). Toccate la voce del certificato per visualizzare la finestra dei dettagli, dove potete controllare il soggetto, l'autorità che lo ha rilasciato, il periodo di validità, il numero di serie e le fingerprint SHA-1/SHA-256. Per rimuovere un certificato, toccate il pulsante *Remove* (Figura 7.6).

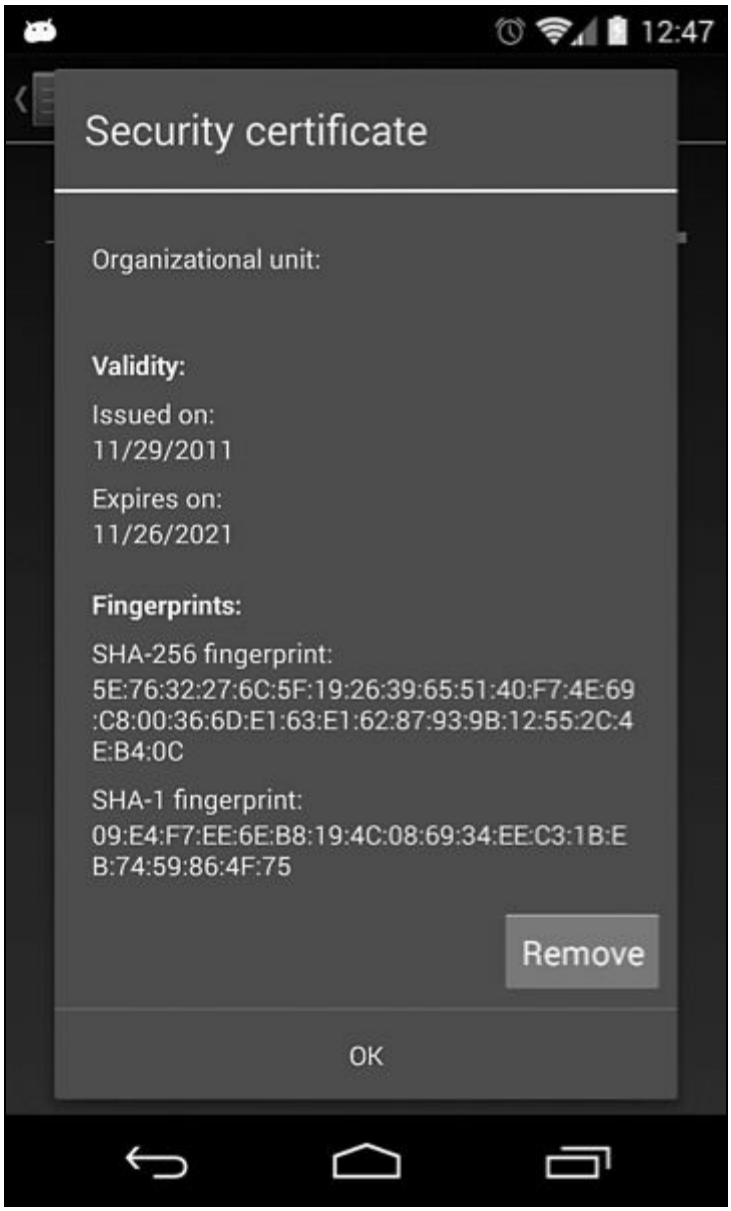

Figura 7.6 Finestra di dialogo con i dettagli del certificato.

Eliminazione di chiavi e certificati utente

Anche se potete eliminare i singoli certificati CA, non c'è modo di eliminare le singole chiavi e i singoli certificati utente (sebbene l'opzione *Clear credentials* nella sezione *Credential storage* delle impostazioni di protezione permetta di eliminare tutte le chiavi e i certificati utente).

NOTA

Finché sono presenti chiavi nell'archivio delle credenziali non potete rimuovere la schermata di blocco, che è utilizzata per proteggere l'accesso al keystore.

Recupero di informazioni sugli algoritmi supportati

Android 4.3 ha aggiunto alla classe `KeyChain` due metodi relativi al supporto hardware. Secondo la documentazione dell'API, `isBoundKeyAlgorithm(String algorithm)` restituisce `true` se l'implementazione `KeyChain` del dispositivo corrente esegue il binding di una `PrivateKey` dell'algoritmo specificato al dispositivo dopo l'importazione o la generazione. In pratica, se passate la stringa `RSA` a questo metodo, dovreste ottenere `true` se le chiavi RSA generate o importate dispongono della protezione hardware e non possono essere copiate esternamente al dispositivo. Il metodo `isKeyAlgorithmSupported(String algorithm)` dovrebbe restituire `true` se l'implementazione attuale di `KeyChain` supporta le chiavi del tipo specificato (RSA, DSA, EC e così via).

Ora che abbiamo presentato le caratteristiche principali dell'API `KeyChain`, vediamone l'implementazione in Android.

Implementazione dell'API KeyChain

La classe pubblica `KeyChain` e le interfacce che la supportano risiedono nel package Java `android.security`, che contiene anche due file AIDL nascosti: `IKeyChainService.aidl` e `IKeyChainAliasCallback`. Possiamo quindi dedurre che la funzionalità del keystore, come la maggior parte dei servizi del sistema operativo Android, è implementata come servizio remoto con cui le API pubbliche possono effettuare il binding. L'interfaccia `IKeyChainAliasCallback` viene chiamata quando selezionate una chiave con `KeyStore.choosePrivateKeyAlias()`, quindi è poco interessante; `IKeyChainService.aidl` definisce invece l'interfaccia di sistema usata effettivamente dai servizi, pertanto ne parleremo nei dettagli.

L'interfaccia `IKeyChainService` dispone di un'implementazione, la classe `KeyChainService` nell'applicazione di sistema `KeyChain`. Oltre a `KeyChainService`, l'applicazione include un'activity, `KeyChain`, e un broadcast receiver, `KeyChainBroadcastReceiver`. `KeyChain` ha il proprio `sharedUserId` impostato su *android.uid.system*, quindi eredita tutti i privilegi dell'utente `system`. I suoi componenti possono inviare comandi di gestione al servizio `keystore` nativo. Vediamo per prima cosa il servizio.

`KeyChainService` è un wrapper per la classe proxy `android.security.KeyStore` che comunica direttamente con il `keystore` nativo. Fornisce quattro servizi principali.

- Gestione del keystore: metodi per recuperare chiavi private e certificati.
- Gestione del trust store: metodi per installare ed eliminare certificati CA nel trust store dell'utente.
- Inizializzazione del keystore e del trust store: un metodo `reset()` che elimina tutte le voci del keystore, compresa la chiave master, riportando il keystore a uno stato non inizializzato; rimuove anche tutti i certificati trusted installati dall'utente.
- Metodi per interrogare e aggiungere voci al database di *grant* (concessione) dell'accesso alle chiavi.

Controllo dell'accesso al keystore

L'applicazione `KeyChain` viene eseguita con l'utente `system`, pertanto qualunque processo che effettua il binding alla sua interfaccia remota è tecnicamente in grado di eseguire tutte le operazioni sul keystore e sul trust store. Per evitare questo problema, `KeyChainService` impone un controllo di accesso supplementare sui suoi utenti, verificando l'accesso alle operazioni dell'archivio delle credenziali in base all'UID del chiamante e utilizzando un database di connessione dell'accesso alle chiavi per regolamentare l'accesso alle singole chiavi. Solo l'utente `system` può eliminare un certificato CA e resettare il keystore e il trust store (operazioni generalmente svolte dall'app *Settings*, eseguita con l'utente `system`). Analogamente, solo l'utente `system` o l'applicazione installer dei certificati (package `com.android.certinstaller`) può installare un certificato CA trusted.

Il controllo dell'accesso alle singole chiavi nell'archivio delle credenziali è un po' più interessante delle restrizioni alle operazioni. `KeyChainService` mantiene un database dei grant (in

`/data/data/com.android.keychain/databases/grants.db`) che mappa gli UID agli alias delle chiavi che possono utilizzare. Vediamolo meglio nel Listato 7.8.

Listato 7.8 Schema e contenuto del database grants.

```
# sqlite3 grants.db
sqlite> .schema
.schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE grants (alias STRING NOT NULL, uid INTEGER NOT NULL, UNIQUE (alias,uid));
sqlite> select * from grants;
select * from grants;
(1)test|10044(2)
(3)key1|10044
```

In questo esempio, l'applicazione con UID `10044` **(2)** può accedere alle chiavi con alias `test` **(1)** e `key1` **(3)**.

Ogni chiamata a `getPrivateKey()` o `getCertificate()` è soggetta a un controllo rispetto al database dei grant e provoca un'eccezione se non viene trovato un grant per l'alias richiesto. Come affermato in precedenza, `KeyChainService` dispone di API per aggiungere e recuperare i grant che solo l'utente `system` può chiamare. Occorre però capire chi ha l'effettiva responsabilità di concedere e revocare l'accesso.

Ricordate la finestra di selezione della chiave privata (Figura 7.4)? Quando chiamate `KeyChain.choosePrivateKeyAlias()`, viene avviato `KeyChainActivity` (presentato in precedenza), che verifica se il keystore è sbloccato; in questo caso, `KeyChainActivity` mostra la finestra di selezione della chiave. Facendo clic sul pulsante *Allow* si ritorna a `KeyChainActivity`, che chiama `KeyChainService.setGrant()` con l'alias selezionato, aggiungendolo al database dei grant. Di conseguenza, anche se l'activity che richiede l'accesso a una chiave privata possiede i permessi necessari, l'utente deve sbloccare il keystore e autorizzare esplicitamente l'accesso a ogni singola chiave.

Oltre a controllare l'archiviazione delle chiavi private, `KeyChainService` offre la gestione del trust store grazie alla nuova classe `TrustedCertificateStore` (parte di *libcore*). Questa classe consente sia di aggiungere certificati CA trusted installati dall'utente sia di rimuovere (o di segnalare come inattendibili) le CA di sistema (preinstallate). Nel Capitolo 6 sono disponibili i dettagli della sua implementazione.

KeyChainBroadcastReceiver

L'ultimo componente dell'app `KeyChain` è `KeyChainBroadcastReceiver`, che attende il broadcast di sistema `android.intent.action.PACKAGE_REMOVED` e inoltra il controllo a `KeyChainService`. Alla ricezione dell'azione `PACKAGE_REMOVED`, il servizio effettua una manutenzione del database dei grant, esaminando tutte le voci ed eliminando eventuali package referenziati che non sono più disponibili (perché sono stati disinstallati).

Riepilogo sulle credenziali e sul trust store

Android 4.0 introduce un nuovo servizio che concede l'accesso sia al keystore di sistema (gestito dal servizio di sistema `keystore`) sia al trust store (gestito dalla classe `TrustedCertificateStore`), sfruttando l'API `KeyChain` esposta nell'SDK pubblico. Questa funzionalità consente di controllare l'accesso alle chiavi in base sia all'UID del processo chiamante sia al database dei grant di accesso, permettendo così un controllo preciso e guidato dall'utente sulle chiavi a cui ogni applicazione può accedere. I componenti dell'archivio delle credenziali e del trust store di Android sono mostrati nella Figura 7.7 con le relative relazioni.

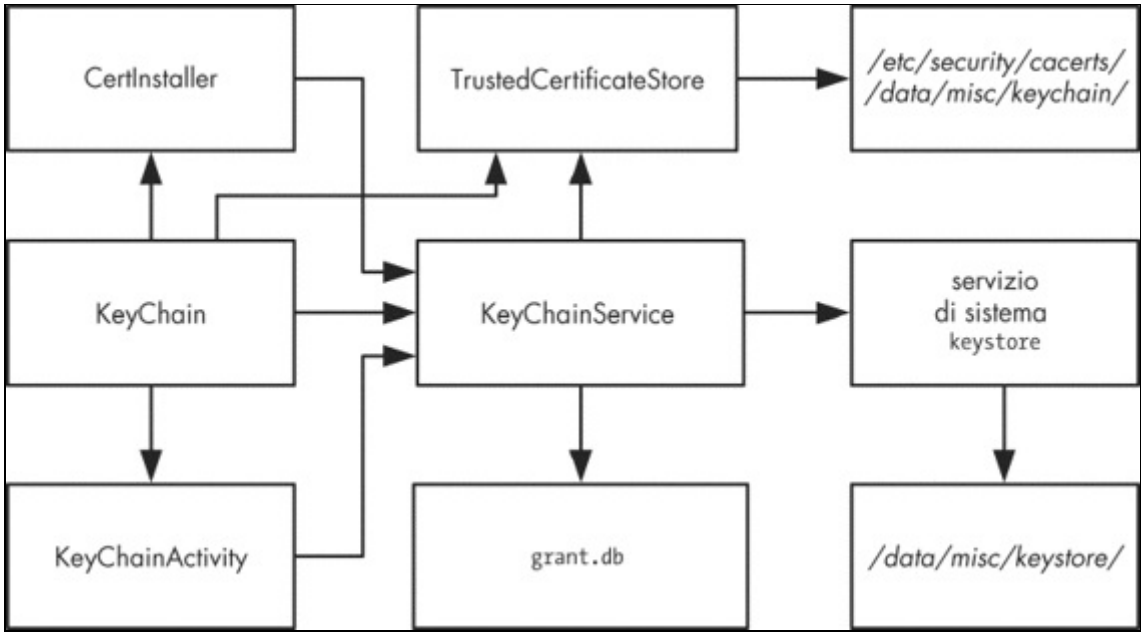

Figura 7.7 Componenti dell'archivio delle credenziali di sistema.

Provider keystore di Android

Anche se l'API `KeyChain` introdotta in Android 4.0 permette alle applicazioni di importare le chiavi nell'archivio delle credenziali di sistema, queste chiavi sono di proprietà dell'utente `system`, e qualunque applicazione può chiedere di accedervi. Android 4.3 aggiunge il supporto per le *chiavi private delle app*, che consente a ogni app di generare e salvare chiavi private a cui può accedere solo l'app in questione e che non sono visibili alle altre app.

Invece che con l'introduzione di un'altra API specifica per Android, l'accesso al keystore viene esposto tramite API JCA standard, nello specifico `java.security.KeyPairGenerator` e `java.security.KeyStore`. Entrambe sono supportate da un nuovo provider JCA di Android, *AndroidKeyStoreProvider*, e vi si accede passando `AndroidKeyStore` come parametro `type` dei rispettivi metodi factory. Il Listato 7.9 mostra come generare e accedere alle chiavi ESA usando *AndroidKeyStoreProvider*.

Listato 7.9 Generazione e accesso alle chiavi RSA con AndroidKeyStoreProvider.

```
// genera una coppia di chiavi
Calendar notBefore = Calendar.getInstance()
Calendar notAfter = Calendar.getInstance();
notAfter.add(1, Calendar.YEAR);
KeyPairGeneratorSpec spec = new KeyPairGeneratorSpec.Builder(ctx)
                .setAlias("key1")
                .setKeyType("RSA")
                .setKeySize(2048)
                .setSubject(new X500Principal("CN=test"))
                .setSerialNumber(BigInteger.ONE).setStartDate(notBefore.getTime())
                .setEndDate(notAfter.getTime()).build();(1)
KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("RSA",
                              "AndroidKeyStore");
kpGenerator.initialize(spec);(2)
KeyPair kp = kpGenerator.generateKeyPair();(3)
// in un'altra parte dell'app, accede alle chiavi
KeyStore ks = KeyStore.getInstance("AndroidKeyStore");
ks.load(null);
KeyStore.PrivateKeyEntry keyEntry = (KeyStore.PrivateKeyEntry)keyStore.getEntry("key1", null);(4)
RSAPublic pubKey = (RSAPublicKey)keyEntry.getCertificate().getPublicKey();
RSAPrivateKey privKey = (RSAPrivateKey) keyEntry.getPrivateKey();
```

Per prima cosa **(1)** occorre creare un `KeyPairGeneratorSpec` che descrive le chiavi da generare e il certificato autofirmato creato automaticamente a cui è associata ogni chiave. È possibile specificare il tipo di chiave (*RSA*, *DSA* o *EC*) usando il metodo `setKeyType()` e la dimensione della chiave con il metodo `setKeySize()`.

NOTA

Ogni `PrivateKeyEntry` gestita da un oggetto `KeyStore` deve essere associata a una catena di certificati. Android crea automaticamente un certificato autofirmato quando generate una chiave, ma in un secondo momento potete sostituire il certificato predefinito con uno firmato da una CA.

A seguire dovete inizializzare un `KeyPairGenerator` **(2)** con l'istanza `KeyPairGeneratorSpec` e quindi generare le chiavi chiamando `generateKeyPair()` **(3)**.

Il parametro più importante è l'alias che passate a `KeyStore.getEntry()` **(4)** per ottenere in un secondo momento un riferimento alle chiavi generate. L'oggetto chiave restituito non contiene il materiale effettivo della chiave, ma è solamente un puntatore a un oggetto chiave gestito dall'hardware. Di conseguenza, non è utilizzabile con i provider di crittografia che fanno affidamento a materiali delle chiavi direttamente accessibili.

Se il dispositivo dispone di un'implementazione hardware del keystore, le chiavi saranno generate esternamente al sistema operativo Android e non saranno accessibili direttamente nemmeno all'utente di sistema (o `root`). Se l'implementazione è solo software, le chiavi saranno crittografate con una chiave master di crittografia per utente derivata dal PIN o dalla password di sblocco.

Riepilogo

Come avete imparato in questo capitolo, Android dispone di un archivio (o store) delle credenziali che può essere usato per memorizzare le credenziali delle funzionalità integrate, quali la connettività Wi-Fi e VPN, nonché delle app di terze parti. Android 4.3 e versioni successive mettono a disposizione API JCA standard per generare e accedere alle chiavi private delle app, facilitando alle app non di sistema la memorizzazione sicura delle loro chiavi senza che sia necessario implementare una protezione specifica delle chiavi. L'archiviazione hardware delle chiavi, disponibile sui dispositivi supportati, garantisce che persino le app con privilegi `system` o `root` non possano estrarre le chiavi. La maggior parte delle attuali implementazioni hardware dell'archivio delle credenziali si basa sulla tecnologia TrustZone di ARM e non usa hardware dedicato a prova di manomissione.

Gestione degli account online

Laddove i servizi aziendali di solito usano PKI per l'autenticazione degli utenti, molti servizi online disponibili al pubblico ricorrono alle password. L'inserimento di password complesse sul touch screen di un dispositivo mobile, per di più diverse volte al giorno per siti diversi, non è certo un esercizio piacevole.

Nel tentativo di migliorare l'esperienza dell'utente durante l'accesso ai servizi online, Android propone un registro centralizzato degli account utente che consente di salvare e riutilizzare le credenziali. Questo registro è accessibile dalle applicazioni di terze parti, che possono quindi accedere ai servizi web per conto dell'utente del dispositivo senza che le app debbano gestire direttamente le password. In questo capitolo vedremo in che modo Android gestisce le credenziali dell'account online di un utente e le API che le applicazioni possono utilizzare per sfruttare le credenziali nella cache e per registrare account personalizzati. Mostreremo quindi come i dispositivi Google Experience (ovvero quelli su cui è preinstallato Google Play Store) archivino le informazioni dell'account Google e consentano l'accesso alle API di Google e ad altri servizi online tramite le credenziali memorizzate.

Panoramica sulla gestione degli account in Android

Anche se i primi dispositivi Android disponevano del supporto integrato per gli account Google e la sincronizzazione automatica e in background dei dati con i servizi Google come Gmail, non fornivano alcuna API per questa funzionalità. Android 2.0 (API livello 5) ha introdotto il concetto di gestione centralizzata degli account con un'API pubblica, il cui elemento fondamentale è la classe `AccountManager`, che consente di accedere a un registro centrale degli account online dell'utente. L'utente inserisce le credenziali (nome utente e password) una sola volta per ogni account, concedendo alle applicazioni l'accesso alle risorse online con l'approvazione "one-click" (`http://bit.ly/1qANbac`). Un'altra caratteristica interessante della classe è la possibilità di ottenere un token di autenticazione per gli account supportati, che permette alle applicazioni di terze parti di effettuare l'autenticazione per i servizi online senza in effetti gestire la password utente. In alcune versioni di Android precedenti, `AccountManager` monitorava anche la SIM ed eliminava le credenziali nella cache quando la si cambiava; questa funzionalità è stata però rimossa in Android 2.3.4 e versioni successive.

Implementazione della gestione degli account

Come per la maggior parte delle API di sistema Android, `AccountManager` è soltanto una facciata per `AccountManagerService`, che svolge il lavoro vero e proprio. Il servizio non fornisce un'implementazione per una forma specifica di autenticazione, ma coordina semplicemente diversi moduli di autenticazione per diversi tipi di account (Google, Twitter, Microsoft Exchange e così via). Qualunque applicazione può registrare un modulo autenticatore implementando un autenticatore di account e le classi correlate, se necessario. Vedremo come scrivere e registrare un modulo personalizzato nel paragrafo "Aggiunta di un modulo autenticatore".

La registrazione di un nuovo tipo di account nel sistema consente di trarre vantaggio da numerosi servizi dell'infrastruttura Android, compresa la possibilità di:

- utilizzare un archivio delle credenziali centralizzato in un database di sistema;
- rilasciare token alle app di terze parti;
- sfruttare la sincronizzazione automatica in background di Android (tramite un adapter di sincronizzazione).

La Figura 8.1 mostra i componenti principali dei sottosistemi di gestione account di Android e le relative relazioni. I vari componenti e i loro ruoli saranno descritti nei prossimi paragrafi.

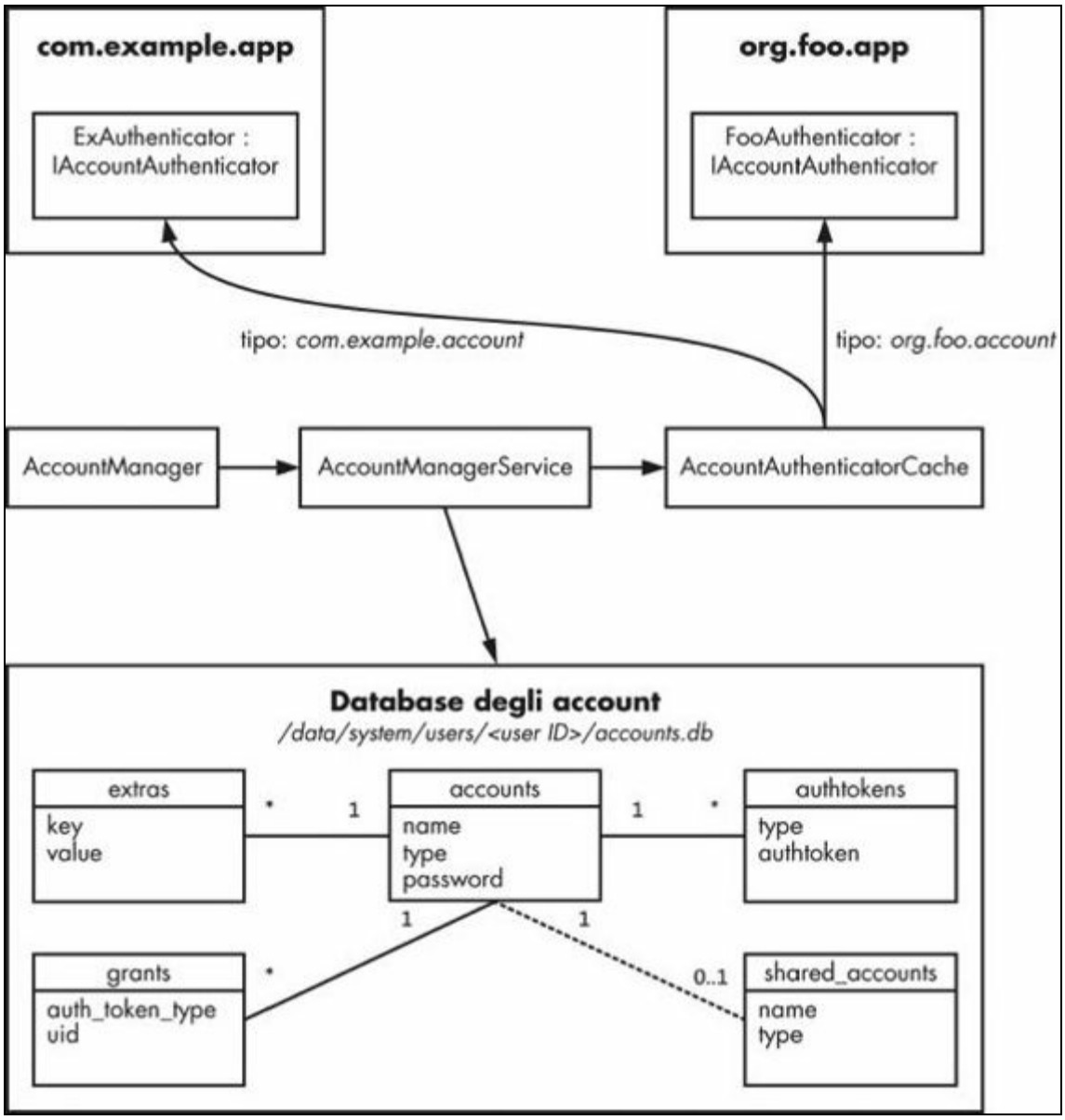

Figura 8.1 Componenti di gestione degli account.

AccountManagerService e AccountManager

L'elemento centrale è `AccountManagerService`, che coordina tutti i componenti e consente la persistenza dei dati degli account nel relativo database. La classe `AccountManager` è la facciata che espone un sottoinsieme delle sue funzionalità alle applicazioni di terze parti, avviando worker thread per i metodi asincroni e inviando i risultati (o i dettagli dell'errore) al chiamante. Inoltre, `AccountManager` mostra un selettore di account quando la funzionalità o il token richiesto può essere fornito da più account. Purtroppo, però, non applica alcun permesso; tutti i permessi del

chiamante sono controllati da `AccountManagerService`. I permessi concreti saranno presentati a breve.

Moduli autenticatori

Come già affermato, la funzionalità di ogni account registrato è fornita da un modulo di autenticazione modulare. I *moduli autenticatori* sono definiti e ospitati dalle applicazioni e non sono altro che servizi di binding che implementano l'interfaccia AIDL `android.accounts.IAccountAuthenticator`. Questa interfaccia dispone dei metodi per aggiungere un account, chiedere all'utente le sue credenziali, ottenere un token di autenticazione e aggiornare i metadati dell'account. Nella pratica, le applicazioni non implementano direttamente questa interfaccia, ma estendono la classe `android.accounts.AbstractAccountAuthenticator` che collega i metodi di implementazione a uno stub AIDL interno.

`AbstractAccountAuthenticator` garantisce inoltre che tutti i chiamanti dello stub AIDL possiedano il permesso `ACCOUNT_MANAGER`, un permesso di firma del sistema che consente solo ai componenti del sistema di chiamare direttamente i moduli autenticatori. Tutti gli altri client devono passare per `AccountManagerService`.

Ogni modulo autenticatore implementa un account definito in maniera univoca grazie a una stringa chiamata *tipo di account*. I tipi di account sono generalmente espressi nella notazione a dominio invertito (come i package Java) e di solito prendono il nome del package di base dell'applicazione che li definisce, concatenato al tipo di account oppure alle stringhe `account` o `auth` (Android comunque non impone questa regola e non vi sono linee guida esplicite). Per esempio, nella Figura 8.1, l'applicazione `com.example.app` definisce un account con tipo `com.example.account`, mentre l'applicazione `org.foo.app` definisce un account con tipo `org.foo.account`.

I moduli autenticatori sono implementati aggiungendo un servizio a cui è possibile effettuare il binding utilizzando l'azione di intent `android.accounts.AccountAuthenticator` nell'applicazione host. Il tipo di account, insieme ad altri metadati, viene collegato al servizio aggiungendo un tag

`<meta-data>` alla dichiarazione del servizio. L'attributo `resource` del tag fa riferimento a un file XML contenente i metadati dell'account (il Listato 8.8 offre un esempio).

NOTA

Un tag `<meta-data>` consente di associare una coppia nome-valore contenente dati arbitrari al suo componente padre. I dati possono essere valori letterali, come stringhe o integer, oppure riferimenti a file di risorse Android. Sono inoltre supportati più tag `<meta-data>` per componente. I valori di tutti i tag `<meta-data>` sono raccolti in un singolo oggetto `Bundle` e sono resi disponibili dal campo `metaData` della classe `PackageItemInfo` (la classe base delle classi concrete che incapsula i valori degli attributi del componente, per esempio `ServiceInfo`). L'interpretazione dei metadati associati è specifica per ogni componente.

Cache del modulo autenticatore

La "modularità" è fornita dalla classe `AccountAuthenticatorCache`, che esamina i package che definiscono i moduli autenticatori e li mettono a disposizione di `AccountManagerService`. `AccountAuthenticatorCache` è un'implementazione della funzionalità generica della cache dei servizi registrati fornita da Android. La cache viene creata su richiesta interrogando `PackageManagerService` in merito ai package installati che registrano azioni di intent e file di metadati specifici. La cache viene mantenuta da un broadcast receiver che attiva un aggiornamento all'aggiunta, all'aggiornamento o alla rimozione dei package. La cache è persistente e viene scritta su disco ogni volta che si rileva un cambiamento; i file della cache sono scritti nella directory */data/system/registered_services/* e prendono il nome dall'azione di intent cercata. La cache dei moduli autenticatori è salvata nel file `android.accounts.AccountAuthenticator.xml` e può essere simile al contenuto del Listato 8.1.

Listato 8.1 Contenuto del file di cache per i servizi registrati AccountAuthenticator.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<services>
    <service uid="10023" type="com.android.exchange" />(1)
    <service uid="10023" type="com.android.email" />(2)
    <service uid="10069" type="com.example.account" />(3)
    <service uid="10074" type="org.foo.account" />(4)
    --altro codice--
    <service uid="1010023" type="com.android.email" />(5)
    <service uid="1010023" type="com.android.exchange" />(6)
    <service uid="1010069" type="com.example.account" />(7)
    --altro codice--
</services>
```

Qui i tipi di account `com.android.exchange` e `com.android.email` (**(1)** e **(2)**) sono registrati dall'applicazione stock Email, mentre `com.example.account` e `org.foo.account` (**(3)** e **(4)**) sono registrati da applicazioni di terze parti. Su un dispositivo multiutente, il file della cache contiene voci per tutti gli account disponibili per ogni utente.

In questo esempio, il primo utente secondario (user ID 10) può utilizzare `com.android.exchange`, `com.android.email` e `com.example.account` (**(5)**, **(6)** e **(7)**), ma non l'account `org.foo.account` (perché non è disponibile una voce relativa nel file). Quando `AccountManagerService` deve eseguire un'azione con un particolare account, interroga `AccountAuthenticatorCache` per conoscere il servizio di implementazione passando il tipo di account. Se per l'utente corrente è registrata un'implementazione per quel tipo di account, `AccountAuthenticatorCache` restituisce i dettagli sul servizio di implementazione, che contengono il nome del componente di implementazione e l'UID del package host. `AccountManagerService` usa queste informazioni per il binding al servizio, al fine di poter chiamare i metodi dell'interfaccia `IAccountAuthenticator` implementata dal servizio.

Permessi e operazioni di AccountManagerService

Come mostrato nella Figura 8.1, `AccountManagerService` implementa la sua funzionalità chiamando i moduli autenticatori o usando i dati nella cache del database degli account. I componenti di terze parti possono usare solo l'API esposta da `AccountManagerService`; non possono invece accedere ai moduli autenticatori o al database degli account. Questa interfaccia centralizzata garantisce il flusso di lavoro delle operazioni e applica le regole di accesso per ogni operazione.

`AccountManagerService` implementa il controllo di accesso usando una combinazione di permessi e controlli dell'UID e della firma del chiamante. Vediamo ora le operazioni che fornisce e i controlli relativi sui permessi.

Elenco e autenticazione degli account

I client possono ottenere un elenco di account corrispondenti a particolari criteri (quali tipo, package dichiarante e altre caratteristiche) chiamando uno dei metodi `getAccounts()`; possono inoltre verificare se un particolare account dispone delle caratteristiche richieste chiamando il metodo `hasFeatures()`. Queste operazioni richiedono il permesso `GET_ACCOUNTS`, che ha un livello di protezione *normal*. Un nuovo account di un tipo specifico può essere aggiunto chiamando il metodo `addAccount()` (che avvia un'*activity di autenticazione* specifica per l'implementazione che raccoglie le credenziali dall'utente), oppure chiamando il metodo `addAccountExplicitly()`, che accetta come parametri l'account, la password e qualunque altro dato utente associato. Il primo metodo richiede che i chiamanti abbiano il permesso `MANAGE_ACCOUNTS`, il secondo che possiedano il permesso `AUTHENTICATE_ACCOUNTS` e che abbiano lo stesso UID dell'autenticatore dell'account. Entrambi i permessi hanno livello di protezione *dangerous* e richiedono quindi la conferma dell'utente in fase di installazione dell'app. La richiesta ai chiamanti di `addAccountExplicitly()` di avere lo stesso UID dell'autenticatore garantisce che solo la medesima app, o le app appartenenti allo stesso user ID condiviso (vedete il Capitolo 2 per i dettagli), possa aggiungere account senza l'interazione dell'utente.

Altre operazioni che richiedono al chiamante di avere sia il permesso `AUTHENTICATE_ACCOUNTS` sia lo stesso UID dell'autenticatore dell'account sono elencate di seguito (per ragioni di chiarezza, nei seguenti paragrafi sono stati omessi i parametri del metodo `AccountManager`; fate riferimento alla documentazione della classe `AccountManager` per conoscere le firme complete del metodo e altre informazioni: `http://bit.ly/1qANbac`).

- `getPassword()`: restituisce la password nella cache "raw".
- `getUserData()`: restituisce i metadati dell'account, specifici per ogni autenticatore, che corrispondono a una chiave specificata.
- `peekAuthToken()`: restituisce un token della cache di tipo specificato (se disponibile).
- `setAuthToken()`: aggiunge o sostituisce un token di autenticazione per un account.

- `setPassword()`: imposta o cancella la password salvata nella cache per un account.
- `setUserData()`: imposta o cancella la voce dei metadati con la chiave specificata.

Gestione degli account

Come l'aggiunta di un nuovo account, anche la rimozione di un account esistente richiede il permesso `MANAGE_ACCOUNTS`. Tuttavia, se per l'utente del dispositivo di chiamata è impostata la restrizione `DISALLOW_MODIFY_ACCOUNTS` (consultate il Capitolo 4 per i dettagli sulle restrizioni utente), questi non potrà aggiungere o rimuovere account, nemmeno se l'applicazione chiamante dispone del permesso `MANAGE_ACCOUNTS`. Altri metodi che richiedono questo permesso sono quelli che modificano le credenziali o le proprietà dell'account, come indicato di seguito.

- `clearPassword()`: cancella una password salvata nella cache.
- `confirmCredentials()`: conferma esplicitamente che l'utente conosce la password (anche se è già nella cache) mostrando una finestra di inserimento password.
- `editProperties()`: mostra una finestra che consente all'utente di cambiare le impostazioni globali dell'autenticatore.
- `invalidateAuthToken()`: rimuove un token di autenticazione dalla cache. Può essere chiamato anche se il chiamante ha il permesso `USE_CREDENTIALS`.
- `removeAccount()`: rimuove un account esistente.
- `updateCredentials()`: chiede all'utente di inserire la password corrente e aggiorna di conseguenza le credenziali salvate.

Uso delle credenziali degli account

L'ultimo permesso che `AccountManagerService` potrebbe richiedere ai suoi client è `USE_CREDENTIALS`. Questo permesso protegge i metodi che restituiscono o modificano un *token di autenticazione*, ovvero una stringa di

credenziali dipendente dal servizio che i client possono usare per autenticare le richieste al server senza inviare la password a ogni richiesta.

Tipicamente, i server restituiscono un token di autenticazione dopo che il client ha effettuato correttamente l'autenticazione con i suoi nome utente e password (o altre credenziali permanenti). Il token è identificato da una stringa chiamata *tipo di token*, che descrive il tipo di accesso offerto dal token (per esempio sola lettura o lettura/scrittura). Il token è riutilizzabile e può essere usato per inviare più richieste, ma presenta un periodo di validità limitato. Inoltre, se si ritiene che un account utente sia stato compromesso, o se l'utente cambia la sua password, tutti i token di autenticazione esistenti per l'utente vengono invalidati sul server. In questo caso, le richieste che usano i token di autenticazione della cache terminano con un errore di autenticazione. Dal momento che `AccountManagerService` è agnostico nei confronti del protocollo e dell'applicazione, non invalida automaticamente i token nella cache, nemmeno se questi sono scaduti o sono stati invalidati dal server. Le applicazioni hanno il compito di eliminare tali token non validi attraverso la chiamata al metodo `invalidateAuthToken()`.

Quelli che seguono sono i metodi che richiedono `USE_CREDENTIALS`.

- `getAuthToken()`: recupera un token di autenticazione del tipo specificato per un account specifico.
- `invalidateAuthToken()`: rimuove un token di autenticazione dalla cache. Può essere chiamato anche se il chiamante ha il permesso `MANAGE_ACCOUNTS`.

Richiesta di accesso con token di autenticazione

Oltre a possedere il permesso `USE_CREDENTIALS`, per ottenere un token di autenticazione di un tipo specifico, i chiamanti del metodo `getAuthToken()` (o di uno dei suoi metodi wrapper forniti dalla classe di facciata `AccountManager`) devono ricevere esplicitamente l'autorizzazione ad accedere al tipo di token richiesto. A tal fine viene visualizzata una finestra di conferma

come quella mostrata nella Figura 8.2. La finestra di dialogo mostra sia il nome dell'applicazione richiedente (in questo esempio nel primo punto, *Account Requestor*), il tipo e il nome dell'account (nel secondo punto, rispettivamente *Example* ed *example_user*) e una breve descrizione (sotto i punti, *Full access to example data*) del tipo di accesso ai dati che sarà consentito a seguito della richiesta di accesso. Se l'utente concede l'accesso, la decisione viene salvata nella cache e la finestra di dialogo non sarà più visualizzata per le successive richieste di un token dello stesso tipo. Le applicazioni in esecuzione con lo stesso UID del modulo autenticatore possono accedere ai relativi token senza mostrare una finestra di dialogo di conferma. Inoltre, alle applicazioni di sistema privilegiate viene concesso implicitamente l'accesso a tutti tipi di token senza necessità di conferma da parte dell'utente, pertanto la finestra di dialogo non viene visualizzata se la richiesta del token proviene da un'applicazione privilegiata.

Database degli account

Ora che abbiamo presentato i moduli autenticatori, la cache dell'autenticatore e le caratteristiche principali di `AccountManagerService`, vediamo come questo servizio usa il *database degli account*, un database SQLite salvato nella directory di sistema di ogni utente con il nome `accounts.db`, per registrare gli account e le credenziali nella cache.

Il database degli account si trova in */data/system/users/0/accounts.db* nei dispositivi monoutente; nei dispositivi multiutente, il file archivia le informazioni sugli account per l'utente primario, mentre gli utenti secondari dispongono di un'istanza personale dello stesso in */data/system/users/<ID utente>/accounts.db*. Il database contiene sei tabelle: `accounts`, `extras`, `authtokens`, `grants`, `shared_users` e `meta`. A oggi la tabella `meta` sembra essere inutilizzata; tutte le altre tabelle con le loro relazioni sono mostrate nella Figura 8.1.

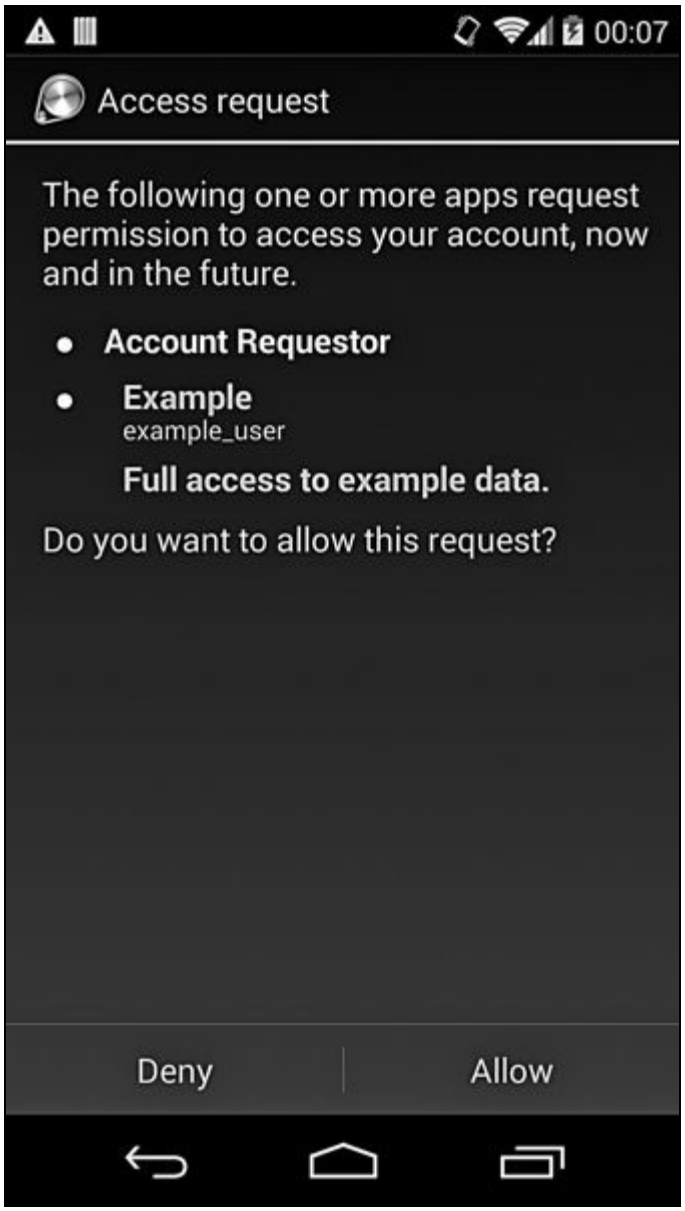

Figura 8.2 Finestra di richiesta di accesso all'account.

Schema delle tabelle

La tabella `accounts` memorizza il nome, il tipo e la password degli account registrati; tutte le altre tabelle sono direttamente o indirettamente collegate a essa. Potrebbe contenere dati simili a quelli mostrati nel Listato 8.2.

Listato 8.2 Contenuto della tabella accounts.

```
sqlite> select * from accounts;
_id|name              |type                    |password
1  |user1@gmail.com   |com.google              |1/...... (1)
2  |user1@example.com|com.google.android.pop3|password (2)
3  |example_user      |com.example.account     |pass1234 (3)
```

Qui **(1)** è un account Google (tipo `com.google`) che consente l'accesso a Gmail, Google Play Store e altri servizi Google. Gli account Google dipendono da componenti di sistema proprietari e sono disponibili solo sui dispositivi Google Experience (maggiori dettagli sono disponibili nel paragrafo "Servizio di login Google" di questo capitolo). L'account in **(2)** è un account di posta POP3 (tipo `com.google.android.pop3`) registrato dall'applicazione stock Email, mentre **(3)** è un account personalizzato (tipo `com.example.account`) registrato da un'applicazione di terze parti. Ogni account può essere associato a zero o più coppie chiave-valore di metadati salvati nella tabella `extras`; il collegamento all'account avviene attraverso la chiave primaria (nella colonna `_id`). Per esempio, se la nostra applicazione personalizzata (**(3)** nel Listato 8.2, `_id=3`) eseguisse la sincronizzazione dei dati in background, potrebbe contenere voci simili a quelle mostrate nel Listato 8.3.

Listato 8.3 Contenuto della tabella extras.

```
sqlite> select * from extras where accounts_id=3;
_id|accounts_id|key      |value
11 |3          |device_id|0123456789
12 |3          |last_sync|1395297374
13 |3          |user_id  |abcdefghij
14 |3          |option1  |1
```

La tabella `authtokens` memorizza i token rilasciati per un account. Per la nostra applicazione personalizzata è simile a quella nel Listato 8.4.

Listato 8.4 Contenuto della tabella authtokens.

```
sqlite> select * from authtokens where accounts_id=3;
_id|accounts_id|type            |authtoken
16 |3          |com.example.auth|abcdefghij0123456789
```

La tabella `grants` associa gli UID delle applicazioni ai tipi di token che possono utilizzare. I grant sono aggiunti quando l'utente consente, nella finestra di dialogo relativa, l'accesso a un particolare tipo di account con relativo token (Figura 8.2). Per esempio, se un'applicazione con UID 10291 ha chiesto e ottenuto l'accesso ai token di tipo `com.example.auth` come nell'applicazione di esempio (Listato 8.4), il grant sarà rappresentato dalla riga riportata di seguito nella tabella `grants` (Listato 8.5); una nuova riga viene aggiunta per ogni combinazione di ID account, tipo di token e UID dell'applicazione.

Listato 8.5 Contenuto della tabella grants.

```
sqlite> select * from grants;
accounts_id|auth_token_type |uid
3          |com.example.auth|10291
```

La tabella `shared_accounts` è usata per condividere gli account del proprietario del dispositivo con uno degli utenti con restrizioni del device (maggiori dettagli sull'uso e sul contenuto sono disponibili nel paragrafo "Supporto multiutente" più avanti in questo capitolo).

Accesso alle tabelle

Passiamo ora alla relazione tra le tabelle e i dati nel database degli account e ai metodi più importanti di `AccountManagerService`. La relazione, se esaminata superficialmente, è piuttosto diretta (se ignoriamo il caching e la sincronizzazione): i metodi che recuperano o manipolano i dettagli degli account accedono alla tabella `accounts`, mentre i metodi che gestiscono i dati utente associati a un account accedono alla tabella `extras`. Le API che gestiscono i token di autenticazione accedono alla tabella `authtokens` e salvano i grant di accesso ai token per ogni applicazione nella tabella `grants`. I metodi e i dati a cui accedono sono descritti più avanti.

Quando si aggiunge un account di un tipo specifico, chiamando uno dei metodi `addAccount()`, `AccountManagerService` inserisce nella tabella `accounts` una riga contenente tipo, nome utente e password. La chiamata a uno dei metodi `getPassword()`, `setPassword()` o `clearPassword()` fa sì che `AccountManagerService` acceda o aggiorni la colonna `password` della tabella `accounts`. Se si ricevono o impostano dati utente per l'account utilizzando i metodi `getUserdata()` o `setUserdata()`, `AccountManagerService` recupera (o salva) la voce corrispondente nella tabella `extras`.

Quando si richiede un token per un account specifico, le cose diventano un po' più complesse. Se non è mai stato rilasciato prima un token del tipo specificato, `AccountManagerService` mostra una finestra di conferma (Figura 8.2) che chiede all'utente di approvare l'accesso per l'applicazione richiedente. Se l'utente accetta, l'UID dell'app richiedente e il tipo di token vengono salvati nella tabella `grants` (gli autenticatori

possono dichiarare l'uso di token personalizzati impostando l'attributo dei metadati dell'account `customTokens` su `true`; in questo caso, sono responsabili della gestione dei token e Android non mostra la finestra di accesso al token né salva automaticamente i token nella tabella `authtokens`). Se esiste già un grant, `AccountManagerService` cerca i token corrispondenti alla richiesta nella tabella `authtokens`. Se ne esiste uno valido, viene restituito il token; se invece non viene trovato un token corrispondente, `AccountManagerService` trova l'autenticatore per il tipo di account specificato nella cache e chiama il suo metodo `getAuthToken()` per richiedere un token. Di solito questa operazione richiede che l'autenticatore recuperi nome utente e password dalla tabella `accounts` (con il metodo `getPassword()`) e chiami il servizio online relativo per ottenere un nuovo token. Una volta restituito un token, questo viene salvato nella tabella `authtokens` e restituito all'app richiedente (di solito in maniera asincrona con un callback). L'invalidazione di un token provoca l'eliminazione della riga relativa dalla tabella `authtokens`. Infine, quando viene rimosso un account chiamando il metodo `removeAccount()`, la riga relativa viene eliminata dalla tabella `accounts` e un trigger del database elimina tutte le righe collegate dalle tabelle `authtokens`, `extras` e `grants`.

Sicurezza mediante password

Le credenziali (di solito nomi utente e password) sono archiviate in un database centrale in */data/system/* accessibile solo alle applicazioni di sistema, ma non sono crittografate; la crittografia o comunque la protezione delle credenziali è lasciata al modulo autenticatore, che la implementa secondo necessità. In effetti, se avete un dispositivo rooted, vedrete facilmente che visualizzando il contenuto della tabella `accounts` potrete leggere alcune password in chiaro, soprattutto per l'applicazione stock Email (package `com.android.email` o `com.google.android.email`). Per esempio, nel Listato 8.2, le stringhe `password` **(2)** e `pass1234` **(3)** sono le password in chiaro per un account POP usato dall'applicazione stock e per un account personalizzato `com.example.account`.

NOTA

Le applicazioni e-mail potrebbero dover archiviare la password, anziché un hash della password o un token di autenticazione, per supportare diversi metodi di autenticazione challenge-response che accettano in input la password, come DIGEST-MD5 e CRAM-MD5.

Visto che il metodo `AccountManager.getPassword()` può essere chiamato solo dalle app con lo stesso UID dell'autenticatore dell'account, le password in chiaro non sono accessibili alle altre applicazioni in fase di runtime, ma possono essere incluse nei backup o nei dump del dispositivo. Per evitare questo rischio per la sicurezza, le applicazioni possono crittografare le password con una chiave specifica per il dispositivo, oppure possono decidere di sostituire la password con un token master revocabile dopo l'autenticazione iniziale. Per esempio, il client Twitter ufficiale non memorizza la password utente nella tabella `accounts`, ma solo i token di autenticazione ottenuti (nella tabella `authtokens`). Gli account Google sono un altro valido esempio (tipo di account `com.google`): come vedremo nel paragrafo "Servizio di login Google", invece della password utente gli account Google salvano un token master che viene scambiato con i token di autenticazione per i servizi specifici.

Supporto multiutente

Nel Capitolo 4 abbiamo visto che, sui dispositivi multiutente, Android consente a ogni utente di avere un proprio set di applicazioni, dati delle applicazioni e impostazioni di sistema. Questo isolamento degli utenti si estende anche agli account online: gli utenti possono infatti registrare account personali nel servizio di gestione degli account del sistema. Android 4.3 ha aggiunto il supporto per i profili con restrizioni, che corrispondono a utenti non del tutto indipendenti che condividono le applicazioni installate con l'utente primario. Inoltre, ai profili con restrizioni si possono applicare numerose limitazioni. Le app che usano le API di `AccountManager` possono aggiungere il supporto esplicito per i profili con restrizioni, permettendo loro di vedere e utilizzare un sottoinsieme degli account dell'utente primario nelle app supportate. Questa funzionalità è descritta nei dettagli nel paragrafo "Account condivisi".

Nei paragrafi seguenti vedremo come Android implementa l'isolamento e la condivisione degli account sui dispositivi multiutente.

Database degli account per utente

Come segnalato nel paragrafo "Database degli account", il database usato da `AccountManagerServices` per salvare le informazioni sugli account e i token di autenticazione viene salvato nella directory di sistema di ogni utente in */data/system/users/<ID utente>/accounts.db*. In questo modo ogni utente può avere un archivio dedicato e utenti diversi possono gestire istanze separate dello stesso tipo di account online. A parte la posizione del database, tutto il resto funziona esattamente come visto per l'utente proprietario, compresi permessi, grant di accesso e così via. Quando viene rimosso un utente, il sistema elimina tutti i suoi dati, compreso il database degli account.

Account condivisi

Gli account dell'utente primario vengono condivisi con un profilo con restrizioni clonando i dati dell'account nel database degli account del profilo con restrizioni. Di conseguenza, i profili con restrizioni non accedono direttamente ai dati degli account dell'utente primario, ma ne possiedono una copia. Quando viene aggiunto un nuovo profilo con restrizioni, il nome e il tipo di tutti gli account correnti dell'utente primario vengono copiati nella tabella `shared_accounts` del database degli account del profilo con restrizioni. Tuttavia, poiché il nuovo utente non è ancora stato avviato, la tabella `accounts` è ancora vuota e gli account condivisi non sono ancora utilizzabili.

La tabella `shared_accounts` ha la stessa struttura della tabella `accounts` ma senza la colonna `password`. È simile al Listato 8.6 per un profilo con restrizioni.

Listato 8.6 Contenuto della tabella shared_accounts.

```
sqlite> select * from shared_accounts;
_id|name             |type
1  |user1@gmail.com  |com.google
2  |user1@example.com|com.google.android.pop3
3  |example_user     |com.example.account
```

Gli account condivisi non vengono clonati direttamente copiando i dati dalla tabella `accounts` del proprietario; la clonazione viene invece eseguita tramite l'autenticatore che ha dichiarato l'account. Per impostazione predefinita `AbstractAccountAuthenticator`, da cui derivano tutte le classi di autenticazione, non supporta la clonazione degli account.

Le implementazioni che intendono supportare gli account condivisi per i profili con restrizioni devono farlo esplicitamente, sostituendo un paio di metodi introdotti in Android 4.3 insieme al supporto per questo tipo di profili: `getAccountCredentialsForCloning()`, che restituisce un `Bundle` contenente tutti i dati necessari per clonare l'account, e `addAccountFromCredentials()`, che riceve questo `Bundle` come parametro ed è responsabile della creazione dell'account in base alle credenziali in `Bundle`. `AccountManagerService` ritarda la clonazione di un account condiviso fino all'avvio vero e proprio dell'utente con restrizioni. Se il proprietario aggiunge nuovi account, questi vengono inseriti nella tabella `shared_accounts` e clonati in maniera analoga.

Anche se gli account vengono clonati correttamente, potrebbero non essere disponibili per un'applicazione avviata da un profilo con restrizioni. Nel Capitolo 4 abbiamo visto che, se un'applicazione vuole supportare gli account condivisi, deve dichiarare esplicitamente il tipo di account di cui necessita usando l'attributo `restrictedAccountType` del tag manifest `<application>`. `AccountManagerServices` usa il valore dell'attributo `restrictedAccountType` per filtrare gli account prima di passarli alle applicazioni in esecuzione con un profilo con restrizioni. Attualmente, un'applicazione può dichiarare un solo tipo di account con questo attributo.

NOTA

Gli utenti secondari non condividono gli account con il proprietario; di conseguenza le loro tabelle `shared_accounts` sono sempre vuote e gli account proprietario non vengono mai clonati.

Aggiunta di un modulo autenticatore

Nel paragrafo "Moduli autenticatori" abbiamo visto che un modulo autenticatore è un servizio associato che implementa l'interfaccia AIDL

`android.accounts.IAccountAuthenticator` e che può essere sottoposto a binding usando l'azione di intent `android.accounts.AccountAuthenticator`. In questo paragrafo vedremo come un'applicazione può implementare e dichiarare un modulo autenticatore.

La maggior parte della logica dell'autenticatore, tra cui l'aggiunta di account, la verifica delle credenziali fornite dall'utente e il recupero dei token di autenticazione, è implementata in una classe autenticatore derivata dalla classe base fornita da Android, ovvero `AbstractAccountAuthenticator` (http://bit.ly/1vyB4kS). La classe autenticatore deve fornire l'implementazione di tutti i metodi astratti, ma se non sono necessarie tutte le funzionalità, i metodi implementati possono restituire `null` o generare `UnsupportedOperationException`. Per salvare la password dell'account, un'implementazione deve implementare almeno il metodo `addAccount()` e visualizzare una finestra che richieda la password all'utente. La password può essere aggiunta esplicitamente al database degli account chiamando il metodo `addAccountExplicitly()` di `AccountManager`. Le activity che implementano la raccolta delle credenziali e il login possono estendersi da `AccountAuthenticatorActivity` (http://bit.ly/1way0rV), che fornisce un metodo comodo per passare le credenziali ottenute ad `AccountManager`.

NOTA

Non dimenticate che il metodo `addAccountExplicitly()` non crittografa né protegge in altro modo la password, che per impostazione predefinita viene salvata come testo normale. Se necessario, la crittografia deve essere implementata separatamente e la password o il token crittografato devono essere passati ad `addAccountExplicitly()` al posto della versione in chiaro.

Una volta disponibile un'implementazione dell'autenticatore di account, dovete creare semplicemente un servizio che restituisca la sua interfaccia Binder quando viene richiamato con l'azione di intent `android.accounts.AccountAuthenticator`, come mostrato nel Listato 8.7 (dove sono state omesse le implementazioni del metodo `AbstractAccountAuthenticator`).

Listato 8.7 Implementazione del servizio autenticatore degli account.

```
public class ExampleAuthenticatorService extends Service {

    public static class ExampleAuthenticator extends
            AbstractAccountAuthenticator{
        // ...
    }

    private ExampleAuthenticator authenticator;
```

```
    @Override
    public void onCreate() {
        super.onCreate();
        authenticator = new ExampleAuthenticator(this);
    }

    @Override
    public IBinder onBind(Intent intent) {
        if (AccountManager.ACTION_AUTHENTICATOR_INTENT.equals(intent.
                getAction())) {
            return authenticator.getIBinder();
        }
        return null;
    }
}
```

Per essere prelevato da `AccountAuthenticatorCache` e messo a disposizione tramite `AccountManagerService`, il servizio deve dichiarare l'azione di intent `android.accounts.AccountAuthenticator` e i metadati corrispondenti, come mostrato nel Listato 8.8. Al manifest devono essere inoltre aggiunti i permessi necessari per accedere ad account e token. In questo esempio aggiungiamo solamente il permesso `AUTHENTICATE_ACCOUNTS`, il minimo richiesto per poter aggiungere un account con `addAccountExplicitly()`.

Listato 8.8 Dichiarazione di un servizio autenticatore degli account in AndroidManifest.xml.

```
<?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" >

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

    <application ...>
        --altro codice--
        <service android:name=".ExampleAuthenticatorService" >
            <intent-filter>
                <action android:name="android.accounts.AccountAuthenticator" />
            </intent-filter>

            <meta-data
                android:name="android.accounts.AccountAuthenticator"
                android:resource="@xml/authenticator" />
        </service>
    </application>
</manifest>
```

Infine, è necessario dichiarare il tipo di account, l'etichetta e le icone nel file di risorse XML referenziato, come mostrato nel Listato 8.9. Qui il tipo di account è `com.example.account` e si utilizza semplicemente l'icona dell'app come icona dell'account.

Listato 8.9 Dichiarazione dei metadati degli account in un file di risorse XML.

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="com.example.account"
    android:label="@string/account_label"
```

```
    android:icon="@drawable/ic_launcher"
    android:smallIcon="@drawable/ic_launcher"/>
```

Dopo l'installazione dell'applicazione che dichiara il nostro nuovo account, gli account `com.example.account` possono essere aggiunti tramite l'API `AccountManager` API o l'app *Settings* di sistema selezionando *Add an account*. Il nuovo account sarà visibile nell'elenco di account supportati, come mostrato nella Figura 8.3.

Gli account personalizzati possono essere usati solo per comodità dall'applicazione dichiarante, oppure durante la creazione di un sync adapter che richiede un account dedicato. Per consentire alle applicazioni di terze parti di autenticarsi utilizzando l'account personalizzato, è necessario implementare i token di autenticazione, perché (come abbiamo visto in "Elenco e autenticazione degli account") le applicazioni di terze parti non possono accedere alla password di un account tramite l'API `AccountManager.getPassword()`, a meno che non siano firmate con la stessa chiave e lo stesso certificato dell'applicazione che ospita il modulo autenticatore dell'account target.

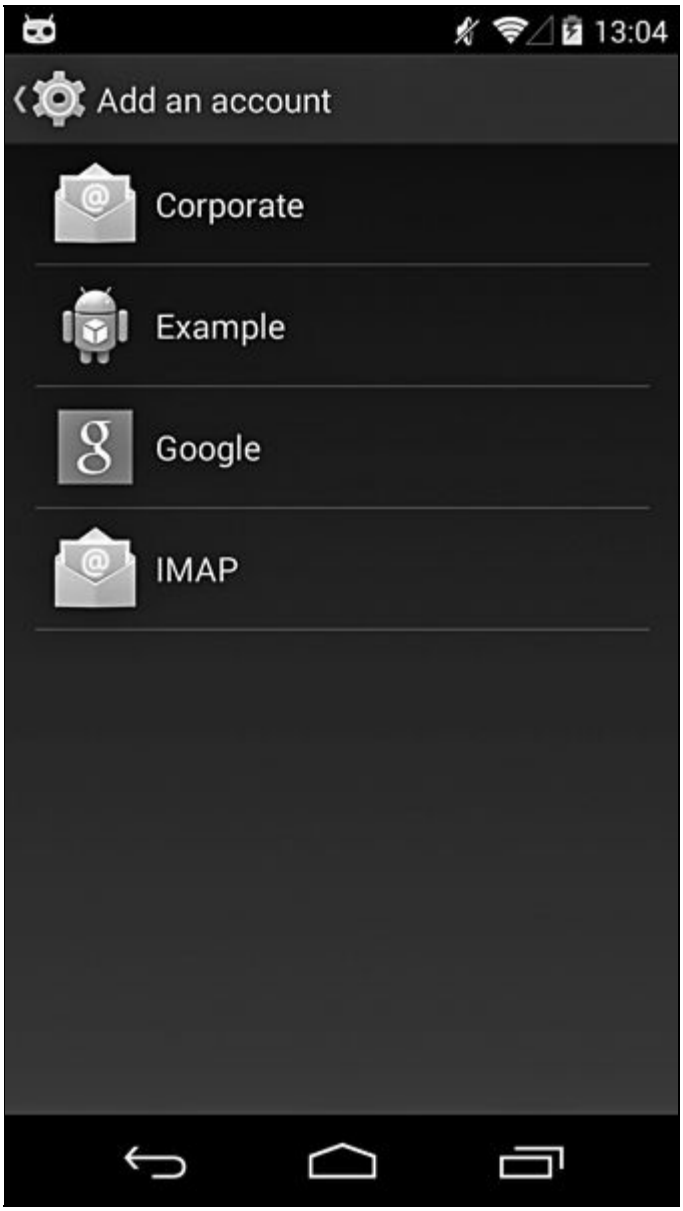

Figura 8.3 Aggiunta di un account personalizzato tramite Settings.

Supporto per gli account Google

L'obiettivo principale del sistema di gestione degli account di Android è l'integrazione agevole dei servizi online nel sistema operativo e la possibilità di accesso ininterrotto ai dati degli utenti attraverso la sincronizzazione in background. Le prime versioni del servizio di gestione degli account di sistema erano pensate per supportare l'integrazione di Android con i servizi online Google; solo in un secondo momento il servizio è stato separato e reso parte del sistema operativo. Nelle versioni di Android dalla 2.0 in poi, il supporto per gli account i servizi online Google è fornito da un set di componenti, comprendenti autenticatori degli account (per il tipo di account `com.google`) e sync adapter (per Gmail, Calendar, contatti e così via), che usa le API standard del sistema operativo. Esistono tuttavia alcune differenze degne di nota rispetto ai moduli autenticatori e ai sync adapter di terze parti.

- I componenti degli account Google sono integrati nel sistema e di conseguenza ricevono permessi supplementari.
- Molte funzionalità sono implementate sul lato server.
- L'autenticatore dell'account non salva le password in chiaro sul dispositivo.

Servizio di login Google

I due componenti principali che implementano il supporto di account e servizi Google sono *Google Services Framework* (GSF) e *Google Login Service* (GLS, indicato come *Google Account Manager* nelle versioni più recenti). Il primo offre servizi comuni a tutte le app Google, quali impostazioni centralizzate e gestione del "feature toggle", il secondo implementa il provider di autenticazione per gli account Google ed è l'argomento principale di questo paragrafo.

Google propone numerosi servizi online e supporta diversi metodi per l'autenticazione con questi servizi, sia tramite un'UI web rivolta all'utente sia mediante varie API di autenticazione dedicate. Il servizio di login

Google per Android, però, non chiama direttamente queste API di autenticazione pubbliche, ma vi accede tramite un servizio online dedicato recuperabile all'indirizzo `https://android.clients.google.com`. Dispone di endpoint per l'autenticazione, il rilascio di token di autorizzazione e vari feed di dati (posta, calendario e così via) utilizzati per la sincronizzazione dei dati.

La maggior parte della logica di autenticazione e autorizzazione è implementata sul lato server, ma sono richieste anche credenziali salvate in locale, soprattutto per la sincronizzazione in background. La gestione delle credenziali sul dispositivo è uno dei servizi offerti da GLS; anche se attualmente non sono disponibili al pubblico né il codice sorgente né la documentazione di riferimento, possiamo osservare quali dati vengono memorizzati da GLS sul dispositivo e dedurre da questi come viene implementata l'autenticazione.

Come già affermato, GLS si inserisce nel framework degli account di sistema, pertanto le credenziali, i token e i dati supplementari associati vengono salvati nel database degli account di sistema per l'utente corrente, come avviene anche per gli altri tipi di account. A differenza di molte altre applicazioni, però, GLS non memorizza direttamente le password degli account Google: al loro posto, GLS salva un token master opaco (presumibilmente una forma di token di aggiornamento OAuth) nella colonna `password` della tabella `accounts` e scambia tale token con i token di autenticazione per i diversi servizi Google chiamando un endpoint del servizio web associato. Il token viene ottenuto alla prima aggiunta di un account Google al dispositivo inviando il nome utente e la password inseriti nell'activity di sign-in mostrata nella Figura 8.4.

Se l'account Google di destinazione usa il metodo di autenticazione predefinito basato solo sulla password, e se viene inserita la password corretta, il servizio online GLS restituisce il token master e l'account viene aggiunto al database degli account dell'utente. Tutte le richieste di autenticazione successive usano il token master per ottenere token specifici per ambito o servizio, utilizzati per la sincronizzazione o per il login web automatico. Se l'account Google è impostato per usare

l’autenticazione a due fattori (2FA), all’utente viene richiesto di inserire la sua password monouso (OTP, *One-Time Password*, chiamata *codice di verifica* nell’UI web) in una vista web incorporata come quella mostrata nella Figura 8.5.

Se l’OTP viene verificata correttamente, il token master viene aggiunto al database degli account, e viene mostrato un elenco dei servizi che supportano la sincronizzazione in background (Figura 8.6). Per gli account Google per cui è abilitato 2FA cambia solo il processo di login iniziale; tutte le richieste di autenticazione successive usano il token master nella cache e non richiedono l’inserimento di una OTP. Di conseguenza, dopo l’archiviazione nella cache, il token master consente accesso completo a un account Google e può essere usato non solo per la sincronizzazione dei dati, ma anche per altri tipi di accesso all’account, compreso il login web.

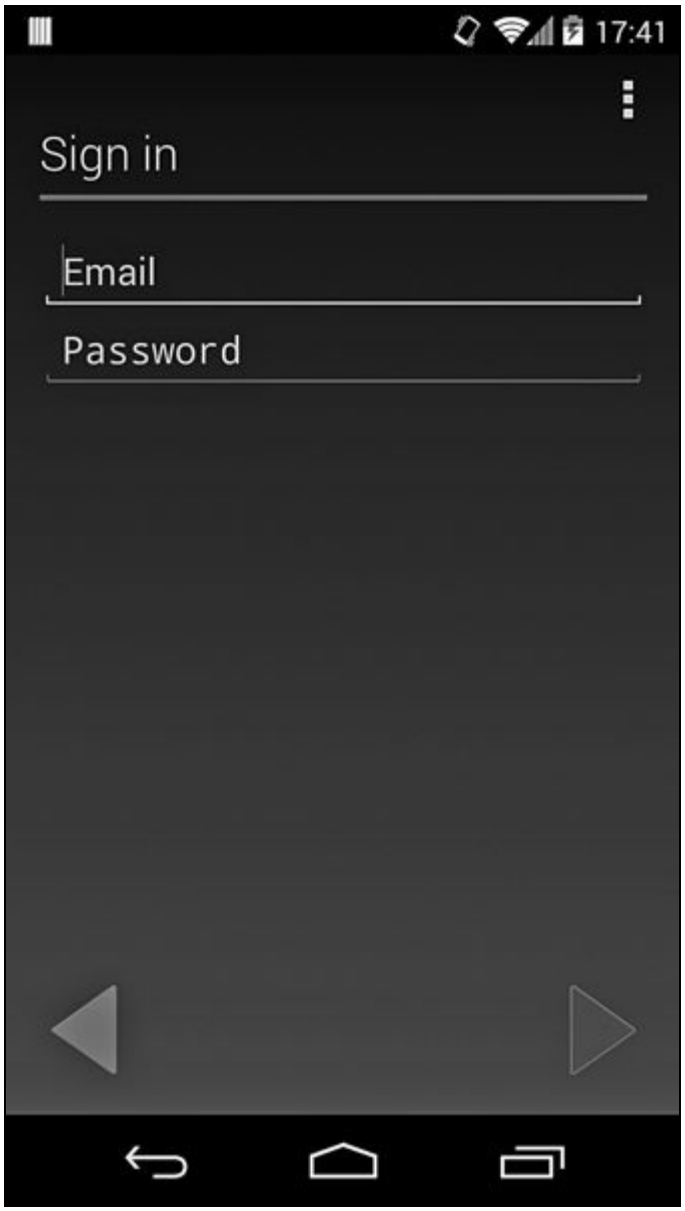

Figura 8.4 Activity di sign-in per gli account Google.

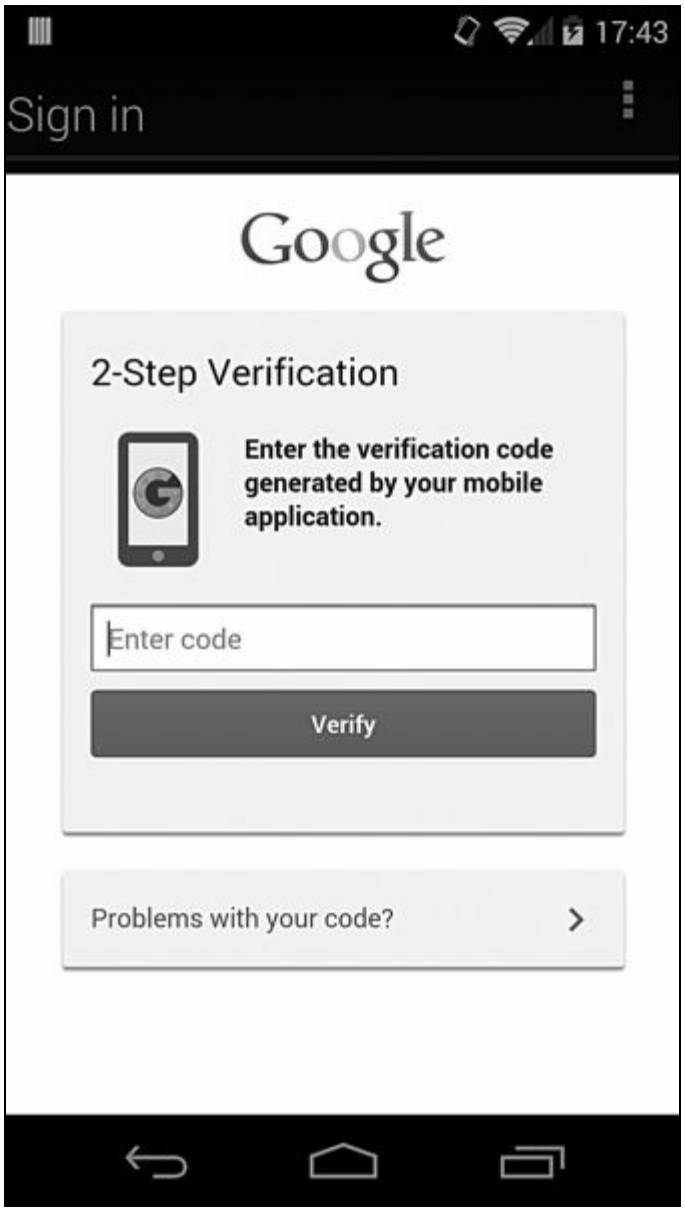

Figura 8.5 Password monouso inserita durante l'aggiunta di un account Google.

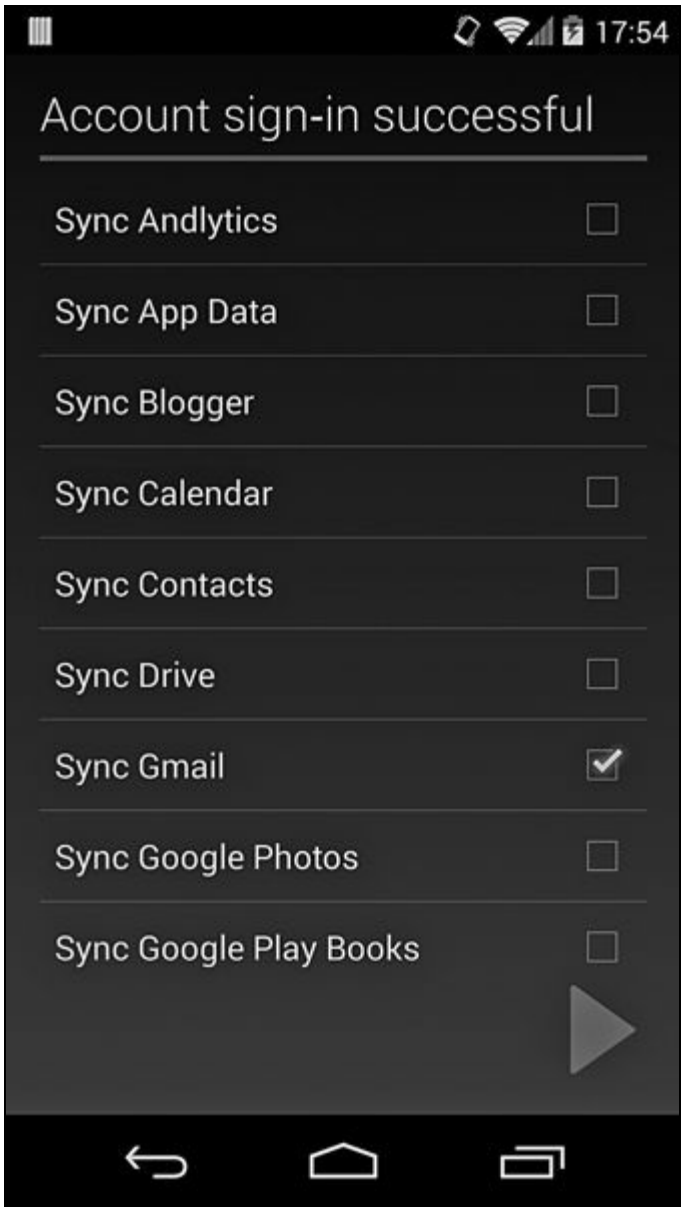

Figura 8.6 Elenco di servizi Google che supportano la sincronizzazione in background.

Anche se è davvero comodo avere un potentissimo token di autenticazione salvato nella cache, questo compromesso ha permesso il proliferare di diversi attacchi agli account Google; per questo molti servizi Google oggi richiedono un'autenticazione supplementare quando vengono visualizzati dati sensibili o vengono cambiate le impostazioni degli account. Il token master può essere invalidato cambiando la password dell'account Google, abilitando l'autenticazione a due fattori o rimuovendo il dispositivo Android dalla pagina *Account Permissions* dell'account Google associato (Figura 8.7). Tutte queste azioni richiedono all'utente di riautenticarsi con le sue nuove credenziali sul

dispositivo al successivo tentativo di recuperare un token di autenticazione Google tramite l'API `AccountManager`.

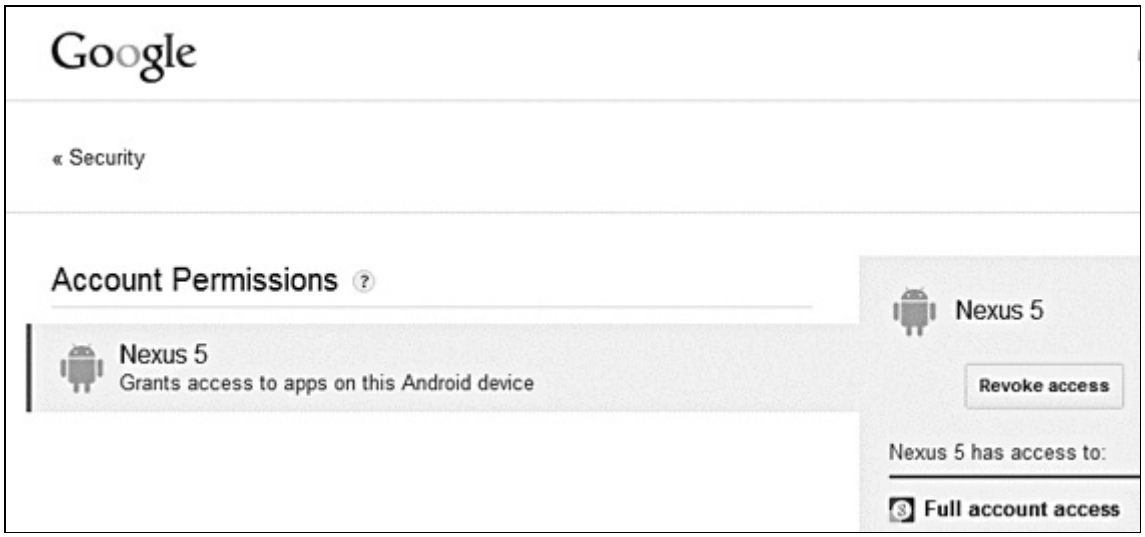

Figura 8.7 Inserimento del dispositivo Android nella pagina Account Permissions di un account Google.

Autorizzazione e autenticazione per i servizi Google

Oltre ai servizi online rivolti all'utente e dotati di UI web, come Gmail, Google Calendar e naturalmente la ricerca, Google permette di accedere in via programmatica a molti dei suoi servizi con API web differenti. La maggior parte di queste richiede l'autenticazione, sia per accedere a un sottoinsieme dei dati di un utente specifico, sia per ragioni di quote e fatturazione. Negli anni sono stati utilizzati diversi metodi di autenticazione e autorizzazione standard o di Google; la tendenza attuale vede la migrazione a OAuth 2.0 (D. Hardt, *The OAuth 2.0 Authorization Framework*, http://tools.ietf.org/html/rfc6749) e a OpenID Connect (N. Sakimura *et al.*, *OpenID Connect Core 1.0*, http://openid.net/specs/openid-connect-core-1_0.html). Tuttavia, molti servizi usano ancora i vecchi protocolli proprietari, quindi vale la pena parlarne brevemente.

La maggior parte dei protocolli di autenticazione è disponibile in due varianti: una per le applicazioni web e una per le applicazioni installate. Le applicazioni web vengono eseguite in un browser e si prevede che possano sfruttare tutte le funzionalità standard dello stesso, quali l'UI avanzata, l'interazione libera con l'utente, l'archiviazione dei cookie e la

capacità di seguire i redirect. Le applicazioni installate, invece, non presentano un modo nativo di preservare le informazioni sulla sessione e potrebbero non disporre di tutte le funzionalità di un browser. Le applicazioni Android native per la maggior parte appartengono alla categoria delle applicazioni installate; vediamo quindi quali protocolli sono disponibili per esse.

ClientLogin

Il più antico e attualmente il più utilizzato protocollo di autorizzazione per le applicazioni installate è *ClientLogin* (http://bit.ly/101pou2). Questo protocollo presume che l'applicazione abbia accesso al nome dell'account e alla password dell'utente e permette di ottenere un token di autorizzazione per un servizio specifico, che può essere salvato e usato per accedere al servizio per conto dell'utente. I servizi sono identificati dai nomi proprietari, come *cl* per Google Calendar e *ah* per l'engine Google App. Un elenco dei numerosi servizi supportati è disponibile nella guida Google Data API (http://bit.ly/ZG5fc1); mancano però alcuni servizi specifici di Android, vale a dire *ac2dm*, *android*, *androidsecure*, *androiddeveloper* e *androidmarket*.

I token di autorizzazione per questi servizi possono essere di lunga durata (fino a due settimane), ma non possono essere aggiornati e l'applicazione deve quindi ottenere un nuovo token alla scadenza di quello attuale. Purtroppo, non c'è modo di convalidare un token senza accedere al servizio associato: se ottenete uno stato HTTP *OK* (200) il token è valido, ma se viene restituito 403 è necessario consultare il codice di errore e riprovare o recuperare un nuovo token.

Un altro limite dei token di autorizzazione ClientLogin è la mancata offerta di un accesso granulare alle risorse di un servizio: l'accesso è del tipo "tutto o niente" e non è quindi possibile specificare l'accesso in sola lettura o l'accesso unicamente a una risorsa specifica. Il più grande svantaggio delle app mobili è il fatto che ClientLogin richiede l'accesso alla password utente effettiva: di conseguenza, a meno che non vogliate costringere gli utenti a inserire la loro password ogni volta che è richiesto

un token, è necessario salvare la password sul dispositivo (causando vari problemi e possibili rischi per la sicurezza). Android evita di salvare la password raw memorizzando un token master sul dispositivo e utilizzando GLS e i servizi online associati per scambiare il token master con i token ClientLogin. Per recuperare un token è sufficiente chiamare il metodo `AccountManager` appropriato, che restituisce un token nella cache o genera una richiesta API per recuperarne uno nuovo.

Nonostante le numerose limitazioni, il protocollo ClientLogin è facile da comprendere e implementare, e pertanto è molto diffuso. È tuttavia stato ufficialmente deprecato nell'aprile 2012, quindi le app che lo utilizzano sono invitate a passare a OAuth 2.0.

OAuth 2.0

Il framework di autorizzazione OAuth 2.0 è divenuto a fine 2012 lo standard Internet ufficiale. Definisce *flussi di autorizzazione* diversi per casi di utilizzo diversi, ma presentarli tutti è impossibile: parleremo quindi solo della relazione di OAuth 2.0 con le applicazioni mobili native (per maggiori dettagli sul protocollo vero e proprio, consultate RFC 6749).

La specifica OAuth 2.0 definisce quattro flussi di base per ottenere un token di autorizzazione per una risorsa. Definisce inoltre due flussi che non richiedono il client (nel nostro scenario un'app Android) per gestire direttamente le credenziali utente (come il nome utente e la password dell'account Google), nello specifico il flusso di *grant del codice di autorizzazione* e il flusso di *grant implicito*. Entrambi richiedono che il server di autorizzazione (di Google) autentichi il proprietario della risorsa (l'utente dell'app Android) per stabilire se concedere o negare l'accesso (per esempio l'accesso in sola lettura alle informazioni del profilo). In una tipica applicazione web basata su browser, l'operazione è semplice: l'utente viene reindirizzato a una pagina di autenticazione e quindi a una pagina di grant dell'accesso che fondamentalmente chiede "Consentire all'app X di accedere ai dati Y e Z?". Se l'utente accetta un altro redirect, che include un token di autorizzazione, riporta l'utente all'applicazione

originale. Il browser deve semplicemente passare il token alla richiesta successiva per ottenere l'accesso alla risorsa target.

Le cose non sono così semplici per le app native. Un'app nativa può utilizzare il browser di sistema per gestire la fase di grant del permesso, oppure incorporare una `WebView` o un controllo simile nella finestra dell'app. L'uso del browser di sistema richiede l'avvio di un'applicazione di terze parti (il browser), la rilevazione dell'esito dell'operazione e l'individuazione di un modo per restituire il token all'applicazione chiamante. La `WebView` è più facile da usare per gli utenti, perché non chiede di spostarsi avanti e indietro tra le applicazioni, ma provoca comunque la visualizzazione di una UI web non nativa e richiede codice complesso per il rilevamento dell'esito e l'estrazione del token di accesso. Nessuna opzione è quindi ideale ed entrambe causano confusione nell'utente.

Questa complessità di integrazione e la mancata corrispondenza delle UI sono i problemi che il supporto di OAuth 2.0 tramite API Android native prova a risolvere. Android offre due API utilizzabili per ottenere token OAuth 2.0: la piattaforma `AccountManager` tramite la sintassi speciale per il tipo di token `oauth2:scope` e Google Play Services (di cui parliamo nel prossimo paragrafo). Quando si utilizza una di queste API per recuperare un token, l'autenticazione utente viene implementata in maniera trasparente passando il token master salvato al componente lato server di GLS, che produce la finestra di dialogo per il grant di accesso `AccountManager` nativa (Figura 8.8) al posto di una `WebView` con una pagina di grant del permesso. Se si concede l'accesso al token all'applicazione richiedente, viene inviata una seconda richiesta per la trasmissione al server, che restituisce il token richiesto. Il token di accesso viene recapitato direttamente all'app, senza passare attraverso un intermediario quale la `WebView`. Il flusso è lo stesso visto per le applicazioni web, tranne per il fatto che non richiede il context switching dall'applicazione nativa al browser (e ritorno) ed è molto più facile da usare. Naturalmente, questo flusso di autorizzazione nativo è utilizzabile solo per gli account Google, pertanto la scrittura di un client per altri servizi online che usano OAuth 2.0 richiede ancora l'integrazione della relativa interfaccia web nell'app. Per

esempio, i client Twitter spesso usano `WebView` per elaborare l'URL di callback per il grant di accesso restituito dall'API Twitter.

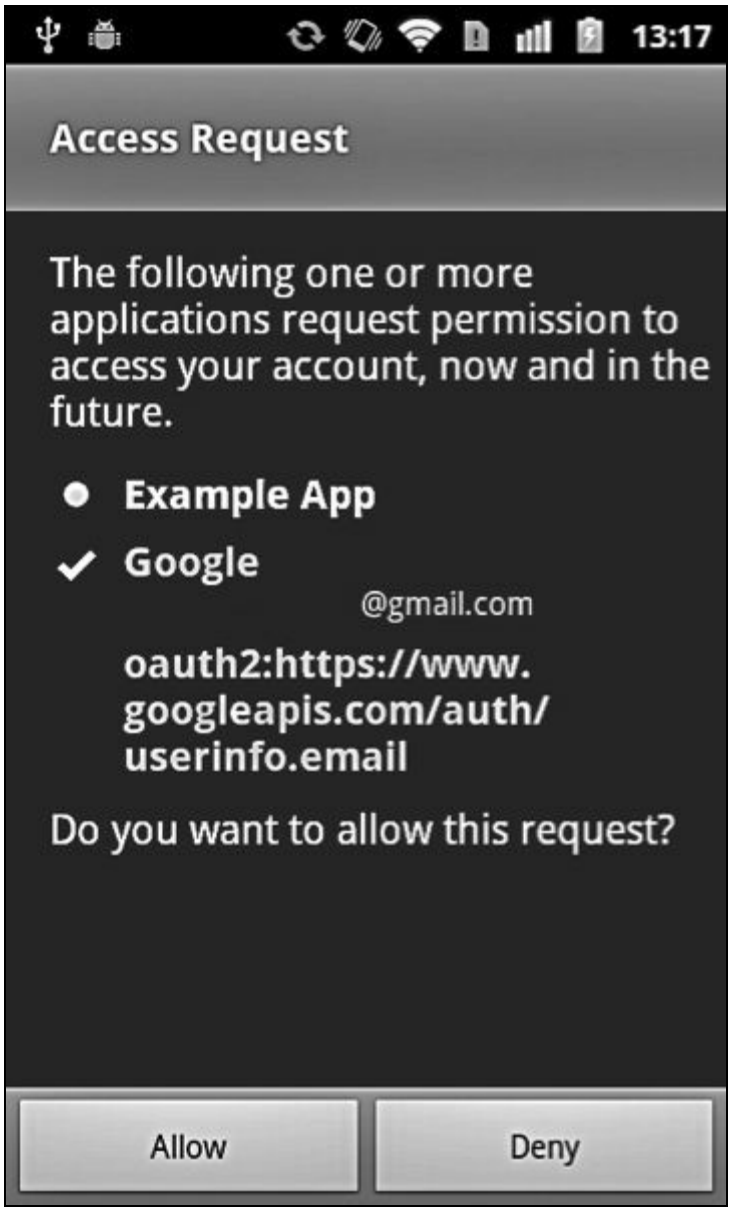

Figura 8.8 Finestra di richiesta dell'accesso al token OAuth.

Google Play Services

Google Play Services (GPS) è stato presentato al Google I/O 2012 come una piattaforma di facile utilizzo che offre alle app Android di terze parti un mezzo per l'integrazione con i prodotti Google (http://bit.ly/ZG5h3E). Da allora, è cresciuto fino a divenire un package all-in-one enorme (con oltre 14.000 metodi Java!) che offre l'accesso alle API Google e alle estensioni proprietarie del sistema operativo.

Come affermato nel paragrafo precedente, il recupero di token OAuth 2.0 tramite l'interfaccia `AccountManager` standard è supporto da Android 2.2 e versioni successive, ma non funziona in maniera affidabile nelle varie build Android, in quanto queste integrano versioni diverse di GLS che causano comportamenti leggermente differenti tra i dispositivi. Inoltre, la finestra di grant dei permessi visualizzata nella fase di richiesta di un token non era particolarmente facile da usare, perché in alcuni casi mostrava l'ambito OAuth 2.0 raw, poco significativo per la maggior parte degli utenti (Figura 8.8). Anche se in parte erano supportati alias leggibili per alcuni ambiti (per esempio, in alcune versioni veniva visualizzata la stringa *Manage your tasks* al posto dell'ambito OAuth raw `oauth2:https://www.googleapis.com/auth/tasks`), tale soluzioni non era né ideale né universalmente disponibile, in quanto dipendeva troppo dalla versione di GLS preinstallata.

In generale, anche se il framework di gestione degli account di Android è ben integrato nel sistema operativo ed è estensibile tramite i moduli autenticatori di terze parti, la sua API non è granché flessibile e l'aggiunta del supporto per i flussi di autenticazione o autorizzazione in più fasi, come quelli usati in OAuth 2.0, è tutt'altro che semplice. GPS mira a raggiungere questo risultato con l'aiuto di un servizio online, che fa del suo meglio per celare la complessità di OAuth 2.0 e fornisce API web compatibili con il framework di gestione degli account di Android. Di seguito vengono presentati i dettagli di questa integrazione.

GPS aggiunge supporti universali per visualizzare una destinazione d'ambito semplice da usare, trasformando il rilascio dei token in un processo in due passaggi.

1. Come in precedenza, la prima richiesta include il nome dell'account, il token master e il servizio richiesto nel formato *oauth2:scope*. GPS aggiunge due nuovi parametri alla richiesta: il nome del package dell'app e l'hash SHA-1 del suo certificato di firma. La risposta include alcuni dettagli leggibili sull'ambito richiesto e sull'applicazione richiedente, che GPS mostra in una finestra di grant dei permessi come quella nella Figura 8.9.

2. Se l'utente concede il permesso, la decisione viene registrata nella tabella `extras` in un formato proprietario che include il nome del package dell'app richiedente, l'hash del certificato di firma e l'ambito OAuth 2.0 concesso (notate che la tabella `grants` non viene utilizzata). GPS invia quindi di nuovo la richiesta di autorizzazione, impostando il parametro `has_permission` a 1. In caso di esito positivo, nella risposta vengono inseriti un token OAuth 2.0 e la relativa data di scadenza: quest'ultima viene salvata nella tabella `extras`, mentre il token viene inserito nella tabella `authtokens` in un formato simile.

L'app GPS usa lo stesso user ID condiviso dei package GSF e GLS (`com.google.uid.shared`), pertanto può interagire direttamente con questi servizi: può quindi, tra le altre cose, recuperare e scrivere direttamente i token e le credenziali dell'account Google nel database degli account. Come previsto, GPS viene eseguito in un servizio remoto a cui si può accedere da una libreria client collegata alle app che usano GPS. Il punto di forza principale rispetto all'API legacy `AccountManager` è il fatto che, anche se i suoi moduli autenticatori sottostanti (GLS e GSF) sono parte del sistema (e in quanto tali non possono essere aggiornati senza un OTA), GPS è un'app installabile dall'utente che può essere aggiornata facilmente tramite Google Play. In effetti l'aggiornamento avviene automaticamente, quindi gli sviluppatori delle app non devono fare affidamento sul fatto che gli utenti si occupino dell'aggiornamento qualora vogliano utilizzare funzionalità nuove (a meno che GPS non sia stato del tutto disabilitato).

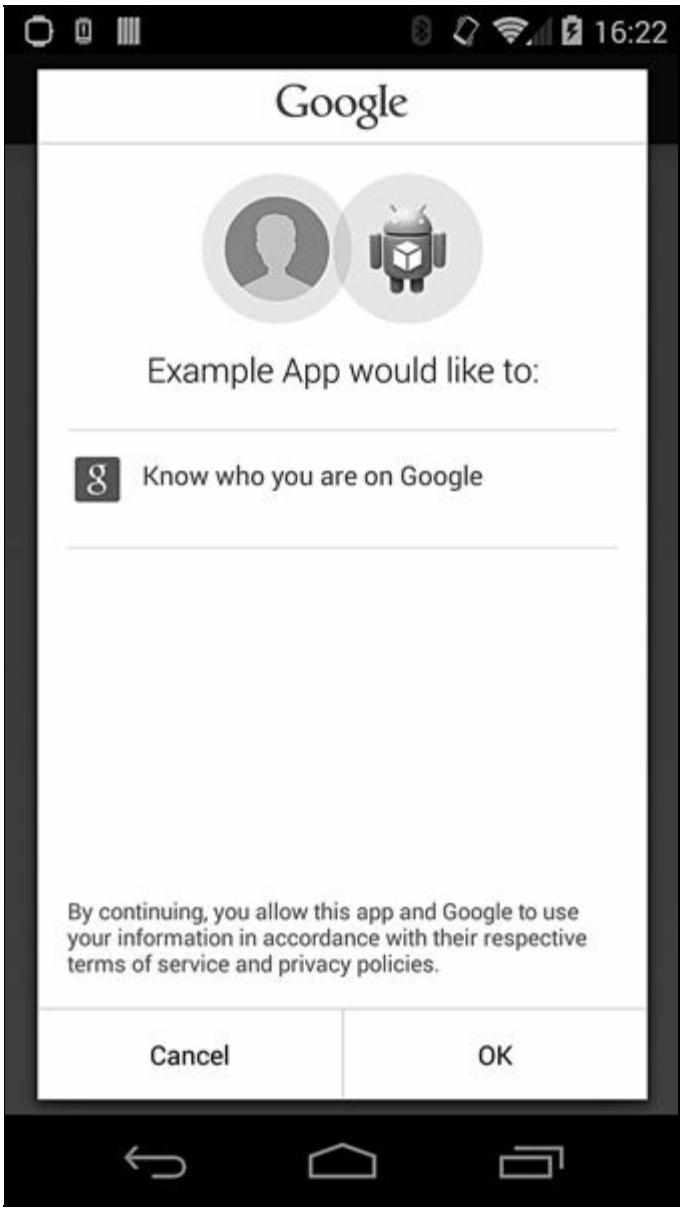

Figura 8.9 Finestra dei permessi di accesso all'account di Google Play Services.

Questo meccanismo di aggiornamento è studiato per garantire l'agilità nella distribuzione di nuove funzionalità della piattaforma; tuttavia, poiché GPS è stato pensato per integrare funzionalità e API diverse che richiedono grandi quantità di test, gli aggiornamenti finora sono stati rari. Detto questo, se la vostra app usa i token OAuth 2.0 per autenticare le API di Google (è questo il metodo attualmente preferito), dovreste prendere in considerazione l'uso di GPS al di sopra dell'accesso ad `AccountManager` "raw".

NOTA

Per usare un'API Google dovete registrare il nome del package e la chiave di firma della vostra app nella console API di Google. La registrazione consente ai servizi che convalidano il token di interrogare Google in merito all'app per cui il token è stato rilasciato, identificando così l'app chiamante. Questo processo di convalida presenta un effetto collaterale lieve ma importante: non

dovete infatti incorporare una chiave API nell'app e inviarla a ogni richiesta. Per un'app pubblicata da terze parti, potete così scoprire facilmente il nome del package e il certificato di firma, quindi non è particolarmente difficile procurarsi un token rilasciato a nome di qualche altra app (naturalmente non tramite l'API ufficiale).

Riepilogo

Android offre un registro centralizzato di account utente online grazie alla classe `AccountManager`, che permette di ottenere token per gli account esistenti senza dover gestire le credenziali utente raw e registrare tipi di account personalizzati. La registrazione di un tipo di account personalizzato consente di accedere a interessanti funzionalità di sistema, come il caching dei token di autenticazione e la sincronizzazione automatica in background. I dispositivi Google Experience includono il supporto integrato per gli account Google, che permette alle app di terze parti di accedere ai servizi online di Google senza dover richiedere direttamente all'utente le informazioni di autenticazione. L'app Google Play Services e la relativa libreria client migliorano ulteriormente il supporto degli account Google facilitando l'uso dei token OAuth 2.0 alle applicazioni di terze parti.

Sicurezza aziendale

Le prime versioni di Android erano orientate principalmente ai consumatori e disponevano di pochissime funzionalità per le aziende. Con la crescita della popolarità della piattaforma, i dispositivi Android hanno iniziato a trovare spazio nei luoghi di lavoro, e oggi sono sempre più usati per accedere all'e-mail aziendale, alle informazioni sui clienti e ad altri dati aziendali. Parallelamente a questa tendenza, è cresciuta anche la necessità di aumentare la sicurezza della piattaforma e di offrire strumenti che consentano una gestione efficace dei dispositivi dei dipendenti. Anche se Android rimane prevalentemente incentrato sui dispositivi consumer, le versioni recenti hanno visto l'introduzione di numerose funzionalità aziendali, ed è probabile che i successivi sviluppi della piattaforma saranno sempre più rivolti alle imprese.

In questo capitolo vedremo le principali caratteristiche di Android per le aziende, analizzandone l'uso al fine di aumentare la sicurezza del dispositivo e di fornire una gestione centralizzata delle policy dei device. Inizieremo dall'amministrazione del dispositivo, spiegando come può essere integrata nelle applicazioni di terze parti, poi esamineremo il supporto per le VPN e descriveremo le API che consentono lo sviluppo di nuove soluzioni VPN sotto forma di applicazioni di terze parti installate dagli utenti. A seguire, vedremo come Android implementa diversi metodi di autenticazione supportati dal framework di autenticazione EAP ed esamineremo la gestione delle credenziali. Per finire, scopriremo come aggiungere un profilo EAP programmaticamente usando l'API di gestione Wi-Fi estesa aggiunta in Android 4.3.

Amministrazione del dispositivo

Android 2.2 ha introdotto il supporto per un'API di amministrazione del dispositivo (*Device Administration*), che consente lo sviluppo di applicazioni in grado sia di applicare una policy di sicurezza a livello di sistema sia di adattare dinamicamente le sue funzionalità in base al livello di sicurezza corrente del dispositivo. Tali applicazioni sono dette *amministratori del dispositivo*. Gli amministratori del dispositivo devono essere abilitati esplicitamente nelle impostazioni di protezione del device e non possono essere disinstallati mentre sono attivi. Quando sono abilitati concedono privilegi speciali che permettono loro di bloccare il dispositivo, cambiare la password della schermata di blocco e persino cancellare tutti i dati utente dal dispositivo. Sono spesso associati a un tipo specifico di account aziendale (come Microsoft Exchange o Google Apps), che consente agli amministratori dell'azienda di controllare l'accesso ai dati aziendali permettendo l'accesso esclusivamente ai dispositivi conformi alla policy di sicurezza richiesta. Le policy di sicurezza possono essere statiche e si integrano nell'applicazione amministratore del dispositivo, oppure possono essere configurate sul lato server e inviate al device come parte di un protocollo di provisioning o sincronizzazione.

Nella versione 4.4 Android supporta i tipi di policy elencati nella Tabella 9.1. Le costanti delle policy sono definite nella classe `DeviceAdminInfo` (`http://bit.ly/1sWxUGB`).

Tabella 9.1 Policy di amministrazione del dispositivo supportate.

Costante della policy/Tag XML	Valore (bit da impostare)	Descrizione	Livello API
`USES_POLICY_LIMIT_PASSWORD` `<limit-password>`	0	Limita le password che l'utente può scegliere stabilendo una lunghezza o una complessità minima.	8
`USES_POLICY_WATCH_LOGIN` `<watch-login>`	1	Controlla i tentativi di login da parte di un utente.	8
`USES_POLICY_RESET_PASSWORD` `<reset-password>`	2	Reimposta la password di un utente.	8
`USES_POLICY_FORCE_LOCK` `<force-lock>`	3	Impone il blocco del dispositivo, oppure limita il timeout massimo per il blocco.	8

USES_POLICY_WIPE_DATA <wipe-data>	4	Riporta il dispositivo alle impostazioni di fabbrica, cancellando tutti i dati utente.	8
USES_POLICY_EXPIRE_PASSWORD <expire-password>	6	Impone all'utente di cambiare la sua password dopo un tempo limite stabilito dall'amministratore.	11
USES_ENCRYPTED_STORAGE <encrypted-storage>	7	Richiede la crittografia dei dati salvati.	11
USES_POLICY_DISABLE_CAMERA <disable-camera>	8	Disabilita l'uso delle fotocamere dei dispositivi.	14
USES_POLICY_DISABLE_KEYGUARD_FEATURES <disable-keyguard-features>	9	Disabilita l'uso delle funzionalità keyguard come i widget di blocco dello schermo o il supporto della fotocamera.	17

Ogni applicazione di amministrazione deve elencare le policy che intende utilizzare in un file dei metadati (vedere il paragrafo "Gestione dei privilegi"). L'elenco di policy supportate viene visualizzato all'utente quando attiva l'app amministratore, come mostrato nella Figura 9.1.

Figura 9.1 Schermata di attivazione dell'amministratore del dispositivo.

Implementazione

Ora che conosciamo le policy che possono essere applicate con l'API di amministrazione del dispositivo, vediamone l'implementazione interna. Come la maggior parte delle API pubbliche di Android, una classe manager chiamata `DevicePolicyManager` (http://bit.ly/1sWxUGB) espone parte delle funzionalità del servizio di sistema sottostante, `DevicePolicyManagerService`. Tuttavia, poiché la classe di facciata `DevicePolicyManager`, pur definendo le costanti e convertendo le eccezioni del servizio per restituire codici, aggiunge ben poche altre funzionalità, ci concentreremo sulla classe `DevicePolicyManagerService`.

Come la maggior parte dei servizi di sistema, `DevicePolicyManagerService` viene avviato ed eseguito nel processo `system_server` con l'utente `system`, e pertanto può eseguire quasi tutte le azioni privilegiate di Android. A differenza della maggior parte dei servizi di sistema, può concedere alle applicazioni di terze parti, che non necessitano di permessi di sistema speciali, l'accesso a determinate azioni privilegiate (come la modifica della password di blocco dello schermo). In questo modo gli utenti possono abilitare e disabilitare gli amministratori del dispositivo su richiesta, garantendo che possano applicare solo le policy che hanno dichiarato esplicitamente. Questo livello di flessibilità non può però essere implementato facilmente con i permessi standard di Android, che sono concessi solamente in fase di installazione e non possono essere revocati (con alcune eccezioni che abbiamo visto nel Capitolo 2). Per questo motivo, `DevicePolicyManagerService` adotta un metodo diverso per la gestione dei privilegi. Un altro aspetto interessante dell'implementazione di Android dell'amministrazione del dispositivo riguarda la gestione e l'applicazione delle policy, di cui parleremo nei dettagli nei paragrafi successivi.

Gestione dei privilegi

In fase di runtime, `DevicePolicyManagerService` mantiene un elenco interno e in memoria delle strutture di policy per ogni utente del dispositivo (le policy

sono disponibili anche su disco in un file XML, di cui parleremo nel prossimo paragrafo).

Ogni struttura contiene la policy attualmente applicata a un particolare utente e un elenco dei metadati per ogni amministratore del dispositivo attivo. Ciascun utente può abilitare più applicazioni con funzionalità di amministrazione del device, pertanto la policy attualmente attiva viene calcolata selezionando quella dalla definizione più rigorosa tra tutti gli amministratori. I metadati su ogni amministratore attivo contengono informazioni sull'applicazione dichiarante e un elenco di policy dichiarate (rappresentato da un maschera di bit).

`DevicePolicyManagerService` decide se concedere a un'applicazione chiamante l'accesso alle operazioni privilegiate basandosi sul suo elenco interno di policy attive: se l'applicazione chiamante è effettivamente un amministratore del dispositivo attivo e ha richiesto una policy corrispondente a quella della richiesta attuale (chiamata API), viene concesso il grant e viene eseguita l'operazione. Per confermare che un componente amministratore attivo appartiene realmente all'applicazione chiamante, `DevicePolicyManagerService` confronta l'UID del processo chiamante (restituito da `Binder.getCallingUid()`) con l'UID associato al componente amministratore target. Per esempio, un'applicazione che chiama `resetPassword()` deve essere un amministratore del dispositivo attivo, deve avere lo stesso UID del componente amministratore registrato e deve aver richiesto la policy `USES_POLICY_RESET_PASSWORD` affinché la chiamata abbia successo.

Le policy vengono richieste aggiungendo un file di risorse XML che elenca tutte le policy che un'applicazione amministratore del dispositivo vuole usare come figli del tag `<uses-policies>`. Prima di abilitare un amministratore del dispositivo, il sistema effettua il parsing del file XML e visualizza una finestra simile a quella nella Figura 9.1, consentendo all'utente di esaminare le policy richieste. Analogamente ai permessi Android, le policy degli amministratori sono concesse su base "tutto o niente" e non c'è modo di abilitarne selettivamente solo alcune. Un file di risorse che richiede tutte le policy potrebbe essere simile a quello nel

Listato 9.1 (per la policy corrispondente a ogni tag potete fare riferimento alla prima colonna della Tabella 9.1). Trovate maggiori dettagli sull'aggiunta di questo file a un'applicazione amministratore del dispositivo nel paragrafo "Aggiunta di un amministratore del dispositivo".

Listato 9.1 Dichiarazione delle policy nell'applicazione di amministrazione del dispositivo.

```
<?xml version="1.0" encoding="utf-8"?>
<device-admin xmlns:android="http://schemas.android.com/apk/res/android">
    <uses-policies>
        <limit-password />
        <watch-login />
        <reset-password />
        <force-lock />
        <wipe-data />
        <expire-password />
        <encrypted-storage />
        <disable-camera />
        <disable-keyguard-features />
        <set-global-proxy />
    </uses-policies>
</device-admin>
```

Per ricevere notifiche sugli eventi di sistema correlati alle policy e per ottenere accesso all'API Device Administration, gli amministratori del dispositivo devono prima essere attivati chiamando il metodo `setActiveAdmin()` di `DevicePolicyManagerService`. Questo metodo richiede il permesso `MANAGE_DEVICE_ADMINS`, che è un permesso di firma di sistema, pertanto solo le applicazioni di sistema possono aggiungere un amministratore del dispositivo senza interagire con l'utente.

Le applicazioni amministratore del dispositivo installate dall'utente possono richiedere l'attivazione solo avviando l'intent implicito `ACTION_ADD_DEVICE_ADMIN` con un codice simile a quello del Listato 9.2. L'unico handler per questo intent è l'applicazione *Settings* di sistema, che possiede il permesso `MANAGE_DEVICE_ADMINS`. Alla ricezione dell'intent, l'applicazione *Settings* verifica se l'applicazione richiedente è un amministratore del dispositivo valido, estrae le policy richieste e mostra la finestra di conferma della Figura 9.1. Premendo il pulsante *Activate* viene chiamato il metodo `setActiveAdmin()`, che aggiunge l'applicazione all'elenco di amministratori attivi per l'utente del dispositivo corrente.

Listato 9.2 Richiesta di attivazione dell'amministratore del dispositivo.

```
Intent intent = new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
ComponentName admin = new ComponentName(this, MyDeviceAdminReceiver.class);
intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, admin);
```

```
intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                "Required for corporate email access.");
startActivityForResult(intent, REQUEST_CODE_ENABLE_ADMIN);
```

Persistenza delle policy

Quando viene attivato o disattivato un amministratore del dispositivo, o quando vengono aggiornate le sue policy, le modifiche vengono scritte nel file `device_policies.xml` dell'utente target. Per l'utente proprietario tale file si trova in */data/system/*, mentre per tutti gli altri utenti è nella directory di sistema dell'utente (*/data/users/<user-ID>/*). Il file è di proprietà dell'utente `system`, che è l'unico a poterlo modificare (permessi sui file 0600).

Il file `device_policies.xml` contiene informazioni su ogni amministratore attivo e sulle sue policy, nonché informazioni globali sulla password corrente di blocco dello schermo. Il file è simile a quello mostrato nel Listato 9.3.

Listato 9.3 Contenuto del file devices_policies.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<policies>
  <admin
name="com.google.android.gms/com.google.android.gms.mdm.receivers.MdmDeviceAdminReceiver">(1)
    <policies flags="28" />
  </admin>
  <admin name="com.example.android.apis/com.example.android.apis.app.DeviceAdminSampleReceiver">
(2)
    <policies flags="1023" />(3)
    <password-quality value="327680" />(4)
    <min-password-length value="6" />
    <min-password-letters value="2" />
    <min-password-numeric value="2" />
    <max-time-to-unlock value="300000" />
    <max-failed-password-wipe value="100" />
    <encryption-requested value="true" />
    <disable-camera value="true" />
    <disable-keyguard-features value="1" />
</admin>
<admin name="com.android.email/com.android.email.SecurityPolicy$PolicyAdmin">(5)
  <policies flags="475" />
</admin>
<password-owner value="10076" />(6)
<active-password quality="327680" length="6"
                 uppercase="0" lowercase="3"
                 letters="3" numeric="3" symbols="0" nonletter="3" />(7)
</policies>
```

In questo esempio sono presenti tre amministratori del dispositivo attivi, ognuno rappresentato da un elemento `<admin>` (**(1)**, **(2)** e **(5)**). Le policy di ogni app amministratore sono archiviate nell'attributo `flags` del tag `<policies>` **(3)**. Una policy è abilitata se il suo bit corrispondente è

impostato (vedete la colonna Valore nella Tabella 9.1). Per esempio, visto che l'applicazione *DeviceAdminSample* ha richiesto tutte le policy attualmente disponibili, il suo attributo `flags` ha il valore `1023` (0x3FF, o 1111111111 in binario).

Se l'amministratore definisce restrizioni qualitative per la password (per esempio alfanumerica o complessa), queste vengono rese persistenti con l'attributo `value` del tag `<password-quality>` **(4)**. In questo esempio il valore `327680` (0x50000) corrisponde a `PASSWORD_QUALITY_ALPHANUMERIC` (le costanti di qualità della password sono definite nella classe `DevicePolicyManager`). I valori di altri requisiti delle policy, come la lunghezza della password e la crittografia del dispositivo, sono salvati come figli di ogni elemento `<admin>`. Se la password è stata impostata programmaticamente utilizzando il metodo `resetPassword()`, `device_policies.xml` contiene un tag `<password-owner>` che archivia l'UID dell'applicazione che imposta la password nel suo attributo `value` **(6)**. Infine, il tag `<active-password>` contiene dettagli sulla complessità della password corrente **(7)**.

Applicazione delle policy

Le policy di amministrazione del dispositivo hanno granularità diverse e possono essere applicate sia all'utente corrente sia a tutti gli utenti di un dispositivo. Alcune policy non vengono per niente applicate dal sistema: quest'ultimo informa solamente l'applicazione di amministrazione dichiarante, che ha l'incarico di compiere un'azione appropriata. In questo paragrafo descriveremo l'implementazione e l'applicazione di ogni tipo di policy.

USES_POLICY_LIMIT_PASSWORD

Dopo l'impostazione di una o più restrizioni per le password, gli utenti non possono inserire una password che non soddisfa la policy corrente. Tuttavia, il sistema non impone la modifica immediata della password, pertanto è possibile che la password corrente resti attiva fino alla modifica vera e propria. Le applicazioni di amministrazione possono

chiedere una nuova password all'utente avviando un intent implicito con l'azione `DevicePolicyManager.ACTION_SET_NEW_PASSWORD`.

Visto che ogni utente del dispositivo dispone di una password di sblocco diversa, le policy sulla qualità della password sono applicate per utente. Quando è impostata la qualità della password, i metodi di sblocco che non consentono una password della qualità prevista vengono disabilitati. Per esempio, se si imposta la qualità della password su `PASSWORD_QUALITY_ALPHANUMERIC`, i metodi di sblocco *Pattern* e *PIN* vengono disabilitati come mostrato nella Figura 9.2.

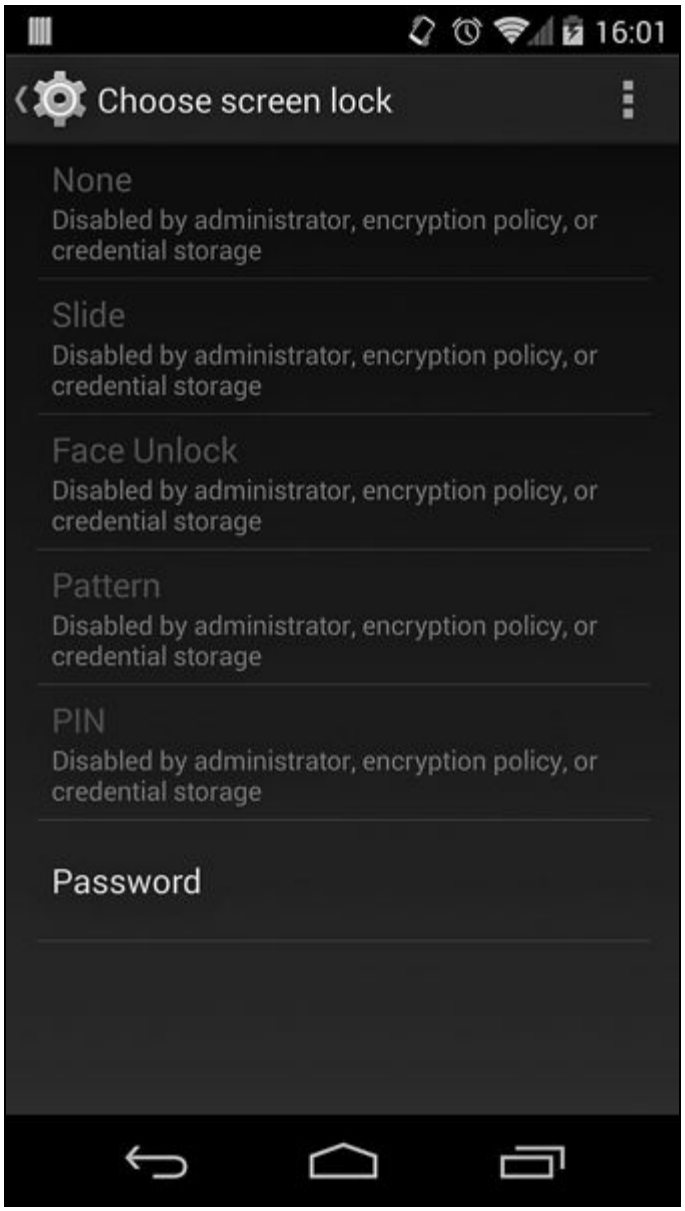

Figura 9.2 L'impostazione di una policy per la qualità della password disabilita i metodi di sblocco non compatibili.

USES_POLICY_WATCH_LOGIN

Questa policy consente agli amministratori del dispositivo di ricevere notifiche sull'esito dei tentativi di login. Le notifiche vengono inviate con i broadcast `ACTION_PASSWORD_FAILED` e `ACTION_PASSWORD_SUCCEEDED`. I broadcast receiver che derivano da `DeviceAdminReceiver` ricevono automaticamente la notifica con i metodi `onPasswordFailed()` e `onPasswordSucceeded()`.

USES_POLICY_RESET_PASSWORD

Questa policy consente alle applicazioni di amministrazione di impostare la password dell'utente corrente tramite l'API `resetPassword()`. La password specificata deve soddisfare i requisiti correnti per la qualità della password e viene applicata immediatamente. Se il dispositivo è crittografato, l'impostazione della password per la schermata di blocco dell'utente proprietario cambia anche la password di crittografia del dispositivo (nel Capitolo 10 sono disponibili maggiori dettagli sulla crittografia del dispositivo).

USES_POLICY_FORCE_LOCK

Questa policy consente agli amministratori di bloccare immediatamente il dispositivo chiamando il metodo `lockNow()`, oppure di specificare il tempo massimo di inattività dell'utente prima che il dispositivo venga bloccato automaticamente utilizzando `setMaximumTimeToLock()`. L'impostazione del tempo massimo ha effetto immediato e limita il tempo di inattività che l'utente può stabilire nelle impostazioni *Display* del sistema.

USES_POLICY_WIPE_DATA

Questa policy consente agli amministratori del dispositivo di cancellare i dati degli utenti chiamando l'API `wipeData()`. Le applicazioni che richiedono anche la policy `USES_POLICY_WATCH_LOGIN` possono impostare il numero di tentativi di login non riusciti prima che il dispositivo venga ripulito interamente e automaticamente tramite l'API `setMaximumFailedPasswordsForWipe()`. Se il numero di tentativi non riusciti è impostato su un valore maggiore di zero, l'implementazione della schermata di blocco informa `DevicePolicyManagerService` e visualizza una finestra di avviso dopo ogni tentativo non riuscito, attivando una cancellazione dei dati al raggiungimento della soglia. Se la cancellazione è attivata da

un tentativo di login non riuscito effettuato dall'utente proprietario, viene eseguita una cancellazione dell'intero dispositivo; se invece la cancellazione è attivata da un utente secondario, vengono eliminati solo l'utente e i dati associati, e il dispositivo passa all'utente primario.

NOTA

La cancellazione dell'intero dispositivo non è immediata, ma viene implementata scrivendo un comando `wipe_data` nella partizione *cache* ed effettuando il riavvio nella modalità di recovery. È il sistema operativo di recovery a effettuare la cancellazione effettiva del dispositivo. Pertanto, se il device dispone di un'immagine di recovery personalizzata che ignora il comando di cancellazione, o se l'utente effettua l'avvio con un recovery personalizzato ed elimina o modifica il file del comando, la cancellazione del dispositivo potrebbe non essere eseguita. Le immagini di recovery sono descritte nei dettagli nei Capitoli 10 e 13.

USES_POLICY_SETS_GLOBAL_PROXY

A partire da Android 4.4, questa policy non è disponibile per le applicazioni di terze parti. Consente agli amministratori del dispositivo di impostare l'host server proxy globale (`Settings.Global.GLOBAL_HTTP_PROXY_HOST`), la porta (`GLOBAL_HTTP_PROXY_PORT`) e l'elenco di host eseguiti (`GLOBAL_HTTP_PROXY_EXCLUSION_LIST`) scrivendo nel provider delle impostazioni di sistema globali. Solo il proprietario del dispositivo è autorizzato a configurare le impostazioni proxy globali.

USES_POLICY_EXPIRE_PASSWORD

Questa policy consente agli amministratori di impostare il timeout di scadenza della password tramite l'API `setPasswordExpirationTimeout()`. Se è impostato un timeout di scadenza, il sistema registra un allarme quotidiano che verifica la scadenza della password. Se la password è già scaduta, `DevicePolicyManagerService` pubblica notifiche giornaliere di modifica della password fino all'effettiva modifica della stessa. Gli amministratori del dispositivo vengono informati dello stato di scadenza della password con il metodo `DeviceAdminReceiver.onPasswordExpiring()`.

USES_ENCRYPTED_STORAGE

Questa policy consente agli amministratori di richiedere che la memoria del dispositivo venga crittografata tramite l'API `setStorageEncryption()`. Solo l'utente proprietario può richiedere la crittografia della memoria. Questa operazione non avvia automaticamente il processo di crittografia del dispositivo se questo non è crittografato; gli

amministratori del dispositivo devono controllare lo stato attuale della memoria usando l'API `getStorageEncryptionStatus()` – che esamina la proprietà di sistema di sola lettura `ro.crypto.state` – e avviare il processo di crittografia. La crittografia del dispositivo può essere iniziata avviando l'activity di sistema associata con l'intent implicito `ACTION_START_ENCRYPTION`.

USES_POLICY_DISABLE_CAMERA

Questa policy consente agli amministratori di disabilitare tutte le fotocamere sul dispositivo tramite l'API `setCameraDisabled()`. La fotocamera viene disabilitata impostando la proprietà di sistema `sys.secpolicy.camera.disabled` a 1. Il servizio `CameraService` del sistema nativo esamina questa proprietà e impedisce tutte le connessioni se l'impostazione è 1, disabilitando così la fotocamera per tutti gli utenti del dispositivo.

USES_POLICY_DISABLE_KEYGUARD_FEATURES

Questa policy consente agli amministratori di disabilitare le personalizzazioni del keyguard, come i widget della schermata di blocco, chiamando il metodo `setKeyguardDisabledFeatures()`. L'implementazione del keyguard di sistema verifica se questa policy è attiva e disabilita le funzionalità corrispondenti per l'utente target.

Aggiunta di un amministratore del dispositivo

Come con le altre applicazioni, gli amministratori del dispositivo possono essere inclusi nell'immagine di sistema o installati dagli utenti. Se un amministratore è parte dell'immagine di sistema, può esser impostato come *app proprietaria del dispositivo* in Android 4.4 e versioni successive: si tratta di un tipo speciale di amministratore del dispositivo che non può essere disabilitato dall'utente né disinstallato. In questo paragrafo vedremo come implementare un'app di questo tipo e dimostreremo come configurare un'app di sistema come proprietario del dispositivo.

Implementazione di un amministratore del dispositivo

Un'applicazione di amministrazione del dispositivo deve dichiarare un broadcast receiver che richiede il permesso `BIND_DEVICE_ADMIN` (**(1)** nel Listato 9.4), dichiara un file di risorse XML che elenca le policy utilizzate **(2)** e risponde all'intent `ACTION_DEVICE_ADMIN_ENABLED` **(3)**. Il Listato 9.4 mostra una dichiarazione di policy di esempio.

Listato 9.4 Dichiarazione di un broadcast receiver per l'amministratore del dispositivo.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.deviceadmin">
    --altro codice--
    <receiver android:name=".MyDeviceAdminReceiver"
        android:label="@string/device_admin"
        android:description="@string/device_admin_description"
        android:permission="android.permission.BIND_DEVICE_ADMIN">(1)
        <meta-data android:name="android.app.device_admin"
                   android:resource="@xml/device_admin_policy" />(2)
        <intent-filter>
            <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />(3)
        </intent-filter>
    </receiver>
    --altro codice--
</manifest>
```

L'SDK di Android offre una classe base da cui si può derivare il receiver, vale a dire `android.app.admin.DeviceAdminReceiver`. Questa classe definisce diversi metodi di callback sostituibili per gestire i broadcast relativi alle policy del dispositivo inviati dal sistema. Le implementazioni predefinite sono vuote, ma come minimo è necessario sostituire i metodi `onEnabled()` e `onDisabled()` per ricevere notifiche all'abilitazione o alla disabilitazione dell'amministratore. Gli amministratori del dispositivo non possono usare API privilegiate prima della chiamata a `onEnabled()` o dopo la chiamata a `onDisabled()`.

Potete utilizzare l'API `isAdminActive()` in qualsiasi momento per valutare se un'applicazione è attualmente un amministratore attivo del dispositivo. Come già spiegato nel paragrafo "Gestione dei privilegi", un amministratore non può attivarsi automaticamente e autonomamente, ma deve avviare un'activity di sistema per chiedere conferma all'utente con un codice simile a quello del Listato 9.2. Tuttavia, se è già attivo, un amministratore può disattivarsi autonomamente chiamando il metodo `removeActiveAdmin()`.

NOTA

Fate riferimento alla guida ufficiale dell'API di amministrazione dei dispositivi per maggiori dettagli e per esaminare un esempio di applicazione pienamente funzionante (http://bit.ly/1waywpZ).

Impostazione del proprietario del dispositivo

Un'applicazione di amministrazione del dispositivo che parte dell'immagine del sistema (vale a dire che il suo file APK è installato nella partizione *system*) può essere impostata come proprietario del dispositivo chiamando il metodo `setDeviceOwner(String packageName, String ownerName)` (non visibile nell'API SDK pubblica). Il primo parametro di questo metodo specifica il nome del package dell'applicazione di destinazione, il secondo il nome del proprietario da visualizzare nell'interfaccia. Anche se questo metodo non richiede permessi speciali, può essere chiamato solo prima del provisioning di un dispositivo (in pratica se l'impostazione globale `Settings.Global.DEVICE_PROVISIONED` è 0) e pertanto può essere chiamata solo dalle applicazioni di sistema eseguite come parte dell'inizializzazione del device.

Una chiamata riuscita a questo metodo scrive un file `device_owner.xml` (come quello nel Listato 9.5) in */data/system/*. Le informazioni sul proprietario del dispositivo corrente possono essere ottenute con i metodi `getDeviceOwner()`, `isDeviceOwner()` – esposto come `isDeviceOwnerApp()` nell'API SDK di Android – e `getDeviceOwnerName()`.

Listato 9.5 Contenuto del file device_owner.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<device-owner package="com.example.deviceadmin" name="Device Owner" />
```

Una volta attivato un proprietario del dispositivo, sia come parte del processo di provisioning sia da parte dell'utente, non è possibile né disabilitarlo né disinstallarlo, come mostrato nella Figura 9.3.

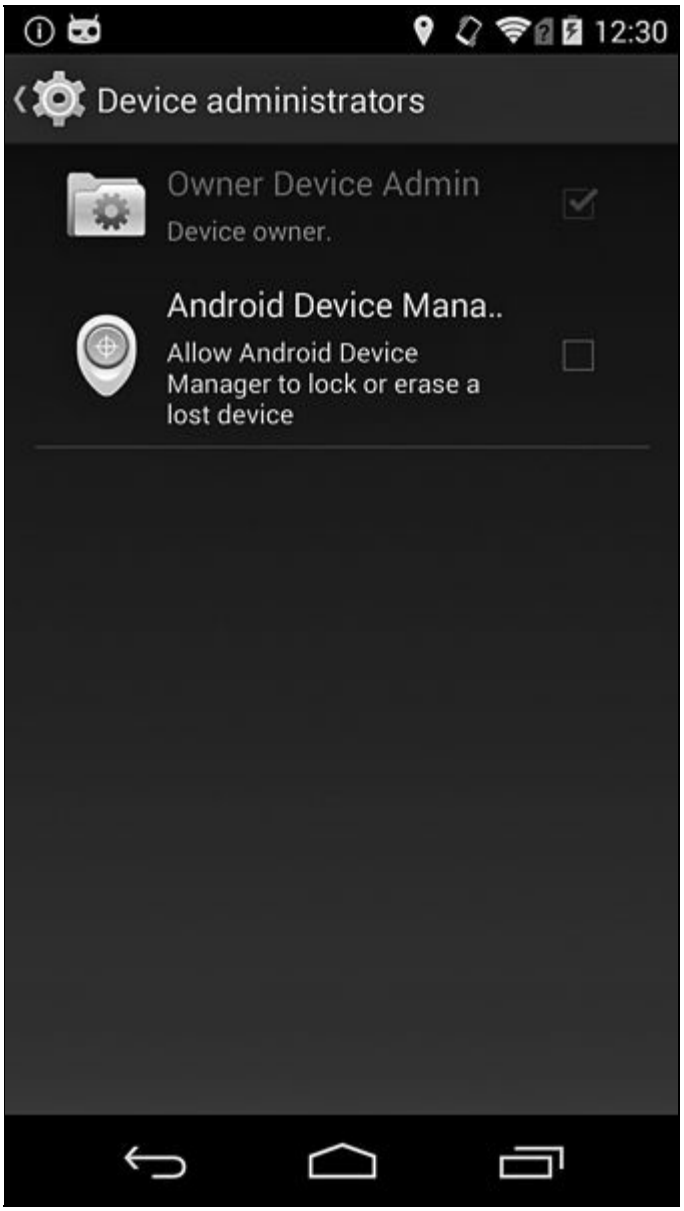

Figura 9.3 Un amministratore proprietario del dispositivo non può essere disabilitato.

Dispositivi gestiti

Un device su cui è installato un amministratore del dispositivo è definito *dispositivo gestito* e reagisce in maniera diversa, rispetto ai dispositivi non gestiti, alle modifiche alla configurazione che incidono sulla sicurezza del device. Nei Capitoli 6 e 7 abbiamo visto che Android consente agli utenti di installare certificati nel trust store di sistema sia tramite l'applicazione *Settings*, sia usando applicazioni di terze parti che chiamano l'API `KeyChain`. Se nel trust store sono presenti certificati installati dall'utente, a partire dalla versione 4.4 Android mostra un avviso (Figura

6.6 nel Capitolo 6) per informare gli utenti che le loro comunicazioni potrebbero essere monitorate.

Le reti aziendali spesso richiedono l'installazione di certificati trusted (per esempio il certificato root di una PKI aziendale) per accedere ai servizi aziendali. Tali certificati possono essere installati o rimossi senza l'intervento dell'utente da parte degli amministratori del dispositivo che possiedono i permessi di sistema `MANAGE_CA_CERTIFICATES` tramite i metodi `installCaCert()` e `uninstallCaCert()` della classe `DevicePolicyManager` (questi metodi sono riservati alle applicazioni di sistema e non sono visibili nell'API SDK pubblica). Se su un dispositivo gestito viene installato un certificato trusted supplementare, l'avviso di monitoraggio della rete viene sostituito da un messaggio meno allarmante, come mostrato nella Figura 9.4.

Figura 9.4 Messaggio informativo per il monitoraggio della rete mostrato sui dispositivi gestiti.

Integrazione degli account aziendali

Come detto nel paragrafo "Amministrazione del dispositivo", le applicazioni di amministrazione del dispositivo sono spesso abbinate agli account aziendali per consentire un certo controllo sui dispositivi che accedono ai dati dell'azienda. In questa sezione vedremo due di queste implementazioni, una nell'applicazione stock Email, utilizzabile con gli account Microsoft Exchange ActiveSync, e l'altra nell'applicazione dedicata Google Apps Device Policy, utilizzabile con gli account Google aziendali.

Microsoft Exchange ActiveSync

Microsoft Exchange ActiveSync (solitamente abbreviato in EAS) è un protocollo che supporta la sincronizzazione di e-mail, contatti, calendari e attività da un server groupware a un dispositivo mobile. È supportato sia da Microsoft Exchange Server, sia dalla maggior parte dei prodotti concorrenti, tra cui Google Apps.

L'applicazione *Email* inclusa in Android supporta gli account ActiveSync e la sincronizzazione dati tramite autenticatori di account (vedete il Capitolo 8) e sync adapter dedicati. Per consentire agli amministratori dell'azienda di applicare una policy di sicurezza ai dispositivi che accedono all'e-mail e ad altri dati aziendali, *Email* non consente la sincronizzazione finché l'utente non abilita l'amministratore del dispositivo predefinito. L'amministratore può impostare le regole per la password della schermata di blocco, cancellare tutti i dati, richiedere la crittografia della memoria e disabilitare le fotocamere del dispositivo, come mostrato nella Figura 9.5. Tuttavia, le policy non sono integrate nell'app, ma vengono recuperate dal servizio utilizzando il protocollo EAS Provision.

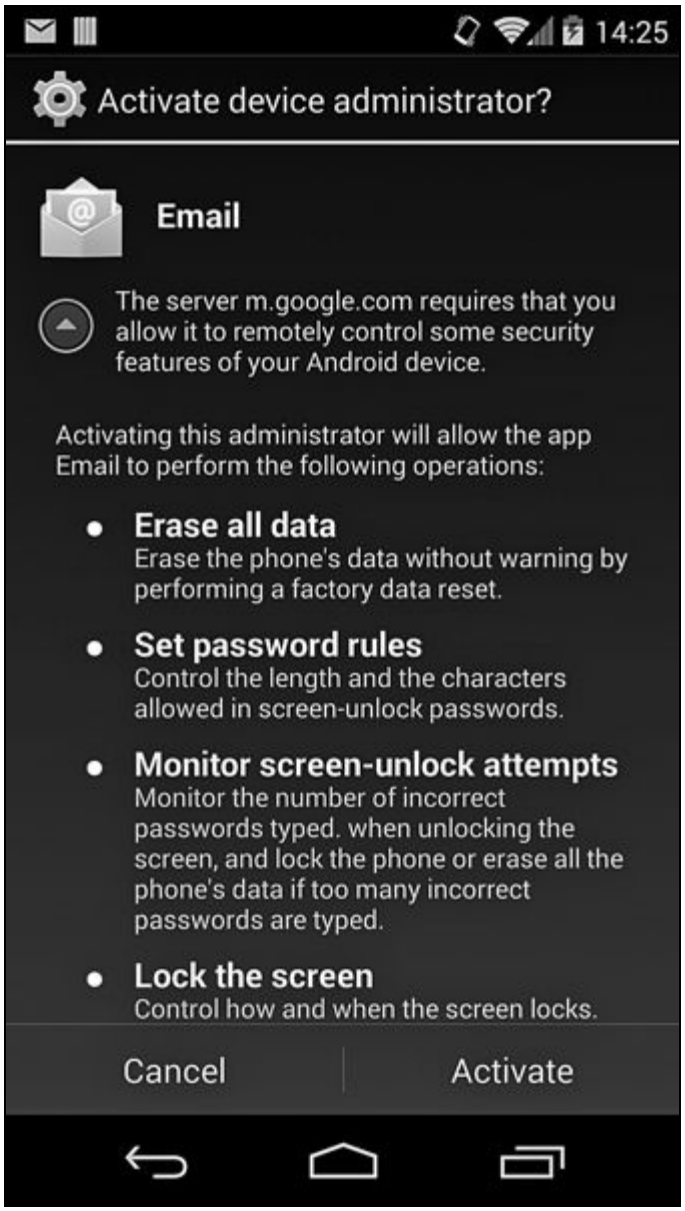

Figura 9.5 Policy di amministrazione del dispositivo richieste per l'uso di un account EAS.

Google Apps

La versione aziendale del servizio Gmail di Google, Google Apps, supporta anche l'impostazione di policy di sicurezza per i dispositivi mobili. Se la funzionalità è stata abilitata dall'amministratore di dominio, i titolari di account Google Apps possono individuare, far suonare, bloccare o cancellare i loro dispositivi Android da remoto. Gli amministratori di dominio possono inoltre eliminare selettivamente un account Google Apps e tutti i contenuti associati da un dispositivo gestito senza eseguire una cancellazione completa. Sia l'applicazione delle policy di sicurezza sia la gestione dei dispositivi da remoto sono implementate

nell'applicazione dedicata Google Apps Device Policy (**(5)** nel Listato 9.3). Al primo avvio, l'applicazione chiede all'utente di abilitare l'amministratore del dispositivo predefinito e visualizza le impostazioni correnti per le policy del dominio, mostrate nella Figura 9.6.

Figura 9.6 Conferma di applicazione delle policy nell'applicazione Google Apps Device Policy.

Gli amministratori di dominio definiscono le policy nella console di amministrazione di Google Apps (Figura 9.7), e le relative impostazioni vengono inviate ai dispositivi utilizzando il protocollo di sincronizzazione proprietario di Google.

Anche se gli account Google gratuiti non supportano l'impostazione di policy del dispositivo, i device Google Experience possono usare l'amministratore del dispositivo di base incluso in Google Play Services

(**(1)** nel Listato 9.3); questo consente ai titolari di account Google di individuare o cancellare da remoto i loro dispositivi utilizzando il sito web Android Device Manager o l'applicazione Android associata.

Supporto VPN

Una *rete privata virtuale* (VPN, *Virtual Private Network*) consente di estendere una rete privata in una rete pubblica senza richiedere una connessione fisica dedicata, permettendo così a tutti i dispositivi connessi di inviare e ricevere dati come se condividessero un percorso e fossero fisicamente collegati alla medesima rete privata. Quando una VPN viene usata per consentire ai singoli dispositivi di connettersi a una rete privata target, si parla di *VPN di accesso remoto*, mentre quando viene usata per connettere due reti remote diventa una *VPN da sito a sito* (o *site-to-site*).

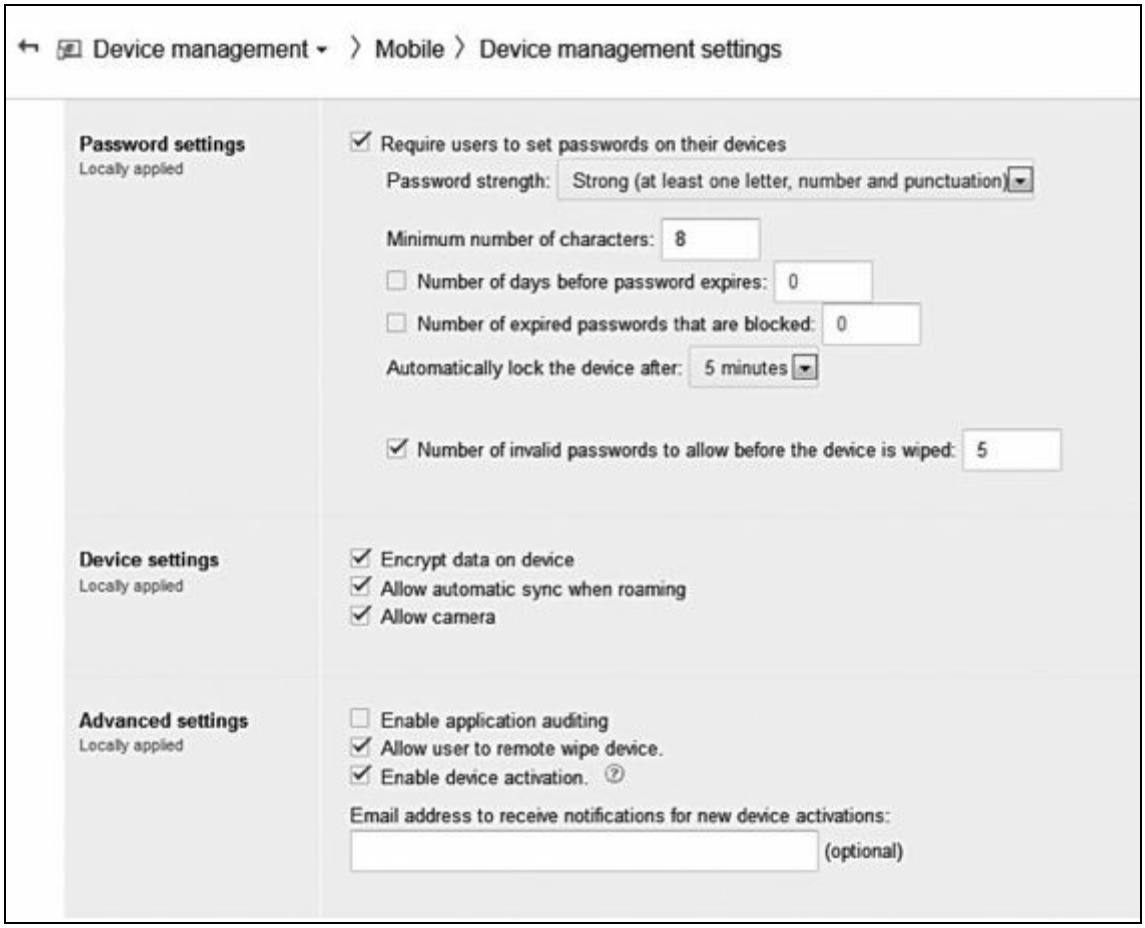

Figura 9.7 Finestra di gestione delle policy del dispositivo Google Apps.

Le VPN di accesso remoto possono connettere dispositivi fissi con un indirizzo IP statico, come un computer in un ufficio remoto, ma sono molto più comuni le configurazioni in cui i client mobili utilizzano connessioni di rete variabili e indirizzi dinamici. Tale configurazione è spesso detta *road warrior*, ed è quella più usata con i client VPN Android.

Per garantire che i dati trasmessi su una VPN restino privati, le VPN di solito effettuano l'autenticazione dei client remoti e garantiscono la riservatezza e l'integrità dei dati utilizzando un protocollo di tunneling sicuro. I protocolli VPN sono complessi perché operano su più strati di rete contemporaneamente e spesso coinvolgono più livelli di incapsulamento per essere compatibili con le varie configurazioni della rete. Una discussione approfondita va oltre l'ambito di questo libro; tuttavia, nei paragrafi successivi troverete una breve descrizione dei tipi principali di protocolli VPN, in particolare di quelli disponibili in Android.

PPTP

Point-to-Point Tunneling Protocol (PPTP) usa un canale di controllo TCP per stabilire le connessioni e il protocollo di tunneling *Generic Routing Encapsulation* (GRE) per incapsulare i pacchetti *Point-to-Point Protoco*l (PPP). Sono supportati diversi metodi di autenticazione, come *Password Authentication Protocol* (PAP), *Challenge-Handshake Authentication Protocol* (CHAP) e la sua estensione Microsoft MS-CHAP v1/v2, nonché EAP-TLS, ma solo quest'ultimo è considerato sicuro.

Il payload PPP può essere crittografato con il protocollo Microsoft *Point-to-Point Encryption* (MPPE), che usa la cifratura a flussi RC4. MPPE non impiega alcuna forma di autenticazione del testo cifrato, pertanto è vulnerabile agli attacchi di bit-flipping; inoltre, negli ultimi anni sono stati scoperti vari problemi della cifratura RC4 che hanno ulteriormente ridotto la sicurezza di MMPE e PPTP.

L2TP/IPSec

Layer 2 Tunneling Protocol (L2TP) è simile a PPTP ed esiste nello strato di data link (strato 2 del modello OSI). L2TP non fornisce di per sé alcuna crittografia o riservatezza (si affida al protocollo di tunneling per implementare queste funzionalità), pertanto una VPN L2TP viene

generalmente implementata usando una combinazione di L2TP e della suite di protocolli *Internet Protocol Security* (IPSec), che aggiunge l'autenticazione, la riservatezza e l'integrità.

In una configurazione L2TP/IPSec, viene per prima cosa stabilito un canale sicuro con IPSec, poi viene creato un tunnel L2TP sul canale sicuro. I pacchetti L2TP sono sempre inseriti nei pacchetti IPSec e sono pertanto sicuri. Una connessione IPSec richiede di stabilire un'*associazione di protezione* (SA, *Security Association*), ovvero una combinazione di algoritmo e modalità di crittografia, chiave di crittografia e altri parametri necessari per stabilire un canale sicuro.

Le SA sono stabilite usando l'*Internet Security Association and Key Management Protocol* (ISAKMP). ISAKMP non definisce un metodo specifico per lo scambio delle chiavi ed è generalmente implementato mediante configurazione manuale dei segreti precondivisi, o usando il protocollo *Internet Key Exchange* (IKE e IKEv2). IKE utilizza i certificati X.509 per l'autenticazione dei peer (analogamente a SSL) e uno scambio di chiavi Diffie-Hellman per stabilire un segreto condiviso, che viene usato per derivare le chiavi di crittografia effettive della sessione.

IPSec Xauth

IPSec Extended Authentication (Xauth) estende IKE per includere altri scambi di autenticazione utente. In questo modo è possibile usare un database di utenti o un'infrastruttura RADIUS esistente per autenticare i client che effettuano l'accesso remoto, nonché integrare l'autenticazione a due fattori.

Mode-configuration (Modecfg) è un'altra estensione di IPSec spesso usata in uno scenario di accesso remoto. Consente ai server VPN di inviare ai client informazioni di configurazione della rete come l'indirizzo IP privato e gli indirizzi del server DNS. Se usati insieme, Xauth e Modecfg permettono di creare una soluzione VPN IPSec pura, che non fa affidamento su altri protocolli per l'autenticazione e il tunneling.

VPN basate su SSL

Le VPN basate su SSL usano SSL o TLS (fate riferimento al Capitolo 6) per stabilire una connessione di rete ed effettuare il tunneling del traffico di rete. Non esiste un singolo standard che definisce le VPN basate su SSL: esistono invece varie implementazioni che usano strategie diverse per stabilire un canale sicuro e incapsulare i pacchetti.

OpenVPN è una famosa applicazione open source che usa SSL per l'autenticazione e lo scambio di chiavi (sono supportate anche le chiavi statiche condivise preconfigurate) e un protocollo di crittografia personalizzato per crittografare e autenticare i pacchetti (http://bit.ly/1wayyhu). OpenVPN effettua il multiplexing della sessione SSL usata per l'autenticazione e lo scambio di chiavi; i pacchetti crittografati vengono così trasferiti su una singola porta UDP (o TCP). Il protocollo di multiplexing offre uno strato di trasporto affidabile per SSL sopra UDP, ma effettua il tunneling dei pacchetti IP crittografati su UDP senza aumentare l'affidabilità. Quest'ultima è fornita dal protocollo di tunneling stesso, solitamente TCP. I principali vantaggi di OpenVPN rispetto a IPSec sono la sua semplicità è la possibilità di effettuare l'implementazione totalmente nello userspace. IPSec, d'altra parte, richiede il supporto a livello di kernel e l'implementazione di più protocolli interoperanti. Inoltre, è più facile far passare il traffico OpenVPN attraverso firewall, NAT e proxy, perché usa i protocolli di rete comuni TCP e UDP e può effettuare il multiplexing del traffico nel tunnel su una singola porta.

Nei paragrafi di seguito vedremo il supporto VPN predefinito di Android e le API fornite alle applicazioni che vogliono implementare soluzioni VPN supplementari. Riesamineremo anche i componenti dell'infrastruttura VPN di Android e vedremo come proteggere le credenziali della VPN.

VPN legacy

Prima di Android 4.0, il supporto VPN era totalmente integrato nella piattaforma e non era estensibile: il supporto per i nuovi tipi di VPN poteva quindi essere aggiunto solo con gli aggiornamenti della piattaforma. Per distinguerlo dalle implementazioni basate sulle applicazioni, il supporto VPN integrato è definito *VPN legacy*.

Le precedenti versioni di Android supportavano diverse configurazioni VPN basate su PPTP e L2TP/IPSec, con supporto per le VPN "pure-IPSec" utilizzando IPSec Xauth (aggiunto nella versione 4.0). Oltre alle nuove configurazioni predefinite per le VPN, Android 4.0 ha introdotto anche le VPN basate sulle applicazioni fornendo la classe base della piattaforma `VpnService`, che le applicazioni possono estendere per implementare una nuova soluzione VPN.

La VPN legacy è controllata dall'applicazione *Settings* di sistema ed è disponibile solo al proprietario (detto anche utente primario) sui dispositivi multiutente. La Figura 9.8 mostra la finestra di dialogo per aggiungere un nuovo profilo VPN legacy IPSec.

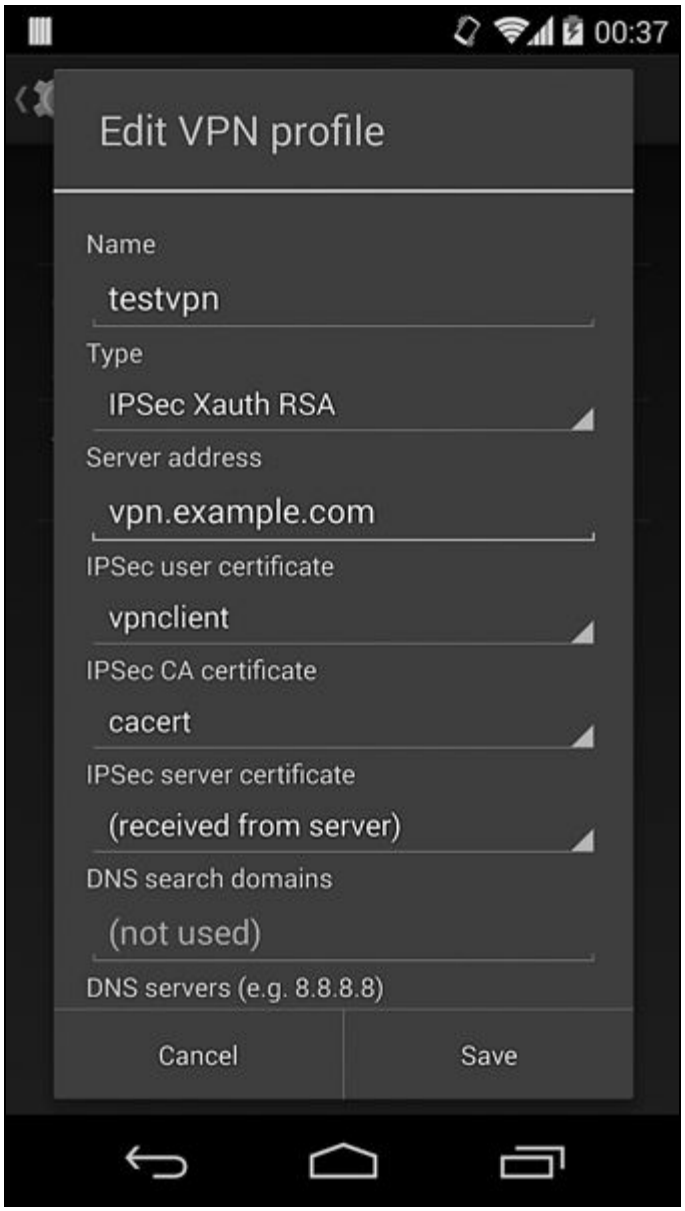

Figura 9.8 Finestra di definizione di un profilo VPN legacy.

Implementazione

Le VPN legacy sono implementate usando una combinazione di driver del kernel e daemon, comandi e servizi di sistema nativi. L'implementazione di livello più basso del tunneling PPTP e L2TP usa un daemon PPP specifico di Android chiamato `mtpd` e i driver del kernel PPPoPNS e PPPoLAC (disponibile solo nei kernel Android).

Le VPN legacy supportano solo una connessione VPN per dispositivo, pertanto `mtpd` può creare solamente una sessione singola. Le VPN IPSec sfruttano il supporto del kernel predefinito per IPSec e un daemon di gestione delle chiavi IKE `racoon` modificato (parte del package di utility

IPSec-Tools, http://ipsec-tools.sourceforge.net, che completa l'implementazione IPSec del kernel Linux; racoon supporta solo IKEv1). Il Listato 9.6 mostra la definizione di questi due daemon in init.rc.

Listato 9.6 Definizione di racoon e mtpd in init.rc.

```
service racoon /system/bin/racoon(1)
    class main
    socket racoon stream 600 system system(2)
    # IKE usa UDP porta 500. Racoon imposterà l'UID su VPN dopo il binding della porta.
    group vpn net_admin inet(3)
    disabled
    oneshot

service mtpd /system/bin/mtpd(4)
    class main
    socket mtpd stream 600 system system(5)
    user vpn
    group vpn net_admin inet net_raw(6)
    disabled
    oneshot
```

Sia racoon **(1)** sia mtpd **(4)** creano socket di controllo (**(2)** e **(5)**), accessibili unicamente all'utente system e non avviati per impostazione predefinita. Per entrambi i daemon vengono aggiunti vpn, net_admin (mappato dal kernel alla capability Linux CAP_NET_ADMIN) e inet ai gruppi supplementari (**(3)** e **(6)**), per consentire loro di creare socket e controllare i dispositivi di interfaccia di rete. Il daemon mtpd riceve inoltre il gruppo net_raw (mappato alla capability Linux CAP_NET_RAW), che consente di creare socket GRE (usati da PPTP).

Quando una VPN è avviata dall'app *Settings* di sistema, Android avvia i daemon racoon e mtpd e invia loro comandi di controllo tramite i relativi socket locali per stabilire la connessione configurata. I daemon creano il tunnel VPN richiesto, poi creano e configurano un'interfaccia di rete tunnel con l'indirizzo IP e la network mask ricevuti. Mentre mtpd esegue internamente la configurazione dell'interfaccia, racoon ricorre al comando helper ip-up-vpn per attivare l'interfaccia del tunnel, solitamente tun0.

Per comunicare i parametri di connessione al framework, i daemon VPN scrivono un file state in */data/misc/vpn/*, come mostrato nel Listato 9.7.

Listato 9.7 Contenuti del file VPN state.

```
# cat /data/misc/vpn/state
tun0(1)
10.8.0.1/24(2)
192.168.1.0/24(3)
192.168.1.1(4)
example.com(5)
```

Il file contiene il nome dell'interfaccia tunnel **(1)**, il suo indirizzo IP e la relativa mask **(2)**, le route configurate **(3)**, i server DNS **(4)** e i domini di ricerca **(5)**, ognuno su una nuova riga. Dopo l'avvio dei daemon VPN, il framework esegue il parsing del file `state` e chiama il `ConnectivityService` di sistema per configurare il routing, i server DNS e i domini di ricerca per la connessione VPN appena stabilita. A sua volta, `ConnectivityService` invia comandi di controllo tramite il socket di controllo locale del daemon `netd`, che può modificare il filtro dei pacchetti e le tabelle di routing del kernel in quanto è in esecuzione come root. Il traffico di tutte le applicazioni avviate dall'utente proprietario e dai profili con restrizioni viene instradato attraverso l'interfaccia VPN aggiungendo una regola del firewall corrispondente all'UID dell'applicazione e alle relative regole di routing (maggiori dettagli sul routing del traffico per applicazione e sul supporto multiutente sono disponibili nel paragrafo "Supporto multiutente" più avanti in questo capitolo).

Archiviazione di profili e credenziali

Ogni configurazione VPN creata con l'app *Settings* è detta *profilo VPN* e viene salvata su disco in forma crittografata. La crittografia viene eseguita dal daemon dell'archivio delle credenziali di Android, `keystore`, con una chiave specifica del dispositivo (consultate il Capitolo 7 per ulteriori informazioni sull'implementazione dell'archivio delle credenziali).

I profili VPN vengono serializzati concatenando tutte le proprietà configurate, che sono delimitate da un carattere *NUL* (`\0`) in una singola stringa di profilo salvata nel keystore di sistema come blob binario. I nomi file dei profili VPN sono generati aggiungendo l'ora corrente in millisecondi (in formato esadecimale) al prefisso `VPN_`. Per esempio, nel

Listato 9.8 è mostrata la directory *keystore* di un utente con tre profili VPN configurati (timestamp del file omessi).

Listato 9.8 Contenuto della directory keystore quando sono configurati profili VPN.

```
# ls -l /data/misc/keystore/user_0
-rw------- keystore keystore      980 1000_CACERT_cacert(1)
-rw------- keystore keystore       52 1000_LOCKDOWN_VPN(2)
-rw------- keystore keystore      932 1000_USRCERT_vpnclient(3)
-rw------- keystore keystore     1652 1000_USRPKEY_vpnclient(4)
-rw------- keystore keystore      116 1000_VPN_144965b85a6(5)
-rw------- keystore keystore       84 1000_VPN_145635c88c8(6)
-rw------- keystore keystore      116 1000_VPN_14569512c80(7)
```

I tre profili VPN sono salvati nei file `1000_VPN_144965b85a6` **(5)**, `1000_VPN_145635c88c8` **(6)** e `1000_VPN_14569512c80` **(7)**. Il prefisso `1000_` rappresenta l'utente proprietario, `system` (UID 1000). I profili VPN sono di proprietà dell'utente `system`, pertanto solo le applicazioni di sistema possono recuperare e decodificare il contenuto del profilo.

Nel Listato 9.9 è mostrato il contenuto decodificato dei tre file dei profili VPN (il carattere *NUL* è stato sostituito con un pipe [`|`] per ragioni di leggibilità).

Listato 9.9 Contenuto dei file dei profili VPN.

```
psk-vpn|1|vpn1.example.com|test1|pass1234||||true|l2tpsecret|l2tpid|PSK|||(1)
pptpvpn|0|vpn2.example.com|user1|password||||true||||||(2)
certvpn|4|vpn3.example.com|user3|password||||true||||vpnclient|cacert||(3)
```

I file dei profili contengono tutti i campi mostrati nella finestra di modifica del profilo VPN (Figura 9.8); le proprietà mancanti sono rappresentate da una stringa vuota. I primi cinque campi rappresentano il nome della VPN, il tipo, l'host gateway, il nome utente e la password. Nel Listato 9.9, il primo profilo VPN **(1)** è per una VPN L2TP/IPSec con chiave precondivisa (tipo `1`); il secondo profilo **(2)** è per una VPN PPTP (tipo `0`); l'ultimo **(3)** è per una VPN IPSec che usa i certificati e l'autenticazione Xauth (tipo `4`).

Oltre al nome utente e alla password, i file dei profili VPN contengono tutte le altre credenziali necessarie per la connessione alla VPN. Nel caso del primo profilo VPN **(1)** del Listato 9.9, la credenziale aggiuntiva è la chiave precondivisa richiesta per stabilire una connessione sicura IPSec (rappresentata dalla stringa `PSK` in questo esempio). Nel caso del terzo profilo **(3)**, le credenziali aggiuntive sono la chiave privata e il certificato

dell'utente. Ad ogni modo, come potete vedere nell'esempio, la chiave e il certificato non sono inclusi nella loro interezza: il profilo contiene solo l'alias (`vpnclient`) della chiave e del certificato (che condividono un alias comune). La chiave privata e il certificato sono salvati nell'archivio delle credenziali del sistema e l'alias incluso nel profilo VPN serve solo come identificatore, usato per accedere alla chiave e al certificato o per recuperarli.

Accesso alle credenziali

Il daemon `racoon`, che in origine usava chiavi e certificati salvati in file PEM, è stato modificato per usare l'engine OpenSSL `keystore` di Android. Come visto nel Capitolo 7, l'engine `keystore` è un gateway per l'archivio delle credenziali di sistema, che può sfruttare le implementazioni hardware dell'archivio delle credenziali (quando disponibili) e che, quando riceve l'alias di una chiave, utilizza la chiave privata corrispondente per firmare i pacchetti di autenticazione, senza estrarla dal keystore.

Il profilo VPN **(3)** nel Listato 9.9 contiene anche l'alias del certificato CA (*cacert*), usato come trust anchor durante la convalida del certificato del server. Durante il runtime, il framework recupera il certificato client (**(3)** nel Listato 9.8) e il certificato CA (**(1)** nel Listato 9.8) dal keystore di sistema e li passa a `racoon` tramite il socket di controllo, insieme agli altri parametri di connessione. Il blob della chiave privata (**(4)** nel Listato 9.8) non viene mai passato direttamente al daemon `racoon`, che riceve solo l'alias della chiave (*vpnclient*).

NOTA

Anche se le chiavi private sono protette dall'hardware sui dispositivi con un keystore hardware, le chiavi precondivise o le password salvate nel contenuto di un profilo VPN non lo sono: attualmente Android non supporta nel keystore hardware l'importazione di chiavi simmetriche, ma solo di chiavi asimmetriche (RSA, DSA ed EC). Di conseguenza, le credenziali per le VPN che usano chiavi precondivise sono salvate nel profilo VPN come testo normale e possono essere estratte dai dispositivi che consentono l'accesso root dopo la decodifica del profilo in memoria.

VPN always-on

Android 4.2 e versioni successive supporta una configurazione VPN *always-on*, che blocca tutte le connessioni di rete dalle applicazioni finché non viene stabilita una connessione al profilo VPN specificato. In questo modo si impedisce alle applicazioni di inviare dati su canali non sicuri, come le reti Wi-Fi pubbliche.

La configurazione di una VPN always-on richiede l'impostazione di un profilo VPN che specifica il gateway VPN come indirizzo IP e dichiara un indirizzo IP del server DNS esplicito. Questa configurazione esplicita è indispensabile per garantire che il traffico DNS non sia inviato al server DNS configurato in locale, bloccato quando la VPN always-on è attiva. La finestra di selezione del profilo VPN è mostrata nella Figura 9.9.

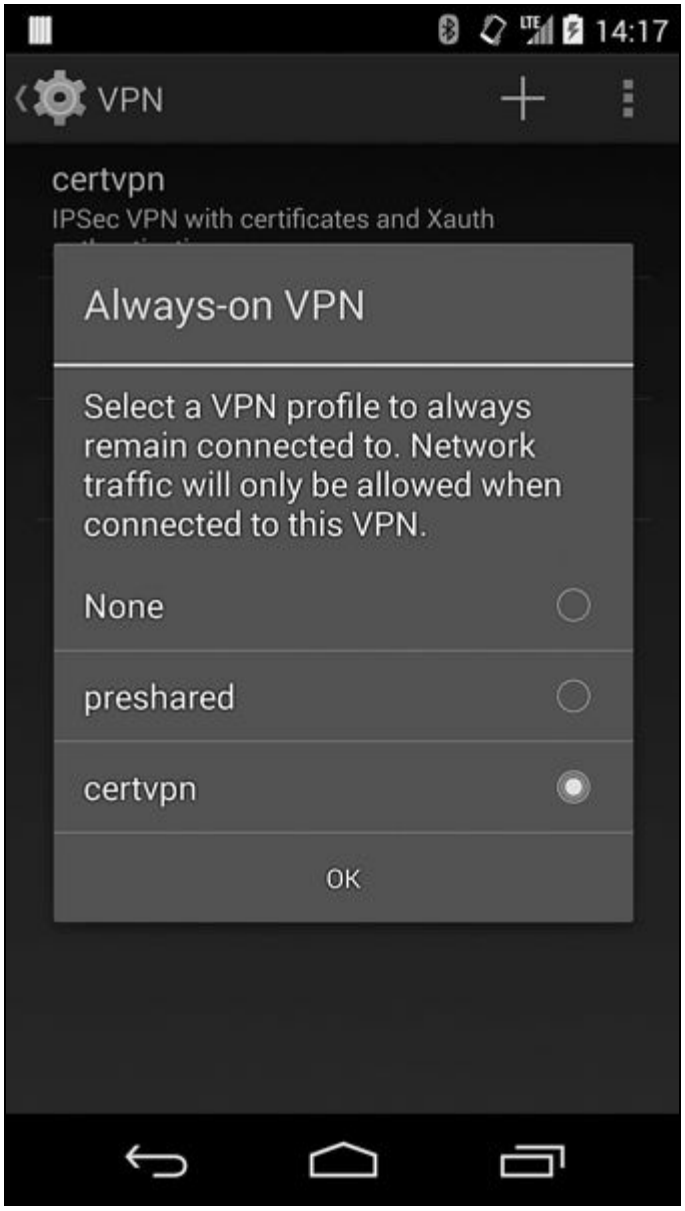

Figura 9.9 Finestra di selezione di un profilo VPN always-on.

La scelta del profilo viene salvata con altri profili VPN nel file crittografato `LOCKDOWN_VPN` (**(2)** nel Listato 9.8), che contiene solo il nome del profilo selezionato, in questo esempio `144965b85a6`. Se è presente il file `LOCKDOWN_VPN`, il sistema si connette automaticamente alla VPN specificata all'avvio del dispositivo. Se la connessione di rete sottostante viene interrotta e ripristinata o subisce comunque modifiche (per esempio se si cambia hotspot Wi-Fi), la VPN viene riavviata automaticamente.

Una VPN always-on garantisce che tutto il traffico passi attraverso la VPN, creando regole del firewall che bloccano tutti i pacchetti tranne quelli che devono passare nell'interfaccia VPN. Le regole sono installate dalla classe `LockdownVpnTracker` (la VPN always-on è definita *VPN lockdown* nel codice sorgente di Android), che monitora lo stato della VPN e adegua lo stato corrente del firewall inviando comandi al daemon `netd`, che a sua volta esegue l'utility `iptables` per modificare le tabelle di filtraggio dei pacchetti del kernel. Per esempio, quando una VPN L2TP/IPSec always-on si è connessa a un server VPN con indirizzo IP `11.22.33.44` e ha creato un'interfaccia di tunneling `tun0` con indirizzo IP `10.1.1.1`, le regole del firewall installate (segnalate da `iptables`; alcune colonne sono state omesse per ragioni di concisione) somigliano a quelle nel Listato 9.10.

Listato 9.10 Regole del firewall VPN always-on.

```
# iptables -v -L n
--altro codice--
Chain fw_INPUT (1 references)
 target   prot opt in    out   source        destination
 RETURN   all  --  *     *     0.0.0.0/0     10.1.1.0/24(1)
 RETURN   all  --  tun0  *     0.0.0.0/0     0.0.0.0/0(2)
 RETURN   udp  --  *     *     11.22.33.44   0.0.0.0/0    udp spt:1701(3)
 RETURN   tcp  --  *     *     11.22.33.44   0.0.0.0/0    tcp spt:1701
 RETURN   udp  --  *     *     11.22.33.44   0.0.0.0/0    udp spt:4500
 RETURN   tcp  --  *     *     11.22.33.44   0.0.0.0/0    tcp spt:4500
 RETURN   udp  --  *     *     11.22.33.44   0.0.0.0/0    udp spt:500
 RETURN   tcp  --  *     *     11.22.33.44   0.0.0.0/0    tcp spt:500
 RETURN   all  --  lo    *     0.0.0.0/0     0.0.0.0/0
 DROP     all  --  *     *     0.0.0.0/0     0.0.0.0/0(4)

Chain fw_OUTPUT (1 references)
 target   prot opt in    out   source        destination
 RETURN   all  --  *     *     10.1.1.0/24   0.0.0.0/0(5)
 RETURN   all  --  *     tun0  0.0.0.0/0     0.0.0.0/0(6)
 RETURN   udp  --  *     *     0.0.0.0/0     11.22.33.44  udp dpt:1701(7)
 RETURN   tcp  --  *     *     0.0.0.0/0     11.22.33.44  tcp dpt:1701
 RETURN   udp  --  *     *     0.0.0.0/0     11.22.33.44  udp dpt:4500
 RETURN   tcp  --  *     *     0.0.0.0/0     11.22.33.44  tcp dpt:4500
 RETURN   udp  --  *     *     0.0.0.0/0     11.22.33.44  udp dpt:500
 RETURN   tcp  --  *     *     0.0.0.0/0     11.22.33.44  tcp dpt:500
```

```
 RETURN     all  --  *      lo     0.0.0.0/0      0.0.0.0/0
 REJECT     all  --  *      *      0.0.0.0/0      0.0.0.0/0    reject-with icmp-port-unreachable(8)
--altro codice--
```

Come potete vedere nel listato, tutto il traffico da e verso la rete VPN è consentito (**(1)** e **(5)**), così come il traffico sull'interfaccia tunnel (**(2)** e **(6)**). Il traffico da e verso il server VPN (**(3)** e **(7)**) è consentito solo sulle porte usate da IPSec (`500` e `4500`) e L2TP (`1701`). Tutto il restante traffico in arrivo viene ignorato **(4)**, così come il traffico in uscita rimanente viene rifiutato **(8)**.

VPN basate sulle applicazioni

Android 4.0 ha aggiunto un'API pubblica `VpnService` (http://bit.ly/ZvZ5Lj) che le applicazioni di terze parti possono usare per creare soluzioni VPN che non sono integrate nel sistema operativo né richiedono permessi a livello di sistema. `VpnService` e la classe `Builder` associata consentono alle applicazioni di specificare parametri di rete come l'indirizzo IP e le route dell'interfaccia, che il sistema impiega per creare e configurare un'interfaccia di rete virtuale. Le applicazioni ricevono un descrittore di file associato a tale interfaccia di rete e possono eseguire il tunneling del traffico di rete leggendo o scrivendo nel descrittore di file dell'interfaccia.

Ogni lettura recupera un pacchetto IP in uscita, mentre ogni scrittura genera un pacchetto IP in entrata. Poiché l'accesso raw ai pacchetti di rete consente alle applicazioni di intercettare e modificare il traffico di rete, le VPN basate sulle applicazioni non possono essere avviate automaticamente e richiedono sempre l'intervento dell'utente. Inoltre, alla connessione di una VPN viene visualizzata una notifica continua. La finestra di avviso di connessione per una VPN basata sulle applicazioni è simile a quella mostrata nella Figura 9.10.

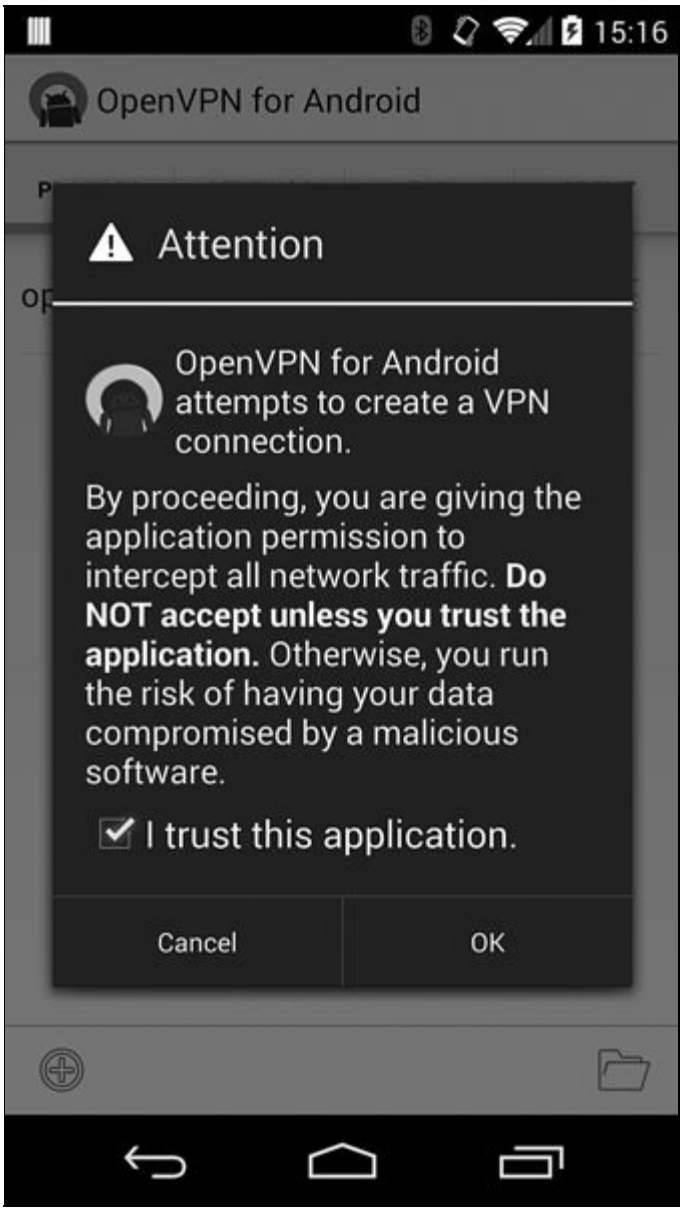

Figura 9.10 Finestra di avviso per una connessione VPN basata sulle applicazioni.

Dichiarazione di una VPN

Una VPN basata sulle applicazioni viene implementata creando un componente del servizio che estende la classe base `VpnService` e registrandola nel manifest dell'applicazione, come mostrato nel Listato 9.11.

Listato 9.11 Registrazione di un servizio VPN nel manifest dell'applicazione.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.vpn">
    --altro codice--
    <application android:label="@string/app">
         --altro codice--
         <service android:name=".MyVpnService"
                  android:permission="android.permission.BIND_VPN_SERVICE">(1)
            <intent-filter>
```

```
            <action android:name="android.net.VpnService"/>(2)
        </intent-filter>
    </service>
</application>
</manifest>:
```

Il servizio deve avere un filtro di intent corrispondente all'azione intent `android.net.VpnService` **(2)** affinché il sistema possa eseguire il binding al servizio e controllarlo. Inoltre, il servizio deve richiedere il permesso di firma di sistema `BIND_VPN_SERVICE` **(1)**, che garantisce che solo le applicazioni di sistema possano effettuare il binding al servizio.

Preparazione della VPN

Per registrare una nuova connessione VPN al sistema, l'applicazione chiama `VpnService.prepare()` per ottenere il permesso di esecuzione e poi chiama il metodo `establish()` per creare un tunnel di rete (come spiegato nel prossimo paragrafo). Il metodo `prepare()` restituisce un intent usato per aprire la finestra di avviso mostrata nella Figura 9.10. La finestra serve per ottenere il permesso dell'utente e assicurare che in un determinato momento sia in esecuzione una sola connessione VPN per utente. Se `prepare()` viene chiamato mentre è attiva una connessione VPN creata da un'altra applicazione, tale connessione viene terminata. Il metodo `prepare()` salva il nome del package dell'applicazione chiamante, che è l'unica a poter avviare una connessione VPN finché non viene chiamato di nuovo il metodo; in caso contrario il sistema interrompe la connessione VPN (per esempio se il processo dell'app VPN subisce un arresto anomalo). Se una connessione VPN viene disattivata per qualsiasi motivo, il sistema chiama il metodo `onRevoke()` dell'implementazione `VpnService` dell'applicazione VPN corrente.

Effettuazione di una connessione VPN

Dopo aver preparato un'applicazione VPN e aver ottenuto il permesso di esecuzione, l'applicazione può avviare il suo componente `VpnService`, che generalmente crea un tunnel al gateway VPN e negozia i parametri di rete per la connessione VPN. A seguire, imposta la classe `VpnService.Builder`

usando questi parametri e chiama `VpnService.establish()` per ricevere un descrittore di file per la lettura e la scrittura dei pacchetti. Il metodo `establish()` verifica di essere stato chiamato dall'applicazione che attualmente ha il permesso di stabilire una connessione VPN confrontando l'UID del chiamante con l'UID dell'applicazione che detiene il grant. `establish()` verifica quindi se l'utente Android corrente può creare connessioni VPN e controlla che il servizio richieda il permesso `BIND_VPN_SERVICE`; in caso contrario, viene considerato non sicuro e viene generata una `SecurityException`. Infine, il metodo `establish()` crea e configura un'interfaccia tunnel usando il codice nativo e configura il routing e i server DNS.

Notifica all'utente della connessione VPN

L'ultimo passo per stabilire una connessione VPN consiste nel mostrare una notifica costante che segnala all'utente che il traffico di rete viene incanalato attraverso una VPN; in questo modo l'utente può monitorare e controllare la connessione dalla finestra di controllo associata. La finestra per l'applicazione *OpenVPN for Android* è mostrata nella Figura 9.11.

Figura 9.11 Finestra di gestione della VPN basata sulle applicazioni.

Questa finestra è parte del package dedicato `com.android.vpndialogs`, l'unico package, oltre all'utente `system`, a cui è esplicitamente consentito gestire le connessioni VPN basate sulle applicazioni. Si garantisce così che una connessione VPN possa essere avviata e gestita solamente dalla finestra di sistema.

Utilizzando il framework della VPN basata sulle applicazioni le applicazioni possono implementare il tunneling di rete con i metodi di autenticazione e crittografia richiesti. Tutti i pacchetti inviati o ricevuti dal dispositivo passano nell'applicazione VPN, che può quindi essere utilizzata, oltre che per il tunneling, per registrare, filtrare o modificare il traffico (per esempio rimuovendo gli annunci pubblicitari).

NOTA

Per un'implementazione completa di una VPN basata sulle applicazioni che sfrutta l'archivio delle credenziali di Android per gestire certificati e chiavi di autenticazione, fate riferimento al codice sorgente di OpenVPN for Android (Arne Schwabe, "Openvpn for Android 4.0+", https://code.google.com/p/ics-openvpn/). Questa applicazione implementa un client VPN SSL totalmente compatibile con il server OpenVPN.

Supporto multiutente

Come già affermato, sui dispositivi multiutente le VPN legacy possono essere controllate solo dall'utente proprietario. Tuttavia, con l'introduzione del supporto multiutente, Android 4.2 e versioni successive consente a tutti gli utenti secondari (fatta eccezione per i profili con restrizioni, che devono condividere la connessione VPN dell'utente primario) di avviare VPN basate sulle applicazioni. Anche se tecnicamente questa modifica permette a ogni utente di avviare una propria VPN, visto che è possibile attivare una sola VPN basata sulle applicazioni per volta, il traffico di tutti gli utenti del dispositivo viene instradato attraverso la VPN attualmente attiva, indipendentemente da chi l'ha avviata. È solo da Android 4.4 che è disponibile un vero supporto VPN multiutente, grazie all'introduzione delle *VPN per utente*, che consentono di instradare il traffico di ogni utente attraverso la sua VPN, isolandolo così dal traffico degli altri utenti.

Routing avanzato di Linux

Android usa diverse funzioni avanzate di routing e filtro dei pacchetti del kernel Linux per implementare le VPN per utente. Queste funzioni (implementate dal framework del kernel *netfilter*) includono il modulo *owner* dello strumento Linux `iptables`, che consente la corrispondenza tra pacchetti generati in locale e basati su UID, GID o PID del processo che li ha creati. Per esempio, il comando nel punto **(1)** del Listato 9.12 crea una regola di packet filtering che scarta tutti i pacchetti in uscita generati dall'utente con UID 1234.

Listato 9.12 Uso della corrispondenza del proprietario e della segnalazione dei pacchetti con iptables.

```
# iptables -A OUTPUT -m owner --uid-owner 1234 -j DROP(1)
# iptables -A PREROUTING -t mangle -p tcp --dport 80 -j MARK --set-mark 0x1(2)
```

```
# ip rule add fwmark 0x1 table web(3)
# ip route add default via 1.2.3.4 dev em3 table web(4)
```

Un'altra importante funzionalità di netfilter è la capacità di contrassegnare i pacchetti che corrispondono a un particolare selettore con un numero specifico (detto *mark*). Per esempio, la regola in **(2)** contrassegna tutti i pacchetti destinati alla porta `80` (tipicamente usata da un web server) con il mark `0x1`. Questo mark può quindi essere usato per la corrispondenza in regole di filtro o routing successive, per esempio per inviare pacchetti con mark attraverso una particolare interfaccia (aggiungendo una regola di routing che invia i pacchetti con mark a una tabella di routing predefinita, nel nostro esempio `web` **(3)**). Infine, è possibile aggiungere una route che invia i pacchetti corrispondenti alla tabella `web` all'interfaccia `em3`, utilizzando il comando mostrato nel punto **(4)**.

Implementazione di VPN multiutente

Android usa queste funzioni di routing e filtraggio dei pacchetti per contrassegnare i pacchetti che hanno origine da tutte le app di un particolare utente Android e per inviarli attraverso l'interfaccia di tunneling creata dall'app VPN avviata dall'utente. Quando è l'utente proprietario ad avviare una VPN, questa viene condivisa con qualunque profilo con restrizioni sul dispositivo che non può avviare la propria VPN creando una corrispondenza con tutti i pacchetti che hanno origine dai profili con restrizioni e instradando i pacchetti nel tunnel VPN del proprietario.

Questo routing diviso (o *split-routing*) è implementato a livello del framework da `NetworkManagementService`, che fornisce le API di gestione del routing e della corrispondenza dei pacchetti in base all'UID o all'intervallo di UID. `NetworkManagementService` implementa queste API inviando comandi al daemon `netd` nativo, eseguito come root, e può quindi modificare le tabelle di routing e di packet filtering del kernel. `netd` manipola la configurazione di routing e filtraggio del kernel chiamando le utility userland `iptables` e `ip`.

Illustriamo il routing VPN per utente di Android sfruttando l'esempio del Listato 9.13. L'utente primario (user ID 0) e il primo utente secondario (user ID 10) hanno avviato ognuno una VPN basata sulle applicazioni. Alla VPN dell'utente proprietario viene assegnata l'interfaccia di tunneling `tun0`, mentre alla VPN dell'utente secondario è assegnata l'interfaccia `tun1`. Sul dispositivo esiste anche un profilo con restrizioni con user ID 13. Il Listato 9.13 mostra lo stato delle tabelle di packet filtering del kernel quando sono connesse entrambe le VPN (alcuni dettagli sono stati omessi).

Listato 9.13 Regole di corrispondenza dei pacchetti per le VPN avviate da due utenti diversi.

```
# iptables -t mangle -L -n
--altro codice--
Chain st_mangle_OUTPUT (1 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0           mark match 0x1(1)
RETURN     all  --  0.0.0.0/0            0.0.0.0/0           owner UID match 1016(2)
--altro codice--
st_mangle_tun0_OUTPUT  all  --  0.0.0.0/0           0.0.0.0/0           [goto]  owner UID match
0-99999(3)
st_mangle_tun0_OUTPUT  all  --  0.0.0.0/0           0.0.0.0/0           [goto]  owner UID match
1300000-1399999(4)
st_mangle_tun1_OUTPUT  all  --  0.0.0.0/0           0.0.0.0/0           [goto]  owner UID match
1000000-1099999(5)

Chain st_mangle_tun0_OUTPUT (3 references)
target     prot opt source               destination
MARK       all  --  0.0.0.0/0            0.0.0.0/0           MARK and 0x0
MARK       all  --  0.0.0.0/0            0.0.0.0/0           MARK set 0x3c(6)

Chain st_mangle_tun1_OUTPUT (2 references)
target     prot opt source               destination
MARK       all  --  0.0.0.0/0            0.0.0.0/0           MARK and 0x0
MARK       all  --  0.0.0.0/0            0.0.0.0/0           MARK set 0x3d(7)
```

I pacchetti in uscita vengono prima inviati alla catena `st_mangle_OUTPUT`, responsabile della corrispondenza e dell'applicazione dei mark ai pacchetti. I pacchetti esentati dal routing per utente (quelli già contrassegnati con `0x1` **(1)**) e i pacchetti che hanno origine da VPN legacy (UID `1016` **(2)**, assegnato all'utente predefinito `vpn`, con cui sono in esecuzione `mtd` e `racoon`) passano senza modifiche.

A seguire, i pacchetti creati dai processi in esecuzione con UID tra `0` e `99999` (l'intervallo degli UID assegnati alle app avviate dall'utente primario, come spiegato nel Capitolo 4) vengono abbinati e inviati alla catena `st_mangle_tun0_OUTPUT` **(3)**. I pacchetti che hanno origine dagli UID `1300000`-`1399999`, l'intervallo assegnato al nostro profilo con restrizioni (user ID `13`), sono

inviati alla stessa catena **(4)**. In pratica, il traffico originato dall'utente proprietario e dal profilo con restrizioni viene trattato allo stesso modo. I pacchetti creati dal primo utente secondario (user ID `10`, intervallo di UID `1000000`-`1099999`) sono invece inviati a una catena diversa, `st_mangle_tun1_OUTPUT` **(5)**. Le catene target in sé sono semplici: `st_mangle_tun0_OUTPUT` cancella il mark dei pacchetti e applica loro il mark `0x3c` **(6)**; `st_mangle_tun1_OUTPUT` fa lo stesso usando il mark `0x3d` **(7)**. Dopo l'applicazione dei mark ai pacchetti, i mark sono usati per implementare e abbinare regole di routing diverse, come mostrato nel Listato 9.14.

Listato 9.14 Regole di routing per le VPN avviate da due utenti diversi.

```
# ip rule ls
0:      from all lookup local
100:    from all fwmark 0x3c lookup 60(1)
100:    from all fwmark 0x3d lookup 61(2)
--altro codice--
# ip route list table 60
default dev tun0  scope link(3)
# ip route list table 61
default dev tun1  scope link(4)
```

Sono state create due regole per la corrispondenza di ogni mark, associate a tabelle di routing diverse. I pacchetti con il mark `0x3c` vengono inviati alla tabella di routing `60` (`0x3c` in esadecimale **(1)**), mentre quelli con il mark `0x3d` vanno alla tabella `61` (`0x3d` in esadecimale **(2)**). La tabella `60` effettua il routing di tutto il traffico attraverso l'interfaccia di tunneling `tun0` **(3)**, creata dall'utente proprietario, mentre la tabella `61` effettua il routing attraverso l'interfaccia `tun1` **(4)**, creata dall'utente secondario.

NOTA

Anche se il metodo di routing del traffico VPN introdotto in Android 4.4 garantisce una flessibilità superiore e consente l'isolamento del traffico VPN utente, attualmente l'implementazione sembra presentare qualche problema, soprattutto per quanto riguarda il passaggio tra reti fisiche diverse (per esempio da mobile a Wi-Fi o viceversa). Questi problemi saranno affrontati nelle versioni future, probabilmente modificando l'associazione delle catene di packet filtering alle interfacce, ma la strategia di implementazione di base rimarrà verosimilmente la stessa.

EAP Wi-Fi

Android supporta diversi protocolli di rete wireless, tra cui *Wi-Fi Protected Acces*s (WPA) e Wi-*Fi Protected Access II* (WPA2), attualmente implementati sulla maggior parte dei dispositivi wireless. Entrambi i protocolli supportano una semplice modalità a *chiave precondivisa* (PSK), detta anche *modalità personale*, in cui tutti i dispositivi che accedono alla rete devono essere configurati con la stessa chiave di autenticazione a 256 bit.

I dispositivi possono essere configurati sia con i byte della chiave raw sia con una passphrase ASCII usata per derivare la chiave di autenticazione utilizzando l'algoritmo di derivazione della chiave PBKDF2. Pur essendo semplice, la modalità PSK non è scalabile all'aumentare del numero di utenti di rete: se per esempio occorre revocare l'accesso a un determinato utente, l'unico modo per cancellare le relative credenziali di rete è cambiare la passphrase condivisa, operazione che impone a tutti gli altri utenti di riconfigurare i loro dispositivi. Inoltre, poiché non vi sono metodi pratici per distinguere utenti e dispositivi, è difficile implementare regole di accesso flessibili o funzioni di accounting.

Per risolvere il problema, sia WPA sia WPA2 supportano lo standard di controllo di accesso alla rete IEEE 802.1X, che offre un incapsulamento di *Extensible Authentication Protocol* (EAP). L'autenticazione in una rete wireless usa 802.1X e coinvolge un supplicant, un autenticatore e un server di autenticazione (Figura 9.12).

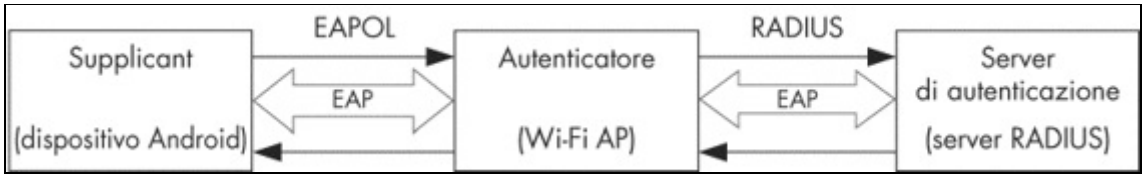

Figura 9.12 Partecipanti all'autenticazione 802.1X.

Il *supplicant* è un dispositivo wireless, per esempio uno smartphone Android, che vuole connettersi alla rete, mentre l'*autenticatore* è il punto di accesso alla rete che convalida l'identità del supplicant e fornisce

l'autorizzazione. In una tipica configurazione Wi-Fi, l'autenticatore è il punto di accesso wireless (AP). Il *server di autenticazione*, solitamente un server RADIUS, verifica le credenziali del client e decide se deve poter accedere sulla base di una policy di accesso preconfigurata. L'autenticazione viene implementata scambiando messaggi EAP tra i tre nodi. Questi sono incapsulati in un formato adatto per il mezzo che connette i due nodi: *EAP over LAN* (EAPOL) tra il supplicant e l'autenticatore e *RADIUS* tra l'autenticatore e il server di autenticazione.

EAP è un framework di autenticazione che supporta diversi tipi di autenticazione concreti e non un meccanismo di autenticazione concreto, pertanto il supplicant e il server di autenticazione (con l'aiuto dell'autenticatore) devono negoziare un metodo di autenticazione supportato da entrambi prima di eseguire l'autenticazione. Esistono vari metodi di autenticazione EAP standard e proprietari; le versioni attuali di Android supportano la maggior parte dei metodi usati nelle reti wireless.

Nel prossimo paragrafo è disponibile una breve panoramica dei metodi di autenticazione EAP supportati da Android, con una descrizione delle modalità di protezione delle credenziali per ognuno. Vedremo inoltre come configurare l'accesso a una rete Wi-Fi che usa EAP per l'autenticazione usando le API di gestione della rete wireless di Android.

Metodi di autenticazione EAP

Nella versione 4.4 Android supporta i metodi di autenticazione PEAP, EAP-TLS, EAP-TTLS ed EAP-PWD. Prima di vedere come Android archivia le credenziali per ogni metodo di autenticazione, vediamo brevemente il funzionamento di ogni metodo.

PEAP

Protected Extensible Authentication Protocol (PEAP) trasmette i messaggi EAP su una connessione SSL per garantire riservatezza e integrità. Usa PKI e un certificato del server per autenticare il server e stabilire una connessione SSL (fase 1), ma non impone la modalità di autenticazione dei client. I client vengono autenticati con un secondo metodo di autenticazione interno (fase 2), trasmesso nel tunnel SSL.

Android supporta i metodi MSCHAPv2 (specificato in PEAPv0; Vivek Kamath, Ashwin Palekar e Mark Woodrich, *Microsoft's PEAP version 0 (Implementation in Windows XP SP1)*, http://bit.ly/1qAPl9T) e *Generic Token Card* (GTC, specificato in PEAPv2; Ashwin Palekar *et al.*, *Protected EAP Protocol (PEAP) Version 2*, http://bit.ly/1txg0Lv) per l'autenticazione della fase 2.

EAP-TLS

Il metodo *EAP-Transport Layer Security* (EAP-TLS; Arne Schwabe, "Openvpn for Android 4.0+", https://code.google.com/p/ics-openvpn/) usa TLS per l'autenticazione reciproca, e in passato era l'unico metodo EAP certificato per l'uso con WPA Enterprise. EAP-TLS usa un certificato del server per autenticare il server ai supplicant e un certificato del client che il server di autenticazione verifica per stabilire l'identità del supplicant. Il grant dell'accesso di rete richiede il rilascio e la distribuzione di certificati client X.509 e di conseguenza la gestione di un'infrastruttura a chiave pubblica (PKI). L'accesso alla rete dei client esistenti può essere impedito revocando i loro certificati supplicant. Android supporta EAP-TLS e gestisce certificati e chiavi del client utilizzando l'archivio delle credenziali di sistema.

EAP-TTLS

Come EAP-TLS, il protocollo *EAP-Tunneled Transport Layer Security* (EAP-TTLS) si basa su TLS (P. Funk e S. Blake-Wilson, *Extensible Authentication Protocol Tunneled Transport Layer Security Authenticated Protocol Version 0 (EAP-TTLSv0)*, https://tools.ietf.org/html/rfc5281/). EAP-TTLS non richiede però l'autenticazione client con i certificati X.509. I client possono essere autenticati usando un certificato nella fase di handshake (fase 1) o con un altro protocollo nella fase di tunneling (fase 2). Android non supporta l'autenticazione nella fase 1, ma supporta i protocolli PAP, MSCHAP, MSCHAPv2 e GTC nella fase 2.

EAP-PWD

Il metodo di autenticazione EAP-PWD usa una password condivisa per l'autenticazione (D. Harkins e G. Zorn, *Extensible Authentication*

Protocol (EAP) Authentication Using Only a Password, https://tools.ietf.org/html/rfc5931/). A differenza degli schemi legacy che fanno affidamento su un semplice meccanismo challenge-response, EAP-PWD è studiato per resistere agli attacchi passivi, attivi e con dizionario. Il protocollo fornisce inoltre la forward secrecy e garantisce che le sessioni precedenti non possano essere decodificate anche in caso di compromissione della password. EAP-PWD è basato sulla crittografia a logaritmo discreto e può essere implementato usando campi finiti o curve ellittiche.

Architettura Wi-Fi di Android

Come la maggior parte del supporto hardware in Android, la sua architettura Wi-Fi prevede uno strato kernel (moduli driver dell'adattatore WLAN), un daemon nativo (`wpa_supplicant`), un *Hardware Abstraction Layer* (HAL), servizi di sistema e un'interfaccia di sistema. I driver del kernel per l'adattatore Wi-Fi sono generalmente specifici per il "system on a chip" (SoC) su cui è basato il dispositivo Android e in genere sono a sorgente chiusa e caricati come moduli del kernel. `wpa_supplicant` è un daemon supplicant WPA che implementa la negoziazione della chiave con un autenticatore WPA e controlla l'associazione 802.1X del driver WLAN (Jouni Malinen, *Linux WPA/WPA2/IEEE 802.1X Supplicant,* http://hostap.epitest.fi/wpa_supplicant/). Tuttavia, è raro che i dispositivi Android includano il codice originale di `wpa_supplicant`; l'implementazione inclusa è spesso modificata per una migliore compatibilità con il SoC sottostante.

HAL è implementato nella libreria nativa *libharware_legacy* ed è responsabile dell'inoltro dei comandi dal framework a `wpa_supplicant` tramite il relativo socket di controllo. Il servizio di sistema che controlla la connettività Wi-Fi è `WifiService`, che offre un'interfaccia pubblica tramite la classe di facciata `WifiManager`. `WifiService` delega la gestione dello stato Wi-Fi a una classe complessa, `WifiStateMachine`, che può attraversare più di una decina di stati durante la connessione a una rete wireless.

La connettività WLAN è controllata tramite la schermata Wi-Fi dell'app *Settings*; lo stato della connettività è visualizzato sulla barra di stato e in *Quick Settings*, entrambi parte del package SystemUI.

Android salva i file di configurazione Wi-Fi nella directory */data/misc/wifi/*, perché i daemon di connettività wireless applicano la persistenza delle modifiche alla configurazione direttamente su disco e necessitano pertanto di una directory scrivibile. La directory è di proprietà dell'utente `wifi` (UID 1010), che è anche l'utente di esecuzione di `wpa_supplicant`. Per i file di configurazione, tra cui `wpa_supplicant.conf`, i permessi sono impostati su 0660, la proprietà è dell'utente `system` e il gruppo è impostato su `wifi`. In questo modo si garantisce che sia le applicazioni di sistema sia il daemon del supplicant possano leggere e modificare i file di configurazione, ma non siano accessibili alle altre applicazioni. Il file `wpa_supplicant.conf` contiene i parametri di configurazione, formattati come coppie chiave-valore, sia globali sia specifici per una rete. I parametri specifici della rete sono racchiusi in blocchi di rete, come mostrato nel Listato 9.15 per una configurazione PSK.

Listato 9.15 Blocco di configurazione della rete PSK in wpa_supplicant.conf.

```
network={
    ssid="psk-ap"(1)
    key_mgmt=WPA-PSK(2)
    psk="password"(3)
    priority=805(4)
}
```

Come potete osservare, il blocco `network` specifica il SSID di rete **(1)**, il protocollo di gestione delle chiavi di autenticazione **(2)**, la chiave precondivisa **(3)** e un valore di priorità **(4)**. PSK è salvato come testo normale; pertanto, anche se i bit di accesso di `wpa_supplicant.conf` impediscono l'accesso alle applicazioni non di sistema, è possibile estrarlo facilmente dai dispositivi che consentono l'accesso root.

Gestione delle credenziali EAP

In questo paragrafo vedremo come Android gestisce le credenziali Wi-Fi per ciascuno dei metodi di autenticazione EAP supportati ed esamineremo le modifiche a `wpa_supplicant` specifiche per Android che

consentono al daemon supplicant di trarre vantaggio dall'archivio delle credenziali di sistema di Android.

Il Listato 9.16 mostra il blocco della rete in `wpa_supplicant.conf` per una rete configurata per l'uso di PEAP.

Listato 9.16 Blocco di configurazione della rete PEAP in wpa_supplicant.conf.

```
network={
    ssid="eap-ap"
    key_mgmt=WPA-EAP IEEE8021X(1)
    eap=PEAP(2)
    identity="android1"(3)
    anonymous_identity="anon"
    password="password"(4)
    ca_cert="keystore://CACERT_eapclient"(5)
    phase2="auth=MSCHAPV2"(6)
    proactive_key_caching=1
}
```

Qui la modalità di gestione delle chiavi è impostata su `WPA-EAP IEEE8021X` **(1)**, il metodo `eap` su `PEAP` **(2)** e l'autenticazione della fase 2 su `MSCHAPv2` **(6)**. Le credenziali, nello specifico l'identità **(3)** e la password **(4)**, sono salvate come testo normale nel file di configurazione, proprio come nella modalità PSK.

Una differenza degna di nota rispetto a un `wpa_supplicant.conf` generico è il formato del percorso dei certificati CA **(5)**. Il percorso dei certificati CA (`ca_cert`) è usato durante la convalida del certificato del server; in Android `ca_cert` è in un formato tipo URI con lo schema `keystore`. Questa estensione specifica per Android consente al daemon `wpa_supplicant` di recuperare certificati dall'archivio delle credenziali di sistema. Quando il daemon incontra un percorso dei certificati che inizia con `keystore://`, si connette all'interfaccia remota `IKeystoreService` del servizio `keystore` nativo e recupera i byte del certificato usando il percorso URI come chiave.

La configurazione di EAP-TLS è simile a quella di PEAP, come mostrato nel Listato 9.17.

Listato 9.17 Blocco di configurazione della rete AP-TLS in wpa_supplicant.conf.

```
network={
    ssid="eap-ap"
    key_mgmt=WPA-EAP IEEE8021X
    eap=TLS
    identity="android1"
    ca_cert="keystore://CACERT_eapclient"
    client_cert="keystore://USRCERT_eapclient"(1)
    engine_id="keystore"(2)
    key_id="USRPKEY_eapclient"(3)
    engine=1
    priority=803
```

```
    proactive_key_caching=1
}
```

La novità qui è l'aggiunta di un URI del certificato client **(1)**, di un engine ID **(2)** e di un key ID **(3)**. Il certificato client viene recuperato dall'archivio delle credenziali di sistema, proprio come il certificato CA. L'engine ID fa riferimento all'engine OpenSSL da usare per le operazioni di crittografia durante la connessione al SSID configurato nel blocco `network`. `wpa_supplicant` dispone del supporto nativo per gli engine OpenSSL configurabili ed è spesso usato con un engine PKCS#11 per impiegare le chiavi salvate su una smart card o su un altro dispositivo hardware.

Come abbiamo già visto nel Capitolo 7, l'engine `keystore` di Android usa le chiavi salvate nell'archivio delle credenziali di sistema. Se un dispositivo supporta l'archiviazione delle credenziali hardware, l'engine `keystore` può sfruttarla in maniera trasparente in virtù del modulo HAL *keymaster* intermedio. Il key ID nel Listato 9.17 fa riferimento all'alias della chiave privata da usare per l'autenticazione.

A partire dalla versione 4.3, Android consente di selezionare il proprietario delle chiavi private e dei certificati in fase di importazione. In precedenza, tutte le chiavi importate erano di proprietà dell'utente `system`, ma se impostate il parametro *Credential use* su *Wi-Fi* nella finestra di importazione (Figura 9.13), il proprietario della chiave viene impostato sull'utente `wifi` (UID 1010) e alla chiave possono accedere solo i componenti del sistema in esecuzione con l'utente `wifi`, come `wpa_supplicant`.

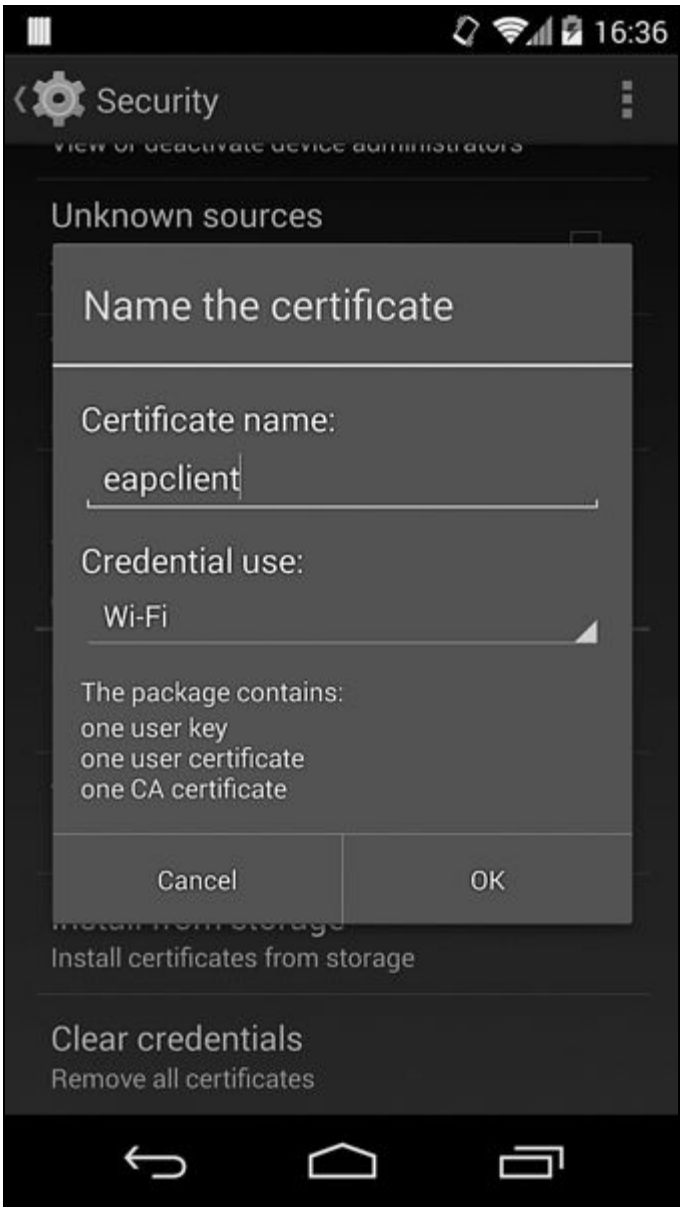

Figura 9.13 Impostazione del proprietario delle credenziali su Wi-Fi nella finestra di importazione PCKS#12.

Visto che Android non supporta l'autenticazione client durante l'uso del metodo di autenticazione EAP-TTLS, la configurazione contiene solo un riferimento al certificato CA **(2)**, come mostrato nel Listato 9.18. La password **(1)** è archiviata in chiaro.

Listato 9.18 Blocco di configurazione della rete EAP-TLS in wpa_supplicant.conf.

```
network={
    ssid="eap-ap"
    key_mgmt=WPA-EAP IEEE8021X
    eap=TTLS
    identity="android1"
    anonymous_identity="anon"
    password="pasword"(1)
    ca_cert="keystore://CACERT_eapclient"(2)
    phase2="auth=GTC"
    proactive_key_caching=1
}
```

Il metodo `EAP-PWD` non dipende da TLS per stabilire un canale sicuro e quindi non richiede la configurazione dei certificati, come mostrato nel Listato 9.19. Le credenziali sono salvate come testo normale (**(1)** e **(2)**), come le altre configurazioni che usano le password.

Listato 9.19 Blocco di configurazione della rete EAP-PWD in wpa_supplicant.conf.

```
network={
    ssid="eap-ap"
    key_mgmt=WPA-EAP IEEE8021X
    eap=PWD
    identity="android1"(1)
    password="password"(2)
    proactive_key_caching=1
}
```

In sintesi, le configurazioni per tutti i metodi `eap` che usano una password per l'autenticazione salvano le informazioni sulle credenziali in testo normale all'interno del file `wpa_supplicant.conf`. Se usate EAP-TLS, che fa affidamento sull'autenticazione del client, la chiave del client è salvata nel keystore di sistema e di conseguenza offre il più alto livello di protezione delle credenziali.

Aggiunta di una rete EAP con WifiManager

Anche se Android supporta numerosi metodi di autenticazione WPA Enterprise, la loro configurazione corretta potrebbe creare difficoltà ad alcuni utenti a causa del numero di parametri da impostare e della necessità di installare e selezionare i certificati di autenticazione. L'API ufficiale di Android per la gestione delle reti Wi-Fi, `WifiManager`, non supportava le configurazioni EAP prima di Android 4.3, pertanto l'unico modo per configurare una rete EAP era aggiungerla tramite l'app *Settings* di sistema e configurarla manualmente. Android 4.3 (API livello 18) ha esteso l'API per consentire la configurazione programmatica di EAP, permettendo così il provisioning automatico della rete negli ambienti aziendali.

In questo paragrafo vedremo come usare `WifiManager` per aggiungere una rete EAP-TLS e ne esamineremo l'implementazione sottostante.

`WifiManager` consente a un'app che ha il permesso `CHANGE_WIFI_STATE` (livello di protezione *dangerous*) di aggiungere una rete Wi-Fi inizializzando

un'istanza di `WifiConfiguration` con il SSID della rete, gli algoritmi di autenticazione e le credenziali e di passare tale istanza al metodo `addNetwork()` di `WifiManager`. Android 4.3 estende questa API aggiungendo un campo `enterpriseConfig` di tipo `WifiEnterpriseConfig` alla classe `WifiConfiguration`, che consente di configurare il metodo di autenticazione EAP da utilizzare, i certificati client e CA, il metodo di autenticazione della fase 2 (se presente) ed eventuali credenziali supplementari quali nome utente e password. Il Listato 9.20 mostra come usare questa API per aggiungere una rete che impiega EAP-TLS per l'autenticazione.

Listato 9.20 Aggiunta di una rete EAP-TLS con WifiManager.

```
X509Certificate caCert = getCaCert();
PrivateKey clientKey = getClientKey();
X509Certificate clientCert = getClientCert();

WifiEnterpriseConfig enterpriseConfig = new WifiEnterpriseConfig();
enterpriseConfig.setCaCertificate(caCert);(1)
enterpriseConfig.setClientKeyEntry(clientKey, clientCert);(2)
enterpriseConfig.setEapMethod(WifiEnterpriseConfig.Eap.TLS);(3)
enterpriseConfig.setPhase2Method(WifiEnterpriseConfig.Phase2.NONE);(4)
enterpriseConfig.setIdentity("android1");(5)

WifiConfiguration config = new WifiConfiguration();
config.enterpriseConfig = enterpriseConfig;(6)
config.SSID = "\"eap-ap\"";
config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.IEEE8021X);(7)
config.allowedKeyManagement.set(WifiConfiguration.KeyMgmt.WPA_EAP);(8)

int netId = wm.addNetwork(config);(9)
if (netId != -1) {
    boolean success = wm.saveConfiguration();(10)
}
```

Per configurare l'autenticazione EAP-TLS è necessario per prima cosa ottenere il certificato CA usato per verificare l'identità del server, nonché la chiave privata e il certificato del client. Queste informazioni sono generalmente distribuite con un file PKCS#12, pertanto possiamo usare un `KeyStore` di tipo PKCS12 per estrarle (procedura non mostrata). Android importerà automaticamente le chiavi e i certificati specificati nel keystore di sistema quando aggiungete un profilo EAP che li utilizza, pertanto non è necessario importare il file PKCS#12. Una volta ottenuti il certificato CA e le credenziali client, li impostiamo sulla nostra istanza di `WifiEnterpriseConfig` utilizzando i metodi `setCaCertificate()` **(1)** e `setClientKeyEntry()` **(2)**. Impostiamo quindi il metodo EAP su `Eap.TLS` **(3)** e il metodo della fase 2 su `NONE` **(4)**, in quanto EAP-TLS autentica gli utenti durante

l'effettuazione della connessione SSL (fase 1). Android richiede inoltre di impostare l'identità **(5)** anche se questa potrebbe non essere impiegata dal server di autenticazione. Dopo aver configurato l'oggetto `WifiEnterpriseConfig`, possiamo aggiungerlo all'istanza `WifiConfiguration` principale **(6)**. È inoltre necessario configurare il set di protocolli di gestione delle chiavi (**(7)** e **(8)**) perché per impostazione predefinita corrisponde a WPA PSK. Infine, possiamo aggiungere la rete **(9)** e salvare la configurazione **(10)**, aggiornando il file `wpa_supplicant.conf` per includere la rete appena configurata.

Android genera automaticamente gli alias per la chiave privata e i certificati configurati e importa le credenziali PKI nel keystore di sistema. Gli alias sono basati sul nome AP, sullo schema di gestione delle chiavi e sul metodo di autenticazione EAP. Una rete configurata programmaticamente viene mostrata in automatico nella schermata *Wi-Fi* dell'applicazione *Settings*, che può apparire come mostrato nella Figura 9.14 per l'esempio del Listato 9.20.

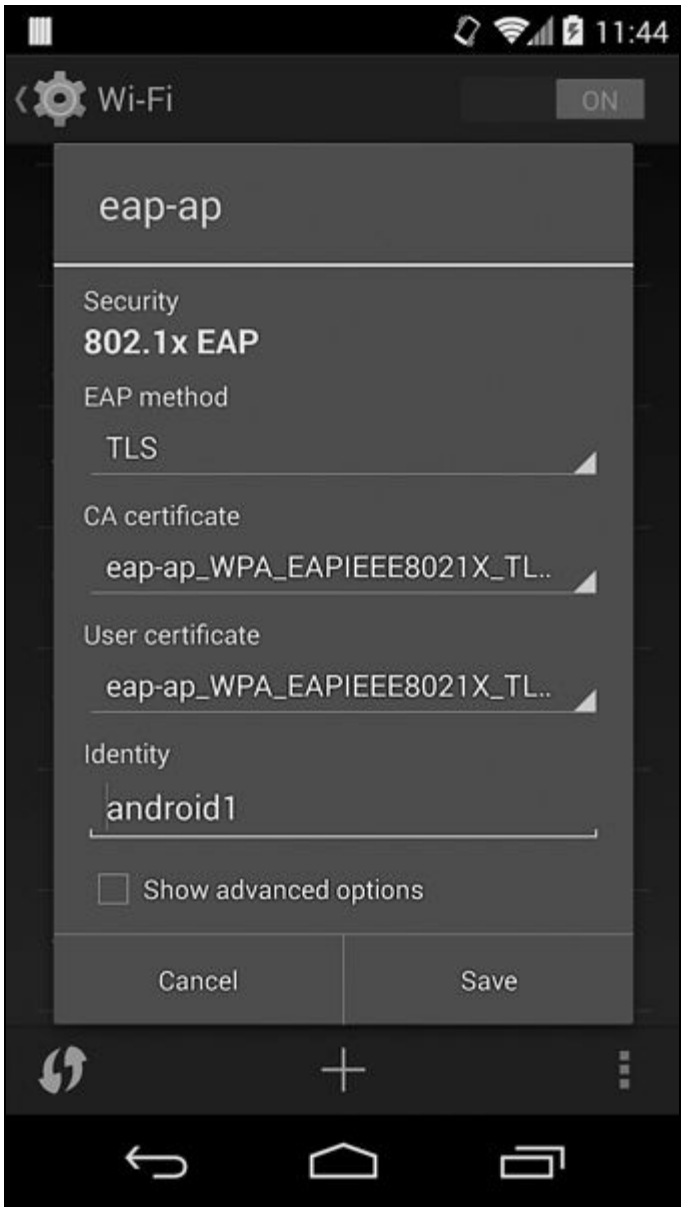

Figura 9.14 Una rete EAP-TLS aggiunta con WifiManager.

Riepilogo

Android supporta un'API Device Administration che consente alle app di amministrazione del dispositivo di configurare una policy di sicurezza contenente i requisiti per la complessità della password della schermata di blocco, la crittografia del dispositivo e l'uso della fotocamera. Gli amministratori del dispositivo sono spesso usati con gli account aziendali, come quelli di Microsoft Exchange e Google Apps, per limitare l'accesso ai dati aziendali sulla base della policy e delle impostazioni del dispositivo. L'API Device Administration fornisce anche funzionalità per abilitare il blocco del dispositivo e la cancellazione dei dati da remoto.

I dispositivi Android possono connettersi a vari tipi di VPN, tra cui PPTP, L2TP/IPSec e basate su SSL. Il supporto per PPTP e L2TP/IPSec è integrato nella piattaforma e può essere esteso solo tramite aggiornamenti del sistema operativo. Android 4.0 aggiunge il supporto per le VPN basate sulle applicazioni, che consentono alle applicazioni di terze parti di implementare soluzioni VPN personalizzate.

Oltre alla diffusa modalità di autenticazione Wi-Fi a chiave precondivisa, Android supporta varie configurazioni WPA Enterprise, nello specifico PEAP, EAP-TLS, EAP-TTLS ed EAP-PWD. I certificati e le chiavi private per i metodi di autenticazione EAP che usano SSL per stabilire un canale sicuro o autenticare gli utenti sono salvati nel keystore di sistema e possono utilizzare la protezione hardware, se disponibile. Le reti Wi-Fi che usano EAP per l'autenticazione possono essere sottoposte a provisioning automatico usando l'API `WifiManager` nelle versioni di Android più recenti (dalla 4.3).

Sicurezza del dispositivo

Finora ci siamo concentrati sull'implementazione in Android del sandboxing e della separazione dei privilegi per isolare le applicazioni l'una dall'altra e dal sistema operativo core. In questo capitolo vedremo come Android garantisce l'integrità del sistema operativo e protegge i dati del dispositivo dai "criminali" che hanno accesso fisico a un device. Cominceremo con una breve descrizione del bootloader e del sistema operativo di recovery, dopodiché affronteremo la funzionalità di boot verificato di Android, che garantisce che la partizione *system* non venga modificata da programmi dannosi. A seguire vedremo come viene crittografata la partizione *userdata*, che ospita i file di configurazione del sistema operativo e i dati delle applicazioni, per garantire che il dispositivo non possa essere avviato senza la password di decodifica e che i dati utente non possano essere estratti nemmeno con l'accesso diretto alla memoria flash del dispositivo. Vedremo poi come viene implementata la funzionalità di blocco dello schermo e come vengono salvati sul dispositivo pattern, PIN e passphrase di sblocco.

Parleremo anche del debug USB sicuro, che autentica gli host che si connettono al daemon *Android Debug Bridge* (ADB) tramite USB e richiede agli utenti di consentire esplicitamente l'accesso per ogni host. Poiché l'accesso ADB su USB permette l'esecuzione delle operazioni privilegiate, come l'installazione delle applicazioni, il backup completo e l'accesso al file system (compreso l'accesso completo alla memoria esterna), questa funzionalità aiuta a prevenire l'accesso non autorizzato ai dati del dispositivo e alle applicazioni sui dispositivi per cui è abilitato il debug ADB. Infine, vedremo l'implementazione e il formato di crittografia degli archivi della funzionalità di backup completo di Android.

Controllo dell'installazione e dell'avvio del sistema operativo

Ottenuto l'accesso fisico a un dispositivo, un hacker può accedere ai dati dell'utente e di sistema non solo tramite i costrutti di alto livello del sistema operativo, quali file e directory, ma anche arrivando direttamente alla memoria o al disco. Tale accesso diretto può essere ottenuto con un interfacciamento fisico ai componenti elettronici del dispositivo, per esempio disassemblando il device, connettendosi a interfacce di debug hardware nascoste o dissaldando la memoria flash e leggendone il contenuto con uno strumento specializzato.

NOTA

Questi attacchi hardware vanno oltre l'ambito del libro; leggete il Capitolo 10 di *Android Hacker's Handbook* (Wiley, 2014) se siete interessati all'argomento.

Un metodo meno intrusivo ma comunque potente per ottenere l'accesso ai dati prevede l'uso del meccanismo di aggiornamento del device per modificare i file di sistema e rimuovere le restrizioni di accesso, oppure l'avvio di un sistema operativo alternativo che consente l'accesso diretto ai dispositivi di archiviazione. La maggior parte dei device Android consumer è bloccata per impostazione predefinita, al fine di evitare l'applicazione di queste tecniche senza la disponibilità di una chiave di firma del codice, generalmente in possesso del solo produttore del dispositivo.

Nei paragrafi successivi vedremo brevemente come il bootloader e il sistema operativo di recovery di Android regolano l'accesso alle immagini di boot e ai meccanismi di aggiornamento del dispositivo (il bootloader e la funzionalità di recovery sono descritti in maggiore dettaglio nel Capitolo 13).

Bootloader

Un *bootloader* è un programma specializzato e specifico per l'hardware che viene eseguito alla prima accensione di un dispositivo (o a seguito di un reset per i dispositivi ARM). Il suo scopo è inizializzare

l'hardware, eventualmente fornire un'interfaccia di configurazione minima e infine trovare e avviare il sistema operativo.

Il boot di un dispositivo di solito richiede il superamento di diverse fasi, che possono prevedere un bootloader distinto per ciascuna di esse; in ogni caso, per semplicità noi faremo riferimento a un unico bootloader aggregato che include tutte le fasi di boot. I bootloader Android sono generalmente proprietari e specifici per il *system on a chip* (SoC) su cui è basato il dispositivo. I produttori di device e SoC forniscono funzionalità e livelli di protezione diversi nei loro bootloader, ma la maggior parte di questi supporta una modalità *fastboot*, detta più generalmente *modalità di download*, che consente la scrittura (operazione detta *flashing*) delle immagini delle partizioni raw nella memoria persistente del dispositivo, nonché il boot di immagini di sistema provvisorie (senza flashing delle stesse sul device). La modalità fastboot viene abilitata da una combinazione di tasti speciale eseguita durante il boot del dispositivo, oppure inviando il comando *reboot bootloader* con ADB.

Per garantire l'integrità del device, i dispositivi consumer sono forniti con bootloader bloccati, che impediscono del tutto il flashing e il boot delle immagini di sistema o consentono tali operazioni solo per le immagini firmate dal produttore del dispositivo. Quasi tutti i dispositivi consumer permettono lo sblocco del bootloader, rimuovendo le restrizioni di fastboot e i controlli sulla firma delle immagini. Lo sblocco del bootloader di solito richiede di formattare la partizione *userdata* per garantire che un'immagine del sistema operativo dannosa non possa accedere ai dati utente esistenti.

Su alcuni dispositivi lo sblocco è una procedura irreversibile, ma la maggior parte offre un mezzo per ribloccare il bootloader e ripristinarne lo stato originale. Questa operazione viene in genere implementata salvando un flag di stato del bootloader su una partizione di sistema dedicata (solitamente chiamata `param` o `misc`) che ospita vari metadati del dispositivo. Il blocco del bootloader provoca il semplice reset del valore di questo flag.

Recovery

Un mezzo più flessibile per aggiornare un dispositivo è tramite il suo *sistema operativo di recovery*, o semplicemente *recovery*. È un sistema operativo minimo basato su Linux che include un kernel, un disco RAM con vari strumenti di basso livello e un'UI minima tipicamente gestita con i tasti del dispositivo. Il recovery è usato per applicare aggiornamenti a seguito del rilascio, generalmente sotto forma di package *over-the-air* (OTA). I package OTA includono le nuove versioni (o una patch binaria) dei file di sistema aggiornati e uno script che li applica. Come spiegato nel Capitolo 3, i file OTA sono inoltre firmati a livello di codice con la chiave privata del produttore del dispositivo. Il recovery include la parte pubblica di tale chiave e verifica i file OTA prima di applicarli, al fine di garantire che il sistema operativo del dispositivo possa essere modificato solo da quelli creati da un'entità affidabile.

Il sistema operativo di recovery è archiviato in una partizione dedicata, come il sistema operativo principale di Android; può quindi essere sostituito attivando la modalità download del bootloader ed effettuando un flashing dell'immagine di recovery personalizzata, sostituendo così la chiave pubblica incorporata o non verificando del tutto le firme OTA. Tale meccanismo consente di sostituire completamente il sistema operativo principale con una build creata da terze parti. Un sistema operativo di recovery personalizzato può inoltre concedere accesso root senza restrizioni tramite ADB, nonché l'acquisizione dei dati nella partizione raw. Anche se la partizione *userdata* può essere crittografata (fate riferimento al paragrafo "Crittografia del disco"), rendendo impossibile l'accesso diretto ai dati, è pur sempre facile installare un programma dannoso (rootkit) nella partizione *system* mentre è attiva la modalità di recovery. Il rootkit può quindi consentire l'accesso remoto al dispositivo all'avvio del sistema operativo principale, permettendo così l'accesso immediato ai dati utente decodificati in maniera trasparente. Il boot verificato (di cui parliamo nel prossimo paragrafo) può evitare questo problema, ma solo se il dispositivo verifica la partizione *boot*

usando una chiave di controllo non modificabile salvata a livello hardware.

Un bootloader sbloccato permette il boot o il flashing di immagini di sistema personalizzate e l'accesso diretto alle partizioni di sistema. Anche se le funzionalità di sicurezza di Android, come il boot verificato e la crittografia del disco, possono limitare i danni causati dal flashing di un'immagine di sistema dannosa tramite un bootloader sbloccato, il controllo di accesso al bootloader è fondamentale per la protezione di un dispositivo Android. Il bootloader dovrebbe quindi essere sbloccato solo sui device di test o di sviluppo, e ribloccato o riportato allo stato originale subito dopo la modifica del sistema.

Boot verificato

L'implementazione di Android del boot verificato è basata sul target di verifica dell'integrità dei blocchi device-mapper dm-verity (consultate Milan Broz, "dm-verity: device-mapper block integrity checking target", http://bit.ly/ZvZcqc). *Device-mapper* (https://www.sourceware.org/dm/) è un framework del kernel Linux che fornisce un mezzo generico per implementare dispositivi a blocchi virtuali. È alla base di *Logical Volume Manager* (LVM) in Linux ed è usato per implementare la crittografia dell'intero disco (con il target dm-crypt), gli array RAID e persino l'archiviazione con replica distribuita.

Device-mapper esegue in sostanza il mapping di un dispositivo a blocchi virtuali con uno o più dispositivi a blocchi fisici e, facoltativamente, modifica i dati in transito. Per esempio, dm-crypt (che è anche la base della crittografia della partizione *userdata* di Android, come spiegato nel paragrafo "Crittografia del disco") decodifica i blocchi fisici letti e crittografa i blocchi scritti prima di salvarli su disco. Di conseguenza, la crittografia del disco è trasparente per gli utenti del dispositivo a blocchi dm-crypt virtuali. I target di device-mapper possono essere impilati per consentire l'implementazione di trasformazioni complesse dei dati.

Informazioni generali su dm-verity

Dm-verity è un target di verifica dell'integrità dei blocchi, pertanto controlla in maniera trasparente l'integrità di ogni blocco del dispositivo durante la lettura del disco. Se il blocco è corretto, la lettura riesce; altrimenti, l'operazione di lettura genera un errore di I/O come se il blocco fosse fisicamente danneggiato.

Dietro le quinte, dm-verity è implementato usando un hash tree precalcolato (detto anche *Merkle tree*) che include gli hash di tutti i blocchi del dispositivo. I nodi foglia (*leaf*) dell'albero (*tree*) includono gli hash dei blocchi fisici, mentre i nodi intermedi sono gli hash dei

relativi nodi figlio (hash degli hash). Il nodo radice (*root*) è chiamato *root hash* ed è basato su tutti gli hash dei livelli inferiori, come mostrato nella Figura 10.1. Di conseguenza, una modifica anche a un solo blocco del dispositivo provoca un cambiamento del root hash: così, basta verificare il root hash per garantire che un hash tree sia autentico.

In fase di esecuzione dm-verity calcola l'hash di ogni blocco durante la lettura, e lo verifica ripercorrendo l'hash tree precalcolato. La lettura dei dati da un dispositivo fisico è un'operazione che in sé richiede tempo, pertanto la latenza aggiunta dall'hashing e dalla verifica è relativamente bassa. Inoltre, dopo la verifica i blocchi del disco sono archiviati nella cache e le letture successive dello stesso blocco non attivano una nuova verifica dell'integrità.

Dm-verity dipende da un hash tree precalcolato su tutti i blocchi del dispositivo, pertanto il dispositivo sottostante deve essere montato in sola lettura perché la verifica sia possibile. La maggior parte dei file system registra i tempi di mounting e gli altri metadati nel relativo superblocco; di conseguenza, anche se nessun file viene modificato durante il runtime, i controlli sull'integrità dei blocchi hanno esito negativo se il dispositivo a blocchi sottostante è montato in lettura/scrittura. Questo può sembrare un limite, ma in realtà è una soluzione perfetta per i dispositivi o le partizioni contenenti file di sistema, che vengono modificati solo dagli aggiornamenti del sistema operativo. Qualunque altra modifica indica un danneggiamento del sistema operativo o del disco, oppure che un programma dannoso sta tentando di modificare il sistema operativo o di fingersi un file di sistema.

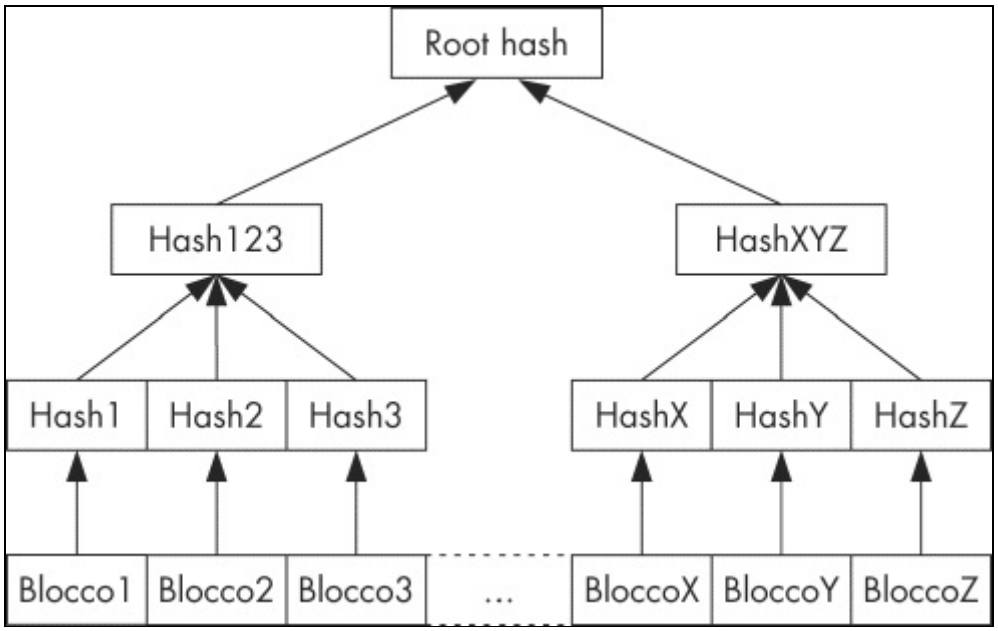

Figura 10.1 Hash tree di 5.2014dm-verity.

Infine, il requisito di sola lettura di dm-verity si adatta bene al modello di sicurezza di Android, che ospita solamente i dati delle applicazioni in una partizione di lettura/scrittura e conserva i file del sistema operativo nella partizione di sola lettura *system*.

Implementazione in Android

Il target device-mapper dm-verity è stato in origine sviluppato per implementare il boot verificato in Chrome OS ed è stato integrato nel kernel Linux mainline nella versione 3.4. Viene abilitato con la voce di configurazione del kernel `CONFIG_DM_VERITY`.

Come Chrome OS, anche Android 4.4 usa il target dm-verity, ma la verifica crittografica del root hash e il mounting delle partizioni verificate sono implementati in maniera diversa. La chiave pubblica RSA usata per la verifica è incorporata nella partizione di boot con il nome file `verity_key` ed è usata per verificare la tabella di mapping dm-verity, che contiene le posizioni del dispositivo target e l'offset della tabella hash, nonché il root hash e il salt.

La tabella di mapping e la sua firma sono parte del blocco dei metadati di verity, che viene scritto su disco subito dopo l'ultimo blocco del file system del dispositivo target. Una partizione viene contrassegnata come

verificabile aggiungendo il flag `verify` al campo `fs_mgr_flags` specifico di Android del file `fstab` del dispositivo. Quando il gestore del file system di Android incontra il flag `verify` in `fstab`, carica i metadati verity dal dispositivo a blocchi specificato in `fstab` e verifica la relativa firma con la chiave verity fornita. Se il controllo della firma riesce, il gestore del file system esegue il parsing della tabella di mapping dm-verity e lo passa al device-mapper Linux, che utilizza le informazioni contenute nella tabella di mapping per creare un dispositivo a blocchi virtuali dm-verity. Questo dispositivo a blocchi virtuali viene quindi montato nel punto di mount specificato in `fstab` al posto del dispositivo fisico corrispondente. Di conseguenza, tutte le letture dal dispositivo fisico sottostante vengono verificate in modo trasparente rispetto all'hash tree pregenerato. La modifica o l'aggiunta di file, o persino il remounting della partizione in lettura/scrittura, provoca una verifica dell'integrità e un errore di I/O.

NOTA

Dm-verity è una funzionalità del kernel; perché la sua protezione dell'integrità sia efficace, il kernel usato dal dispositivo per il boot deve essere attendibile. In Android, a tal fine occorre verificare la partizione di boot, che contiene anche il disco RAM del file system root (*initrd*) e la chiave pubblica di verity. La verifica dell'immagine di boot o del kernel è un processo specifico del dispositivo, tipicamente implementato nel suo bootloader e basato su una chiave di verifica della firma non modificabile salvata nell'hardware.

Abilitazione del boot verificato

La documentazione ufficiale di Android descrive la procedura richiesta per abilitare il boot verificato in Android come un processo in più fasi, che prevede la generazione di un hash tree, la creazione di una tabella di mapping dm-verity per l'hash tree, la firma della tabella e la generazione e la scrittura di un blocco di metadati verity sul dispositivo target (`http://bit.ly/1DeQJGX`). In questo paragrafo descriviamo brevemente i passaggi principali di questo processo.

Un hash tree dm-verity viene generato dal programma `veritysetup`, che fa parte del package di strumenti per la gestione dei volumi crittografici *cryptsetup*. Il programma `veritysetup` può operare direttamente sui dispositivi a blocchi o generare un hash tree utilizzando un'immagine del

file system e quindi scrivere la tabella hash in un file. L'implementazione dm-verity di Android prevede che i dati dell'hash tree siano archiviati sullo stesso dispositivo del file system target; pertanto, è necessario specificare un offset di hash esplicito che punta a una posizione successiva al blocco dei metadati verity quando si richiama `veritysetup`. La Figura 10.2 mostra il layout di una partizione del disco preparata per l'uso con dm-verity.

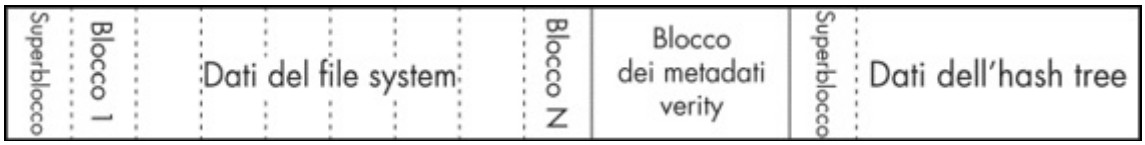

Figura 10.2 Layout di una partizione del disco preparata per la verifica con dm-verity.

La generazione dell'hash tree produce il root hash, utilizzato per costruire la tabella di mapping dm-verity per il dispositivo target. Nel Listato 10.1 è mostrata una tabella di mapping di esempio.

Listato 10.1 Tabella dei mapping del dispositivo dm-verity in Android.

```
1(1) /dev/block/mmcblk0p21(2) /dev/block/mmcblk0p21(3) 4096(4) 4096(5)
204800(6) 204809(7) sha256(8)
1F951588516c7e3eec3ba10796aa17935c0c917475f8992353ef2ba5c3f47bcb(9)
5f061f591b51bf541ab9d89652ec543ba253f2ed9c8521ac61f1208267c3bfb1(10)
```

Come mostrato nel listato, la tabella contiene una sola riga (divisa su più righe per ragioni di leggibilità) che, oltre al root hash **(9)**, contiene la versione di dm-verity **(1)**, il nome dei dati sottostanti e del dispositivo hash (**(2)** e **(3)**), le dimensioni dei dati e dei blocchi hash (**(4)** e **(5)**), gli offset su disco di dati e hash (**(6)** e **(7)**), l' algoritmo hash **(8)** e il salt **(10)**.

La tabella di mapping viene firmata utilizzando una chiave RSA a 2048 bit e, insieme con la firma PKCS#1 v1.5 risultante, è utilizzata per formare il blocco dei metadati verity da 32 KB. La Tabella 10.1 illustra il contenuto e la dimensione di ciascun campo del blocco dei metadati.

Tabella 10.1 Contenuto del blocco dei metadati 2014Verity.

Campo	Descrizione	Dimensione	Valore
Magic number	Usato `da fs_mgr` come controllo di integrità	4 byte	0xb001b001
Version	Versione del blocco dei metadati	4 byte	Attualmente 0
Signature	Firma della tabella di mapping (PKCS#1 v1.5)	256 byte	
Mapping table length	Lunghezza della tabella di mapping in byte	4 byte	
Mapping table	Tabella di mapping dm-verity	Variabile	

Padding	Padding con byte zero fino a una lunghezza di 32K byte	Variabile	

La chiave pubblica RSA utilizzata per la verifica deve essere nel formato mincrypt (una libreria crittografica minimalista, utilizzata anche dallo stock recovery durante la verifica delle firme dei file OTA), che è una serializzazione della struttura `RSAPublicKey` di mincrypt. L'aspetto interessante di questa struttura è che non include semplicemente il modulo della chiave e i valori degli esponenti pubblici, ma contiene anche i valori precalcolati usati dall'implementazione RSA di mincrypt (basata sulla riduzione di Montgomery). La chiave pubblica è inclusa nel root dell'immagine di boot con il nome file `verity_key`.

L'ultimo passo necessario per abilitare il boot verificato è modificare il file `fstab` del dispositivo per abilitare la verifica dell'integrità dei blocchi per la partizione *system*. A tal fine è sufficiente aggiungere il flag `verify` come mostrato nel Listato 10.2 (file `fstab` di esempio per Nexus 4).

Listato 10.2 Voce fstab per una partizione verificata formattata con dm-verity.

```
/dev/block/platform/msm_sdcc.1/by-name/system /system ext4 ro,barrier=1 wait,verify
```

All'avvio del dispositivo, Android crea automaticamente un dispositivo dm-verity virtuale basato sulla voce `fstab` e sulle informazioni nella tabella di mapping (contenuta nel blocco dei metadati) e lo monta in */system* come mostrato nel Listato 10.3.

Listato 10.3 Dispositivo di blocco virtuale dm-verity montato in /system.

```
# mount|grep system
/dev/block/dm-0 /system ext4 ro,seclabel,relatime,data=ordered 0 0
```

Ora qualunque modifica alla partizione di sistema provocherà errori durante la lettura dei file corrispondenti. Purtroppo, anche le modifiche al sistema effettuate da aggiornamenti OTA basati su file, che modificano i blocchi di file senza aggiornare i metadati di verity, invalidano l'hash tree. La documentazione ufficiale indica che, per la compatibilità con il boot verificato basato su dm-verity, gli aggiornamenti OTA devono operare a livello di blocco, garantendo l'aggiornamento sia dei blocchi di file, sia dell'hash tree e dei metadati. Occorre quindi cambiare l'attuale infrastruttura di aggiornamento OTA: probabilmente questo è uno dei

motivi per cui il boot verificato deve ancora essere distribuito nei dispositivi di produzione.

Crittografia del disco

Android 3.0 ha introdotto la crittografia del disco insieme alle policy di amministrazione del dispositivo (per i dettagli vedete il Capitolo 9), che consentono di imporre la crittografia obbligatoria del device come uno dei numerosi "miglioramenti per le aziende" inclusi in tale versione. La crittografia del disco è stata disponibile in tutte le versioni successive, con ben pochi cambiamenti fino alla versione 4.4, che ha introdotto una nuova funzione di derivazione della chiave (scrypt). In questo paragrafo vediamo come Android implementa la crittografia del disco e come vengono memorizzati e gestiti metadati e chiavi di crittografia.

NOTA

La definizione di compatibilità di Android indica che, se il dispositivo dispone della schermata di blocco, deve supportare anche la crittografia dell'intero disco (*Android 4.4 Compatibility Definition*, "9.9. Full-Disk Encryption", http://static.googleusercontent.com/media/source.android.com/it//compatibility/android-cdd.pdf).

La *crittografia del disco* usa un algoritmo di crittografia per convertire ogni bit di dati da salvare su disco in testo cifrato, garantendo che i dati non possano essere letti dal disco senza la chiave di decodifica. La *crittografia dell'intero disco* (FDE, *Full-Disk Encryption*) garantisce che tutto il contenuto del disco sia crittografato, compresi i file temporanei, la cache e i file del sistema operativo. Nella pratica, una piccola parte del sistema operativo, o un loader separato dello stesso, deve essere mantenuta non crittografata affinché possa ottenere la chiave di decodifica e successivamente decodificare e montare i volumi del disco utilizzati dal sistema operativo principale. La chiave di decodifica del disco viene solitamente salvata in forma crittografata e richiede una *Key-Encryption Key* (KEK) supplementare per la decodifica. La chiave KEK può essere memorizzata in un modulo hardware, per esempio una smart card o un TPM, oppure può essere derivata da una passphrase ottenuta dall'utente a ogni avvio. Se è memorizzata in un modulo hardware, anche la KEK può essere protetta da un PIN o da una password forniti dall'utente.

L'implementazione FDE di Android crittografa solamente la partizione *userdata*, che ospita i file di configurazione del sistema e i dati delle applicazioni. Le partizioni *boot* e *system*, che mantengono il kernel e i file del sistema operativo, non sono invece crittografate, ma volendo *system* può essere verificata utilizzando il target device-mapper dm-verity come descritto nel paragrafo "Boot verificato". La crittografia del disco di Android non è abilitata per impostazione predefinita e il processo di crittografia del disco deve essere attivato dall'utente o da una policy del dispositivo sui dispositivi gestiti. L'implementazione della crittografia del disco di Android è descritta nei paragrafi che seguono.

Modalità di cifratura

La crittografia del disco di Android usa dm-crypt (Milan Broz, "dm-crypt: Linux kernel device-mapper crypto target", http://bit.ly/1zcQdKV), attualmente il sottosistema di crittografia del disco standard nel kernel Linux. Come dm-verity, dm-crypt è un target di device-mapper che esegue il mapping di un dispositivo a blocchi fisici crittografato con un dispositivo device-mapper virtuale. L'accesso ai dati sul dispositivo virtuale viene decodificato (per la lettura) o crittografato (per la scrittura) in maniera trasparente.

Il meccanismo di crittografia impiegato in Android usa una chiave di 128 bit generata in maniera casuale insieme ad AES nella modalità CBC. Come abbiamo imparato nel Capitolo 5, la modalità CBC richiede un vettore di inizializzazione (IV) casuale e imprevedibile affinché la crittografia sia sicura. Si presenta quindi un problema durante la crittografia dei dispositivi a blocchi, perché ai blocchi si accede in maniera non sequenziale, e quindi ogni settore (o blocco del dispositivo) richiede un IV distinto.

Android usa il metodo *Encrypted Salt-Sector Initialization Vector* (ESSIV) con l'algoritmo hash SHA-256 (ESSIV:SHA256) per generare gli IV per settore. ESSIV usa un algoritmo hash per derivare una chiave secondaria *s* dalla chiave di crittografia del disco *K*, detta *salt*. Usa quindi il salt come chiave di crittografia e crittografa il numero del settore *SN* di

ciascun settore per produrre un IV per settore; in altre parole, *IV(SN) = AESs(SN)*, dove *s = SHA256(K)*.

Visto che l'IV di ciascun settore dipende da un'informazione segreta (la chiave di crittografia del disco), gli IV per settore non possono essere individuati da un hacker. Tuttavia, ESSIV non cambia la proprietà di malleabilità di CBC e non assicura l'integrità dei blocchi crittografati. In effetti, è stato dimostrato che un hacker che conosce il testo normale originale memorizzato su disco può manipolare i dati archiviati e persino inserire una backdoor sui volumi che utilizzano CBC per la crittografia del disco (Jakob Lell, "Practical malleability attack against CBC-Encrypted LUKS partitions", `http://bit.ly/1cPdSjD`).

MODALITÀ DI CIFRATURA ALTERNATIVE: XTS

Questo particolare attacco contro la modalità ESSIV può essere evitato passando a una modalità di cifratura con crittografia regolabile, come XTS (modalità tweaked-codebook basata su XEX con sottrazione del testo cifrato), che usa una combinazione dell'indirizzo di settore e dell'indice del blocco di cifratura all'interno del settore per ricavare un "tweak" (parametro variabile) univoco per ogni settore. L'uso di un tweak distinto per ciascun settore produce lo stesso effetto della crittografia con una chiave unica: lo stesso testo in chiaro produce un testo cifrato diverso se archiviato in settori diversi, ma le prestazioni sono superiori rispetto al derivare una chiave (o IV) separata per ogni settore. Tuttavia, seppure superiore alla modalità ESSIV CBC, XTS è ancora suscettibile alla manipolazione dei dati in alcuni casi e non fornisce l'autenticazione del testo cifrato.

A oggi Android non supporta la modalità XTS per la crittografia del disco; tuttavia, il target di device-mapper dm-crypt sottostante supporta XTS e può essere facilmente abilitato con alcune modifiche all'implementazione del daemon dei volumi di Android (`vold`).

Derivazione della chiave

La chiave di crittografia del disco (chiamata *master key* nel codice sorgente di Android) è codificata con un'altra chiave AES a 128 bit (KEK), derivata da una password fornita dall'utente. Nelle versioni di Android da 3.0 a 4.3, la funzione di derivazione della chiave utilizzata era PBKDF2, con 2000 iterazioni e un valore salt casuale a 128 bit. La chiave master crittografata e il salt risultanti vengono salvati, insieme ad altri metadati come il numero di tentativi di decodifica non riusciti, in una struttura footer che occupa gli ultimi 16 KB della partizione crittografata detta *footer crypto*. La memorizzazione di una chiave crittografata su disco, al posto dell'uso di una chiave derivata dalla password fornita

direttamente dall'utente, permette di cambiare rapidamente la password di decodifica, perché l'unico elemento che deve essere ricrittografato con la chiave derivata dalla nuova password è la chiave master (16 byte).

Anche se l'uso di un salt casuale rende impossibile utilizzare le tabelle precalcolate per accelerare il cracking della chiave, il numero di iterazioni (2000) utilizzato per PBKDF2 non è sufficientemente grande per gli standard attuali (il processo di derivazione della chiave del keystore usa 8192 iterazioni, come spiegato nel Capitolo 7; la crittografia del backup usa 10.000 iterazioni, come descritto più avanti nel paragrafo "Backup in Android"). Inoltre, PBKDF2 è un algoritmo iterativo, basato su funzioni hash standard e relativamente facili da implementare, che rende possibile applicare la parallelizzazione alla derivazione di chiavi PBKDF2, sfruttando al massimo la potenza di elaborazione dei dispositivi multicore come le GPU. Diventa quindi possibile forzare le passphrase alfanumeriche complesse in pochi giorni, o persino in poche ore.

Per rendere più difficoltosi gli attacchi di forza bruta sulle password di crittografia del disco, Android 4.4 ha introdotto il supporto per una nuova funzione di derivazione della chiave chiamata *scrypt* (fate riferimento a Jakob Lell, "Practical malleability attack against CBC-Encrypted LUKS partitions", http://bit.ly/1cPdSjD). Scrypt usa un algoritmo di derivazione della chiave progettato specificamente per richiedere grandi quantità di memoria e molteplici iterazioni (un algoritmo di questo tipo è detto *memory hard*). Rende pertanto più difficile il mounting di attacchi di forza bruta su hardware specializzato come ASIC o GPU, che in genere operano con una quantità di memoria limitata.

Scrypt può essere regolato specificando i parametri variabili N, r e p, che influenzano rispettivamente le risorse della CPU, la quantità di memoria e il costo di parallelizzazione richiesti. I valori utilizzati per impostazione predefinita in Android sono $N = 32768$ (2^{15}), $r = 8$ e $p = 2$. Per cambiarli è sufficiente impostare il valore della proprietà di sistema `ro.crypto.scrypt_params` utilizzando il formato `N_factor:r_factor:p_factor`, per esempio *15:3:1* (impostazione predefinita). Il valore di ogni parametro viene calcolato elevando 2 alla potenza del rispettivo fattore. I dispositivi

Android 4.4 aggiornano automaticamente l'algoritmo di derivazione della chiave nel footer crypto da PBKDF2 a scrypt e ricrittografano la chiave master utilizzando una chiave di crittografia derivata da scrypt. Quando viene aggiornata la chiave master crittografata, i parametri `N`, `r` e `p` utilizzati per la derivazione KEK vengono scritti nel footer crypto.

NOTA

Sullo stesso computer desktop, un attacco di forza bruta su un PIN di 4 cifre (utilizzando un algoritmo a thread singolo che genera tutti i PIN possibili a partire da 0000) richiede circa 5 millisecondi per PIN quando si utilizza PBKDF2 e circa 230 millisecondi per PIN quando si utilizza scrypt come funzione di derivazione KEK. In altre parole, un attacco di forza bruta su PBKDF2 è quasi 50 volte più conveniente (in termini di velocità) rispetto a scrypt.

Password di crittografia del disco

Come discusso nel paragrafo precedente, la chiave KEK utilizzata per crittografare la chiave di crittografia del disco è derivata da una password fornita dall'utente. Al primo avvio del processo di crittografia del dispositivo, viene chiesto di confermare il PIN o la password di sblocco del dispositivo, o di impostarne uno se non avete mai eseguito l'operazione o state utilizzando il blocco dello schermo basato su pattern (Figura 10.3). La password o il PIN inserito viene quindi utilizzato per ricavare la chiave di crittografia della chiave master; dovrete quindi inserire la password o il PIN ogni volta che accenderete il dispositivo e ancora una volta per sbloccare lo schermo dopo l'avvio.

Figura 10.3 Schermata di crittografia del dispositivo.

Android non dispone di un'impostazione dedicata per gestire la password di crittografia dopo la crittografia del dispositivo; inoltre, la modifica del PIN o della password di blocco dello schermo provoca anche una modifica della password di crittografia. Questa decisione è molto probabilmente basata sull'usabilità: la maggior parte degli utenti troverebbe scomodo dover ricordare e inserire due password diverse in tempi diversi, e probabilmente dimenticherebbe presto la password di crittografia del disco, usata meno frequentemente e in genere più complessa. Anche se l'idea è valida ai fini dell'usabilità, in effetti costringe gli utenti a utilizzare una password di crittografia del disco semplice, visto che devono inserirla ogni volta che sbloccano il

dispositivo, di solito decine di volte al giorno. Nessuno vuole inserire ripetutamente una password complessa e così molti utenti optano per un semplice PIN numerico (a meno che una policy del dispositivo richieda di procedere diversamente). Inoltre, le password sono limitate a 16 caratteri (un limite predefinito del framework e non configurabile), quindi l'uso di una passphrase non è una scelta percorribile.

Qual è il problema legato all'uso della stessa password per la crittografia del disco e la schermata di blocco? Dopotutto, per ottenere i dati sullo smartphone è necessario indovinare comunque la password della schermata di blocco, quindi perché bisognerebbe preoccuparsi di scegliere una password diversa per la crittografia del disco? Il motivo è che le due password proteggono lo smartphone da due tipi diversi di attacco. La gran parte degli attacchi contro la schermata di blocco avviene online e con modalità di forza bruta: sostanzialmente, qualcuno prova diverse password su un dispositivo funzionante fino a ottenere l'accesso. Dopo alcuni tentativi infruttuosi, Android blocca lo schermo per 30 secondi (limitazione del rate) e può addirittura cancellare il contenuto del dispositivo a seguito di ulteriori tentativi di sblocco non riusciti (se richiesto dalla policy del dispositivo). Pertanto, il più delle volte anche un PIN relativamente breve per la schermata di blocco offre un'adeguata protezione contro gli attacchi online (leggete il paragrafo "Protezione contro gli attacchi di forza bruta" per i dettagli).

Naturalmente, se qualcuno ha accesso fisico al dispositivo o a una sua immagine del disco, può estrarre gli hash delle password ed effettuarne il cracking offline senza preoccuparsi della limitazione del rate o del wiping del dispositivo. È proprio questo lo scenario per cui è stata ideata la protezione mediante crittografia dell'intero disco: quando un dispositivo viene rubato o confiscato, l'hacker può sia eseguire attacchi di forza bruta sul dispositivo sia copiarne i dati e analizzarli anche dopo averlo restituito o gettato. Come accennato in precedenza nel paragrafo "Derivazione della chiave", la chiave master crittografata è archiviata sul disco, pertanto se la password utilizzata per derivare la chiave di crittografia è basata su un PIN numerico breve, può essere ottenuta con

un attacco di forza bruta in pochi minuti, o addirittura in qualche secondo sui dispositivi precedenti alla versione 4.4, che utilizzano PBKDF2 per la derivazione della chiave (la dimostrazione è stata fornita durante l'intervento "Into The Droid" al DEF CON 20; le diapositive sono disponibili all'indirizzo `http://bit.ly/1vgLOCU`). Una soluzione di cancellazione remota potrebbe prevenire questo attacco eliminando la chiave master, operazione che richiede un istante e rende inutilizzabile il dispositivo, ma spesso questa non è un'opzione perché il dispositivo potrebbe essere offline o spento.

Modifica della password di crittografia del disco

La sezione user-level della crittografia del disco viene implementata nel modulo *cryptfs* del daemon di gestione dei volumi di Android (`vold`). *crypfs* dispone di comandi per creare e montare un volume crittografato, nonché per verificare e cambiare la password di crittografia della chiave master. I servizi di sistema Android comunicano con *cryptfs* inviando comandi a `vold` attraverso un socket locale (anch'esso denominato `vold`); `vold` imposta le proprietà di sistema che descrivono lo stato attuale del processo di crittografia o di mounting in base al comando ricevuto. Si ottiene così una procedura di boot piuttosto complessa, descritta nei dettagli nei paragrafi "Abilitazione della crittografia" e "Avvio di un dispositivo crittografato").

Android non fornisce un'interfaccia utente per cambiare solo la password di crittografia del disco, ma è possibile eseguire questa operazione comunicando direttamente con il daemon `vold` utilizzando l'utility a riga di comando `vdc`. Tuttavia, l'accesso al socket di controllo `vold` è limitato all'utente root e ai membri del gruppo `mount`; inoltre, i comandi `cryptfs` sono disponibili solo agli utenti `root` e `system`. Se state utilizzando una build di engineering, o se il vostro dispositivo consente l'accesso root tramite un'app "superuser" (Capitolo 13), potete inviare il comando `cryptfs` mostrato nel Listato 10.4 a `vold` per cambiare la password di crittografia del disco.

Listato 10.4 Modifica della password di crittografia del disco con vdc.

```
# vdc cryptfs changepw <newpass>
200 0 0
```

NOTA

Se cambiate la password della schermata di blocco, la password di crittografia del disco sarà cambiata anch'essa automaticamente (questo tuttavia non vale per gli utenti secondari sui dispositivi multiutente).

Abilitazione della crittografia

Come spiegato nel paragrafo precedente, la sezione user-level della crittografia del disco di Android viene implementata da un modulo *cryptfs* dedicato del daemon `vold`. *cryptfs* fornisce i comandi `checkpw`, `restart`, `cryptocomplete`, `enablecrypto`, `changepw`, `verifypw`, `getfield` e `setfield`, che il framework invia in diversi momenti della crittografia o del processo di mounting dei volumi crittografati. Oltre ai permessi impostati sul socket locale `vold`, *crypfs* verifica esplicitamente l'identità del sender dei comandi e consente l'accesso solo agli utenti `root` e `system`.

Controllo della crittografia del dispositivo con le proprietà di sistema

Il daemon `vold` imposta diverse proprietà del sistema per attivare le varie fasi di mounting o crittografia del dispositivo e per comunicare lo stato di crittografia attuale ai servizi del framework. La proprietà `ro.crypto.state` mantiene lo stato di crittografia corrente, impostato su *encrypted* se la partizione dati è stata correttamente crittografata e su *unencrypted* se non è ancora stata crittografata. La proprietà può anche essere impostata su *unsupported* se il dispositivo non supporta la crittografia del disco. Il daemon `vold` imposta anche diversi valori predefiniti per la proprietà `vold.decrypt` al fine di segnalare lo stato attuale di mounting o crittografia del dispositivo. La proprietà `vold.encrypt_progress` contiene l'avanzamento attuale della crittografia (da 0 a 100) o una stringa di errore se si è verificato un errore durante il mounting o l'operazione di crittografia.

La proprietà di sistema `ro.crypto.fs_crypto_blkdev` contiene il nome del dispositivo virtuale allocato dal device-mapper. Dopo una corretta decodifica della chiave di crittografia del disco, questo dispositivo virtuale viene montato in */data* al posto del volume fisico sottostante, come mostrato nel Listato 10.5 (output diviso per ragioni di leggibilità).

Listato 10.5 Dispositivo di blocco virtuale crittografato montato in /data.

```
# mount|grep '/data'
/dev/block/dm-0 /data ext4 rw,seclabel,nosuid,nodev,noatime,
errors=panic,user_xattr,barrier=1,nomblk_io_submit,data=ordered 0 0
```

Unmounting di /data

Il framework Android richiede che */data* sia disponibile, ma è necessario effettuarne l'unmounting per la crittografia: si crea così una situazione catch-22, che Android risolve effettuando l'unmounting della partizione fisica *userdata* e montando al suo posto un file system in memoria (`tempfs`) durante l'esecuzione della crittografia. La sostituzione delle partizioni in fase di esecuzione richiede a sua volta l'arresto e il riavvio di alcuni servizi di sistema, che `vold` effettua impostando il valore della proprietà di sistema `vold.decrypt` su `trigger_restart_framework`, `trigger_restart_min_framework` o `trigger_shutdown_framework`. Questi valori attivano diverse parti di `init.rc`, come mostrato nel Listato 10.6.

Listato 10.6 Trigger vold.decrypt in init.rc.

```
--altro codice--
on post-fs-data(1)
    chown system system /data
    chmod 0771 /data
    restorecon /data
    copy /data/system/entropy.dat /dev/urandom
--altro codice--
on property:vold.decrypt=trigger_reset_main(2)
    class_reset main

on property:vold.decrypt=trigger_load_persist_props
    load_persist_props

on property:vold.decrypt=trigger_post_fs_data(3)
    trigger post-fs-data

on property:vold.decrypt=trigger_restart_min_framework(4)
    class_start main

on property:vold.decrypt=trigger_restart_framework(5)
    class_start main
    class_start late_start

on property:vold.decrypt=trigger_shutdown_framework(6)
    class_reset late_start
```

```
    class_reset main
--altro codice--
```

Attivazione del processo di crittografia

Quando l'utente avvia il processo di crittografia tramite l'UI *Settings* con *Security* > *Encrypt phone*, l'app *Settings* chiama `MountService`, che a sua volta invia il comando `cryptfs enablecrypto inplace` *password* a `vold` (dove *password* è la password della schermata di blocco). A sua volta, `vold` effettua l'unmounting della partizione *userdata* e imposta `vold.decrypt` su `trigger_shutdown_framework` (**(6)** nel Listato 10.6), arrestando la maggior parte dei servizi di sistema a eccezione di quelli che fanno parte della classe di servizio `core`. Il daemon `vold` smonta quindi */data*, monta un file system `tempfs` al suo posto e infine imposta `vold.encrypt_progress` su `0` e `vold.decrypt` su `trigger_restart_min_framework` (**(4)** nel Listato 10.6). Viene così avviato un numero di servizi di sistema (nella classe `main`) inferiore a quello richiesto per mostrare l'UI di avanzamento della crittografia.

Aggiornamento del footer crypto e crittografia dei dati

A seguire, `vold` configura il device virtuale dm-crypt e scrive il footer crypto. Il footer può essere scritto alla fine della partizione *userdata* oppure in un file o partizione dedicato; il percorso è specificato nel file `fstab` come valore del flag `encryptable`. Per esempio, su Nexus 5 il footer crypto viene scritto nella partizione dedicata *metadata*, come mostrato al punto **(1)** del Listato 10.7 (dove la riga è stata suddivisa per ragioni di leggibilità). Se il footer crypto viene scritto alla fine della partizione crittografata, il flag `encryptable` viene impostato sulla stringa `footer`.

Listato 10.7 Il flag crittografabile fstab specifica la posizione del footer crypto.

```
--altro codice--
/dev/block/platform/msm_sdcc.1/by-name/userdata  /data  ext4
noatime,nosuid,nodev,barrier=1,data=ordered,nomblk_io_submit,noauto_da_alloc,errors=panic
wait,check,encryptable=/dev/block/platform/msm_sdcc.1/by-name/metadata(1)
--altro codice--
```

Il footer crypto contiene la chiave di crittografia del disco crittografato (chiave master), il salt usato per la derivazione KEK e altri metadati e parametri di derivazione della chiave. Il suo campo `flags` è impostato su

`CRYPT_ENCRYPTION_IN_PROGRESS` (0x2) per segnalare che la crittografia del dispositivo è stata avviata ma non completata.

Infine, ogni blocco viene letto dalla partizione fisica *userdata* e scritto sul dispositivo virtuale dm-crypt, che crittografa i blocchi di lettura e li scrive su disco, crittografando così la partizione *userdata*. Se la crittografia viene completata senza errori, `vold` cancella il flag `CRYPT_ENCRYPTION_IN_PROGRESS` e riavvia il dispositivo.

Avvio di un dispositivo crittografato

Per il boot di un dispositivo crittografato è necessario che l'utente inserisca la password di crittografia del disco. Invece di ricorrere a un'UI specializzata del bootloader, Android imposta la proprietà di sistema `vold.decrypt` a `1` e quindi avvia un set minimo di servizi di sistema per mostrare un'UI Android standard. Come per la crittografia del dispositivo, è richiesto il mounting di un file system `tmpfs` in */data* per consentire l'avvio dei servizi di sistema core. Una volta predisposto il framework core, Android rileva che `vold.decrypt` è impostato su `1` e avvia il processo di mounting della partizione *userdata*.

Recupero della password di crittografia del disco

Il primo passo di questo processo è la verifica della corretta crittografia della partizione con l'invio del comando `cryptfs cryptocomplete` a `vold`, che a sua volta controlla se il footer crypto è formattato adeguatamente e se il flag `CRYPT_ENCRYPTION_IN_PROGRESS` non è impostato. Se la partizione è ritenuta correttamente crittografata, il framework avvia la finestra di inserimento della password, mostrata nella Figura 10.4, fornita da `CryptKeeper` (parte dell'app *Settings*). Questa activity agisce come una schermata principale (launcher) e, poiché ha una priorità più alta rispetto al launcher predefinito, viene avviata per prima subito dopo il boot del dispositivo.

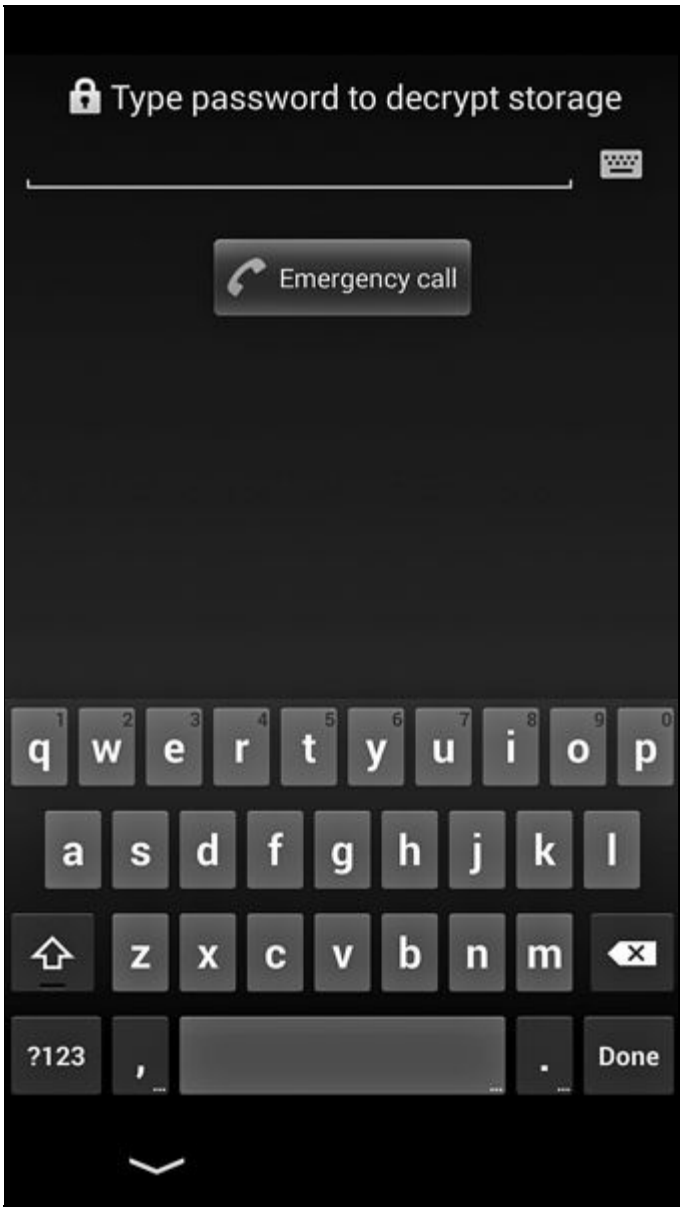

Figura 10.4 Finestra di inserimento della password di crittografia del dispositivo.

Se il dispositivo non è crittografato, `CryptKeeper` si disabilita e si chiude autonomamente, imponendo all'activity manager di sistema di avviare l'applicazione predefinita per la schermata principale. Se il dispositivo è crittografato, o se la crittografia è in corso (la proprietà `vold.crypt` non è vuota né impostata su `trigger_restart_framework`), l'activity `CryptKeeper` viene avviata e nasconde le barre di stato e di sistema. Inoltre, `CryptKeeper` ignora le pressioni dei tasti, disabilitando così l'uscita dalla finestra di inserimento della password. Se il dispositivo crittografato è danneggiato, o se il processo di crittografia si è interrotto e la partizione *userdata* è stata crittografata solo parzialmente, il dispositivo non può essere avviato. In questo caso, `CryptKeeper` mostra la finestra della Figura 10.5 per

consentire all'utente di attivare un reset di fabbrica che riformatti la partizione *userdata*.

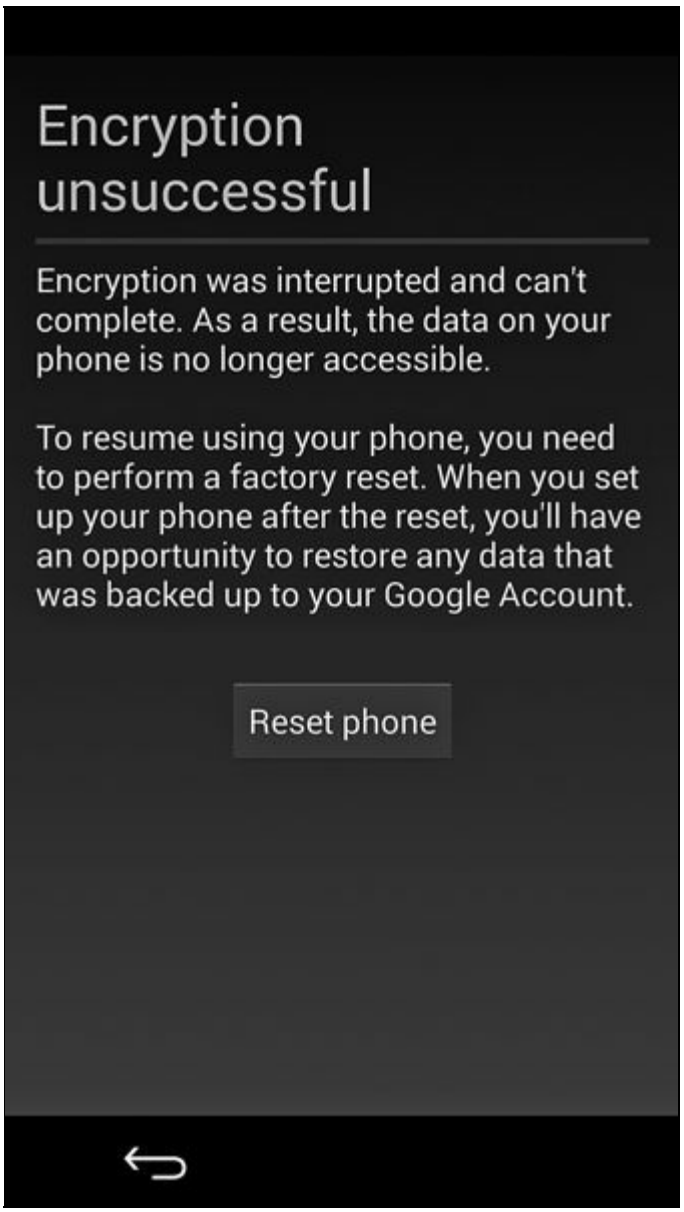

Figura 10.5 Messaggio mostrato se la crittografia del dispositivo ha esito negativo.

Decodifica e mounting di /data

Quando l'utente inserisce la sua password, `CryptKeeper` invia il comando `cryptfs checkpw` a `vold` chiamando il metodo `decryptStorage()` del servizio di sistema `MountService`. Questo comando indica a `vold` di verificare se la password inserita è corretta provando a montare la partizione crittografata in un punto di mount temporaneo e quindi a smontarla. Se la procedura riesce, `vold` imposta il nome del dispositivo a blocchi virtuali allocato da device-mapper come valore della proprietà

`ro.crypto.fs_crypto_blkdev` e restituisce il controllo a `MountService`, che a sua volta invia il comando `cryptfs restart` per chiedere a `vold` di riavviare tutti i servizi di sistema nella classe `main` (**(2)** nel Listato 10.6). In questo modo è possibile smontare il file system `tempfs` e montare il dispositivo a blocchi dm-crypt virtuale appena allocato in */data*.

Avvio di tutti i servizi di sistema

Dopo aver montato e preparato la partizione crittografata, `vold` imposta `vold.decrypt` su `trigger_post_fs_data` (**(3)** nel Listato 10.6), attivando la sezione `post-fs-data` **(1)** di `init.rc`. I comandi in questa sezione configurano i permessi di file e directory, ripristinano i contesti SELinux e creano le directory richieste in */data*, se necessario.

Infine, `post-fs-data` imposta a `1` la proprietà `vold.post_fs_data_done`, sottoposta periodicamente a polling da `vold`. Se `vold` rileva il valore `1`, imposta la proprietà `vold.decrypt` a `trigger_restart_framework` (**(5)** nel Listato 10.6) per riavviare tutti i servizi nella classe `main` e avviare tutti i servizi ritardati (classe `late_start`). A questo punto, il framework è completamente inizializzato e il dispositivo effettua il boot utilizzando la vista decodificata della partizione *userdata* montata in */data*. D'ora in poi tutti i dati scritti dalle applicazioni o dal sistema vengono automaticamente crittografati prima di essere salvati su disco.

LIMITI DELLA CRITTOGRAFIA DEL DISCO

La crittografia del disco protegge i dati unicamente "a riposo", vale a dire quando il dispositivo è spento. Dal momento che è trasparente ed è implementata a livello del kernel, dopo il mounting un volume crittografato risulta indistinguibile da un volume in chiaro per i processi a livello utente. Per questo motivo la crittografia del disco non protegge i dati dai programmi dannosi in esecuzione sul device. Le applicazioni che hanno a che fare con dati sensibili non dovrebbero affidarsi unicamente alla crittografia dell'intero disco, ma dovrebbero implementare la propria crittografia basata su file. La chiave di crittografia dei file deve essere crittografata con una KEK derivata da una password fornita dall'utente, oppure da una proprietà hardware immutabile se i dati devono essere associati al dispositivo. Per garantire l'integrità dei file, i dati crittografati devono essere autenticati utilizzando uno schema di crittografia autenticato come GCM, oppure con una funzione di autenticazione aggiuntiva come HMAC.

Sicurezza dello schermo

Un metodo per controllare l'accesso a un dispositivo Android consiste nel richiedere l'autenticazione dell'utente prima di consentire l'accesso all'UI di sistema e alle applicazioni. L'autenticazione dell'utente viene implementata mostrando una *schermata di blocco* (detta anche *lockscreen*) ogni volta che il dispositivo viene avviato o si accende il suo schermo. La schermata di blocco su un dispositivo monoutente, configurata per richiedere un PIN numerico per lo sblocco, è simile a quella nella Figura 10.6.

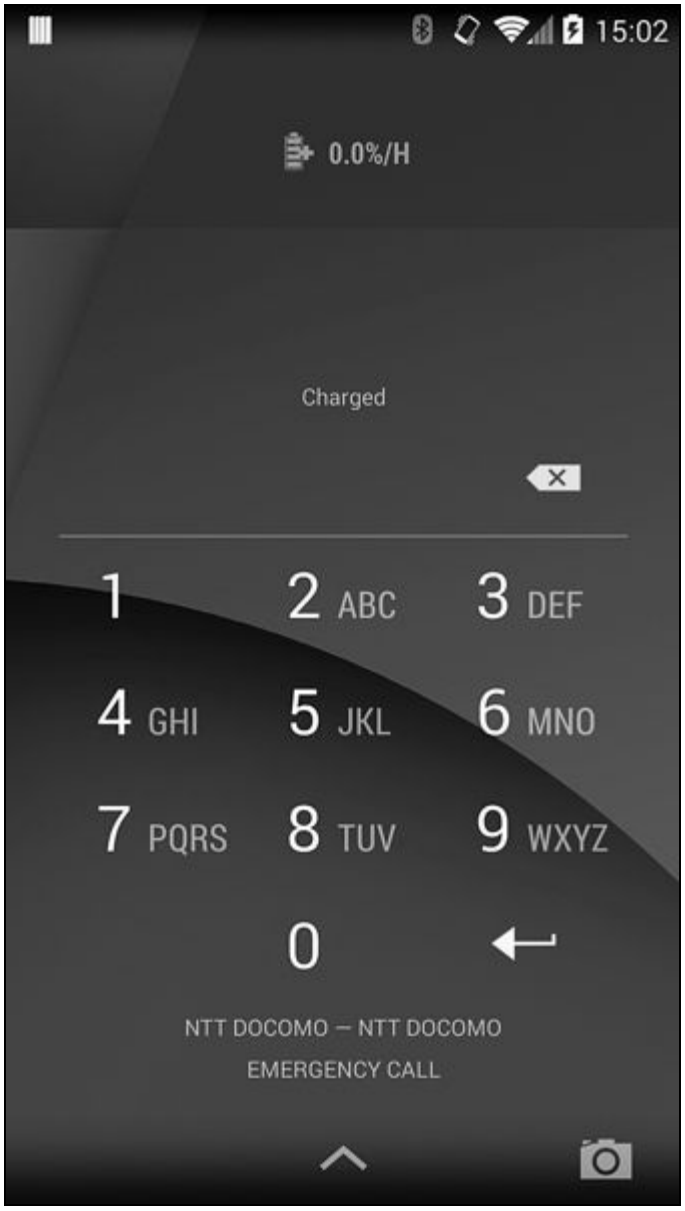

Figura 10.6 Schermata di blocco con PIN.

Nelle prime versioni di Android la schermata di blocco era progettata per proteggere solamente l'accesso alla UI del dispositivo. Con l'evoluzione della piattaforma, è stata ampliata con funzionalità per la visualizzazione di widget che mostrano lo stato aggiornato del dispositivo o delle applicazioni, per il passaggio da un utente all'altro sui dispositivi multiutente e per lo sblocco del keystore di sistema. Analogamente, il PIN o la password di sblocco oggi è usato per derivare la chiave di crittografia dell'archivio delle credenziali (implementazione software) e la KEK della chiave di crittografia del disco.

Implementazione della schermata di blocco

La schermata di blocco di Android (o *keyguard*) viene implementata come le normali applicazioni Android, ovvero con dei widget disposti in una finestra. La sua peculiarità è che la finestra risiede su un livello superiore in cui le altre applicazioni non possono agire e su cui non hanno controllo. Inoltre, il keyguard intercetta i normali pulsanti di navigazione: essendo quindi impossibile aggirarlo, si ottiene il "blocco" del dispositivo.

Il livello della finestra del keyguard non è tuttavia il più alto: le finestre di dialogo che questo genera e la barra di stato sono disegnate al di sopra. Un elenco delle finestre attualmente mostrate può essere ottenuto con lo strumento Hierarchy Viewer disponibile in ADT. Quando lo schermo è bloccato, la finestra attiva è *Keyguard*, come mostrato nella Figura 10.7.

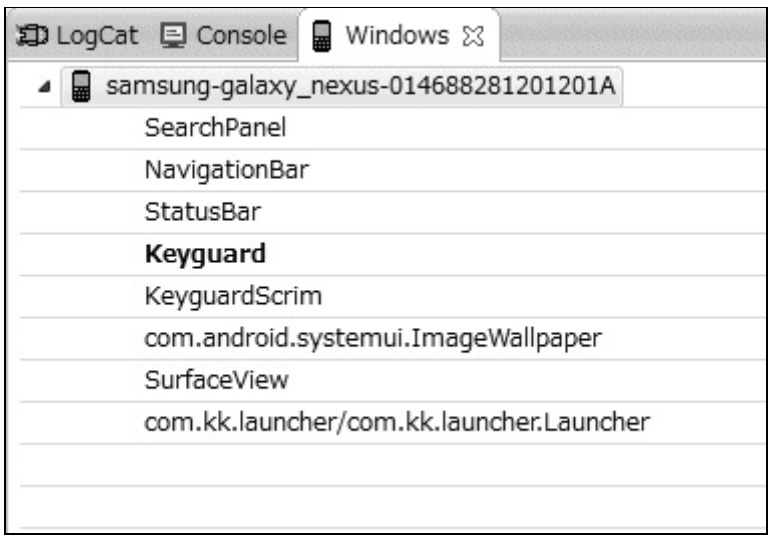

Figura 10.7 Posizione della finestra del keyguard nello stack di finestre di Android.

NOTA

Precedentemente ad Android 4.0, le applicazioni di terze parti potevano mostrare le finestre nel livello del keyguard, e di conseguenza erano in grado di intercettare il tasto Home e implementare funzionalità in stile "kiosk". Tuttavia, a seguito dell'abuso di questa funzionalità da parte di alcune applicazioni malware, a partire da Android 4.0 l'aggiunta di finestre al livello del keyguard richiede il permesso di firma `INTERNAL_SYSTEM_WINDOW`, disponibile solo per le applicazioni di sistema.

Per lungo tempo il keyguard è stato un dettaglio di implementazione del sistema di finestre di Android e non era separato in un componente dedicato. Con l'introduzione dei widget della schermata di blocco, degli screen saver e del supporto per più utenti, il keyguard ha acquisito nuove funzionalità e alla fine, in Android 4.4, è divenuto un'applicazione di sistema dedicata, chiamata appunto *Keyguard*. L'app *Keyguard* risiede nel processo `com.android.systemui` insieme all'implementazione dell'UI Android core.

L'UI per ogni metodo di sblocco (di cui parliamo più avanti) è implementata come un componente di visualizzazione specializzato, ospitato da una classe contenitore dedicata, chiamata `KeyguardHostView`, insieme ai widget del keyguard e ad altri componenti helper. Per esempio, la visualizzazione di sblocco mediante PIN mostrata nella Figura 10.6 è implementata nella classe `KeyguardPINView`, mentre lo sblocco mediante password è implementato dalla classe `KeyguardPasswordView`. La classe `KeyguardHostView` sceglie e mostra automaticamente la visualizzazione del keyguard più appropriata per il metodo di sblocco attualmente configurato e per lo stato del dispositivo. Le visualizzazioni di sblocco delegano il controllo della password alla classe `LockPatternUtils`, responsabile del confronto dell'input utente con le credenziali di sblocco salvate, nonché del salvataggio su disco delle modifiche alla password e dell'aggiornamento dei metadati di autenticazione.

Oltre alle implementazioni delle visualizzazioni di sblocco del keyguard, l'applicazione di sistema `Keyguard` include il servizio esportato `KeyguardService`, che espone un'interfaccia AIDL remota chiamata `IKeyguardService`. Questo servizio permette ai client di controllare lo stato attuale del keyguard, impostare l'utente corrente, avviare la fotocamera e nascondere o disabilitare il keyguard. Le operazioni che modificano lo

stato del keyguard sono protette dal permesso di firma del sistema `CONTROL_KEYGUARD`.

Metodi di sblocco del keyguard

Android Stock offre diversi metodi di sblocco del keyguard (detti anche *modalità di protezione* nel codice sorgente di Android). Di questi, cinque possono essere selezionati direttamente nella schermata *Choose screen lock*: si tratta di *Slide*, *Face Unlock*, *Pattern*, *PIN* e *Password*, come mostrato nella Figura 10.8.

Il metodo di sblocco *Slide* non richiede l'autenticazione dell'utente e il suo livello di sicurezza è quindi equivalente alla selezione di *None* (Nessuno). Entrambi gli stati vengono rappresentati internamente impostando la modalità di protezione corrente sul valore enum `KeyguardSecurityModel.SecurityMode.None`. A oggi *Face Unlock* è l'unica implementazione della modalità di protezione `SecurityMode.Biometric` e internamente è definita "biometrica debole" (una "biometrica forte" potrà essere implementata in una versione futura con il riconoscimento delle impronte digitali o delle iridi). I metodi di sblocco non compatibili con la policy di sicurezza corrente del dispositivo (i primi tre nella Figura 10.8) sono disabilitati e non possono essere selezionati. La policy di sicurezza può essere impostata esplicitamente da un amministratore del dispositivo oppure implicitamente abilitando una funzione di sicurezza del sistema operativo come l'archiviazione delle credenziali o la crittografia dell'intero disco.

Il metodo di sblocco *Pattern* (`SecurityMode.Pattern`) è specifico per Android e richiede, per sbloccare il dispositivo, di tracciare un pattern predefinito su una griglia 3×3, come mostrato nella Figura 10.9.

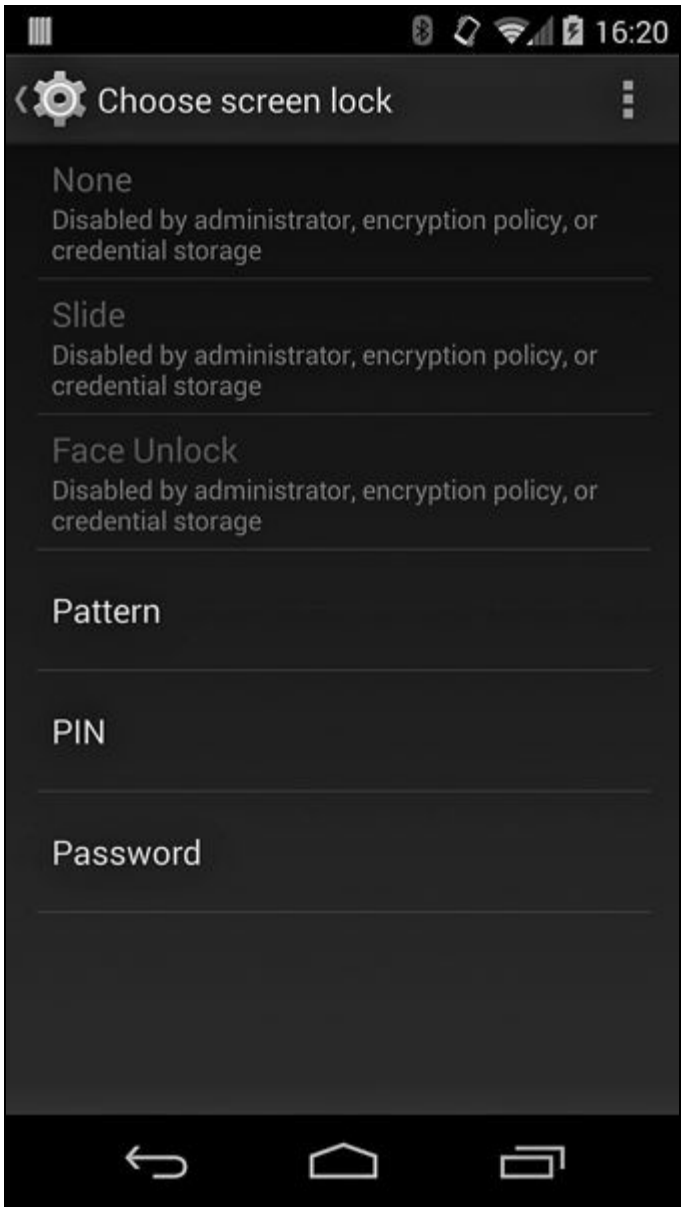

Figura 10.8 Metodi di sblocco del keyguard selezionabili direttamente.

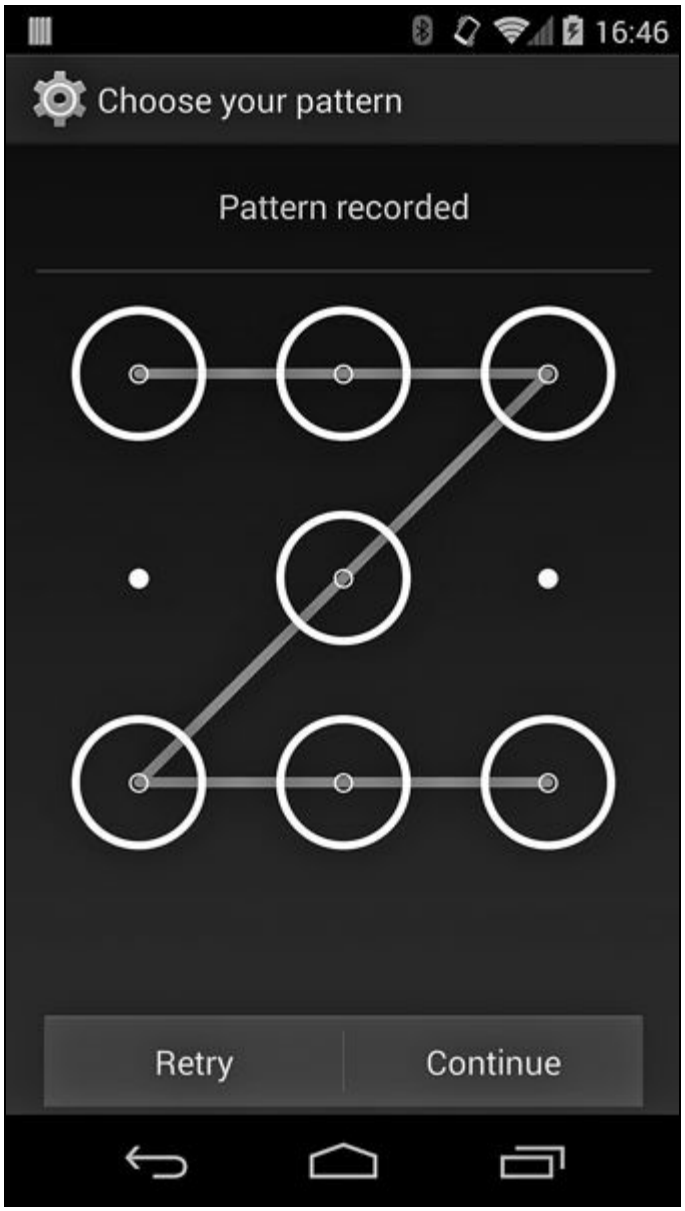

Figura 10.9 Configurazione del metodo di sblocco Pattern.

I metodi di sblocco *PIN* (`SecurityMode.PIN`) e *Password* (`SecurityMode.Password`) sono implementati in modo simile e differiscono solo per l'ambito dei caratteri accettati: solo numeri (0-9) per *PIN* e caratteri alfanumerici per *Password*.

L'enum `SecurityMode` definisce altri tre metodi di sblocco che non sono selezionabili direttamente nella schermata *Choose screen lock*: `SecurityMode.Account`, `SecurityMode.SimPin` e `SecurityMode.SimPuk`. Il metodo `SecurityMode.Account` è disponibile solo sui dispositivi che supportano gli account Google (device Google Experience) e non è un metodo di sblocco indipendente; può tuttavia essere usato come metodo di fallback per un'altra modalità di protezione. Allo stesso modo, `SecurityMode.SimPin` e

`SecurityMode.SimPuk` di per sé non sono metodi di sblocco della schermata; sono disponibili solo qualora la SIM del dispositivo richieda un PIN prima dell'uso. Visto che la SIM ricorda lo stato di autenticazione del PIN, il PIN o il PUK devono essere inseriti una sola volta all'avvio del dispositivo (o se lo stato della SIM viene reimpostato per altri motivi). Le implementazioni di ciascuna modalità di protezione della schermata di blocco sono descritte nei dettagli nei prossimi paragrafi.

Face Unlock

Face Unlock è un metodo di sblocco relativamente nuovo introdotto in Android 4.0. Usa la fotocamera anteriore del dispositivo per registrare un'immagine del volto del proprietario (Figura 10.10) e fa affidamento sulla tecnologia di riconoscimento delle immagini per riconoscere il volto acquisito durante lo sblocco del dispositivo. Per quanto l'accuratezza di *Face Unlock* sia stata notevolmente migliorata dai tempi della sua introduzione, è comunque considerato il metodo di sblocco meno sicuro, persino se la schermata di configurazione avverte l'utente che una persona dall'aspetto simile al suo potrebbe essere in grado di sbloccare lo smartphone. Inoltre, *Face Unlock* richiede un metodo di sblocco di riserva (un pattern o un PIN) per le situazioni in cui il riconoscimento del volto non è possibile (illuminazione scadente, malfunzionamento della fotocamera e così via). L'implementazione di *Face Unlock* si basa sulla tecnologia di riconoscimento facciale sviluppata dalla società PittPatt (Pittsburgh Pattern Recognition), acquisita da Google nel 2011. Il codice è proprietario e non sono disponibili dettagli sul formato dei dati archiviati o sugli algoritmi di riconoscimento utilizzati. Attualmente l'implementazione di *Face Unlock* risiede nel package `com.android.facelock`.

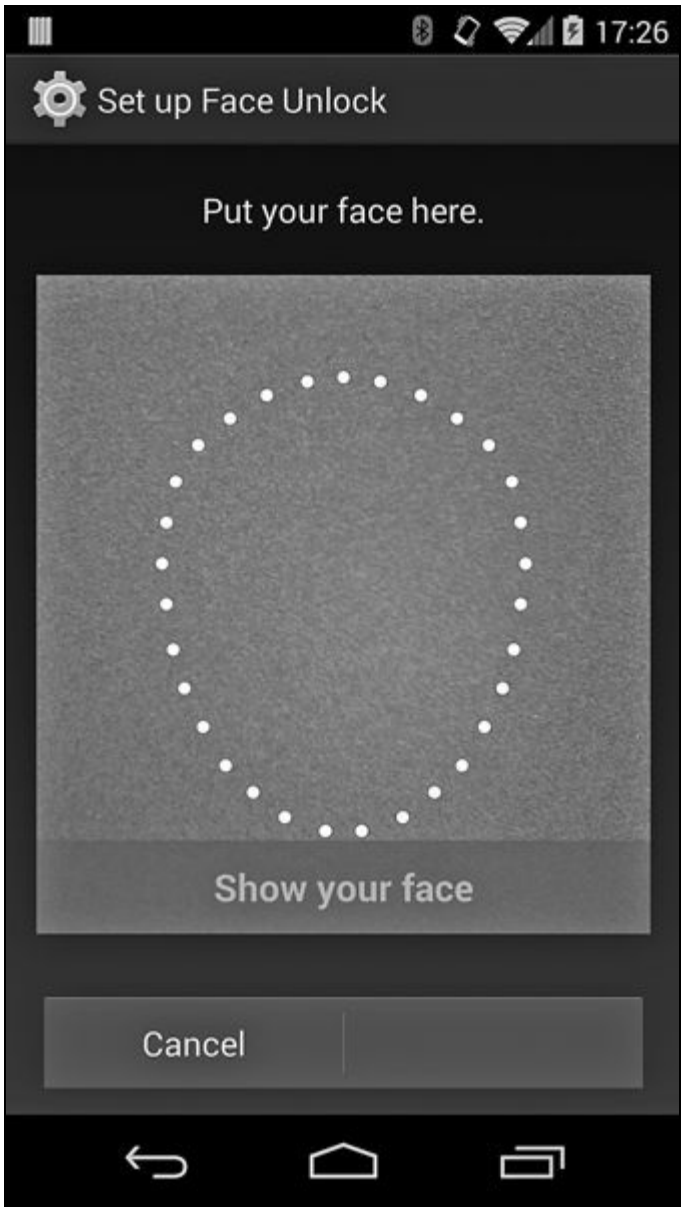

Figura 10.10 Schermata di configurazione di Face Unlock.

Sblocco mediante pattern

Come mostrato nella Figura 10.9, il codice per lo sblocco mediante pattern viene inserito congiungendo almeno quattro punti su una matrice 3×3. Ogni punto può essere usato una sola volta (i punti incrociati vengono ignorati) e il numero massimo di punti è nove. Internamente il pattern viene memorizzato come una sequenza di byte, con ciascun punto rappresentato dal suo indice, dove 0 è il punto in alto a sinistra e 8 è quello in basso a destra. Il pattern è quindi simile a un PIN con un minimo di quattro e un massimo di nove cifre, che usa solo nove cifre distinte (da 0 a 8). Tuttavia, poiché i punti non possono essere ripetuti, il

numero di varianti in un pattern di sblocco è considerevolmente inferiore rispetto a quello di un PIN di nove cifre.

L'hash per il blocco mediante pattern è salvato in */data/system/gesture.key* (*/data/system/users/<user ID>/gesture.key* sui dispositivi multiutente) come valore SHA-1 senza salt. Con un semplice dump di questo file possiamo vedere facilmente che il contenuto del file `gesture.key` per il pattern nella Figura 10.9 (rappresentato come *00010204060708* in esadecimale), mostrato nel Listato 10.8, corrisponde all'hash SHA-1 della sequenza di byte del pattern, *6a062b9b3452e366407181a1bf92ea73e9ed4c48* in questo esempio.

Listato 10.8 Contenuti del file /data/system/gesture.key.

```
# od -t x1 /data/system/gesture.key
0000000 6a 06 2b 9b 34 52 e3 66 40 71 81 a1 bf 92 ea 73
0000020 e9 ed 4c 48
```

Visto che non viene utilizzato un valore salt casuale nel calcolo dell'hash, ogni pattern genera sempre lo stesso valore hash, e quindi è relativamente facile generare una tabella precalcolata di tutti i possibili pattern con i rispettivi hash (le tabelle sono già disponibili online). Di conseguenza, una volta ottenuto il file `gesture.key`, è possibile ricavare il pattern in un istante. Va detto però che il file è di proprietà dell'utente `system` e che i suoi permessi sono impostati a 0600, quindi il recupero di solito è impossibile sui dispositivi di produzione. Il pattern inserito viene confrontato con l'hash salvato utilizzando il metodo `checkPattern()` della classe `LockScreenUtils`; l'hash del pattern viene calcolato e salvato usando il metodo `saveLockPattern()` della stessa classe. Il salvataggio del pattern provoca anche l'impostazione del valore di qualità della password corrente su `DevicePolicyManager.PASSWORD_QUALITY_SOMETHING`.

Un'altra proprietà "sfortunata" del metodo di sblocco mediante pattern è legato all'uso diretto del dito (e non di uno stilo o di uno strumento simile) sui touch screen capacitivi: la ripetizione del pattern di sblocco lascia una traccia distinta sul touch screen, che rende il dispositivo vulnerabile al cosiddetto *attacco smudge* (*smudge* in inglese significa sbavatura). Con l'illuminazione e la fotocamera adeguate è possibile rilevare le tracce lasciate dal dito sullo schermo e dedurre agevolmente il

pattern. Per questi motivi il livello di sicurezza del metodo di sblocco mediante pattern è considerato molto basso. Inoltre, poiché il numero di combinazioni è limitato, il pattern è una fonte scadente di entropia e il suo uso è vietato quando le credenziali di sblocco dell'utente sono impiegate per derivare una chiave di crittografia, come quella utilizzata per il keystore del sistema e la crittografia del dispositivo.

Analogamente a *Face Unlock*, il metodo di sblocco mediante pattern supporta un meccanismo di sblocco di riserva reso disponibile solo dopo che l'utente ha inserito più di cinque volte un pattern non valido. L'autenticazione di riserva deve essere attivata manualmente premendo il pulsante *Forgot pattern* nella parte inferiore della schermata di blocco. Una volta premuto il pulsante, il dispositivo entra nella modalità di protezione `SecurityMode.Account` e mostra la schermata della Figura 10.11.

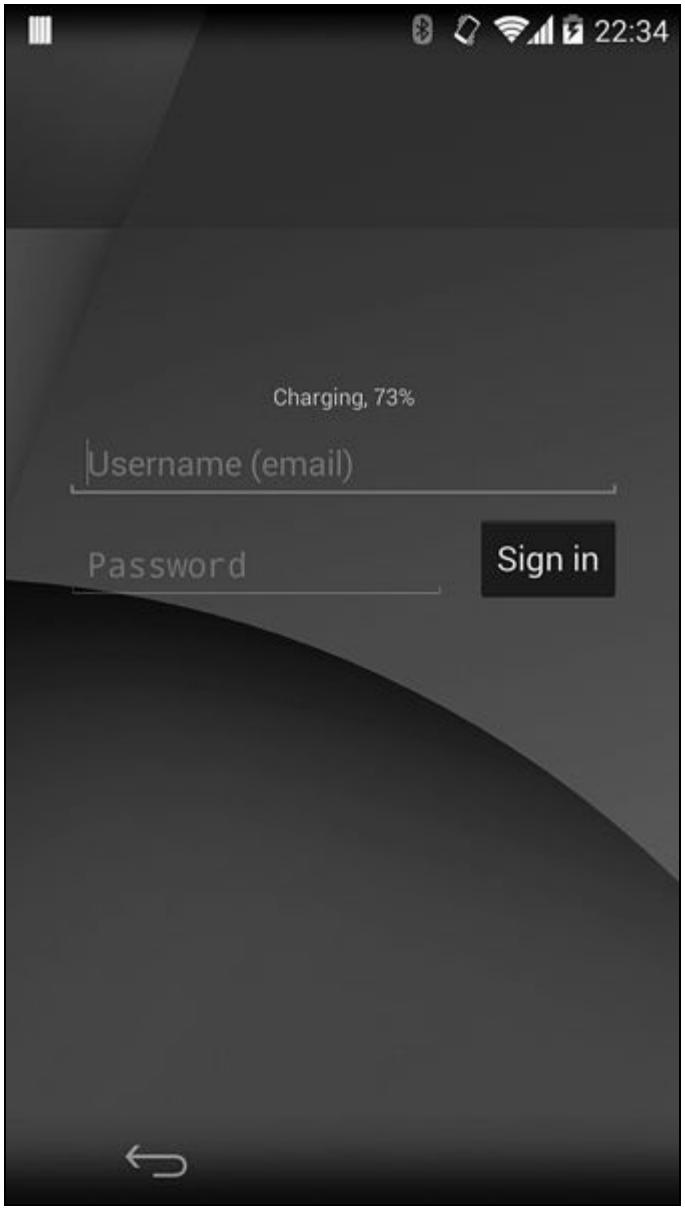

Figura 10.11 Modalità di sblocco dell'account Google.

L'utente può inserire le credenziali di qualsiasi account Google registrato sul dispositivo per sbloccarlo e quindi reimpostare o cambiare il metodo di sblocco. Di conseguenza, la presenza di un account Google con una password facile da indovinare (o condivisa) registrata sul dispositivo può rappresentare una possibile backdoor per la schermata di blocco del device.

NOTA

Attualmente gli account Google configurati per richiedere l'autenticazione a due fattori non possono essere usati per sbloccare il dispositivo.

Sblocco mediante PIN e password

I metodi basati su PIN e password fondamentalmente si equivalgono: in pratica, confrontano l'hash dell'input utente con un hash con salt memorizzato sul dispositivo ed effettuano lo sblocco se i due valori corrispondono. L'hash del PIN o della password è una combinazione dei valori hash SHA-1 e MD5 dell'input utente, a cui viene applicato il salt con un valore casuale a 64 bit. L'hash calcolato è salvato in */data/misc/password.key* (*/data/system/users/<user ID>/password.key* sui dispositivi multiutente) come una stringa esadecimale dall'aspetto simile a quello mostrato nel Listato 10.9.

Listato 10.9 Contenuti del file /data/misc/password.key.

```
# cat /data/system/password.key && echo
9B93A9A846FE2FC11D49220FC934445DBA277EB0AF4C9E324D84FFC0120D7BAE1041FAAC
```

Nelle versioni di Android precedenti alla 4.2 il salt usato per calcolare i valori hash era salvato nella tabella `secure` del content provider `SettingsProvider` di sistema, all'interno della chiave `lockscreen.password_salt`; nelle successive è stato trasferito in un database dedicato, insieme ad altri metadati relativi alla schermata di blocco, al fine di supportare più utenti per ogni dispositivo. A partire da Android 4.4, il database si trova in */data/system/locksettings.db* e vi si accede tramite l'interfaccia AIDL `ILockSettings` di `LockSettingsService`.

L'accesso al servizio richiede il permesso di firma `ACCESS_KEYGUARD_SECURE_STORAGE`, concesso solo alle applicazioni di sistema. Il database `locksettings.db` contiene una singola tabella, anch'essa chiamata `locksettings`, che può racchiudere dati simili a quelli del Listato 10.10 per un particolare utente (la colonna `user` contiene lo user ID Android).

Listato 10.10 Contenuti di /data/system/locksettings.db per l'utente proprietario.

```
sqlite> select name, user, value from locksettings where user=0;
name                              |user|value
--altro codice--
lockscreen.password_salt          |0   |6909501022570534487(1)
--altro codice--
lockscreen.password_type_alternate|0   |0(2)
lockscreen.password_type          |0   |131072(3)
lockscreen.passwordhistory        |0   |5BFE43E89C989972EF0FA0EC00BA30F356EE7B
7C7BF8BC08DEA2E067FF6C18F8CD7134B8,EE29A531FE0903C2144F0618B08D1858473C50341A7
8DEA85D219BCD27EF184BCBC2C18C(4)
```

Qui l'impostazione `lockscreen.password_salt` **(1)** memorizza il valore del salt a 64 bit (rappresentato nel tipo Java `long`), mentre l'impostazione `lockscreen.password_type_alternate` **(2)** contiene il tipo del metodo di sblocco di riserva (o alternativo, 0 per nessuno) per il metodo di sblocco attuale. `lockscreen.password_type` **(3)** memorizza il tipo di password attualmente selezionato, rappresentato dal valore della costante `PASSWORD_QUALITY` corrispondente definita nella classe `DevicePolicyManager`. In questo esempio, `131072` (0x00020000 in esadecimale) corrisponde alla costante `PASSWORD_QUALITY_NUMERIC`, ovvero la qualità della password offerta da un PIN numerico. Infine, `lockscreen.passwordhistory` **(4)** contiene la cronologia delle password, salvata come una sequenza di hash di PIN o password precedenti separati da virgole. La cronologia viene salvata solo se la sua lunghezza è stata impostata a un valore maggiore di zero utilizzando uno dei metodi `setPasswordHistoryLength()` della classe `DevicePolicyManager`. Se è disponibile la cronologia delle password, è proibito inserire una nuova password identica a una già presente nella cronologia.

L'hash della password può essere calcolato facilmente concatenando la stringa della password o del PIN (`1234` in questo esempio) con il valore del salt formattato come una stringa esadecimale (`5fe37a926983d657` in questo esempio) e calcolando gli hash SHA-1 e MD5 della stringa risultante, come mostrato nel Listato 10.11.

Listato 10.11 Calcolo dell'hash di un PIN o di una password con sha1sum e md5sum.

```
$ SHA1='echo -n '12345fe37a926983d657'|sha1sum|cut -d- -f1|tr '[a-z]' '[A-Z]''1
$ MD5='echo -n '12345fe37a926983d657'|md5sum|cut -d- -f1|tr '[a-z]' '[A-Z]''2
$ echo "$SHA1$MD5"|tr -d ' '3
9B93A9A846FE2FC11D49220FC934445DBA277EB0AF4C9E324D84FFC0120D7BAE1041FAAC
```

In questo esempio gli hash vengono calcolati utilizzando i comandi `sha1sum` **(1)** e `md5sum` **(2)**. Dopo la concatenazione **(3)**, l'output dei due comandi produce la stringa contenuta nel file `password.key` e mostrata nel Listato 10.9.

Anche se l'uso di un hash casuale rende impossibile l'utilizzo di un'unica tabella precalcolata per sferrare attacchi di forza bruta al PIN o alla password di un dispositivo, il calcolo della password o dell'hash richiede una singola chiamata dell'hash; pertanto, la generazione di una tabella di hash mirata a un particolare dispositivo (presumendo che sia disponibile anche il valore del salt) è una procedura relativamente economica. Inoltre, anche se Android calcola entrambi gli hash SHA-1 e MD5 del PIN o della password, questa operazione non aumenta la sicurezza perché è sufficiente prendere di mira l'hash più corto (MD5) per svelare il PIN o la password.

La password inserita viene confrontata con l'hash memorizzato utilizzando il metodo `LockPatternUtils.checkPassword()`; l'hash di una password fornita dall'utente viene calcolato e salvato utilizzando uno dei metodi `saveLockPassword()` della classe. La chiamata a `saveLockPassword()` aggiorna il file `password.key` per l'utente target (o corrente). Come `gesture.key`, questo file è di proprietà dell'utente `system` e presenta i permessi 0600. Oltre ad aggiornare l'hash della password, `saveLockPassword()` calcola la complessità della password inserita e aggiorna la colonna `value` corrispondente alla chiave `lockscreen.password_type` (**(3)** nel Listato 10.10) in `locksettings.db` con il valore di complessità calcolato. Se è abilitata la cronologia delle password, `saveLockPassword()` aggiunge poi l'hash del PIN o della password alla tabella `locksettings`.

Non dimenticate che, se il dispositivo è crittografato, il PIN o la password permette anche di derivare una chiave KEK che crittografa la chiave di crittografia del disco. Di conseguenza, la modifica del PIN o

della password dell'utente proprietario provoca anche una nuova crittografia della chiave del disco attraverso una chiamata al metodo `changeEncryptionPassword()` del servizio di sistema `MountService` (la modifica del PIN o della password di un utente secondario non influisce sulla chiave di crittografia del disco).

Sblocco mediante PIN e PUK

Le modalità di protezione PIN e PUK non sono di per sé metodi di sblocco dello schermo, perché dipendono dallo stato della SIM del dispositivo e sono mostrati solo se questa è in uno stato di blocco. Una SIM può richiedere agli utenti di inserire un codice PIN preconfigurato per sbloccare la scheda e ottenere l'accesso alle chiavi di autenticazione di rete memorizzate al suo interno, richieste per registrarsi nella rete mobile ed effettuare chiamate non di emergenza.

Visto che la SIM ricorda il suo stato di blocco fino al reset, di solito il codice PIN deve essere inserito una sola volta al primo avvio del dispositivo. Se si inserisce più di tre volte un codice errato, la SIM si blocca e l'utente deve inserire un codice diverso per sbloccarla: tale codice è chiamato PUK (*PIN Unlock Key*) o PUC (*Personal Unblocking Code*).

Quando è visibile la schermata di blocco, Android controlla lo stato della SIM: se è `State.PIN_REQUIRED` (definito nella classe `IccCardConstants`), viene mostrato il keyguard di sblocco della SIM (Figura 10.12). Quando l'utente inserisce un PIN di sblocco della SIM, questo viene passato al metodo `supplyPinReportResult()` dell'interfaccia `ITelephony` (implementata nell'applicazione di sistema *TeleService*), che a sua volta lo passa al processore baseband del dispositivo (il componente che implementa la comunicazione di rete mobile, a volte chiamato anche *modem* o *radio*) tramite il daemon dell'interfaccia radio (`rild`). Infine, il processore baseband, connesso direttamente alla SIM, invia il PIN a questa e riceve in cambio un codice di stato, che viene restituito alla vista di sblocco seguendo lo stesso percorso. Se il codice di stato indica che la SIM ha accettato il PIN e che non è configurato alcun blocco dello schermo,

viene visualizzata la schermata principale (*launcher*); se invece è stato configurato un blocco dello schermo, questo viene mostrato subito dopo lo sblocco della SIM, pertanto l'utente deve inserire le sue credenziali per sbloccare il dispositivo.

Figura 10.12 Schermata di sblocco della SIM.

Se la SIM è bloccata (ovvero nello stato `PUK_REQUIRED`), Android mostra una schermata di inserimento del PUK e consente all'utente di configurare un nuovo PIN dopo aver sbloccato la scheda. Il PUK e il nuovo PIN vengono passati al metodo `supplyPukReportResult()` dell'interfaccia `ITelephony`, che li trasferisce alla SIM. Se è configurato un blocco dello schermo, questo viene mostrato a seguito della convalida del PUK e della configurazione del nuovo PIN.

L'applicazione di sistema `Keyguard` monitora i cambiamenti di stato della SIM registrando il broadcast `TelephonyIntents.ACTION_SIM_STATE_CHANGED` e mostra la schermata di blocco se la scheda è bloccata o disabilitata in modo permanente. Gli utenti possono attivare o disattivare la protezione mediante PIN della SIM selezionando *Settings* > *Security* > *Set up SIM card lock* e selezionando la casella di controllo *Lock SIM card*.

Protezione contro gli attacchi di forza bruta

Le password complesse sono difficili da inserire sulla tastiera di un touch screen, pertanto spesso gli utenti scelgono credenziali di sblocco relativamente brevi ma facili da indovinare o da recuperare con un attacco di forza bruta. Android protegge dagli attacchi di forza bruta eseguiti direttamente sul dispositivo (attacchi online) chiedendo agli utenti di attendere 30 secondi ogni cinque tentativi di autenticazione consecutivi non riusciti, come mostrato nella Figura 10.13. Questa tecnica è detta *limitazione del rate* (o *rate limiting*).

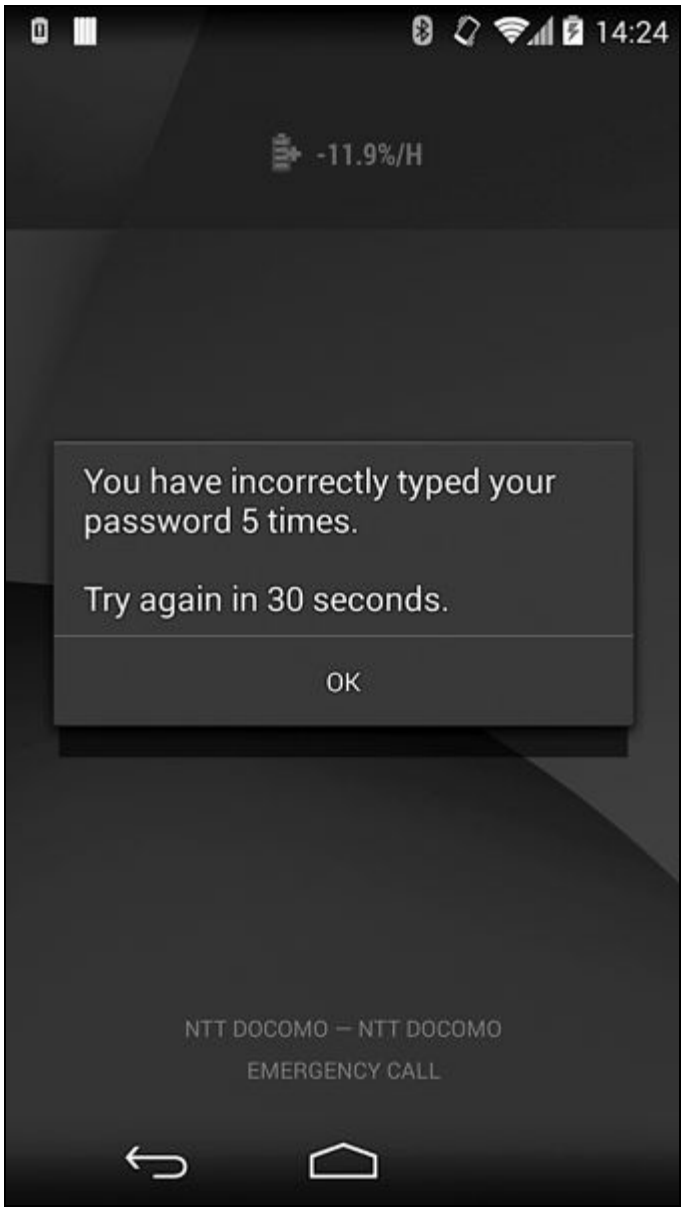

Figura 10.13 Limitazione del rate dopo cinque tentativi di autenticazione non riusciti consecutivi.

Per scoraggiare ulteriormente questi attacchi, è possibile impostare regole di complessità, scadenza e cronologia delle password utilizzando l'API `DevicePolicyManager`, come già spiegato nel Capitolo 9. Se il dispositivo contiene o consente l'accesso a dati aziendali sensibili, gli amministratori del dispositivo possono inoltre impostare una soglia per il numero di tentativi di autenticazione non riusciti utilizzando il metodo `DevicePolicyManager.setMaximumFailedPasswordsForWipe()`. Al raggiungimento della soglia tutti i dati utente sul dispositivo vengono automaticamente eliminati, impedendo ai criminali di ottenere accesso non autorizzato ai dati.

Debug USB sicuro

Una delle ragioni del successo di Android è la limitata barriera di accesso allo sviluppo di applicazioni; le app possono essere sviluppate su qualunque sistema operativo, in un linguaggio di alto livello, senza la necessità di investire in strumenti o hardware per sviluppatori (se si utilizza l'emulatore di Android). Lo sviluppo di software per dispositivi incorporati o comunque dedicati è sempre stato difficile, perché di solito è complicato (e in alcuni casi impossibile) esaminare lo stato interno di un programma o comunque interagire con il dispositivo per eseguire il debug dei programmi.

Sin dalle sue prime versioni Android ha incluso un potente toolkit di interazione con il dispositivo che permette il debug interattivo e l'analisi dello stato del dispositivo: tale strumento è chiamato *Android Debug Bridge* (ADB). ADB è generalmente disattivato sui dispositivi consumer, ma può essere attivato dall'UI di sistema per abilitare lo sviluppo di app e il debug sul dispositivo. Offre l'accesso privilegiato al file system e alle applicazioni del dispositivo, quindi può essere usato per ottenere l'accesso non autorizzato ai dati. Nei prossimi paragrafi vedremo l'architettura di ADB e i passi compiuti dalle più recenti versioni di Android per limitare l'accesso ad ADB.

Panoramica su ADB

ADB tiene traccia di tutti i dispositivi (o emulatori) connessi a un host e offre vari servizi ai suoi client (client a riga di comando, IDE e così via). È costituito da tre componenti principali: il server ADB, il daemon ADB (`adbd`), e il client a riga di comando predefinito (*adb*). Il server ADB viene eseguito sulla macchina host come un processo in background e separa i client dai dispositivi o dagli emulatori veri e propri. Monitora la connettività dei dispositivi e ne imposta di conseguenza lo stato (`CS_CONNECTED`, `CS_OFFLINE`, `CS_RECOVERY` e così via).

Il daemon ADB viene eseguito su un dispositivo Android (o emulatore) e offre i servizi veri e propri usati dai client; si connette al server ADB tramite USB o TCP/IP e riceve ed elabora i comandi da esso. Il client a riga di comando *adb* consente di inviare comandi a un particolare dispositivo: in pratica, è implementato nello stesso binario del server ADB e quindi condivide gran parte del suo codice. La Figura 10.14 mostra l'architettura di ADB.

NOTA

Oltre all'implementazione nativa nel comando `adb` e a quella basata su Java nel plug-in Eclipse di *Android Development Tools* (ADT), sono disponibili anche varie implementazioni di terze parti del protocollo ADB, tra cui un client Python (Anthony King, "PyAdb: basic ADB core for python using TCP", http://bit.ly/11rLQwL) e un server ADB implementato in JavaScript (Kenny Root, "adb-on-chrome: ADB (Android Debug Bridge) server as a Chrome extension", http://bit.ly/1vqMkAO), che può essere incorporato nel browser Chrome come estensione.

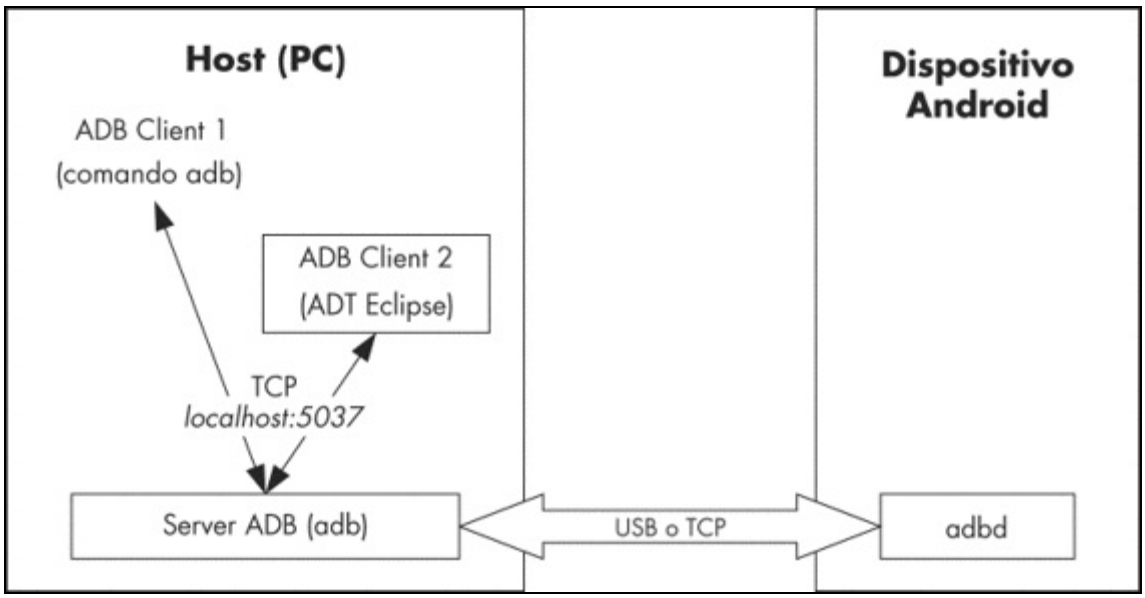

Figura 10.14 Architettura di ADB.

Il client comunica con il server ADB locale tramite TCP (in genere su *localhost:5037*) utilizzando comandi di testo e riceve in cambio le risposte *OK* o *FAIL*. Alcuni comandi, come l'enumerazione dei dispositivi, il port forwarding o il riavvio del daemon sono gestiti dal daemon locale; altri (come la shell o l'accesso ai log) richiedono una connessione al dispositivo Android target. L'accesso al dispositivo è generalmente ottenuto inoltrando i flussi di input e output da e verso l'host. Lo strato di trasporto che implementa questa funzionalità utilizza semplici messaggi con un header da 24 byte, contenente un identificatore

del comando, due argomenti, la lunghezza e il CRC32 del payload opzionale che segue e un valore magico che inverte tutti i bit del comando. La struttura del messaggio è definita in *system/core/adb/adb.h* ed è mostrata come riferimento nel Listato 10.12. I messaggi sono a loro volta incapsulati in pacchetti, inviati sul link USB o TCP al server ADB in esecuzione sul dispositivo.

Listato 10.12 Struttura del messaggio ADB.

```
struct amessage {
    unsigned command;       /* costante di identificazione del comando   */
    unsigned arg0;          /* primo argomento                           */
    unsigned arg1;          /* secondo argomento                         */
    unsigned data_length;   /* lunghezza del payload (0 consentito)      */
    unsigned data_check;    /* checksum dei payload dati                 */
    unsigned magic;         /* comando ^ 0xffffffff                      */
};
```

Non scenderemo ulteriormente nei dettagli del protocollo ADB se non per osservare i comandi di autenticazione aggiunti al protocollo per implementare il debug USB sicuro. Per maggiori dettagli su ADB consultate la descrizione del protocollo nel file *system/core/adb/protocol.txt* del source tree di Android.

NOTA

Potete abilitare i trace log per tutti i servizi ADB impostando la variabile di ambiente `ADB_TRACE` a `1` sull'host e la proprietà di sistema `persist.adb.trace_mask` sul dispositivo. Per scegliere i servizi di cui eseguire il tracing è sufficiente impostare il valore di `ADB_TRACE` o `persist.adb.trace_mask` su un elenco di tag di servizio separati da virgole o spazi (come separatore sono supportati anche i due punti e il punto e virgola). Fate riferimento a *system/core/adb/adb.c* per l'elenco completo dei tag supportati.

Esigenza di ADB sicuri

Se vi siete mai occupati di sviluppo, sapete che "debug" e "sicuro" sono due termini opposti: il debug, infatti, di solito richiede l'esame (e talvolta anche la modifica) dello stato interno del programma, il dump dei dati di comunicazione crittografati in file di log, l'accesso root universale e altre attività terrificanti ma necessarie. Il debug è già abbastanza difficile quando non ci si preoccupa della sicurezza, quindi perché complicare le cose ulteriormente aggiungendo anche altri strati di protezione? Il debug in Android, così come fornito da ADB, è piuttosto versatile e offre un controllo quasi completo su un dispositivo, quando è

abilitato. La funzionalità è naturalmente la benvenuta durante lo sviluppo o il test di un'applicazione (o del sistema operativo stesso), ma può essere utilizzata anche per altri scopi.

Ecco un elenco selettivo di operazioni consentite da ADB.

- Copiare file da e verso il dispositivo.
- Eseguire il debug di app in esecuzione sul dispositivo (tramite JWDP o `gdbserver`).
- Eseguire comandi di shell sul dispositivo.
- Ottenere i log di sistema e applicazioni.
- Installare e rimuovere app.

Se il debug è abilitato su un dispositivo, è possibile eseguire tutte le suddette operazioni e molte altre ancora (per esempio inviare eventi di tocco o inserire testo nell'UI) semplicemente collegando il dispositivo a un computer tramite un cavo USB. Visto che ADB non dipende dalla schermata di blocco del dispositivo, non è necessario sbloccare il dispositivo per eseguire i suoi comandi; inoltre, sulla maggior parte dei dispositivi che forniscono accesso root, la connessione tramite ADB permette di accedere e modificare ogni file, compresi i file di sistema e i database delle password. Peggio ancora, non è nemmeno necessario un computer dotato di strumenti di sviluppo per accedere a un dispositivo Android tramite ADB: sono sufficienti un altro dispositivo Android e un cavo USB *On-The-Go* (OTG). Sono facilmente reperibili strumenti Android in grado di estrarre quanti più dati possibile da un altro dispositivo in un tempo molto breve (Kyle Osborn, "p2p-adb Framework", https://github.com/kosborn/p2p-adb/). Se il dispositivo è stato sottoposto a rooting, tali strumenti possono estrarre tutte le vostre credenziali, disabilitare o forzare la schermata di blocco e persino accedere al vostro account Google. Anche senza rooting, comunque, tutto il contenuto della memoria esterna (in particolare le foto) è accessibile, così come i contatti e gli SMS.

Protezione di ADB

Android 4.2 è stata la prima versione a cercare di rendere l'accesso ADB più difficile nascondendo la schermata delle impostazioni *Developer options*, che ora richiede l'uso di un "codice segreto" (basta toccare sette volte il numero della build) per la relativa abilitazione. Per quanto non sia un metodo di protezione particolarmente efficace, almeno garantisce che la maggior parte degli utenti non abiliti accidentalmente l'accesso ADB. Questa è ovviamente solo una misura tampone: non appena attivate il debug USB il vostro dispositivo risulta ancora una volta vulnerabile.

Android 4.2.2 ha introdotto una soluzione adeguata con la cosiddetta funzionalità di debug USB sicuro: qui il termine "sicuro" si riferisce al fatto che solo gli host esplicitamente autorizzati dall'utente possono connettersi al daemon `adbd` sul dispositivo ed eseguire comandi di debug. Di conseguenza, se qualcuno prova a connettersi via USB a un dispositivo da un altro device per accedere ad ADB, deve prima sbloccare il dispositivo target e autorizzare l'accesso dall'host di debug facendo clic su *OK* nella finestra di dialogo di conferma mostrata nella Figura 10.15.

Figura 10.15 Finestra di autorizzazione del debug USB.

La decisione può anche essere resa definitiva selezionando la casella di controllo *Always allow from this computer*: il debug funzionerà esattamente come in precedenza, finché si rimane sulla stessa macchina.

Naturalmente, questo debug USB sicuro è efficace solo se è attiva una password della schermata di blocco ragionevolmente sicura.

NOTA

Sui tablet con supporto multiutente, la finestra di conferma viene mostrata solo all'utente primario (proprietario).

Implementazione sicura di ADB

La funzionalità di autenticazione dell'host ADB è attivata per impostazione predefinita quando la proprietà di sistema `ro.adb.secure` è

impostata su 1 e non esiste alcun modo per disabilitarla tramite l'interfaccia di sistema. Quando un dispositivo si connette a un host, inizialmente è nello stato CS_UNAUTHORIZED e passa allo stato CS_DEVICE solo dopo l'autenticazione dell'host. Gli host usano le chiavi RSA per autenticarsi con il daemon ADB sul dispositivo, generalmente seguendo questo processo in tre fasi.

Quando un host prova a connettersi, il dispositivo invia un messaggio A_AUTH con un argomento di tipo ADB_AUTH_TOKEN, che include un valore casuale di 20 byte (letto da */dev/urandom/*).

L'host risponde con un messaggio A_AUTH con un argomento di tipo ADB_AUTH_SIGNATURE, che include una firma *SHA1withRSA* del token casuale con una delle chiavi private dell'host.

Il dispositivo tenta di verificare la firma ricevuta: se tale verifica riesce, risponde con un pacchetto A_CNXN e passa allo stato CS_DEVICE; se la verifica ha esito negativo, perché il valore della firma non corrisponde o perché non esiste una chiave pubblica corrispondente con cui effettuare la verifica, il dispositivo invia un altro ADB_AUTH_TOKEN con un nuovo valore casuale, in modo che l'host possa ritentare l'autenticazione (rallentando se il numero di errori va oltre una determinata soglia).

La verifica della firma in genere non riesce la prima volta che si connette il dispositivo a un nuovo host, perché non è ancora disponibile la chiave dell'host. In tal caso l'host invia la sua chiave pubblica in un messaggio A_AUTH con un argomento ADB_AUTH_RSAPUBLICKEY. Il dispositivo prende l'hash MD5 di quella chiave e lo visualizza nella finestra di conferma *Allow USB debugging* mostrata nella Figura 10.15. adbd è un daemon nativo, pertanto la chiave deve essere passata al sistema operativo Android principale affinché il suo hash sia visualizzato sullo schermo: questo risultato si ottiene semplicemente scrivendo la chiave in un socket locale (chiamato anch'esso adbd) monitorato dal daemon adbd.

Quando si abilita il debug ADB dalla schermata delle impostazioni per gli sviluppatori, viene avviato un thread che "ascolta" tale socket adbd. Quando il thread riceve un messaggio che inizia con *PK*, lo tratta come

una chiave pubblica, quindi ne esegue il parsing, ne calcola l'hash MD5 e visualizza la finestra di conferma (implementata in un'activity dedicata, `UsbDebuggingActivity`, parte del package SystemUI). Toccando *OK*, l'activity invia una semplice risposta *OK* ad `adbd`, che usa la chiave per verificare il messaggio di autenticazione. Se selezionate la casella di controllo *Always allow from this computer*, la chiave pubblica viene scritta su disco e sarà usata automaticamente per la verifica della firma la prossima volta che vi connetterete al medesimo host.

NOTA

Dalla versione 4.3, Android consente di cancellare le chiavi di autenticazione dell'host salvate. Questa funzionalità può essere attivata selezionando *Settings* > *Developer options* > *Revoke USB debugging authorizations*.

La classe `UsbDeviceManager` offre metodi pubblici per consentire e vietare il debug USB, cancellare le chiavi di autenticazione nella cache e avviare e arrestare il daemon `adbd`. Questi metodi sono messi a disposizione delle altre applicazioni tramite l'interfaccia AIDL `IUsbManager` del servizio di sistema `UsbService`. La chiamata a metodi di `IUsbManager` che modificano lo stato del dispositivo richiede il permesso di firma di sistema `MANAGE_USB`.

Chiavi di autenticazione ADB

Pur avendo descritto il protocollo di autenticazione ADB, non abbiamo detto molto sulle chiavi effettive utilizzate nel processo, vale a dire le chiavi RSA a 2048 bit generate dal server ADB locale. Queste chiavi sono tipicamente memorizzate in *$HOME/.android* (*%USERPROFILE%\.android* in Windows) come `adbkey` (chiave privata) e `adbkey.pub` (chiave pubblica). La directory predefinita delle chiavi può essere sostituita impostando la variabile di ambiente `ANDROID_SDK_HOME`. Se è impostata la variabile di ambiente `ADB_VENDOR_KEYS`, le chiavi vengono cercate anche nella directory indicata dalla variabile. Se non vengono trovate chiavi in questi percorsi, viene generata e salvata una nuova coppia di chiavi.

Il file della chiave privata (`adbkey`), salvato esclusivamente sull'host, è nel formato PEM OpenSSL standard. Il file della chiave pubblica

(`adbkey.pub`) contiene la rappresentazione compatibile con mincrypt e codifica Base 64 della chiave pubblica: fondamentalmente è una serializzazione della struttura `RSAPublicKey` di mincrypt (vedete "Abilitazione del boot verificato" in precedenza nel capitolo), seguita da uno spazio e da un identificatore utente `user@host`. L'identificatore utente non sembra essere utilizzato, almeno attualmente, e ha un senso solo sui sistemi operativi basati su Unix; in Windows è sempre `unknown@unknown`.

Le chiavi sono memorizzate sul dispositivo nel file */data/misc/adb/adb_keys/*; le nuove chiavi autorizzate vengono aggiunte allo stesso file nel momento in cui le accettate. Le "chiavi del vendor" di sola lettura sono memorizzate nel file */adb_keys*, che però non sembra esistere sui dispositivi Nexus attuali. Le chiavi pubbliche sono nello stesso formato usato sull'host: è quindi facile caricarle in libmincrypt, a cui `adbd` si collega in modo statico. Il Listato 10.13 mostra alcuni esempi di `adb_keys`. Il file è di proprietà dell'utente `system`, il suo gruppo è impostato su `shell` e i suoi permessi su `0640`.

Listato 10.13 Contenuto del file adb_keys.

```
# cat data/misc/adb/adb_keys
QAAAAJs1UDFt17wyV+Y2GNGF+EgWoiPfsByfC4frNd3s64w3IGt25fKERnl7O8/A+iVPGv1W
--altro codice--
yZ61cFd7R6ohLFYJRPB6Dy7tISUPRpb+NF4pbQEAAQA= unknown@unknown
QAAAAKFLvP+fp1cB4Eq/6zyV+hnm1S1eV9GYd7cYe+tmwuQZFe+O4vpeow6huIN8YbBRkr7
--altro codice--
m7+bGd6F0hRkO82gopy553xywXU7rI/aMl6FBAEAAQA= user1@host2
```

Verifica del fingerprint della chiave host

Anche se la finestra di conferma del debug USB visualizza un fingerprint della chiave per consentirvi di verificare agevolmente di essere connessi all'host previsto, il client `adb` non dispone di un comando utile per visualizzare il fingerprint della chiave host. Anche se all'apparenza è difficile fare confusione (dopotutto, c'è un unico cavo collegato a un'unica macchina), quando si eseguono un paio di VM le cose possono diventare complicate. Il Listato 10.14 mostra un metodo per visualizzare il fingerprint nello stesso formato utilizzato dalla finestra di conferma della Figura 10.15 (va eseguito in *$HOME/.android* o specificando il percorso del file della chiave pubblica).

Listato 10.14 Visualizzazione del fingerprint della chiave host.

```
$ cut -d' ' -f1 adbkey.pub|openssl base64 -A -d -a | \
openssl md5 -c|cut -d' ' -f2|tr '[a-z]' '[A-Z]'
69:D4:AC:0D:AF:6B:17:88:BA:6B:C4:BE:0C:F7:75:9A
```

Backup in Android

Android include un framework specifico che consente di eseguire il backup dei dati delle applicazioni nel cloud storage di Google e che supporta il backup completo dei file APK installati, dei dati delle applicazioni e dei file nella memoria esterna su una macchina host connessa tramite USB. Anche se il backup del dispositivo non è esattamente una funzionalità di sicurezza, consente tuttavia di estrarre i dati delle applicazioni dal dispositivo, causando un problema di sicurezza.

Panoramica sul backup in Android

Il framework di backup di Android è stato annunciato pubblicamente in Android 2.2, ma era probabilmente già disponibile internamente nelle versioni precedenti. Il framework consente alle applicazioni di dichiarare componenti speciali, detti *agent di backup*, chiamati dal sistema durante la creazione di un backup per un'applicazione e durante il ripristino dei suoi dati. Anche se il framework disponeva internamente del supporto per i trasporti di backup pluggable, inizialmente l'unico trasporto utilizzabile nella pratica era quello proprietario che salvava i dati delle applicazioni nel cloud storage di Google.

Backup nel cloud

I backup sono associati all'account Google dell'utente; pertanto, quando si installa un'applicazione che dispone di un agent di backup su un nuovo dispositivo, i dati dell'applicazione possono essere ripristinati automaticamente se l'utente ha registrato lo stesso account Google utilizzato al momento della creazione del backup. Il backup e il ripristino sono gestiti dal sistema e in genere non possono essere attivati o controllati dagli utenti (anche se la shell di Android permette di accedere a comandi per sviluppatori che attivano il backup nel cloud). Per impostazione predefinita i backup vengono attivati periodicamente,

mentre il ripristino solo alla prima installazione di un'app su un dispositivo.

Backup locale

Android 4.0 ha aggiunto un nuovo trasporto di backup locale che consente agli utenti di salvare i backup anche in un file sul proprio computer desktop. Il backup locale (chiamato anche backup completo) richiede che il debug ADB sia abilitato e autorizzato, in quanto i dati di backup vengono inviati al computer host utilizzando lo stesso metodo che ADB impiega (con `adb pull`) per trasferire i file del dispositivo a un host.

Il backup completo viene avviato eseguendo il comando `adb backup` in una shell: questo comando avvia un nuovo processo Java sul dispositivo, che effettua il binding con il servizio di sistema `BackupManagerService` e richiede un backup con i parametri specificati per `adb backup`. `BackupManagerService` a sua volta avvia un'activity di conferma come quella mostrata nella Figura 10.16, invitando l'utente ad autorizzare il backup e a specificare, se necessario, una password di crittografia per il backup. Se il dispositivo è già crittografato, l'utente deve inserire la password di crittografia del dispositivo per procedere: questa password sarà utilizzata per crittografare anche il backup, visto che l'uso di una password di crittografia dedicata al backup non è supportato. Il processo di backup completo viene avviato quando l'utente seleziona il pulsante *Back up my data*.

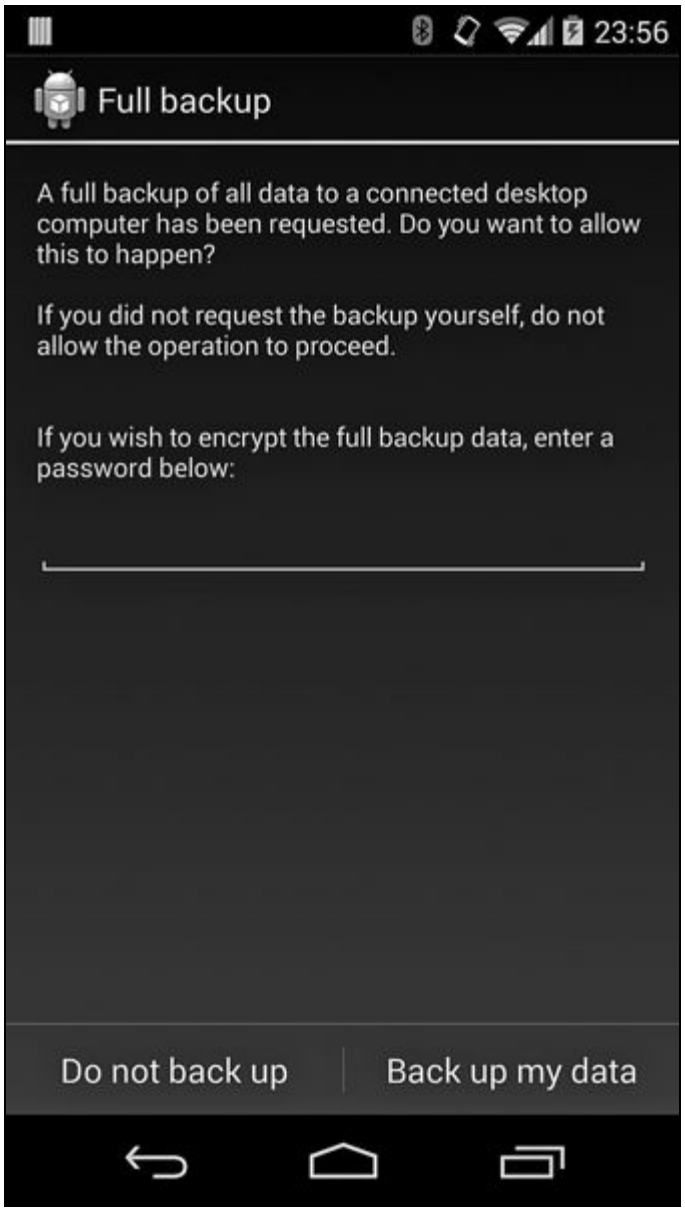

Figura 10.16 Finestra di conferma del backup.

Il backup completo chiama l'agent di backup di ciascun package target al fine di ottenere una copia dei suoi dati. Se non è definito un agent di backup, `BackupManagerService` usa una classe interna `FullBackupAgent` che copia tutti i file del package. Il backup completo rispetta l'attributo `allowBackup` del tag `<application>` nel file `AndroidManifest.xml` del package e non estrae i dati del package se `allowBackup` è impostato su `false`.

Oltre ai dati delle applicazioni, il backup completo può includere i file APK installati dall'utente e quelli delle applicazioni di sistema, nonché il contenuto della memoria esterna, con alcune limitazioni: il backup completo, infatti, non riguarda le app protette (con DRM) e ignora alcune

impostazioni di sistema come i dettagli di connessione agli APN di rete mobile e ai punti di accesso Wi-Fi.

I backup vengono ripristinati utilizzando il comando `adb restore`. La funzione di ripristino di un backup è piuttosto limitata e non consente la selezione di opzioni: è pertanto possibile eseguire solamente un ripristino completo.

Formato di file di backup

I file di backup Android iniziano con poche righe di testo seguite da dati binari; queste righe rappresentano l'header del backup e specificano il formato e i parametri di crittografia (se è stata specificata una password di backup) usati per creare il backup. L'header di un backup non crittografato è riportato nel Listato 10.15.

Listato 10.15 Header di backup non crittografato.

```
ANDROID BACKUP(1)
1(2)
1(3)
none(4)
```

La prima riga **(1)** è l'identificatore di formato del file, la seconda **(2)** è la versione del formato di backup (`1` fino ad Android 4.4.2, 2 per le versioni successive; la versione 2 indica una modifica del metodo di derivazione della chiave, che ora tiene conto dei caratteri multibyte nelle password degli account), la terza **(3)** è un flag di compressione (`1` se compresso), l'ultima **(4)** è l'algoritmo di crittografia utilizzato (`none` o `AES-256`).

I dati di backup effettivi corrispondono a un file tar compresso e facoltativamente crittografato, che include un manifest di backup seguito dall'APK dell'applicazione (se presente) e dai dati dell'app (file, database e preferenze condivise). I dati sono compressi utilizzando l'algoritmo `deflate` e possono essere decompressi utilizzando il comando `zlib` di OpenSSL, come mostrato nel Listato 10.16.

Listato 10.16 Decompressione di un backup Android con OpenSSL.

```
$ dd if=mybackup.ab bs=24 skip=1|openssl zlib -d > mybackup.tar
```

Dopo la decompressione del backup, è possibile visualizzarne o estrarne i contenuti utilizzando il comando standard `tar`, come mostrato nel Listato 10.17.

Listato 10.17 Visualizzazione del contenuto di un backup non compresso con tar.

```
$ tar tvf mybackup.tar
-rw------- 1000/1000      1019 apps/org.myapp/_manifest(1)
-rw-r--r-- 1000/1000   1412208 apps/org.myapp/a/org.myapp-1.apk(2)
-rw-rw---- 10091/10091     231 apps/org.myapp/f/share_history.xml(3)
-rw-rw---- 10091/10091       0 apps/org.myapp/db/myapp.db-journal(4)
-rw-rw---- 10091/10091    5120 apps/org.myapp/db/myapp.db
-rw-rw---- 10091/10091    1110 apps/org.myapp/sp/org.myapp_preferences.xml(5)
```

All'interno del file tar i dati dell'app sono salvati nella directory *apps/*, che contiene una sottodirectory per ogni package sottoposto a backup. La directory di ogni package contiene un file `_manifest` **(1)** nella sua radice, il file APK (se richiesto) in *a/* **(2)**, i file dell'app in *f/* **(3)**, i database in *db/* **(4)** e le preferenze condivise in *sp/* **(5)**. Il manifest contiene il nome del package e il codice di versione dell'app, il codice di versione della piattaforma, un flag che indica se l'archivio contiene l'APK dell'app e il certificato di firma dell'app.

`BackupManagerService` usa queste informazioni durante il ripristino di un'app per verificare se è stata firmata con lo stesso certificato che è attualmente installato. Se i certificati non corrispondono, l'installazione dell'APK viene saltata, tranne per i package di sistema che potrebbero essere firmati con un certificato diverso (di proprietà del produttore) su dispositivi diversi. Inoltre, `BackupManagerService` si aspetta che i file siano nell'ordine mostrato nel Listato 10.17; il ripristino ha esito negativo se l'ordine non viene rispettato. Per esempio, se il manifest afferma che il backup include un APK, `BackupManagerService` proverà a leggere a installare l'APK prima di ripristinare i file dell'app. L'ordine di ripristino è necessario perché non è possibile ripristinare i file di un'app che non è installata. Tuttavia, `BackupManagerService` non cerca l'APK nell'archivio, e se non lo trova subito dopo il manifest ignora tutti gli altri file.

Se l'utente ha richiesto il backup della memoria esterna (passando l'opzione `-shared` ad `adb backup`), l'archivio conterrà anche una directory *shared/* in cui sono presenti i file della memoria esterna.

Crittografia del backup

Se l'utente ha fornito una password di crittografia al momento della richiesta di backup, il file di backup è crittografato con una chiave derivata dalla password. La password è usata per generare una chiave AES a 256 bit utilizzando 10.000 cicli di PBKDF2 con un salt a 512 bit generato in maniera casuale. Questa chiave viene quindi impiegata per crittografare un'altra chiave master AES a 256 bit, sempre generata in maniera casuale, a sua volta utilizzata per crittografare i dati effettivi dell'archivio nella modalità CBC (utilizzando la trasformazione *AES/CBC/PKCS5Padding* `cipher`). Viene inoltre calcolato e salvato nell'header del file di backup un checksum della chiave master. Per generare il checksum, la chiave master raw generata viene convertita in un array di caratteri Java effettuando il cast di ogni byte in `char`; il risultato viene trattato come una stringa di password e passato alla funzione PBKDF2 per generare infine un'altra chiave AES, i cui byte sono impiegati come checksum.

NOTA

Una chiave AES non è altro che una sequenza di byte casuali, pertanto di solito la chiave raw contiene diversi byte che non sono associabili a caratteri stampabili. PKCS#5 non specifica la codifica effettiva di una stringa di password, quindi il metodo di generazione del checksum di crittografia usato in Android produce risultati che dipendono dall'implementazione e dalla versione.

Il checksum viene utilizzato per verificare se la password di decodifica fornita dall'utente è corretta prima di effettuare la decodifica dei dati di backup. Una volta decodificata la chiave master, ne viene calcolato il checksum utilizzando il metodo descritto sopra; tale checksum viene quindi confrontato con quello nell'header dell'archivio. Se i checksum non corrispondono, la password viene considerata errata e il processo di ripristino viene interrotto. Il Listato 10.18 mostra un esempio di header di backup per un archivio crittografato.

Listato 10.18 Header di backup crittografato.

```
ANDROID BACKUP
1
1
AES-256(1)
68404C30DF8CACA5FA004F49BA3A70...(2)
909459ADCA2A60D7C2B117A6F91E3D...(3)
10000(4)
```

```
789B1A01E3B8FA759C6459AF1CF1F0FD (5)
8DC5E483D3893EC7F6AAA56B97A6C2...(6)
```

Qui AES-256 **(1)** è l'algoritmo di crittografia utilizzato; la riga successiva **(2)** contiene il salt della password utente sotto forma di stringa esadecimale, seguito dal salt del checksum della chiave master **(3)**, dal numero di cicli PBKDF2 usati per derivare una chiave **(4)** e dall'IV della chiave utente **(5)**. L'ultima riga **(6)** è il key blob della chiave master, che contiene l'IV di crittografia dei dati dell'archivio, la chiave master effettiva e il suo checksum, tutti crittografati con la chiave derivata dalla password fornita dall'utente. Nel Listato 10.19 è mostrato il formato dettagliato del key blob della chiave master.

Listato 10.19 Formato del key blob della chiave master.

```
byte Niv(1)
byte[Niv] IV(2)
byte Nmk(3)
byte [Nmk] MK(4)
byte Nck(5)
byte [Nck] MKck(6)
```

Il primo campo **(1)** è la lunghezza dell'IV, seguito dal valore IV **(2)**, dalla lunghezza della chiave master (MK) **(3)** e dalla chiave master vera e propria **(4)**. Gli ultimi due campi contengono la lunghezza dell'hash di checksum della chiave master **(5)** e lo stesso hash di checksum della chiave master **(6)**.

Controllo dell'ambito del backup

Il modello di sicurezza di Android garantisce che ogni applicazione sia eseguita all'interno della propria sandbox e che i suoi file non siano accessibili ad altre applicazioni o all'utente del dispositivo, a meno che l'applicazione non consenta esplicitamente l'accesso. Per questo, la maggior parte delle applicazioni non effettua la crittografia dei suoi dati prima di salvarli su disco. Tuttavia, sia gli utenti legittimi sia i criminali che riescono a ottenere la password di sblocco del dispositivo possono facilmente estrarre dati dalle applicazioni utilizzando la funzione di backup completo di Android. Per questo motivo le applicazioni che memorizzano dati sensibili dovrebbero utilizzare la crittografia o fornire

un agent di backup esplicito che limiti i dati esportabili, garantendo così che i dati sensibili non possano essere facilmente estratti da un backup.

Come spiegato nel paragrafo “Informazioni generali sul backup in Android”, se il backup dei dati delle applicazioni non è necessario né desiderabile, le applicazioni possono disabilitarlo completamente impostando il loro attributo `allowBackup` su `false` in `AndroidManifest.xml`, come mostrato nel Listato 10.20.

Listato 10.20 Impedimento del backup dei dati delle applicazioni in AndroidManifest.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="org.example.app"
    android:versionCode="1"
    android:versionName="1.0" >
    --altro codice--
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"
        android:allowBackup="false">
        --altro codice--
    </application>
</manifest>
```

Riepilogo

Android adotta varie misure per proteggere i dati degli utenti e le applicazioni, nonché per garantire l'integrità del sistema operativo. Sui dispositivi di produzione, il bootloader è bloccato e il sistema operativo di recovery consente solo gli aggiornamenti OTA firmati dal produttore del dispositivo, garantendo così che si possano avviare o caricare su un dispositivo solo le build del sistema operativo autorizzate. Se abilitato, il boot verificato basato su dm-verity garantisce che la partizione *system* non venga modificata confrontando il valore hash di ogni blocco del dispositivo con un hash tree attendibile, impedendo l'installazione di programmi dannosi, quali i rootkit, nella partizione *system*. Android può inoltre crittografare la partizione *userdata*, rendendo più difficile l'estrazione dei dati delle applicazioni tramite accesso diretto ai dispositivi di memoria.

Android supporta vari metodi di blocco della schermata e applica il rate limiting ai tentativi di autenticazione non riusciti, creando così un deterrente agli attacchi online contro un dispositivo avviato. Le applicazioni di amministratore del dispositivo possono specificare e imporre il tipo e la complessità del PIN o della password di sblocco. È supportata anche una policy che cancella l'intero contenuto del dispositivo dopo un numero eccessivo di tentativi di autenticazione non riusciti. Il debug USB sicuro richiede che gli host di debug siano autorizzati esplicitamente dall'utente e aggiunti a una whitelist, impedendo così l'estrazione di informazioni tramite USB.

Infine, i backup completi dei dispositivi possono essere crittografati con una chiave derivata da una password fornita dall'utente, rendendo più difficile l'accesso ai dati del dispositivo estratti in un backup. Per ottenere un alto livello di sicurezza dei dispositivi è opportuno che tutte le misure di protezione siano abilitate e configurate di conseguenza.

NFC ed elementi sicuri

Questo capitolo offre una breve introduzione a *Near Field Communication* (NFC) e agli elementi di sicurezza (SE), spiegando come vengono integrati nei dispositivi mobili. Anche se NFC è utilizzabile in molti modi, ci concentreremo sulla sua modalità di emulazione card, usata per fornire un'interfaccia a un SE integrato in un dispositivo mobile. Gli elementi sicuri offrono uno spazio di archiviazione protetta per i dati privati, per esempio le chiavi di autenticazione, e un ambiente di esecuzione sicuro in grado di proteggere il codice critico per la sicurezza. Descriveremo i tipi di SE supportati da Android e presenteremo le API che le applicazioni Android possono utilizzare per la comunicazione con i SE. Infine, vedremo la tecnologia e le applicazioni *Host-based Card Emulation* (HCE) e la relativa implementazione in Android.

Panoramica su NFC

NFC è una tecnologia che permette ai dispositivi vicini tra loro (di solito entro 10 centimetri) di stabilire una comunicazione via radio e di scambiare dati. Non è uno standard univoco, ma è basato su una serie di standard che definiscono le frequenze radio, i protocolli di comunicazione e i formati di scambio dei dati. NFC si fonda sulla tecnologia *Radio-Frequency Identification* (RFID) e opera alla frequenza di 13,56 MHz, permettendo varie velocità di trasmissione dati come 106 kbps, 212 kbps e 424 kbps.

La comunicazione NFC coinvolge due dispositivi: un iniziatore e un target. Nella *modalità attiva*, sia l'iniziatore sia il target dispongono di alimentatori propri e ciascuno può trasmettere un segnale radio per comunicare con l'altra entità; nella *modalità passiva*, il dispositivo target non ha una propria fonte di alimentazione e viene attivato e alimentato dal campo elettromagnetico emesso dall'iniziatore.

Durante la comunicazione nella modalità passiva, l'iniziatore è spesso chiamato *reader* e il target *tag*. Il reader può essere un dispositivo dedicato, oppure può essere incorporato in un dispositivo di uso generale, come un personal computer o un telefono cellulare. I tag assumono varie forme e dimensioni, da semplici adesivi con quantità molto limitate di memoria fino a smart card contactless dotate di CPU integrata.

I dispositivi NFC possono operare in tre diverse modalità: *Reader/Writer* (R/W), *Peer-to-Peer* (P2P) e *Card Emulation* (CE). Nella modalità R/W, un dispositivo agisce come un iniziatore attivo e può leggere e scrivere dati sui tag esterni. Nella modalità P2P, due dispositivi NFC possono scambiarsi attivamente dati utilizzando un protocollo di comunicazione bidirezionale. La modalità CE permette a un dispositivo NFC di emulare un tag o una smart card contactless. Android supporta tutte e tre le modalità con alcune limitazioni. Nel prossimo paragrafo introdurremo l'architettura NFC di Android e mostreremo come utilizzare ciascuna modalità.

Supporto NFC in Android

Il supporto NFC in Android è stato introdotto nella versione 2.3; l'architettura e le funzionalità correlate sono rimaste sostanzialmente immutate fino alla versione 4.4, in cui è stato inserito il supporto per HCE.

L'implementazione NFC di Android risiede nel servizio di sistema `NfcService`, che fa parte dell'applicazione di sistema *Nfc* (package `com.android.nfc`); racchiude le librerie native richieste per gestire ogni controller NFC supportato, implementa il controllo di accesso, il tag discovery e il tag dispatching, e controlla l'emulazione delle card. Android non espone un'API di basso livello per la funzionalità di `NfcService`, ma offre un framework guidato dagli eventi che consente alle applicazioni interessate di registrare gli eventi NFC. Questo approccio guidato dagli eventi è utilizzato in tutte e tre le modalità di funzionamento di NFC.

Modalità Reader/Writer

Le applicazioni Android abilitate per NFC non possono impostare direttamente il dispositivo nella modalità R/W; devono invece dichiarare il tipo di tag a cui sono interessate e lasciare che il sistema di tag dispatching di Android scelga e avvii l'applicazione corrispondente all'individuazione di un tag.

Il sistema di tag dispatching usa una tecnologia specifica (presentata a breve) ed esegue il parsing del contenuto dei tag per decidere per quale applicazione approntare il tag. Utilizza tre azioni di intent per informare le applicazioni del tag individuato: `ACTION_NDEF_DISCOVERED`, `ACTION_TECH_DISCOVERED` e `ACTION_TAG_DISCOVERED`. L'intent `ACTION_NDEF_DISCOVERED` ha la priorità più alta e viene inviato quando Android individua un tag nel formato standard *NFC Data Exchange Format* (NDEF) e contenente un tipo di dati riconosciuto. Il formato NDEF e la sua implementazione per le varie tecnologie di tag sono descritti nella specifica di NFC Forum disponibile

all'indirizzo http://bit.ly/1qATk6d. L'intent ACTION_TECH_DISCOVERED viene inviato quando il tag rilevato non contiene dati NDEF o se il formato dei dati non è riconosciuto dalle applicazioni in grado di gestire la tecnologia dei tag individuata. Se nessuna applicazione può gestire ACTION_NDEF_DISCOVERED o ACTION_TECH_DISCOVERED, NfcService invia l'intent generico ACTION_TAG_DISCOVERED. Gli eventi di tag dispatching vengono recapitati solo alle activity e di conseguenza non possono essere elaborati in background senza interazione con l'utente.

Registrazione per il tag dispatching

Le applicazioni registrano gli eventi NFC utilizzando il sistema di filtraggio degli intent standard, dichiarando gli intent supportati da un'activity abilitata per NFC in AndroidManifest.xml, come mostrato nel Listato 11.1.

Listato 11.1 File manifest di un'applicazione con tecnologia NFC.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.nfc" ...>
    --altro codice--

    <uses-permission android:name="android.permission.NFC" />(1)
    --altro codice--
    <application ...>
        <activity
            android:name=".NfcActivity"(2)
            android:launchMode="singleTop" >
            <intent-filter>
                <action android:name="android.nfc.action.NDEF_DISCOVERED"/>(3)
                <category android:name="android.intent.category.DEFAULT"/>
                <data android:mimeType="text/plain" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.nfc.action.TECH_DISCOVERED" />(4)
            </intent-filter>
            <intent-filter>
                <action android:name="android.nfc.action.TAG_DISCOVERED" />(5)
            </intent-filter>

            <meta-data
                android:name="android.nfc.action.TECH_DISCOVERED"(6)
                android:resource="@xml/filter_nfc" >
            </meta-data>
        </activity>
        --altro codice--
    </application>
</manifest>
```

Come potete vedere nel listato, l'applicazione per prima cosa richiede il permesso android.permission.NFC **(1)**, necessario per accedere al controller NFC, e quindi dichiara un'activity che gestisce gli eventi NFC, NfcActivity

(2). L'activity registra tre filtri intent, uno per ogni evento di individuazione dei tag. L'applicazione dichiara di poter gestire i dati NDEF con il tipo MIME `text/plain` specificando l'attributo `mimeType` del tag `<data>` nel filtro intent `NDEF_DISCOVERED` **(3)**. `NfcActivity` dichiara inoltre di poter gestire l'intent `TECH_DISCOVERED` **(4)**, inviato se il tag esaminato usa una delle tecnologie specificate nel file di risorse XML dei metadati associato **(6)**. Infine, l'applicazione richiede di ricevere una notifica per tutti i tag NFC individuati aggiungendo il filtro intent "catch-all" `TAG_DISCOVERED` **(5)**.

Se vengono rilevate più activity che supportano il tag esaminato, Android mostra una finestra di selezione che consente all'utente di scegliere quale activity dovrebbe gestire il tag. Le applicazioni già in foreground possono "cortocircuitare" questa selezione chiamando il metodo `NfcAdapter.enableForegroundDispatch()`. Tale applicazione ottiene la priorità su tutte le altre applicazioni corrispondenti e riceve automaticamente l'intent NFC quando è in foreground.

Tecnologie dei tag

Una *tecnologia dei tag* è un termine astratto che descrive un tag NFC concreto. La tecnologia dei tag è determinata dal protocollo di comunicazione utilizzato dal tag, dalla sua struttura interna o dalle funzionalità che offre. Per esempio, un tag che usa il protocollo NFC-A (basato su ISO 14443-3A; le versioni ufficiali degli standard ISO possono essere acquistate all'indirizzo web `http://www.iso.org/iso/home/store/catalogue_ics.htm`, mentre le bozze degli standard possono solitamente essere ottenute dal sito web del gruppo che si occupa dello standard) per la comunicazione corrisponde alla tecnologia *NfcA*, mentre un tag contenente dati in formato NDEF corrisponde alla tecnologia *Ndef*, indipendentemente dal protocollo di comunicazione sottostante. Consultate la documentazione di riferimento per la classe `TagTechnology` per un elenco completo di tecnologie dei tag supportate da Android, `http://bit.ly/1F2iB33`.

Un'activity che specifica l'intent `TECH_DISCOVERED` deve fornire un file di risorse XML che specifichi le tecnologie concrete supportate con un elemento `<tech-list>`. Un'activity è considerata corrispondente a un tag se una delle tech list dichiarate è un sottoinsieme delle tecnologie supportate dal tag. Per la corrispondenza con tag diversi è possibile dichiarare più tech list, come mostrato nel Listato 11.2.

Listato 11.2 Dichiarazione delle tecnologie di corrispondenza per le tech list.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <tech-list>(1)
        <tech>android.nfc.tech.IsoDep</tech>
        <tech>android.nfc.tech.NfcA</tech>
    </tech-list>

    <tech-list>(2)
        <tech>android.nfc.tech.NfcF</tech>
    </tech-list>
</resources>
```

Qui la prima tech list **(1)** corrisponde ai tag che forniscono un'interfaccia di comunicazione compatibile con ISO 14443-4 (ISO-DEP) e che sono implementate usando la tecnologia NFC-A (solitamente impiegata dalle smart card contactless NXP); le seconda tech list **(2)** corrisponde tag che utilizzano la tecnologia NFC-F (generalmente le schede Felica). Entrambe le liste sono definite in maniera indipendente, quindi nell'esempio `NfcActivity` (Listato 11.1) riceverà una notifica sia quando viene rilevata una smart card contactless NXP sia quando viene rilevata una card Felica.

Lettura di un tag

Dopo che il sistema di tag dispatching ha selezionato un'activity per la gestione del tag acquisito, viene creato un oggetto intent NFC da passare all'activity selezionata. L'activity può quindi utilizzare l'extra `EXTRA_TAG` per ottenere un oggetto `Tag` che rappresenti il tag acquisito e chiamare i suoi metodi per leggere o scrivere il tag. I tag che contengono dati NDEF forniscono anche l'extra `EXTRA_NDEF_MESSAGES`, che contiene un array di messaggi NDEF analizzati dal tag.

Un oggetto `Tag` concreto che rappresenta la tecnologia dei tag sottostante può essere ottenuto utilizzando il metodo statico `get()` della classe della

tecnologia corrispondente, come mostrato nel Listato 11.3. Se l'oggetto `Tag` non supporta la tecnologia richiesta, il metodo `get()` restituisce `null`.

Listato 11.3 Recupero di un'istanza Tag concreta dall'intent NFC.

```
protected void onNewIntent(Intent intent) {
    setIntent(intent);

    Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    IsoDep isoDep = IsoDep.get(tag);
    if (isoDep != null) {
         isoDep.connect();
         byte[] command = {...};
         byte[] response = isoDep.transceive(command);
         --altro codice--
    }
}
```

Uso della modalità Reader

Oltre al sistema di tag dispatching basato sugli intent, Android 4.4 introduce un nuovo metodo utilizzabile dalle activity per ottenere un oggetto `Tag` live: si tratta della modalità Reader. La modalità Reader garantisce che, mentre l'activity target è in foreground, tutte le altre modalità operative supportate dal controller NFC (come il peer-to-peer e l'emulazione di card) siano disabilitate. Questa modalità è utile durante la scansione di un dispositivo NFC attivo, per esempio un altro dispositivo Android nella modalità di emulazione host-based, che potrebbe attivare la comunicazione point-to-point e togliere il controllo all'attuale activity in foreground.

Le activity possono abilitare la modalità Reader chiamando il metodo `enableReaderMode()` della classe `NfcAdapter`, come mostrato nel Listato 11.4 (`http://bit.ly/1sLBHVY`).

Listato 11.4 Abilitazione della modalità Reader e recupero di un oggetto Tag con ReaderCallback.

```
public class NfcActivity extends Activity implements NfcAdapter.ReaderCallback {
    private NfcAdapter adapter;
    --altro codice--
    @Override
    public void onResume() {
        super.onResume();
        if (adapter != null) {
            adapter.enableReaderMode(this, this, NfcAdapter.FLAG_READER_NFC_A(1)
                    | NfcAdapter.FLAG_READER_SKIP_NDEF_CHECK, null);
        }
    }

    @Override
    public void onTagDiscovered(Tag tag) {(2)
        IsoDep isoDep = IsoDep.get(tag);
        if (isoDep != null) {
             isoDep.connect();
```

```
            byte[] command = {...};
            byte[] response = isoDep.transceive(command);
            --altro codice--
        }
    }
    --altro codice--
}
```

In questo caso, l'activity abilita la modalità Reader quando passa in foreground chiamando il metodo `enableReaderMode()` **(1)** (l'activity dovrebbe disabilitare la modalità Reader utilizzando il metodo `disableReaderMode()` corrispondente quando non è più in foreground) e ottiene un'istanza `Tag` direttamente (senza un intent intermedio) attraverso il callback `onTagDiscovered()` **(2)**. L'oggetto `Tag` viene quindi utilizzato nello stesso modo visto per il dispatching basato sugli intent.

Modalità Peer-to-Peer

Android implementa uno scambio di dati NFC in modalità P2P tra dispositivi che utilizzano la modalità push NDEF proprietaria e i protocolli standard *Simple NDEF Exchange Protocol* (SNEP; NFC Forum, http://bit.ly/1qATk6d). I dispositivi Android possono scambiare un singolo messaggio NDEF con qualsiasi device che supporta questi protocolli, ma la modalità P2P è in genere utilizzata con un altro dispositivo Android per implementare la cosiddetta funzione Android Beam.

Oltre ai messaggi NDEF, Android Beam consente il trasferimento di grandi oggetti dati, quali foto e video, che non possono rientrare in un singolo messaggio NDEF creando una connessione Bluetooth temporanea tra i device. Questo processo è detto *NFC handover* ed è stato aggiunto in Android 4.1.

Lo scambio di messaggi NDEF nella modalità P2P viene abilitato chiamando i metodi `setNdefPushMessage()` o `setNdefPushMessageCallback()` della classe `NfcAdapter` (consultate la guida ufficiale all'API NFC per ottenere maggiori dettagli ed esempi di codice: http://bit.ly/1sV4A26).

Modalità di emulazione delle card

Come affermato nel paragrafo “Informazioni generali su NFC”, la modalità CE consente a un dispositivo Android di emulare una smart card contactless o un tag NFC. Nella modalità CE, il dispositivo riceve i comandi su NFC, li elabora e invia le risposte sempre su NFC. Il componente responsabile dell’elaborazione dei comandi può essere sia un elemento sicuro hardware (come spiegato nel prossimo paragrafo) connesso al controller NFC del dispositivo, sia un’applicazione Android in esecuzione sul dispositivo (nella modalità di emulazione card host-based, o HCE).

Nei prossimi paragrafi parleremo degli elementi sicuri nei dispositivi mobili e delle API Android che le applicazioni possono utilizzare per comunicare con i SE. Vedremo inoltre come Android implementa HCE e come creare un’applicazione che consente l’emulazione delle card.

Elementi sicuri

Un *elemento sicuro* (*secure element*, SE) è un chip di smart card a prova di manomissione in grado di eseguire applicazioni per smart card (dette *applet* o *cardlet*) con un determinato livello di sicurezza e isolamento. Una smart card è fondamentalmente un ambiente di elaborazione minimo su un singolo chip, completo di CPU, ROM, EEPROM, RAM e porta di I/O. Le card recenti comprendono anche coprocessori di crittografia che implementano algoritmi comuni come AES e RSA.

Le smart card utilizzano diverse tecniche per implementare la resistenza alle manomissioni, rendendo piuttosto difficile l'estrazione dei dati tramite disassemblaggio o analisi del chip. Quelle moderne sono pre-programmate con un sistema operativo multi-applicazione che utilizza le funzioni di protezione della memoria dell'hardware per garantire che i dati di ogni applicazione siano disponibili solo all'applicazione stessa. L'installazione dell'applicazione e, facoltativamente, l'accesso alla stessa sono controllati richiedendo l'uso di chiavi crittografiche per ogni operazione.

Il SE può essere integrato nei dispositivi mobili, per esempio *Universal Integrated Circuit Card* (UICC, comunemente note come *schede SIM*), incorporati nel device o connessi a uno slot per schede SD. Se il dispositivo supporta NFC, il SE è solitamente connesso (o incorporato) al controller NFC e diventa così possibile la comunicazione wireless con il SE.

Le smart card sono disponibili dagli anni Settanta e sono ora utilizzate in applicazioni che spaziano dalle tessere telefoniche prepagate ai biglietti dei mezzi pubblici, dalle carte di credito agli archivi delle credenziali per le VPN. Un SE installato in un dispositivo mobile dispone di capability equivalenti o superiori a quelle di una smart card, quindi, almeno teoricamente, può essere usato per qualsiasi applicazione per cui oggi si usano le smart card. Inoltre, poiché un SE può ospitare più applicazioni, ha il potenziale per sostituire tutte le schede oggi usate quotidianamente

dalle persone con un singolo dispositivo. Ancora, visto che il SE può essere controllato dal sistema operativo del device, l'accesso allo stesso può essere limitato richiedendo un'autenticazione supplementare (PIN, passphrase o firma del codice) per la sua abilitazione.

Una delle principali applicazioni dei SE nei dispositivi mobili è l'emulazione delle carte di pagamento contactless: in effetti, la forza che ha trainato la distribuzione dei SE è stato proprio il desiderio di consentire i pagamenti mobili. Oltre alle applicazioni finanziarie, i SE mobili possono essere usati per emulare altre carte contactless ampiamente diffuse, come i badge di accesso, le tessere fedeltà e così via.

I SE mobili possono inoltre essere utilizzati per migliorare la sicurezza delle app che hanno a che fare con algoritmi o informazioni sensibili: la parte critica per la sicurezza dell'app, come l'archivio delle credenziali o la verifica della licenza, può essere implementata nel SE per garantire che sia davvero difficile effettuare il reverse engineering e l'estrazione delle informazioni. Altre app che possono trarre vantaggio dall'implementazione nel SE sono i generatori di password monouso (OTP, *One Time Password*) e naturalmente gli archivi delle credenziali (per chiavi segrete condivise o chiavi private in una PKI).

Anche se è possibile implementare app abilitate per SE con gli strumenti e le tecnologie standard, il loro uso negli attuali dispositivi commerciali Android non è così semplice: ne parleremo nei dettagli nel paragrafo "Ambiente di esecuzione SE di Android". Prima, però, esaminiamo i tipi di SE disponibili sui device mobili e il livello di supporto possibile in Android.

Fattori di forma SE nei dispositivi mobili

La Figura 11.1 mostra un diagramma a blocchi semplificato dei componenti di un dispositivo Android, illustrando la relazione con il supporto per SE (compresi i SE incorporati, o eSE, e UICC) e NFC. Faremo riferiamo ai componenti di questo diagramma nella successiva descrizione degli elementi sicuri e dell'emulazione delle card host-based nel resto di questo capitolo.

Nei paragrafi successivi riesamineremo brevemente i tipi di SE disponibili nei dispositivi Android, la loro relazione con gli altri componenti del dispositivo e i metodi usati dal sistema operativo per comunicare con ogni tipo di SE.

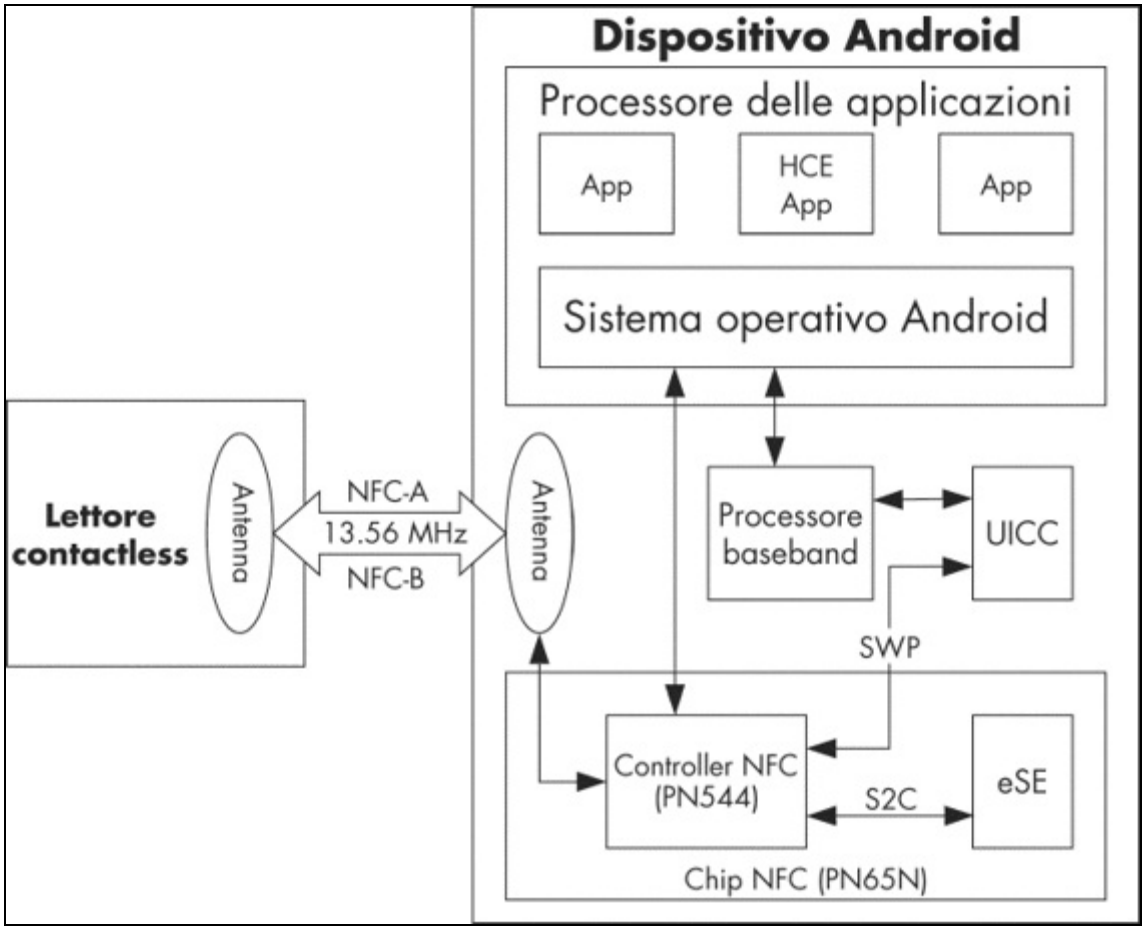

Figura 11.1 Componenti NFC e SE di Android.

UICC

La maggior parte dei dispositivi mobili di oggi dispone di qualche tipo di UICC. Anche se gli UICC sono smart card in grado di ospitare le applicazioni, dal momento che lo UICC è stato tradizionalmente connesso solo al processore baseband (non al processore applicativo che esegue il sistema operativo del dispositivo principale), non è possibile accedervi direttamente da Android. Tutta la comunicazione passa attraverso lo strato *Radio Interface Layer* (RIL), che non è altro che un'interfaccia IPC proprietaria alla baseband.

La comunicazione con il SE UICC avviene utilizzando comandi AT estesi (`AT+CCHO`, `AT+CCHC`, `AT+CGLA` come definito da 3GPP TS 27.007,

http://bit.ly/1xURa6r), che il gestore di telefonia Android attuale non supporta. Il progetto SEEK for Android (http://bit.ly/1sWAPyT) offre patch per implementare i comandi necessari, consentendo la comunicazione con lo UICC tramite l'API SmartCard, un'implementazione di riferimento della specifica API OpenMobile SIMalliance (di cui parleremo tra poco nel paragrafo "Uso dell'API OpenMobile"; http://bit.ly/1sWAPyT). In ogni caso, come quasi tutti i componenti che comunicano direttamente con l'hardware in Android, RIL è costituito da una sezione open source (*rild*) e da una libreria proprietaria (*libXXX-ril.so*). Per supportare la comunicazione con l'elemento sicuro UICC, è quindi necessario aggiungere il supporto sia a *rild* sia alla libreria sottostante. La scelta di aggiungere tale supporto è lasciata ai vendor dell'hardware.

A oggi, l'API SmartCard non è stata integrata in Android mainline (sebbene il source tree AOSP includa una directory vuota *packages/apps/SmartCardService/*); tuttavia, i dispositivi Android dei vendor più importanti vengono forniti con un'implementazione dell'API SmartCard che consente la comunicazione tra lo UICC e le applicazioni di terze parti (con varie limitazioni di accesso).

Single Wire Protocol (SWP) offre un mezzo alternativo per utilizzare lo UICC come un SE: SWP è usato per connettere lo UICC a un controller NFC, permettendo a tale controller di esporre lo UICC ai lettori esterni nella modalità di emulazione delle card. I controller NFC integrati nei dispositivi Nexus più recenti (come Broadcom BCM20793M in Nexus 5) supportano SWP, anche se la funzionalità è disabilitata per impostazione predefinita (per abilitarla è sufficiente cambiare il file di configurazione della libreria *libnfc-brcm* in Nexus 5). Un'API standard per il passaggio tra lo UICC, il SE incorporato (se disponibile) e HCE nella modalità di emulazione card non è ancora stata esposta, ma la funzionalità di routing "off-host" disponibile in Android 4.4 può, in linea teorica, instradare comandi allo UICC (leggete il paragrafo "Routing APDU" per i dettagli).

SE basati su microSD

Un altro fattore di forma dei SE è la scheda *Advanced Security SD* (https://www.sdcard.org/developers/overview/ASSD/), in pratica una scheda SD con un chip SE integrato. Quando viene connessa a un dispositivo Android dotato di slot per schede SD, che esegue una versione di Android con le patch SEEK, è possibile accedere al SE tramite l'API SmartCard. Tuttavia, i dispositivi Android con tale slot sono sempre più rare, quindi è improbabile che il supporto Android di ASSD entri a far parte del sistema operativo mainstream. Inoltre, anche quando disponibili, le versioni recenti di Android trattano le schede SD come dispositivi di archiviazione secondari e consentono l'accesso alle stesse solo tramite un'API restrittiva di altissimo livello.

SE incorporati

Un *SE incorporato* (eSE, *embedded SE*) non è un dispositivo distinto ma di solito è integrato nel controller NFC e ospitato nello stesso enclosure. Un esempio di eSE è il chip PN65N di NXP, che unisce un controller radio NFC PN544 con il SE P5CN072 (parte della serie SmartMX).

Il primo dispositivo Android mainstream a disporre di un SE incorporato è stato il Nexus S, che ha introdotto anche il supporto NFC in Android ed è stato costruito utilizzando il controller PN65N. Anche i suoi successori, il Galaxy Nexus e il Nexus 4, sono stati dotati di un eSE. Tuttavia, i dispositivi a marchio Google più recenti, come il Nexus 5 e il Nexus 7 (2013), hanno deprecato l'eSE in favore della Host-based Card Emulation e non includono un eSE.

Il SE incorporato è connesso al controller NFC attraverso una connessione *SignalIn/SignalOut* (S2C), standardizzata come *NFC Wired Interface* (NFC-WI; ECMA International, http://bit.ly/1occVyc), e dispone di tre modalità di funzionamento: off, wired e virtuale. Nella modalità off non avviene alcuna comunicazione con il SE; nella modalità wired, il SE è visibile al sistema operativo Android come se fosse una smart card contactless connessa al lettore NFC; nella modalità virtuale, il SE è visibile ai lettori esterni come se lo smartphone fosse una smart card

contactless. Queste modalità si escludono reciprocamente, quindi possiamo comunicare con il SE tramite l'interfaccia contactless (cioè da un lettore esterno) oppure attraverso l'interfaccia wired (cioè da un'applicazione Android). Nel prossimo paragrafo vedremo come usare la modalità wired per comunicare con l'eSE da un'app Android.

Accesso ai SE incorporati

Attualmente nessuna API dell'SDK Android pubblico permette la comunicazione con il SE incorporato, ma le versioni recenti di Android includono una libreria opzionale, chiamata *nfc_extras*, che offre un'interfaccia stabile all'eSE. In questo paragrafo viene mostrato come configurare Android per consentire a determinate applicazioni di Android di accedere all'eSE, nonché come utilizzare la libreria *nfc_extras*.

L'emulazione delle card e di conseguenza le API interne per l'accesso al SE incorporato sono state introdotte in Android 2.3.4 (la versione che ha presentato Google Wallet). Queste API sono nascoste alle applicazioni SDK e per utilizzarle sono necessari permessi di firma del sistema (`WRITE_SECURE_SETTINGS` o `NFCEE_ADMIN`) in Android 2.3.4 e nelle successive release 2.3.x, nonché nella release iniziale di Android 4.0 (API livello 14). Un permesso di firma è alquanto restrittivo, perché permette solo alle entità che controllano le chiavi di firma della piattaforma di distribuire le app che possono utilizzare l'eSE.

Android 4.0.4 (API livello 15) ha eliminato questa limitazione sostituendo il permesso di firma con la whitelist dei certificati di firma a livello del sistema operativo. Per quanto sia ancora necessario modificare i file del sistema operativo core, e di conseguenza occorra la cooperazione del vendor, non è necessario firmare le applicazioni SE con la chiave del vendor, semplificando così notevolmente la distribuzione. Inoltre, poiché la whitelist è mantenuta in un file, è possibile aggiornarla facilmente con un OTA per introdurre il supporto per altre applicazioni SE.

Concedere l'accesso ai SE

La nuova strategia di controllo di accesso basata sul whitelisting è implementata dalla classe `NfceeAccessControl` e applicata dal servizio `NfcService` di sistema. La classe `NfceeAccessControl` legge la whitelist da `/etc/nfcee_access.xml`, un file XML che archivia un elenco di certificati di firma e nomi di package autorizzati ad accedere all'eSE. L'accesso può essere concesso sia a tutte le applicazioni firmate dalla chiave privata di un certificato specifico (se non viene specificato alcun nome di package), sia a un singolo package (app). Il Listato 11.5 mostra l'aspetto del contenuto del file `nfcee_access.xml`.

Listato 11.5 Contenuto del file nfcee_access.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <signer android:signature="308204a830820390a003020102020900b399...">(1)
        <package android:name="com.example.nfc">(2)
        </package>
    </signer>
</resources>
```

Questa configurazione consente al package `com.example.nfc` **(2)** di accedere al SE se è firmato con il certificato di firma specificato **(1)**. Sui dispositivi di produzione questo file solitamente contiene solo il certificato di firma dell'app Google Wallet, limitando così l'accesso eSE a Google Wallet.

NOTA

Ad aprile 2014 Google Wallet era supportato solo su Android 4.4 e versioni successive e utilizzava HCE anziché l'eSE.

Dopo aver aggiunto il certificato di firma di un'applicazione a `nfcee_access.xml`, non servono permessi diversi dal permesso NFC standard per accedere all'eSE. Oltre al whitelisting del certificato di firma dell'app, per abilitare l'accesso all'eSE è necessario aggiungere esplicitamente la libreria *nfc_extras* al manifest dell'app e contrassegnarla come obbligatoria con il tag `<uses-library>` (la libreria è opzionale, quindi non viene caricata per impostazione predefinita), come mostrato nel Listato 11.6 al punto **(1)**.

Listato 11.6 Aggiunta della libreria nfc_extras ad AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.nfc" ...>
    --altro codice--
    <uses-permission android:name="android.permission.NFC" />
    <application ...>
```

```
        --altro codice--
        <uses-library
            android:name="com.android.nfc_extras"(1)
            android:required="true" />
    </application>
</manifest>
```

Uso dell'API NfcExecutionEnvironment

L'API di accesso eSE di Android non è basata su un'API di comunicazione smart card standard, come JSR 177 (https://jcp.org/en/jsr/detail?id=177) o l'API OpenMobile, ma offre un'interfaccia di comunicazione basilare implementata nella classe `NfcExecutionEnvironment`. La classe contiene solo tre metodi pubblici, mostrati nel Listato 11.7.

Listato 11.7 API NfcExecutionEnvironment.

```
public class NfcExecutionEnvironment {
    public void open() throws EeIOException {...}

    public void close() throws IOException {...}

    public byte[] transceive(byte[] in) throws IOException {...}
}
```

Questa semplice interfaccia è sufficiente per comunicare con il SE, ma per utilizzarla è necessario prima ottenere un'istanza della classe `NfcExecutionEnvironment`: l'istanza può essere ottenuta dalla classe `NfcAdapterExtras`, a cui si accede con il metodo statico `get()` come mostrato nel Listato 11.8.

Listato 11.8 Uso dell'API NfcExecutionEnvironment.

```
NfcAdapterExtras adapterExtras =
     NfcAdapterExtras.get(NfcAdapter.getDefaultAdapter(context));(1)
NfcExecutionEnvironment nfceEe =
     adapterExtras.getEmbeddedExecutionEnvironment();(2)
nfcEe.open();(3)
byte[] emptySelectCmd = { 0x00, (byte) 0xa4, 0x04, 0x00, 0x00 };
byte[] response = nfcEe.transceive(emptySelectCmd);(4)
nfcEe.close();(5)
```

Qui recuperiamo per prima cosa un'istanza di `NfcAdapterExtras` **(1)**, quindi chiamiamo il suo metodo `getEmbeddedExecutionEnvironment()` per ottenere un'interfaccia all'eSE **(2)**. Per comunicare con l'eSE apriamo una connessione **(3)** e utilizziamo il metodo `transceive()` per inviare un comando e ottenere una risposta **(4)**. Infine, chiudiamo la connessione con il metodo `close()` **(5)**.

Broadcast correlati agli eSE

Un'app abilitata per i SE deve ricevere una notifica degli eventi NFC, quali il rilevamento di un campo RF, e degli eventi pertinenti agli eSE e alle applet installati su tali eSE, come la selezione di un'applet dall'interfaccia NFC, per cambiare stato di conseguenza. La diffusione di tali eventi alle applicazioni dannose può portare alla fuga di informazioni sensibili e ad attacchi Denial of Service, pertanto l'accesso agli eventi relativi agli eSE deve essere limitato alle applicazioni attendibili.

In Android, gli eventi globali sono implementati utilizzando i broadcast e le applicazioni possono creare e registrare broadcast receiver che ricevono i broadcast a cui l'app è interessata. L'accesso ai broadcast legati agli eSE può essere controllato con permessi basati su firme Android standard, ma questa strategia presenta uno svantaggio: visto che solo le app firmate con il certificato della piattaforma possono ricevere eventi eSE, le uniche app abilitate per i SE sono quelle create dal produttore del dispositivo o dal gestore della rete mobile (MNO, *Mobile Network Operator*). Per evitare questa limitazione, Android utilizza lo stesso meccanismo adottato per controllare l'accesso agli eSE, vale a dire il whitelisting dei certificati delle applicazioni, al fine di controllare l'ambito delle applicazioni che possono ricevere broadcast legati agli eSE. Qualunque applicazione il cui certificato di firma (e, facoltativamente, nome del package) sia registrato in `nfcee_access.xml` può ricevere broadcast relativi agli eSE registrando un receiver come quello mostrato nel Listato 11.9.

Listato 11.9 Dichiarazione di un broadcast receiver per gli eventi correlati a eSE in AndroidManifest.xml.

```
<receiver android:name="com.example.nfc.SEReceiver" >
  <intent-filter>
   <action android:name="com.android.nfc_extras.action.RF_FIELD_ON_DETECTED" />(1)
   <action android:name="com.android.nfc_extras.action.RF_FIELD_OFF_DETECTED" />(2)
   <action android:name="com.android.nfc_extras.action.APDU_RECEIVED" />(3)
   <action android:name="com.android.nfc_extras.action.AID_SELECTED" />(4)
   <action android:name="com.android.nfc_extras.action.MIFARE_ACCESS_DETECTED" />(5)
   <action android:name="com.android.nfc_extras.action.EMV_CARD_REMOVAL" />(6)
   <action android:name="com.android.nfc.action.INTERNAL_TARGET_DESELECTED" />(7)
   <action android:name="android.intent.action.MASTER_CLEAR_NOTIFICATION" />(8)
  </intent-filter>
</receiver>
```

Come potete osservare, Android propone notifiche per gli eventi di comunicazione di livello inferiore, come il rilevamento di campi `RF` **(12)**,

la ricezione APDU **(3)** e la selezione di applet **(4)**, ma anche per eventi di livello superiore, come l'accesso ai settori MIFARE **(5)** e la rimozione di card EMV **(6)**. Gli APDU (*Application Protocol Data Unit*) sono gli elementi fondamentali dei protocolli smart card; fate riferimento a "Protocolli di comunicazione SE" più avanti in questo capitolo. Il broadcast APDU_RECEIVED non è implementato, perché in pratica il controller NFC instrada gli APDU in entrata direttamente all'eSE, rendendoli invisibili al sistema operativo. Le app abilitate per i SE registrano questi broadcast per poter cambiare il loro stato interno o avviare un'activity correlata al verificarsi di ogni evento (per esempio per avviare un'activity di inserimento del PIN quando si seleziona un'applet EMV). Il broadcast INTERNAL_TARGET_DESELECTED **(7)** viene inviato quando l'emulazione delle card è disattivata, mentre il broadcast MASTER_CLEAR_NOTIFICATION **(8)** viene inviato alla cancellazione del contenuto dell'eSE (le versioni precedenti ad HCE di Google Wallet offrivano agli utenti la possibilità di cancellare l'eSE da remoto in caso di furto o perdita del dispositivo).

Ambiente di esecuzione SE di Android

Il SE di Android è essenzialmente una smart card in un package differente, quindi può essere utilizzata la maggior parte degli standard e dei protocolli originariamente sviluppati per le smart card. Riesaminiamo brevemente i più importanti.

Le smart card sono per tradizione orientate al file system e il ruolo principale del loro sistema operativo è quello di gestire l'accesso ai file e applicare i permessi di accesso. Le schede più nuove supportano una macchina virtuale in esecuzione sopra il sistema operativo nativo, che consente l'esecuzione di applicazioni indipendenti dalla piattaforma, dette *applet*, che usano una libreria di runtime ben definita per implementare le loro funzionalità. Per quando esistano diverse implementazioni di questo paradigma, a oggi la più popolare è *Java Card Runtime Environment* (JCRE). Le applet sono implementate in una versione limitata del linguaggio Java e utilizzano una libreria di runtime limitata,

che offre classi di base per l'I/O, il parsing dei messaggi e le operazioni di crittografia. La specifica JCRE (http://bit.ly/1vgP69s), pur definendo interamente l'ambiente di runtime delle applet, non indica come caricare, inizializzare ed eliminare le applet sulle schede fisiche (gli strumenti sono disponibili solo per l'emulatore JCRE).

Visto che tra le applicazioni principali delle smart card vi sono vari servizi di pagamento, il processo di caricamento e inizializzazione delle applicazioni (spesso chiamato *personalizzazione della card*) deve essere controllato, e solo le entità autorizzate dovrebbero essere in grado di modificare lo stato della card e delle applicazioni installate. Visa in origine ha sviluppato una specifica per la gestione sicura delle applet, chiamata Open Platform, oggi gestita e sviluppata da GlobalPlatform (GP) con il nome di GlobalPlatform Card Specification (http://www.globalplatform.org/specificationscard.asp). L'essenza di questa specifica è che ogni card conforme a GP dispone di un componente obbligatorio *Issuer Security Domain* (ISD, chiamato *Card Manager* in termini meno formali) che offre un'interfaccia ben definita per la gestione del ciclo di vita della card e delle applicazioni. L'esecuzione di operazioni ISD richiede l'autenticazione mediante chiavi di crittografia salvate sulla card; di conseguenza, solo un'entità che conosce queste chiavi può cambiare lo stato della scheda (tra `OP_READY`, `INITIALIZED`, `SECURED`, `CARD_LOCKED` e `TERMINATED`) o gestire le applet. Inoltre, la specifica delle card GP definisce vari protocolli di comunicazione sicura (detti *Secure Channel*) che offrono l'autenticazione, la riservatezza e l'integrità dei messaggi durante la comunicazione con la card.

Protocolli di comunicazione SE

Come abbiamo visto nel paragrafo "Uso dell'API NfcExecutionEnvironment", l'interfaccia di Android per la comunicazione con i SE è il metodo `byte[] transceive(byte[] command)` della classe `NfcExecutionEnvironment`. I messaggi scambiati utilizzando questa API sono in pratica APDU e la loro struttura è definita nella norma *ISO/IEC 7816-4: Organization, security and commands for interchange* (un sunto di ISO

7816 e di altri standard relativi alle smart card è disponibile sul sito web di CardWerk, http://www.cardwerk.com/smartcards/smartcard_standards.aspx). Il lettore (detto anche *Card Acceptance Device* o CAD) invia alla card APDU di comando (a volte detti C-APDU) contenenti un header obbligatorio di quattro byte con una classe di comando (CLA), istruzioni (INS) e due parametri (P1 e P2). Seguono la lunghezza opzionale dei dati di comando (Lc), i dati effettivi e infine il numero massimo di byte di risposta previsti, se presente (Le). La card restituisce un APDU di risposta (R-APDU) con una status word obbligatoria (SW, composta da due byte: SW1 e SW2) e dati di risposta opzionali.

Storicamente, i dati degli APDU di comando erano limitati a 255 byte (lunghezza totale dell'APDU 261 byte) e i dati degli APDU di risposta a 256 byte (lunghezza totale dell'APDU 258 byte). Le card e i lettori più recenti supportano APDU estesi con lunghezza massima dei dati pari a 65.536 byte; tuttavia, gli APDU estesi non sono sempre utilizzabili, più che altro per ragioni di compatibilità. La comunicazione di livello inferiore tra il lettore e la card viene eseguita da uno di diversi protocolli di trasmissione, di cui i più diffusi sono T=0 (orientato ai byte) e T=1 (orientato ai blocchi). Entrambi sono definiti in *ISO 7816-3: Cards with contacts – Electrical interface and transmission protocols*. Lo scambio di APDU non è completamente indipendente dal protocollo, perché T=0 non può inviare direttamente i dati di risposta, ma solo avvertire il lettore del numero di byte disponibili. Per recuperare i dati di risposta devono essere inviati altri APDU di comando (GET RESPONSE).

Gli standard originali ISO 7816 sono stati sviluppati per le card "a contatto", ma lo stesso modello di comunicazione basato su APDU è utilizzato anche per le card contactless: è organizzato sopra il protocollo di trasmissione wireless definito da ISO/IEC 14443-4, che si comporta in maniera molto simile a T=1 per le card a contatto.

Interrogazione dell'ambiente di esecuzione eSE

Come abbiamo visto nel paragrafo “SE incorporati”, l’eSE di Galaxy Nexus è un chip della serie SmartMX di NXP che esegue un sistema operativo compatibile con Java Card ed è fornito con un ISD conforme a GlobalPlatform. L’ISD è configurato per richiedere l’autenticazione per la maggior parte delle operazioni di gestione della card, e le chiavi di autenticazione sono naturalmente non disponibili al pubblico. Inoltre, un certo numero di tentativi consecutivi di autenticazione non riusciti (solitamente 10) causa il blocco dell’ISD e rende impossibile l’installazione o la rimozione di applet, impedendo gli attacchi di forza bruta sulle chiavi di autenticazione. Ad ogni modo, l’ISD fornisce alcune informazioni su se stesso e sull’ambiente di runtime della card senza che sia necessaria l’autenticazione, affinché i client possano adeguare il loro comportamento in maniera dinamica e risultare compatibili con diverse card.

Sia Java Card sia GlobalPlatform definiscono un ambiente multi-applicazione, pertanto ogni applicazione necessita di un identificatore univoco chiamato *Application Identifier* (AID). L’AID è costituito da un *Registered Application Provider Identifier* (RID, detto anche *Resource Identifier*) di 5 byte e da una *Proprietary Identifier eXtension* (PIX) che può raggiungere una lunghezza di 11 byte. La lunghezza di un AID può quindi essere compresa tra 5 e 16 byte. Prima di poter inviare comandi a una particolare applet, è necessario attivarla, o selezionarla, con il comando `SELECT` (`CLA`=00, `INS`=A4) e il suo AID. Come tutte le applicazioni, l’ISD è identificato anche da un AID che varia in base al produttore della card e all’implementazione GP. Per scoprire l’AID dell’ISD potete inviare un comando `SELECT` vuoto, che provocherà la selezione dell’ISD e restituirà informazioni sulla card e sulla configurazione dell’ISD. Un comando `SELECT` vuoto rappresenta una selezione senza indicazione dell’AID, quindi l’APDU di comando `SELECT` diventa `00 A4 04 00 00`. Se inviamo questo comando con il metodo `transceive()` della classe `NfcExecutionEnvironment` (Listato 11.8, punto **(4)**), la risposta restituita è simile a quella nel Listato 11.10 al punto **(2)** (**(1)** è il comando `SELECT`).

Listato 11.10 Risposta eSE di Galaxy Nexus a un comando SELECT vuoto.

```
--> 00A4040000(1)
<-- 6F658408A000000003000000A5599F6501FF9F6E06479100783300734A06072A86488
6FC6B01600C060A2A864886FC6B02020101630906072A864886FC6B03640B06092A86488
6FC6B040215650B06092B851086486402010366 0C060A2B060104012A026E0102 9000(2)
```

La risposta include uno stato di operazione riuscita (0x9000) e una lunga stringa di byte. Il formato di questi dati è definito in "APDU Command Reference", Capitolo 9 della *GlobalPlatform Card Specification*, e come la maggior parte degli elementi nel mondo delle smart card è nel formato *tag-lunghezza-valore* (TLV). In TLV, ogni unità di dati è descritta da un tag univoco seguito dalla sua lunghezza in byte e dai dati effettivi. La maggior parte delle strutture è ricorsiva, quindi i dati possono ospitare un'altra struttura TLV, che a sua volta ne racchiude un'altra e così via. La struttura del Listato 11.10 è chiamata *File Control Information* (FCI) e in questo caso racchiude una struttura Security Domain Management Data che descrive l'ISD. Dopo il parsing, FCI assume l'aspetto mostrato nel Listato 11.11.

Listato 11.11 FCI dell'ISD sottoposto a parsing sull'eSE in Galaxy Nexus.

```
SD FCI: Security Domain FCI
   AID: a0 00 00 00 03 00 00 00(1)
    RID: a0 00 00 00 03 (Visa International [US])
    PIX: 00 00 00

   Data field max length: 255
   Application prod. life cycle data: 479100783300
   Tag allocation authority (OID): globalPlatform 01
   Card management type and version (OID): globalPlatform 02020101
   Card identification scheme (OID): globalPlatform 03
   Global Platform version: 2.1.1(2)
   Secure channel version: SC02 (options: 15)(3)
   Card config details: 06092B8510864864020103(4)
   Card/chip details: 060A2B060104012A026E0102(5)
```

Qui l'`AID` dell'ISD è `a0 00 00 00 03 00 00 00` **(1)**, la versione dell'implementazione GlobalPlatform è la `2.1.1` **(2)** e il protocollo Secure Channel supportato è `SC02` **(3)**; gli ultimi due campi della struttura contengono alcuni dati proprietari sulla configurazione della card (**(4)** e **(5)**). L'unico altro comando GP che non richiede l'autenticazione è `GET DATA`, utilizzabile per restituire altri dati sulla configurazione dell'ISD.

UICC come elementi sicuri

Nel paragrafo "Fattori di forma SE nei dispositivi mobili" abbiamo visto che lo UICC in un dispositivo mobile può essere utilizzato come SE

di uso generico quando vi si accede con l'API OpenMobile o con un'interfaccia di programmazione simile. In questo paragrafo è disponibile una breve introduzione agli UICC e alle applicazioni che generalmente ospitano; viene poi mostrato come accedervi dall'API OpenMobile.

Schede SIM e UICC

Il predecessore dello UICC è la SIM e colloquialmente gli UICC sono spesso chiamati ancora "schede SIM". *SIM* è l'abbreviazione di *Subscriber Identity Module* e fa riferimento a una smart card che memorizza in modo sicuro l'identificativo dell'abbonato e la chiave associata usata per identificare e autenticare un dispositivo in una rete mobile. Le SIM inizialmente erano usate sulle reti GSM; gli standard GSM furono poi estesi per supportare 3G e LTE. Visto che le SIM sono smart card, sono conformi alle norme ISO-7816 per le caratteristiche fisiche e l'interfaccia elettrica. Le prime SIM avevano le stesse dimensioni delle smart card "normali" (Full-size, FF), ma le più popolari oggi sono le Mini-SIM (2FF), le Micro-SIM (3FF) e le Nano-SIM (4FF), queste ultime introdotte nel 2012 e pronte a conquistare il mercato.

È ovvio che non tutte le smart card che possono essere inserite nello slot della SIM sono utilizzabili in un dispositivo mobile. Dobbiamo quindi chiederci quali caratteristiche "trasformano" una smart card in una SIM. Tecnicamente, sono la conformità agli standard di comunicazione mobile come 3GPP TS 11.11 e la certificazione di SIMalliance; in pratica, è la capacità di eseguire un'applicazione che permette di comunicare con il telefono (si parla di *Mobile Equipment* o *Mobile Station* negli standard correlati) e di connettersi a una rete mobile. Anche se lo standard GSM originale non distingueva tra la smart card fisica e il software richiesto per connettersi alla rete mobile, con l'introduzione degli standard 3G è stata applicata una distinzione chiara ed evidente. La smart card fisica è chiamata *Universal Integrated Circuit Card* (UICC); sono state quindi definite diverse applicazioni di rete mobile eseguibili sullo UICC, come GSM, CSIM, USIM, ISIM e così via. Uno UICC può ospitare e gestire

più applicazioni di rete (da qui il nome *universale*) e di conseguenza può essere utilizzato per connettersi a reti diverse. Anche se le funzionalità specifiche delle applicazioni di rete dipendono dalla rete mobile, le funzionalità di base sono piuttosto simili: memorizzare in modo sicuro i parametri di rete, identificarsi presso la rete, autenticare l'utente (facoltativo) e memorizzare i dati dell'utente.

Applicazioni UICC

Prendiamo come esempio GSM e riesaminiamo brevemente il funzionamento di un'applicazione di rete. Per GSM, i parametri di rete principali sono l'identità di rete (*International Mobile Subscriber Identity*, IMSI, associato alla SIM), il numero di telefono (MSISDN, usato per il routing delle chiamate e modificabile) e una chiave di autenticazione di rete condivisa *Ki*. Per la connessione alla rete il telefono deve autenticarsi e negoziare una chiave di sessione. Entrambe le chiavi di autenticazione e di sessione vengono derivate usando *Ki*, nota anche alla rete e ricercata da IMSI. Il telefono invia una richiesta di connessione contenente il suo IMSI, che la rete utilizza per trovare la *Ki* corrispondente. La rete usa quindi *Ki* per generare un challenge (*RAND*), la risposta prevista al challenge (*SRES*) e una chiave di sessione *Kc*. Una volta generati questi parametri, la rete invia *RAND* al telefono e l'applicazione GSM in esecuzione sulla SIM entra in gioco: il dispositivo mobile passa *RAND* alla SIM e questa genera i propri *SRES* e *Kc*. *SRES* viene inviato alla rete e, se corrisponde al valore previsto, viene stabilita una comunicazione crittografata utilizzando la chiave di sessione *Kc*.

Come potete osservare, la sicurezza di questo protocollo si affida solamente alla segretezza della chiave *Ki*. Visto che tutte le operazioni che coinvolgono *Ki* sono implementate all'interno della SIM e che la chiave non viene mai a contatto diretto con il telefono o la rete, lo schema risulta ragionevolmente sicuro. Naturalmente, la sicurezza dipende anche dagli algoritmi di crittografia usati, e i punti deboli che consentono la decodifica delle chiamate GSM intercettate mediante hardware pronto

all'uso sono stati riscontrati nelle versioni originali della cifratura a flussi A5/1 (inizialmente segreto).

In Android, l'autenticazione di rete è implementata dal processore baseband (ulteriori informazioni nel paragrafo "Accesso allo UICC") e non è mai direttamente visibile al sistema operativo principale.

Implementazione e installazione di applicazioni UICC

Abbiamo visto che gli UICC devono eseguire le applicazioni; vediamo ora come sono implementate e installate queste applicazioni. Le prime smart card erano basate su un modello file system, dove i file (detti *file elementari*, o EF) e le directory (dette *file dedicati*, o DF) utilizzavano come nome un identificatore a due byte. Lo sviluppo di "un'applicazione" richiedeva quindi di selezionare un ID per il DF che ospitava i file dell'applicazione (detto ADF) e di specificare i formati e i nomi degli EF che memorizzavano i dati. Per esempio, l'applicazione GSM fa parte dell'ADF *7F20*, mentre l'ADF USIM ospita `EF_imsi`, `EF_keys`, `EF_sms` e altri file necessari.

Quasi tutti gli UICC in uso oggi sono basati sulla tecnologia Java Card e implementano le specifiche delle card GlobalPlatform, pertanto tutte le applicazioni di rete sono implementate come applet Java Card ed emulano la struttura basata su file legacy per la compatibilità con le versioni precedenti. Le applet sono installate in base alle specifiche GlobalPlatform, autenticando l'ISD e generando comandi `LOAD` e `INSTALL`.

Una funzionalità di gestione delle applicazioni specifica per le SIM è il supporto per gli aggiornamenti OTA tramite SMS binari: questa funzionalità non è utilizzata da tutti i gestori telefonici, ma permette loro di installare in remoto applet sulle SIM da loro rilasciate. OTA viene implementato racchiudendo comandi per la scheda (APDU) all'interno di T-PDU (*Transport Protocol Data Unit*) SMS, che il telefono inoltra allo UICC. Nella maggior parte degli UICC questo è l'unico modo per caricare applet sulla scheda, anche durante la personalizzazione iniziale.

Il caso di utilizzo più importante per questa funzionalità OTA è l'installazione e la gestione delle applicazioni *SIM Toolkit* (STK), che

possono interagire con il device tramite comandi "proattivi" standard (che in realtà sono implementati tramite polling), nonché visualizzare menu, aprire pagine web e inviare SMS. Android supporta STK con un'app di sistema STK dedicata, che viene disabilitata automaticamente se sulla card UICC non sono installate applet STK.

Accesso allo UICC

Come abbiamo visto nel paragrafo "Applicazioni UICC", le funzionalità correlate alla rete mobile in Android, compreso l'accesso allo UICC, sono implementate dal software baseband. Il sistema operativo principale (Android) presenta dei limiti legati alle operazioni eseguibili con lo UICC grazie alle funzionalità esposte dalla baseband. Android supporta le applicazioni STK e può cercare e salvare contatti sulla SIM, quindi è evidente che dispone di un supporto interno per la comunicazione con la SIM; tuttavia, l'informativa sulla sicurezza in Android dichiara esplicitamente che l'accesso di basso livello alla SIM non è disponibile per le app di terze parti (http://bit.ly/1zcRwd0). Come possiamo, allora, utilizzare la SIM (UICC) come SE? Alcune build Android dei vendor più importanti, in particolare Samsung, offrono un'implementazione dell'API OpenMobile di SIMalliance; un'implementazione open source dell'API (per i dispositivi compatibili) è inoltre messa a disposizione dal progetto SEEK for Android. L'API OpenMobile mira a fornire un'interfaccia unificata per l'accesso ai SE su Android, UICC compresi.

Per capire il funzionamento dell'API OpenMobile e le ragioni delle sue limitazioni, dobbiamo riesaminare come è implementato in Android l'accesso alla SIM. Sui dispositivi Android, tutta la funzionalità di rete mobile (composizione, invio di SMS e così via) è fornita dal processore baseband (detto anche *modem* o *radio*). Le applicazioni Android e i servizi di sistema comunicano con la baseband solo indirettamente tramite il daemon *Radio Interface Layer* (RIL) (*rild*). *rild* comunica con l'hardware utilizzando una libreria HAL RIL fornita dal produttore, che racchiude l'interfaccia proprietaria fornita dalla baseband. La card UICC

è tipicamente connessa solo al processore baseband (anche se a volte si connette anche al controller NFC tramite SWP) e di conseguenza tutte le comunicazioni devono passare attraverso RIL.

Se l'implementazione RIL proprietaria può sempre accedere allo UICC per eseguire l'identificazione e l'autenticazione di rete, nonché la lettura e la scrittura di contatti e l'accesso alle applicazioni STK, il supporto per uno scambio di APDU trasparente non è sempre disponibile. Come spiegato nel paragrafo "UICC", il metodo standard per fornire questa funzionalità è l'uso di comandi AT estesi come `AT+CSIM` (*Generic SIM Access*) e `AT+CGLA` (*Generic UICC Logical Channel Access*); tuttavia, alcuni vendor implementano lo scambio di APDU utilizzando estensioni proprietarie, pertanto il supporto per i comandi AT necessari non fornisce automaticamente l'accesso allo UICC.

SEEK for Android implementa un servizio di gestione delle risorse (`SmartCardService`) che può connettersi a qualunque SE supportato (eSE, ASSD o UICC) e alle estensioni del framework di telefonia Android che consentono lo scambio trasparente di APDU con lo UICC. L'accesso tramite RIL dipende dall'hardware e da HAL, pertanto occorrono sia un dispositivo compatibile sia una build che includa `SmartCardService` e le estensioni del framework correlate, come quelle presenti nei più recenti dispositivi Samsung Galaxy.

Uso dell'API OpenMobile

L'API OpenMobile è relativamente piccola e definisce classi che rappresentano il lettore di schede a cui è connesso un SE (`Reader`), una sessione di comunicazione con un SE (`Session`) e un canale di base (`channel 0`, secondo ISO 7816-4) o logico con il SE (`Channel`). La classe `Channel` consente alle applicazioni di scambiare APDU con il SE utilizzando il metodo `transmit()`. Il punto di accesso all'API è la classe `SEService`, che si connette al servizio di gestione delle risorse remoto (`SmartcardService`) e fornisce un metodo che restituisce un elenco di oggetti `Reader` disponibili sul dispositivo (per ulteriori informazioni sull'API OpenMobile e

sull'architettura di `SmartcardService`, fate riferimento a *SEEK for Android Wiki*, `http://bit.ly/1DeSOmr`).

Per utilizzare l'API OpenMobile, le applicazioni devono richiedere il permesso `org.simalliance.openmobileapi.SMARTCARD` e aggiungere la libreria di estensioni *org.simalliance.openmobileapi* al loro manifest, come mostrato nel Listato 11.12.

Listato 11.2 Configurazione di AndroidManifest.xml richiesta per l'uso dell'API OpenMobile.

```
<manifest ...>
    --altro codice--
    <uses-permission android:name="org.simalliance.openmobileapi.SMARTCARD" />

    <application ...>
        <uses-library
            android:name="org.simalliance.openmobileapi"
            android:required="true" />
     --altro codice--
    </application>
</manifest>
```

Il Listato 11.13 mostra come un'applicazione può utilizzare l'API OpenMobile per connettersi e inviare un comando al primo SE sul dispositivo.

Listato 11.13 Invio di un comando al primo SE utilizzando l'API OpenMobile.

```
Context context = getContext();
SEService.CallBack callback = createSeCallback();
SEService seService = new SEService(context, callback);(1)
Reader[] readers = seService.getReaders();(2)
Session session = readers[0].openSession();(3)
Channel channel = session.openLogicalChannel(aid);(4)
byte[] command = { ... };
byte[] response = channel.transmit(command);(5)
```

Qui l'applicazione crea un'istanza di `SEService` **(1)** che si connette a `SmartCardService` in modalità asincrona e informa l'applicazione tramite il metodo `serviceConnected()` (non mostrato) dell'interfaccia `SEService.CallBack` una volta stabilita la connessione. L'app può ottenere un elenco dei lettori SE disponibili utilizzando il metodo `getReaders()` **(2)** e quindi aprire una sessione per il lettore selezionato utilizzando il metodo `openSession()` **(3)**. Se il dispositivo non contiene un eSE (o un'altra forma di SE oltre allo UICC), o se `SmartCardService` non è stato configurato per utilizzare tale eSE, l'elenco di lettori contiene una singola istanza `Reader` che rappresenta il lettore UICC incorporato nel dispositivo. Quando l'app dispone di una `Session` aperta con il SE target, chiama il metodo `openLogicalChannel()` **(4)** per

ottenere un `Channel`, che viene poi usato per inviare gli APDU e ricevere risposte utilizzando il suo metodo `transmit()` **(5)**.

Emulazione di card software

L'*emulazione di card software* (detta anche *Host-based Card Emulation* o HCE) consente di recapitare i comandi ricevuti dal controller NFC al processore delle applicazioni (sistema operativo principale) e di permetterne l'elaborazione da parte delle normali applicazioni Android, anziché dalle applet installate su un SE hardware. Le risposte sono inviate al lettore tramite NFC, pertanto un'app può comportarsi come una smart card contactless virtuale.

Prima dell'aggiunta ufficiale all'API Android, HCE era disponibile come funzione sperimentale della distribuzione di Android CyanogenMod (http://www.cyanogenmod.org/). A partire dalla versione 9.1, CyanogenMod integrava un set di patch (sviluppato da Doug Yeager) che sbloccava la funzionalità HCE del famoso controller NFC PN544 e forniva un'interfaccia framework ad HCE. Per supportare HCE, al framework NFC sono state aggiunte due nuove tecnologie tag (`IsoPcdA` e `IsoPcdB`, che rappresentano i lettori contactless esterni basati rispettivamente sulle tecnologie NFC Type A e Type B). Le lettere *Pcd* sono un'abbreviazione di *Proximity Coupling Device*, un altro termine tecnico per indicare il lettore contactless.

Le classi `IsoPcdA` e `IsoPcdB` hanno capovolto il ruolo degli oggetti `Tag` nell'API NFC di Android: visto che il lettore contactless esterno si presenta come un "tag", i "comandi" inviati dallo smartphone sono in effetti risposte a una comunicazione avviata dal lettore. A differenza del resto dello stack NFC di Android, questa architettura non era guidata dagli eventi e richiedeva alle applicazioni di gestire l'I/O di blocco in attesa che il lettore inviasse il suo comando successivo. Android 4.4 ha introdotto un framework standard guidato dagli eventi per lo sviluppo di applicazioni HCE, di cui parleremo più avanti.

Architettura HCE di Android 4.4

A differenza delle modalità R/W e P2P, disponibili solo per le activity, le applicazioni HCE possono operare in background e sono implementate definendo un servizio che elabora i comandi ricevuti dal lettore esterno e restituisce le risposte. Tali servizi HCE estendono la classe del framework astratta `HostApduService` e ne implementano i metodi `onDeactivated()` e `processCommand()`. `HostApduService` è una piccolissima classe di mediazione che abilita la comunicazione bidirezionale con il servizio di sistema `NfcService` utilizzando oggetti `Messenger` (http://bit.ly/1vgPuEG). Per esempio, quando `NfcService` riceve un APDU che deve essere instradato (il routing degli APDU è presentato nel prossimo paragrafo) a un servizio HCE, invia un `MSG_COMMAND_APDU` a `HostApduService`, che estrae l'APDU dal messaggio e lo passa alla sua implementazione concreta chiamando il metodo `processCommand()`. Se `processCommand()` restituisce un APDU, `HostApduService` lo incapsula in un messaggio `MSG_RESPONSE_APDU` e lo invia a `NfcService`, che a sua volta lo inoltra al controller NFC. Se il servizio HCE concreto non è in grado di restituire subito un APDU di risposta, restituisce `null` e invia la risposta in un secondo momento (quando è disponibile) chiamando `sendResponseApdu()`, che invia la risposta al servizio `NfcService` racchiudendola in un messaggio `MSG_RESPONSE_APDU`.

Routing APDU

Quando il dispositivo è nella modalità di emulazione di card, il controller NFC riceve tutti gli APDU provenienti dai lettori esterni e decide se inviarli a un SE fisico (se presente) o a un servizio HCE in base alla sua tabella di routing APDU interna. La tabella di routing è basata su AID e viene popolata usando i metadati delle applicazioni abilitate per i SE e dei servizi HCE dichiarati nei rispettivi manifest delle applicazioni. Quando il lettore esterno invia un comando `SELECT` che non è indirizzato direttamente al SE, il controller NFC lo inoltra a `NfcService`, che estrae l'AID target dal comando e cerca nella tabella di routing un servizio HCE corrispondente chiamando il metodo `resolveAidPrefix()` della classe `RegisteredAidCache`.

Se viene trovato un servizio corrispondente, `NfcService` effettua il binding con tale servizio e ottiene un'istanza `Messenger` che usa per inviare gli APDU successivi (racchiusi nei messaggi `MSG_COMMAND_APDU`, come spiegato nel paragrafo precedente). Affinché tutto questo funzioni, il servizio HCE dell'app deve essere dichiarato in `AndroidManifest.xml`, come mostrato nel Listato 11.14.

Listato 11.14 Dichiarazione di un servizio HCE in AndroidManifest.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hce" ...>
    --altro codice--
    <uses-permission android:name="android.permission.NFC" />

    <application ...>
        --altro codice--
        <service
            android:name=".MyHostApduService"(1)
            android:exported="true"
            android:permission="android.permission.BIND_NFC_SERVICE" >(2)
            <intent-filter>
                <action
                    android:name="android.nfc.cardemulation.action.HOST_APDU_SERVICE" />(3)
            </intent-filter>

            <meta-data
                android:name="android.nfc.cardemulation.host_apdu_service"(4)
                android:resource="@xml/apduservice" />
        </service>
        --altro codice--
    </application>
</manifest>
```

L'applicazione dichiara il suo servizio HCE **(1)** come di consueto, usando il tag `<service>`; in questo caso vi sono però alcuni requisiti aggiuntivi. Per prima cosa, il servizio deve essere protetto con il permesso di firma di sistema `BIND_NFC_SERVICE` **(2)** al fine di garantire che solo le app di sistema (in pratica solo `NfcService`) possano effettuare il binding al servizio. A seguire, il servizio deve dichiarare un filtro intent corrispondente all'azione `android.nfc.cardemulation.action.HOST_APDU_SERVICE` **(3)** per identificarla come servizio HCE durante la scansione dei package installati e per ottenere il binding alla ricezione di un APDU corrispondente. Infine, il servizio deve disporre di una voce di metadati per le risorse XML, con il nome `android.nfc.cardemulation.host_apdu_service` **(4)**, che faccia riferimento a un file di risorse XML contenente l'elenco degli AID che il servizio può gestire. Il contenuto di questo file è utilizzato per

costruire la tabella di routing AID, consultata dallo stack NFC alla ricezione di un comando `SELECT`.

Specifica del routing per i servizi HCE

Per le applicazioni HCE, il file XML deve includere un elemento root `<host-apdu-service>`, come mostrato nel Listato 11.15.

Listato 11.15 File dei metadati AID del servizio HCE.

```
<host-apdu-service
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/servicedesc"
    android:requireDeviceUnlock="false">(1)
    <aid-group android:description="@string/aiddescription"(2)
               android:category="other">(3)
        <aid-filter android:name="A0000000010101"/>(4)
    </aid-group>
</host-apdu-service>
```

Il tag `<host-apdu-service>` dispone di un attributo `description` e di un attributo `requireDeviceUnlock` **(1)**, che specifica se il servizio HCE corrispondente deve essere attivato al blocco del dispositivo (lo schermo del dispositivo deve essere acceso perché NFC sia utilizzabile). L'elemento root contiene una o più voci `<aid-group>` **(2)**, ognuna delle quali ha un attributo `category` **(3)** e contiene uno o più tag `<aid-filter>` **(4)** che specificano un AID nel loro attributo `name` (`A0000000010101` in questo esempio).

Un gruppo AID definisce un set di AID che vengono sempre gestiti da un particolare servizio HCE. Il framework NFC garantisce che, se un singolo AID è gestito da un servizio HCE, tutti gli altri AID nel gruppo sono anch'essi gestiti dallo stesso servizio. Se due o più servizi HCE definiscono lo stesso AID, il sistema mostra una finestra di selezione lasciando all'utente il compito di scegliere l'applicazione che deve gestire il comando `SELECT` in arrivo. Dopo che l'utente ha confermato la selezione dell'app nella finestra di dialogo mostrata nella Figura 11.2, tutti i comandi successivi vengono indirizzati all'app scelta.

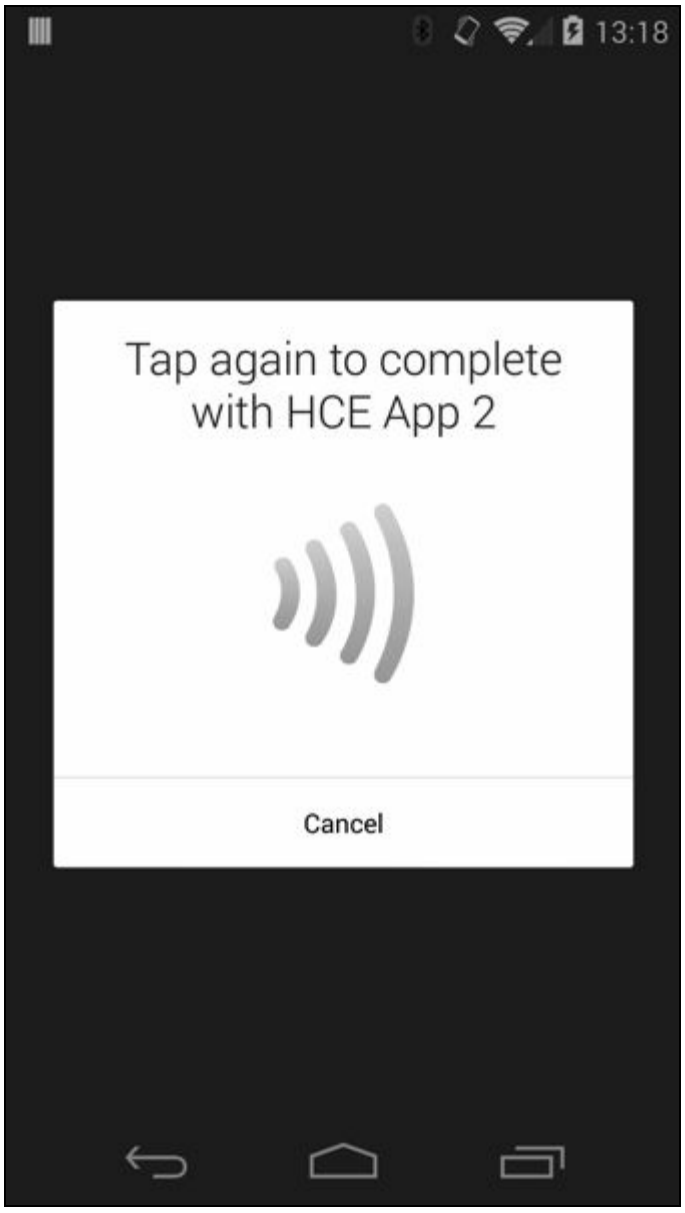

Figura 11.2 Finestra di conferma della selezione dell'applicazione HCE.

Ogni gruppo AID è associato a una categoria (specificata con l'attributo `category`), che consente al sistema di impostare un gestore predefinito per categoria, anziché per AID. Un'applicazione può verificare se un determinato servizio è il gestore predefinito per una categoria chiamando il metodo `isDefaultServiceForCategory()` della classe `CardEmulation`, nonché ottenere la modalità di selezione per una categoria chiamando il metodo `getSelectionModeForCategory()`. Attualmente sono definite solo due categorie: `CATEGORY_PAYMENT` e `CATEGORY_OTHER`.

Android usa una singola categoria di pagamento attiva al fine di garantire che l'utente abbia scelto esplicitamente quale app deve gestire le transazioni di pagamento. L'app predefinita per la categoria di pagamento

viene scelta nella schermata *Tap & pay* dell'app *Settings*, come mostrato nella Figura 11.3 (consultate la documentazione HCE ufficiale per ulteriori informazioni sulle app di pagamento; http://bit.ly/1vgPAfI.)

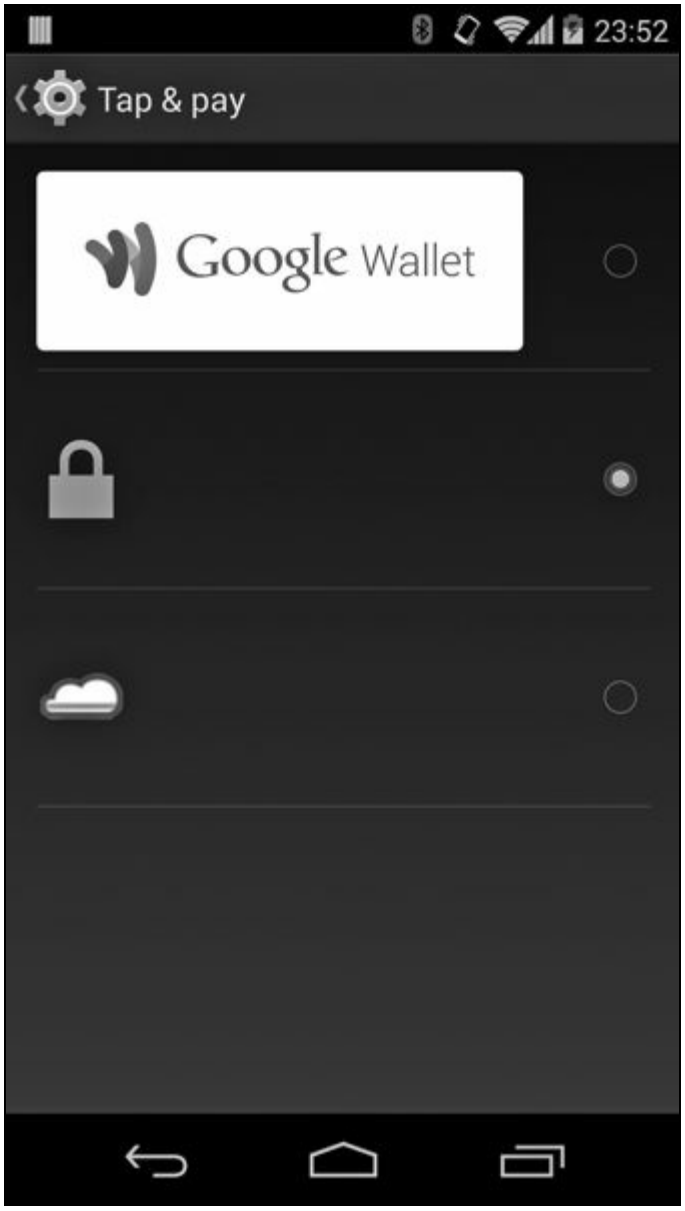

Figura 11.3 Scelta dell'applicazione di pagamento predefinita nella schermata Tap & pay.

Specifica del routing per le applet SE

Se un dispositivo supporta HCE e dispone anche di un SE fisico, un comando SELECT inviato da un lettore esterno può essere destinato sia a un servizio HCE sia a un'applet installata sul SE. Android 4.4 indirizza all'host tutti gli AID non elencati nella tabella di routing degli AID, pertanto gli AID delle applet installate sul SE devono essere aggiunti esplicitamente alla tabella di routing del controller NFC. Questo risultato

si ottiene con lo stesso meccanismo adottato per la registrazione dei servizi HCE: aggiungendo una voce di servizio al manifest dell'applicazione e collegandolo a un file XML di metadati che specifica un elenco di AID da instradare al SE. Una volta stabilita la route, gli APDU di comando vengono inviati direttamente al SE (che li elabora e restituisce una risposta tramite il controller NFC), pertanto il servizio è utilizzato solo come marker e non fornisce alcuna funzionalità.

L'SDK di Android include un servizio helper (`OffHostApduService`) utilizzabile per elencare gli AID da instradare direttamente al SE. Questa classe `OffHostApduService` definisce alcune costanti utili, ma per il resto è vuota; un'applicazione può estenderla e dichiarare il componente del servizio risultante nel suo manifest, come mostrato nel Listato 11.16.

Listato 11.16 Dichiarazione di un servizio APDU esterno all'host in AndroidManifest.xml.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.hce" ...>
    --altro codice--
    <uses-permission android:name="android.permission.NFC" />

    <application ... >
        --altro codice--
        <service android:name=".MyOffHostApduService"
                 android:exported="true"
                 android:permission="android.permission.BIND_NFC_SERVICE">
        <intent-filter>
            <action
                android:name="android.nfc.cardemulation.action.OFF_HOST_APDU_SERVICE"/>(1)
        </intent-filter>
        <meta-data
            android:name="android.nfc.cardemulation.off_host_apdu_service"(2)
            android:resource="@xml/apduservice"/>
    </service>
    --altro codice--
    </application>
</manifest>
```

La dichiarazione del servizio è simile a quella del Listato 11.14, tranne per il fatto che l'azione di intent dichiarata è `android.nfc.cardemulation.action.OFF_HOST_APDU_SERVICE` **(1)** e che il nome dei metadati XML è `android.nfc.cardemulation.off_host_apdu_service` **(2)**. Anche il file dei metadati è leggermente diverso, come mostrato nel Listato 11.17.

Listato 11.17 File dei metadati del servizio APDU esterno all'host.

```
<offhost-apdu-service (1)
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/servicedesc">
    <aid-group android:description="@string/se_applets"
               android:category="other">(2)
        <aid-filter android:name="F0000000000001"/>(3)
        <aid-filter android:name="F0000000000002"/>(4)
```

```
    </aid-group>
</offhost-apdu-service>
```

Come potete osservare, il formato è lo stesso di un servizio HCE, ma l'elemento root del file è `<offhost-apdu-service>` **(1)** anziché `<host-apdu-service>`. Un'altra piccola differenza è che `<offhost-apdu-service>` non supporta l'attributo `requireDeviceUnlock`, perché le transazioni sono inviate direttamente al SE, e di conseguenza l'host non può intervenire indipendentemente dallo stato della schermata di blocco. Gli AID delle applet che risiedono nel SE (**(3)** e **(4)**) sono inclusi in un `<aid-group>` **(2)**. Questi AID sono inviati direttamente al controller NFC, che li salva nella sua tabella di routing interna per poter inviare gli APDU corrispondenti direttamente al SE, senza interagire con il sistema operativo Android. Se l'APDU ricevuto non è nella tabella di routing del controller NFC, il controller lo inoltra a `NfcService`, che lo invia al servizio HCE corrispondente o restituisce un errore se non vengono rilevate corrispondenze.

Scrittura di un servizio HCE

Quando il servizio HCE di un'applicazione è stato dichiarato nel suo manifest, come mostrato nel Listato 11.14, la funzionalità HCE può essere aggiunta estendendo la classe base `HostApduService` e implementando i suoi metodi astratti come mostrato nel Listato 11.18.

Listato 11.18 Implementazione di un servizio HostApduService.

```
public class MyHostApduService extends HostApduService {
    --altro codice--
    static final int OFFSET_CLA = 0;(1)
    static final int OFFSET_INS = 1;
    static final int OFFSET_P1 = 2;
    static final int OFFSET_P2 = 3;
    --altro codice--
    static final short SW_SUCCESS = (short) 0x9000;(2)
    static final short SW_CLA_NOT_SUPPORTED = 0x6E00;
    static final short SW_INS_NOT_SUPPORTED = 0x6D00;
    --altro codice--
    static final byte[] SELECT_CMD = { 0x00, (byte) 0xA4,
            0x04, 0x00, 0x06, (byte) 0xA0,
            0x00, 0x00, 0x00, 0x01, 0x01, 0x01 };(3)

    static final byte MY_CLA = (byte) 0x80;(4)
    static final byte INS_CMD1 = (byte) 0x01;
    static final byte INS_CMD2 = (byte) 0x02;

    boolean selected = false;

    public byte[] processCommandApdu(byte[] cmd, Bundle extras) {
        if (!selected) {
            if (Arrays.equals(cmd, SELECT_CMD)) {(5)
                selected = true;
```

```
                return toBytes(SW_SUCCESS);
            }
            --altro codice--
        }

        if (cmd[OFFSET_CLA] != MY_CLA) {(6)
            return toBytes(SW_CLA_NOT_SUPPORTED);
        }

        byte ins = cmd[OFFSET_INS];(7)
        switch (ins) {
            case INS_CMD1:(8)
                byte p1 = cmd[OFFSET_P1];
                byte p2 = cmd[OFFSET_P2];
                --altro codice--
                return toBytes(SW_SUCCESS);
            case INS_CMD2:
                --altro codice--
                return null;(9)
            default:
                return toBytes(SW_INS_NOT_SUPPORTED);
        }
    }

    @Override
    public void onDeactivated(int reason) {
        --altro codice--
        selected = false;(10)
    }
    --altro codice--
}
```

Qui, il servizio HCE di esempio dichiara alcune costanti utili per la fase di accesso ai dati APDU **(1)** e di restituzione di un risultato standard con lo stato **(2)**. Il servizio definisce il comando `SELECT` usato per attivarlo, comprensivo dell'AID **(3)**. Le costanti successive **(4)** dichiarano la classe delle istruzioni (`CLA`) e le istruzioni che il servizio può gestire.

Quando il servizio HCE riceve un APDU, lo passa al metodo `processCommandApdu()` come array di byte, che viene analizzato dal servizio. Se il servizio non è ancora stato selezionato, il metodo `processCommandApdu()` verifica se l'APDU contiene un comando `SELECT` **(5)** e in tal caso imposta il flag `selected`. Se l'APDU contiene qualche altro comando, il codice verifica se dispone di un byte di classe (`CLA`) supportato dal servizio **(6)** e quindi estrae il byte di istruzioni (`INS`) incluso nel comando **(7)**. Se l'APDU di comando contiene l'istruzione `INS_CMD1` **(8)**, il servizio estrae i parametri `P1` e `P2`, effettua il parsing dei dati inclusi nell'APDU (operazione non mostrata), imposta lo stato interno e restituisce uno stato di operazione riuscita.

Se il comando include `INS_CMD2`, che nell'esempio è associato a un'operazione ipotetica che richiede un certo tempo di elaborazione (per

esempio la generazione di chiavi asimmetriche), il servizio avvia un worker thread (non mostrato) e restituisce `null` **(9)** per non bloccare il thread principale dell'applicazione. Al termine dell'esecuzione, il worker thread può restituire il suo risultato usando il metodo ereditato `sendResponseApdu()` (definito nella classe padre `HostApduService`). Quando viene selezionato un altro servizio o applet SE, il sistema chiama il metodo `onDeactivated()`, che dovrebbe liberare le risorse usate prima di restituire il controllo; nel nostro esempio, però, imposta semplicemente il flag `selected` a `false` **(10)**.

Visto che, fondamentalmente, un servizio HCE effettua il parsing degli APDU di comando e restituisce le risposte, il modello di programmazione è molto simile a quello delle applet Java Card. Tuttavia, poiché un servizio HCE risiede all'interno di una normale applicazione Android, non viene eseguito in un ambiente vincolato e può sfruttare tutte le funzionalità Android disponibili. Diventa così più facile implementare funzionalità complesse, ma anche influire sulla sicurezza delle app HCE, come vedremo nel prossimo paragrafo.

Sicurezza delle applicazioni HCE

Visto che qualunque applicazione Android può dichiarare un servizio HCE e ricevere ed elaborare APDU, il sistema garantisce che un'applicazione dannosa non possa iniettare comandi APDU falsi in un servizio HCE richiedendo il permesso di firma di sistema `BIND_NFC_SERVICE` per il binding ai servizi HCE. Inoltre, il modello di sandboxing di Android garantisce che le altre applicazioni non possano accedere ai dati sensibili memorizzati dall'applicazione HCE leggendo i suoi file o chiamando le API di accesso ai dati esposte senza permesso (naturalmente supponendo che tali API siano state adeguatamente protette).

Ciononostante, un'applicazione dannosa che tenta di ottenere privilegi root su un dispositivo (per esempio sfruttando una vulnerabilità di escalation dei privilegi) può ispezionare e iniettare APDU mirati a un servizio HCE e leggere i suoi dati privati. L'applicazione HCE può

prendere alcune misure per rilevare questa situazione, per esempio controllando l'identità e il certificato di firma del chiamante del suo metodo `processCommandApdu()`, ma queste possono essere vanificate ottenendo un accesso senza restrizioni al sistema operativo. Come tutte le applicazioni che memorizzano dati sensibili, le applicazioni HCE dovrebbero prendere provvedimenti anche per proteggere i dati memorizzati, per esempio crittografandoli su disco o depositandoli nell'archivio delle credenziali di sistema nel caso delle chiavi di crittografia. Un altro modo per proteggere il codice e i dati delle applicazioni HCE è l'inoltro di tutti i comandi ricevuti a un server remoto, utilizzando un canale crittografato, affidandosi quindi solo alle sue risposte. Tuttavia, poiché la maggior parte di queste misure è implementata nel software, è comunque possibile disabilitarle o aggirarle con un'applicazione dannosa sufficientemente ricercata e con accesso root.

Al contrario, gli elementi di sicurezza hardware offrono una resistenza fisica alla manomissione, una superficie di attacco ridotta grazie alla loro funzionalità limitate e un rigoroso controllo sulle applet installate. Di conseguenza, i SE fisici sono più difficili da attaccare e offrono una protezione più solida dei dati sensibili utilizzati nei tipici scenari di emulazione della card, come i pagamenti contactless, anche quando la sicurezza predefinita offerta dal sistema operativo host è stata aggirata.

NOTA

Per una discussione dettagliata sulle differenze a livello di sicurezza tra le applicazioni di emulazione delle card implementate negli elementi sicuri o nel software utilizzando HCE, fate riferimento alla serie di post di blog "HCE vs embedded secure element" di Cem Paya (che ha lavorato all'implementazione originale di Google Wallet con supporto SE; parti da I a VI, http://bit.ly/1ocd8By).

Riepilogo

Android supporta tre modalità NFC: *Reader/Writer* (R/W), *Peer-to-Peer* (P2P) e *Card Emulation* (CE). Nella modalità Reader/Writer, i dispositivi Android possono accedere a NFC, carte contactless e dispositivi di emulazione NFC, mentre la modalità point-to-point fornisce semplici funzionalità di scambio dei dati. La modalità di emulazione delle card può essere ottenuta con un elemento sicuro (SE) fisico come uno UICC, un elemento sicuro integrato nel controller NFC (SE incorporato) o le normali applicazioni Android a partire da Android 4.4. Gli elementi di sicurezza hardware garantiscono la massima sicurezza offrendo la resistenza fisica alla manomissione e un controllo rigoroso sulla gestione dell'applicazione SE (totalmente implementata come applet Java Card). Tuttavia, poiché le chiavi di autenticazione che richiedono l'installazione di un'applicazione su un SE sono generalmente controllate da una singola entità (come il produttore del dispositivo o un MNO), la distribuzione di applicazioni SE può essere problematica. La tecnologia *Host-based Card Emulation* (HCE), introdotta in Android 4.4, facilita lo sviluppo e la distribuzione di applicazioni che operano nella modalità Card Emulation, ma fa affidamento solo sul sistema operativo per applicare la sicurezza, e di conseguenza offre una protezione debole del codice e dei dati delle applicazioni sensibili.

SELinux

Anche se nei capitoli precedenti abbiamo accennato a *Security-Enhanced Linux* (SELinux) e alla sua integrazione in Android, la nostra descrizione del modello di sicurezza di Android finora si è concentrata sull'implementazione sandbox "tradizionale" di Android, che fa grande affidamento sul *Discretionary Access Control* (DAC) di Linux. Il DAC di Linux è leggero e ormai collaudato, ma presenta alcuni svantaggi, in particolare la granularità ridotta dei permessi DAC, l'eventualità di fughe di dati dovute a programmi configurati erroneamente e l'impossibilità di applicare vincoli precisi ai privilegi dei processi in esecuzione con l'utente root. (Anche se le funzionalità POSIX, implementate come estensione al DAC tradizionale di Linux, offrono un mezzo per concedere solo certi privilegi ai processi root, la granularità delle funzioni POSIX è limitata e i privilegi concessi si estendono a tutti gli oggetti a cui il processo in questione accede.)

Mandatory Access Control (MAC), nell'implementazione di SELinux, prova a superare i limiti del DAC di Linux applicando una policy di sicurezza a livello di sistema più granulare, che può essere modificata solo dall'amministratore del sistema e non dagli utenti o dai programmi senza privilegi.

Questo capitolo fornisce una breve introduzione all'architettura e ai concetti utilizzati in SELinux e descrive le principali modifiche apportate a SELinux per il supporto di Android. Infine, vedremo brevemente la policy SELinux distribuita nella versione corrente di Android.

Introduzione a SELinux

SELinux è un meccanismo di controllo di accesso obbligatorio per il kernel Linux, implementato come modulo di sicurezza Linux. Il framework *Linux Security Modules* (LSM) consente di collegare al kernel meccanismi di controllo di accesso di terze parti e di modificare l'implementazione predefinita del DAC. LSM è implementato come una serie di hook di funzione di sicurezza (*upcall*) e strutture di dati correlati integrate nei vari moduli del kernel Linux responsabile del controllo di accesso.

Alcuni dei principali servizi del kernel in cui sono inseriti hook LSM sono l'esecuzione dei programmi, le operazioni su file e inode, la messaggistica netlink e le operazioni sui socket. Se non è installato alcun modulo di sicurezza, Linux utilizza il suo meccanismo DAC predefinito per regolamentare l'accesso agli oggetti kernel gestiti da questi servizi. Se è installato un modulo di sicurezza, Linux lo consulta insieme al DAC per prendere una decisione finale di sicurezza quando viene richiesto l'accesso a un oggetto del kernel.

Oltre a fornire hook nei principali servizi del kernel, il framework LSM estende anche il file system virtuale procfs (*/proc*) per includere gli attributi di sicurezza (*thread*) per processo e per task; aggiunge inoltre il supporto per l'utilizzo di attributi estesi del file system come archivio persistente degli attributi di sicurezza. SELinux è stato il primo modulo LSM integrato nel kernel Linux ed è ufficialmente disponibile a partire dalla versione 2.6 (le precedenti implementazioni erano distribuite come un set di patch). Dopo l'integrazione di SELinux, sono stati accettati nel kernel mainline anche altri moduli di sicurezza, che attualmente comprendono AppArmor, Smack e TOMOYO Linux. Questi moduli forniscono implementazioni MAC alternative e sono basati su modelli di sicurezza diversi da quelli di SELinux.

Nei prossimi paragrafi esamineremo il modello di sicurezza SELinux e la sua architettura.

Architettura di SELinux

Anche se l'architettura di SELinux è piuttosto complessa, ad alto livello è costituita da quattro componenti principali: i gestori di oggetti (OM), una cache di vettori di accesso (AVC), un server di protezione e una policy di sicurezza, come mostrato nella Figura 12.1.

Quando un soggetto chiede di eseguire un'azione su un oggetto SELinux (per esempio quando un processo tenta di leggere un file), il gestore dell'oggetto associato interroga AVC per valutare se l'azione è consentita. Se AVC contiene nella cache una decisione di sicurezza per la richiesta, AVC la restituisce all'OM, che applica la decisione consentendo o negando l'azione (passi 1, 2 e 5 nella Figura 12.1). Se la cache non contiene una decisione di sicurezza corrispondente, AVC contatta il server di protezione, il quale prende una decisione di sicurezza basata sulla policy attualmente caricata e la restituisce ad AVC, che la memorizza nella cache. AVC a sua volta la restituisce all'OM, che applica la decisione (passi 1, 2, 3, 4 e 5 nella Figura 12.1). Il server di protezione è parte del kernel, mentre la policy viene caricata dallo userspace con una serie di funzioni contenute nella libreria userspace di supporto.

OM e AVC possono risiedere sia nello spazio del kernel (quando l'OM gestisce oggetti a livello di kernel) sia nello userspace (quando l'OM è parte di una cosiddetta applicazione SELinux-aware, che dispone di supporto MAC incorporato).

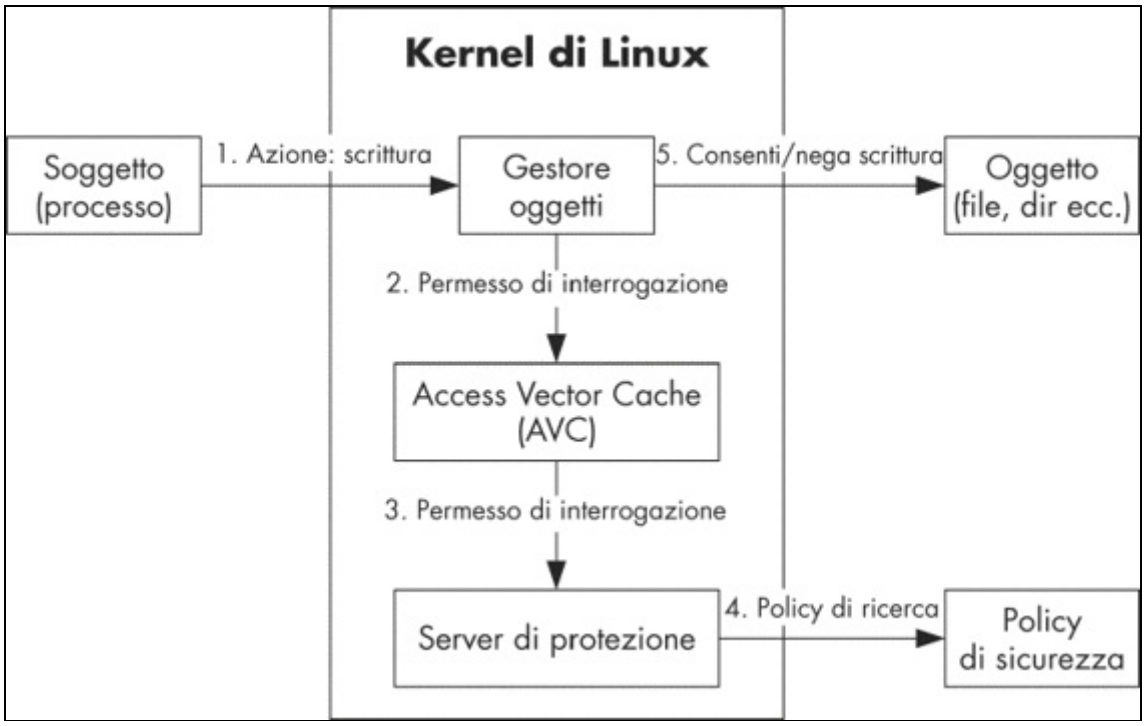

Figura 12.1 Componenti di SELinux.

Mandatory Access Control

Il modello MAC di SELinux si basa su tre concetti principali: soggetti, oggetti e azioni. In questo modello, i soggetti sono gli attori attivi che eseguono le azioni sugli oggetti; l'azione viene eseguita solo se la policy di sicurezza lo permette.

In pratica, i soggetti di solito sono processi in esecuzione (un processo può anche essere un oggetto), mentre gli oggetti sono risorse a livello di sistema operativo gestite dal kernel, come file e socket. Soggetti e oggetti dispongono di una serie di attributi di sicurezza (noti collettivamente come *contesto di protezione*; ne parliamo nel prossimo paragrafo), che il sistema operativo interroga per decidere se l'azione richiesta deve essere consentita o no. Quando SELinux è abilitato, i soggetti non possono aggirare o influenzare le regole della policy; di conseguenza, la policy è obbligatoria.

NOTA

La policy MAC viene consultata solo se il DAC consente l'accesso a una risorsa; se il DAC impedisce l'accesso (per esempio sulla base dei permessi dei file), la negazione viene considerata come decisione di sicurezza finale.

SELinux supporta due forme di MAC: *Type Enforcement* (TE) e *Multi-Level Security* (MLS). MLS è solitamente utilizzato per applicare diversi livelli di accesso alle informazioni riservate e non è utilizzato in Android. Il type enforcement implementato in SELinux richiede che tutti i soggetti e gli oggetti siano associati a un tipo, che SELinux usa per applicare le regole della sua policy di sicurezza.

In SELinux, un *tipo* non è altro che una stringa definita nella policy e associata a oggetti o soggetti. I tipi dei soggetti fanno riferimento a processi o gruppi di processi e sono detti anche *domini*; i tipi che si riferiscono a oggetti di solito specificano il ruolo svolto da un oggetto all'interno della policy, per esempio file di sistema, file di dati dell'applicazione e così via. Il tipo (o dominio) è parte integrante del contesto di protezione, come descritto nel paragrafo "Contesti di protezione" più avanti in questo capitolo.

Modalità di SELinux

SELinux ha tre modalità di funzionamento: Disabled, Permissive ed Enforcing. Quando SELinux è disabilitato (modalità disabled), non vengono caricate policy e viene applicata solo la sicurezza DAC predefinita. Nella modalità Permissive, la policy viene caricata e viene controllato l'accesso agli oggetti, ma la negazione dell'accesso viene solamente registrata (non applicata). Infine, nella modalità Enforcing, la policy di sicurezza viene caricata e applicata, registrando le violazioni.

In Android, è possibile controllare e cambiare la modalità di SELinux con i comandi `getenforce` e `setenforce`, come mostrato nel Listato 12.1. Tuttavia, la modalità impostata con `setenforce` non è persistente, e al riavvio del dispositivo viene riapplicata la modalità predefinita.

Listato 12.1 Uso dei comandi getenforce e setenforce.

```
# getenforce
Enforcing
# setenforce 0
# getenforce
Permissive
```

Inoltre, anche quando SELinux è nella modalità Enforcing, la policy può specificare la modalità Permissive per dominio (processo) con

l'istruzione `permissive` (vedete "Istruzioni per classi di oggetti e permessi" per un esempio).

Contesti di protezione

In SELinux, un *contesto di protezione* (detto anche *etichetta di protezione*, o semplicemente *etichetta*) è una stringa con quattro campi delimitati da due punti: nome utente, ruolo, tipo e un intervallo di sicurezza MLS facoltativo. Un nome utente SELinux è in genere associato a un gruppo o una classe di utenti, per esempio `user_u` per gli utenti non privilegiati e `admin_u` per gli amministratori.

Gli utenti possono essere associati a uno o più ruoli per implementare il controllo di accesso basato su ruoli, dove ogni ruolo è associato a uno o più tipi di dominio. Il tipo è usato per raggruppare i processi in un dominio o per specificare un tipo di logica per gli oggetti.

L'intervallo (o livello) di sicurezza è usato per implementare MLS e specifica i livelli di sicurezza a cui un soggetto può accedere. Attualmente Android utilizza solo il campo tipo del contesto di protezione; l'utente e l'intervallo di sicurezza sono sempre impostati a `u` e `s0`. Il ruolo è impostato su `r` per i domini (processi) o sul ruolo predefinito `object_r` per gli oggetti.

Il contesto di protezione dei processi può essere visualizzato specificando l'opzione `-Z` nel comando `ps`, come mostrato nel Listato 12.2 (colonna `LABEL`).

Listato 12.2 Contesti di protezione dei processi in Android.

```
# ps -Z
LABEL                          USER     PID   PPID   NAME
u:r:init:s0(1)                  root     1     0      /init
u:r:kernel:s0                  root     2     0      kthreadd
u:r:kernel:s0                  root     3     2      ksoftirqd/0
--altro codice--
u:r:healthd:s0(2)               root     175   1      /sbin/healthd
u:r:servicemanager:s0(3)        system   176   1      /system/bin/servicemanager
u:r:vold:s0(4)                  root     177   1      /system/bin/vold
u:r:init:s0                    nobody   178   1      /system/bin/rmt_storage
u:r:netd:s0                    root     179   1      /system/bin/netd
u:r:debuggerd:s0               root     180   1      /system/bin/debuggerd
u:r:rild:s0                    radio    181   1      /system/bin/rild
--altro codice--
u:r:platform_app:s0            u0_a12   950   183    com.android.systemui
u:r:media_app:s0               u0_a5    1043  183    android.process.media
u:r:radio:s0                   radio    1141  183    com.android.phone
u:r:nfc:s0                     nfc      1163  183    com.android.nfc
```

```
u:r:untrusted_app:s0          u0_a7    1360  183    com.google.android.gms
--altro codice--
```

Analogamente, il contesto dei file può essere visualizzato passando `-Z` al comando `ls`, come mostrato nel Listato 12.3.

Listato 12.3 Contesti di protezione di file e directory in Android.

```
# ls -Z
drwxr-xr-x root     root              u:object_r:cgroup:s0 acct
drwxrwx--- system   cache             u:object_r:cache_file:s0 cache
-rwxr-x--- root     root              u:object_r:rootfs:s0 charger
--altro codice--
drwxrwx--x system   system            u:object_r:system_data_file:s0 data
-rw-r--r-- root     root              u:object_r:rootfs:s0 default.prop
drwxr-xr-x root     root              u:object_r:device:s0 dev
lrwxrwxrwx root     root              u:object_r:rootfs:s0 etc -> /system/etc
-rw-r--r-- root     root              u:object_r:rootfs:s0 file_contexts
dr-xr-x--- system   system            u:object_r:sdcard_external:s0 firmware
-rw-r----- root     root              u:object_r:rootfs:s0 fstab.hammerhead
-rwxr-x--- root     root              u:object_r:rootfs:s0 init
--altro codice--
```

Assegnazione e persistenza del contesto di protezione

Abbiamo stabilito che tutti i soggetti e gli oggetti hanno un contesto di protezione. Occorre adesso capire come questo viene assegnato e reso persistente. Per gli oggetti (che di solito sono associati a un file sul file system), il contesto di protezione è persistente ed è di solito memorizzato come attributo esteso nei metadati del file.

Gli attributi estesi non sono interpretati dal file system e possono contenere dati arbitrari (anche se questi sono in genere limitati in termini di dimensioni). Il file system `ext4`, quello predefinito nella maggior parte delle distribuzioni Linux e nelle versioni attuali di Android, supporta gli attributi estesi nella forma di coppie nome-valore, dove il nome è una stringa terminata da null. SELinux usa il nome `security.selinux` per salvare il contesto di protezione degli oggetti file. Il contesto può essere impostato esplicitamente come parte di un'inizializzazione del file system (si parla anche di *etichettatura*), oppure può essere assegnato implicitamente quando si crea un oggetto. Gli oggetti in genere ereditano l'etichetta tipo del loro padre (per esempio, i file appena creati in una directory ereditano l'etichetta della directory); tuttavia, se la policy di sicurezza lo

permette, gli oggetti possono ricevere un'etichetta diversa da quella del padre con un processo definito *transizione del tipo*.

Come gli oggetti, i soggetti (processi) ereditano il contesto di protezione del loro processo padre, oppure possono cambiare il loro contesto con una *transizione di dominio*, se consentito dalla policy di sicurezza. La policy può anche specificare una transizione di dominio automatica, che imposta il dominio dei nuovi processi avviati in base al dominio del padre e al tipo di binario eseguito. Per esempio, visto che tutti i daemon di sistema sono avviati dal processo `init`, che ha il contesto di protezione `u:r:init:s0` (**(1)** nel Listato 12.2), essi normalmente ereditano questo contesto, ma la policy SELinux di Android usa le transizioni di dominio automatiche per impostare un dominio dedicato a ogni daemon in base alle necessità (**(2)**, **(3)** e **(4)** nel Listato 12.2).

Policy di sicurezza

La policy di sicurezza SELinux è utilizzata dal server di protezione nel kernel per consentire o negare l'accesso agli oggetti del kernel in fase di esecuzione. Per motivi di prestazioni, la policy è in genere in una forma binaria generata compilando un certo numero di file sorgente di policy. Questi sono scritti in un linguaggio dedicato, costituito da istruzioni e regole. Le *istruzioni* definiscono le entità delle policy, quali tipi, utenti e ruoli; le *regole* consentono o negano l'accesso agli oggetti (regole dei vettori di accesso), specificano il tipo di transizioni ammesse (regole di applicazione del tipo) e definiscono l'assegnazione di utenti, ruoli e tipi predefiniti (regole predefinite). Una discussione approfondita della grammatica delle policy di SELinux va oltre l'ambito di questo libro; tuttavia, nei paragrafi successivi troverete un'introduzione alle istruzioni e alle regole usate più spesso.

Istruzioni per le policy

Il linguaggio per le policy di SELinux supporta vari tipi di istruzioni; nondimeno, una policy di sicurezza è costituita perlopiù da istruzioni per

tipi, attributi e permessi. Vedremo quindi questi tre tipi di istruzioni nei paragrafi che seguono.

Istruzioni per tipi e attributi

Le istruzioni `type` e `attribute` dichiarano i tipi e i loro attributi, come mostrato nel Listato 12.4.

Listato 12.4 Istruzioni per tipi e attributi.

```
attribute file_type;(1)
attribute domain;(2)

type system_data_file, file_type, data_file_type;(3)
type untrusted_app, domain;(4)
```

Qui la prima **(1)** e la seconda **(2)** istruzione dichiarano gli attributi `file_type` e `domain`, mentre l'istruzione successiva **(3)** dichiara il tipo `system_data_file` e lo associa agli attributi `file_type` e `data_file_type`. Il codice al punto **(4)** dichiara il tipo `untrusted_app` e lo associa all'attributo `domain` (che contrassegna tutti i tipi utilizzati per i processi).

In base alla granularità, una policy di SELinux può avere decine o addirittura centinaia di dichiarazioni di tipi e attributi sparse in molteplici file sorgente. Tuttavia, poiché l'accesso a tutti gli oggetti kernel deve essere verificato rispetto alla policy in fase di esecuzione, una policy troppo estesa può avere un effetto negativo sulle prestazioni. L'incidenza sulle performance è particolarmente evidente sui dispositivi con risorse di elaborazione limitate: ecco perché Android cerca di mantenere relativamente piccola la sua policy SELinux.

Istruzioni per utenti e ruoli

L'istruzione `user` dichiara un identificatore utente SELinux, lo associa al suo ruolo e, facoltativamente, specifica il suo livello di sicurezza predefinito e l'intervallo di livelli di sicurezza a cui l'utente può accedere. Il Listato 12.5 mostra le dichiarazioni dell'identificatore utente predefinito (e unico) in Android.

Listato 12.5 Dichiarazioni dell'identificatore utente SELinux predefinito in Android.

```
user u roles { r } level s0 range s0 - mls_systemhigh;
```

Come potete osservare nel Listato 12.5, l'utente `u` è associato al ruolo `r` (tra parentesi graffe), che a sua volta è dichiarato utilizzando l'istruzione `role` **(1)** come mostrato nel Listato 12.6.

Listato 12.6 Dichiarazione del ruolo SELinux predefinito in Android.

```
role r;(1)
role r types domain;(2)
```

La seconda istruzione **(2)** associa il ruolo `r` all'attributo `domain`, che lo contrassegna come un ruolo assegnato ai processi (domini).

Istruzioni per classi di oggetti e permessi

L'istruzione `permissive` consente a un dominio denominato di essere eseguito nella modalità Permissive (una modalità che registra le violazioni alla policy MAC ma che in effetti non applica la policy, come vedremo più avanti), anche se SELinux è in esecuzione nella modalità Enforcing. Come vedremo nel paragrafo "Applicazione dei domini", la maggior parte dei domini nella policy base attuale di Android è "permissive": per esempio, i processi nel dominio `adbd` (`in` pratica i processi del daemon `adbd`) sono eseguiti nella modalità Permissive, come mostrato nel Listato 12.7 **(1)**.

Listato 12.7 Impostazione di un dominio denominato sulla modalità permissiva.

```
type adbd, domain;
permissive adbd;(1)
--altro codice--
```

L'istruzione `class` definisce una classe di oggetti SELinux, come mostrato nel Listato 12.8. Le classi di oggetti e i permessi associati sono determinati dalle rispettive implementazioni del gestore degli oggetti nel kernel di Linux e sono statici all'interno di una policy. Le classi di oggetti sono solitamente definite nel file sorgente di policy `security_classes`.

Listato 12.8 Dichiarazioni delle classi di oggetti nel file security_classes.

```
--altro codice--
# file-related classes
class filesystem
class file
class dir
class fd
class lnk_file
class chr_file
class blk_file
class sock_file
```

```
class fifo_file
--altro codice--
```

I permessi SELinux (detti anche *vettori di accesso*) sono in genere definiti e associati alle classi di oggetti in un file sorgente di policy chiamato `access_vectors`. Possono essere specifici per una classe (definiti con la parola chiave `class`) o ereditabili da una o più classi di oggetti, caso in cui sono definiti con la parola chiave `common`. Il Listato 12.9 mostra la definizione del set di permessi comune a tutti gli oggetti file **(1)**, l'associazione della classe `dir` (che rappresenta le directory) con tutti i permessi di file comuni (utilizzando la parola chiave `inherits`), nonché un set di permessi specifici per le directory (`add_name`, `remove_name` e così via) **(2)**.

Listato 12.9 Definizioni dei permessi nel file access_vectors.

```
--altro codice--
common file
{
    ioctl
    read
    write
    create
    getattr
    setattr
    lock
    --altro codice--
}(1)
--altro codice--
class dir
inherits file
{
    add_name
    remove_name
    reparent
    search
    rmdir
    --altro codice--
}(2)
--altro codice--
```

Regole di transizione dei tipi

Le regole di applicazione del tipo e le regole dei vettori di accesso (di cui si parla nei paragrafi "Regole di transizione dei domini" e "Regole dei vettori di accesso") costituiscono di solito la parte più consistente di una policy SELinux. Il tipo più comune di regola di applicazione è `type_transition`, che specifica quando sono consentite le transizioni di tipi e domini. A titolo di esempio, il daemon `wpa_supplicant`, che gestisce le connessioni Wi-Fi in Android, usa la regola di transizione dei tipi mostrata nel Listato 12.10 al punto **(4)** per associare i socket di controllo

creati nella directory */data/misc/wifi/* con il tipo `wpa_socket`. In assenza di questa regola, i socket erediterebbero il tipo della loro directory padre, `wifi_data_file`.

Listato 12.10 Transizioni dei tipi nel dominio wpa (da wpa_supplicant.te).

```
# wpa - wpa supplicant or equivalent
type wpa, domain;
permissive wpa;(1)
type wpa_exec, exec_type, file_type;

init_daemon_domain(wpa)(2)
unconfined_domain(wpa)(3)
type_transition wpa wifi_data_file:sock_file wpa_socket;(4)
```

Qui `wpa`, `wifi_data_file:sock_file` e `wpa_socket` sono rispettivamente il tipo di origine (in questo caso il dominio del processo `wpa_supplicant`), il tipo e la classe di destinazione (il tipo e la classe dell'oggetto prima della transizione) e il tipo di oggetto dopo la transizione.

NOTA

Per creare il file del socket e cambiarne l'etichetta, il dominio `wpa` necessita di permessi aggiuntivi sulla directory padre e sul file del socket stesso; la sola regola `type_transition` non è sufficiente. Tuttavia, poiché il dominio `wpa` è permissivo **(1)** e senza restrizioni (riceve la maggior parte dei permessi per impostazione predefinita) **(3)**, la transizione è consentita anche senza grant esplicito di ogni permesso necessario.

Regole di transizione dei domini

In Android, i daemon di sistema nativi come `wpa_supplicant` sono avviati dal processo `init` e di conseguenza, per impostazione predefinita, ereditano il suo contesto di protezione. Tuttavia, quasi tutti i daemon sono associati a un dominio dedicato e usano le transizioni dei domini per cambiare il relativo dominio in fase di avvio. Questo risultato di norma si ottiene utilizzando la macro `init_daemon_domain()` (**(2)** nel Listato 12.10), che dietro le quinte è implementata utilizzando la parola chiave `type_transition`, proprio come le transizioni dei tipi.

Il processo di generazione della policy SELinux binaria usa il preprocessore macro `m4` per espandere le macro prima di unire tutti i file di origine per creare il file della policy binario (*GNU M4 - GNU Project - Free Software Foundation (FSF)*, https://www.gnu.org/software/m4/). La macro `init_daemon_domain()` accetta un solo parametro (il nuovo dominio del

processo) ed è definita nel file `te_macros` utilizzando altre due macro, `domain_trans()` e `domain_auto_trans()`, che servono rispettivamente per consentire la transizione a un nuovo dominio e per eseguire automaticamente la transizione. Il Listato 12.11 mostra le definizioni di queste tre macro (**(1)**, **(2)** e **(3)**). Le righe che iniziano con la parola chiave `allow` sono regole dei *vettori di accesso* (AV), di cui parleremo nel prossimo paragrafo.

Listato 12.11 Definizione delle macro di transizione dei domini nel file te_macros.

```
# domain_trans(olddomain, type, newdomain)
define('domain_trans', '
allow $1 $2:file { getattr open read execute };
allow $1 $3:process transition;
allow $3 $2:file { entrypoint read execute };
allow $3 $1:process sigchld;
dontaudit $1 $3:process noatsecure;
allow $1 $3:process { siginh rlimitinh };
')(1)
# domain_auto_trans(olddomain, type, newdomain)
define('domain_auto_trans', '
domain_trans($1,$2,$3)
type_transition $1 $2:process $3;
')(2)
# init_daemon_domain(domain)
define('init_daemon_domain', '
domain_auto_trans(init, $1_exec, $1)
tmpfs_domain($1)
')(3)
--altro codice--
```

Regole dei vettori di accesso

Le regole AV definiscono i privilegi a disposizione dei processi in fase di esecuzione specificando il set di permessi disponibili sui rispettivi oggetti target. Nel Listato 12.12 è mostrato il formato generico di una regola AV.

Listato 12.12 Formato delle regole AV.

```
rule_name source_type target_type : class perm_set;
```

`rule_name` può essere `allow`, `dontaudit`, `auditallow` o `neverallow`. Per costruire una regola, gli elementi `source_type` e `target_type` vengono sostituiti con uno o più identificatori `type` o `attribute` definiti in precedenza, dove `source_type` è l'identificatore di un soggetto (processo) e `target_type` è l'identificatore di un oggetto a cui il processo tenta di accedere. L'elemento `class` viene sostituito con la classe oggetto del target, mentre `perm_set` specifica il set di permessi che il processo di origine ha sull'oggetto target. È possibile

specificare più tipi, classi e permessi racchiudendoli tra parentesi graffe ({}). Inoltre, alcune regole supportano l'uso degli operatori carattere jolly (*) e complemento (~), che consentono rispettivamente di specificare l'inclusione di tutti i tipi o di tutti i tipi tranne quelli indicati esplicitamente.

Regole allow

La regola usata più comunemente è `allow`, che specifica le operazioni che un soggetto (processo) del tipo di origine specificato può eseguire su un oggetto del tipo e della classe di destinazione specificati nella regola. Prendiamo come esempio la policy SELinux per il daemon `vold` (Listato 12.13) per illustrare l'uso.

Listato 12.13 Regole allow per il dominio vold (da vold.te).

```
type vold, domain;
type vold_exec, exec_type, file_type;
init_daemon_domain(vold)
--altro codice--
allow vold sdcard_type:filesystem { mount remount unmount };(1)
--altro codice--
allow vold self:capability { sys_ptrace kill };(2)
--altro codice--
```

Nel listato, la regola **(1)** consente al daemon `vold` (in esecuzione nel dominio `vold`) di montare, smontare e rimontare i file system di tipo `sdcard_type`. La regola **(2)** permette al daemon di utilizzare le capability Linux `CAP_SYS_PTRACE` (che consente di chiamare `ptrace()` su qualunque processo) e `CAP_KILL` (che consente di inviare segnali a qualunque processo), che corrispondono al set di permessi specificato nella regola (tra `{}`). Nella regola **(2)**, la parola chiave `self` indica che il dominio di destinazione è lo stesso dell'origine, in questo caso `vold`.

Regole auditallow

La regola `auditallow` è utilizzata con `allow` per registrare gli eventi di audit quando un'operazione è consentita. È utile perché, per impostazione predefinita, SELinux registra solo gli eventi di accesso negato. Tuttavia, `auditallow` da sola non consente l'accesso, pertanto occorre usare una regola `allow` corrispondente per concedere i permessi di accesso necessari.

Regole dontaudit

La regola `dontaudit` è usata per evitare l'audit dei messaggi di negazione quando un evento specificato è considerato sicuro. Per esempio, la regola al punto **(1)** del Listato 12.14 specifica di non creare log di audit se al daemon `installd` viene negata la capability `CAP_SYS_ADMIN` . Tuttavia, queste regole possono mascherare gli errori di programma, quindi l'uso di `dontaudit` è sconsigliato.

Listato 12.14 Regola dontaudit per il dominio installd (da installd.te).

```
type installd, domain;
--altro codice--
dontaudit installd self:capability sys_admin;(1)
--altro codice--
```

Regole neverallow

La regola `neverallow` afferma che l'operazione dichiarata non deve mai essere consentita, nemmeno se esiste una regola `allow` esplicita che la permette. Per esempio, la regola del Listato 12.15 vieta a tutti i domini, tranne `init`, di caricare la policy SELinux.

Listato 12.15 Regola neverallow che vieta ai domini diversi da init di caricare la policy SELinux (da domain.te).

```
--altro codice--
neverallow { domain -init } kernel:security load_policy;
```

NOTA

Questo paragrafo offre solo una breve introduzione a SELinux, incentrata sulle funzionalità usate in Android. Per una descrizione dettagliata dell'architettura e dell'implementazione di SELinux, nonché del suo linguaggio per le policy, consultate *SELinux Notebook* (Richard Haines, *The SELinux Notebook: The Foundations*, terza edizione, 2012, http://bit.ly/101qDJG).

Implementazione in Android

Come spiegato nei Capitoli 1 e 2, il modello di sicurezza di sandboxing di Android si basa principalmente sull'utilizzo di UID Linux separati per applicazioni e daemon di sistema. L'isolamento dei processi e il controllo di accesso sono applicati in ultima analisi dal kernel di Linux sulla base di GID e UID di processo. Visto che anche SELinux è parte del kernel di Linux, è un candidato naturale per il rafforzamento del modello di sandboxing di Android mediante una policy MAC.

SELinux è integrato nel kernel Linux mainline, pertanto potrebbe sembrare che la sua abilitazione in Android abbia richiesto solamente di configurare il kernel e progettare una policy MAC adeguata. In realtà, poiché Android introduce alcune estensioni univoche nel kernel Linux e poiché la sua struttura userspace è abbastanza diversa da quella delle distribuzioni Linux per desktop e server, sono state necessarie diverse modifiche al kernel e allo userspace. Anche se il lavoro per l'integrazione di SELinux è stato avviato da Google, gran parte delle modifiche richieste è stata implementata nel progetto Security Enhancements for Android (precedentemente Security-Enhanced Android, o SEAndroid, https://bitbucket.org/seandroid/manifests/) e successivamente è stata integrata nel source tree Android mainline. Nei paragrafi successivi vengono presi in esame questi importanti cambiamenti. Per un elenco completo delle modifiche e della logica che ha portato ad attuarle, fate riferimento al documento *Security Enhanced (SE) Android: Bringing Flexible MAC to Android* degli autori originali del progetto SEAndroid (https://bitbucket.org/seandroid/manifests/).

Modifiche al kernel

Abbiamo già visto che SELinux è un modulo di sicurezza che implementa i vari hook LSM inseriti nei servizi kernel relativi al controllo di accesso agli oggetti. Il meccanismo IPC Binder di Android è anch'esso implementato come driver del kernel, ma visto che la sua

implementazione in origine non conteneva hook LSM, il suo comportamento in fase di esecuzione non poteva essere controllato da alcuna policy SELinux. Per aggiungere il supporto SELinux a Binder, sono stati inseriti hook LSM nel driver Binder, e al codice SELinux è stato aggiunto il supporto per la classe di oggetti `binder` e i permessi correlati.

Gli hook di sicurezza SELinux sono dichiarati in *include/linux/security.h*; il Listato 12.16 mostra le dichiarazioni correlate a Binder aggiunte per supportare Android.

Listato 12.16 Dichiarazioni degli hook di sicurezza di Binder in include/linux/security.h.

```
--altro codice--
int security_binder_set_context_mgr(struct task_struct *mgr);(1)
int security_binder_transaction(struct task_struct *from,
                                struct task_struct *  to);(2)
int security_binder_transfer_binder(struct task_struct *from,
                                    struct task_struct *to);(3)
int security_binder_transfer_file(struct task_struct *from,
                                  struct task_struct *to, struct file *file);(4)
--altro codice--
```

Il primo hook **(1)** controlla quali processi possono divenire gestori del contesto di Binder, mentre il secondo **(2)** controlla la capacità di un processo di richiamare una transazione Binder. Le due funzioni successive sono usate per stabilire chi può trasferire un riferimento Binder a un altro processo **(3)** e chi può trasferire un file aperto a un altro processo **(4)** utilizzando Binder.

Per consentire alla policy SELinux di impostare restrizioni per Binder, al kernel è stato aggiunto anche il supporto per la classe di oggetti `binder` e i suoi permessi (`impersonate`, `call`, `set_context_mgr` e `transfer`), come mostrato nel Listato 12.17.

Listato 12.17 Dichiarazione dei permessi e delle classi di oggetti Binder in selinux/include/classmap.h.

```
--altro codice--
struct security_class_mapping secclass_map[] = {
    --altro codice--
    {"binder", {"impersonate", "call", "set_context_mgr", "transfer", NULL} },
    { NULL }
  };
```

Modifiche allo userspace

Oltre alle modifiche al kernel, per l'integrazione di SELinux in Android sono state necessarie diverse modifiche ed estensioni dello userspace. Tra queste, le più importanti sono il supporto per l'etichettatura del file system nella libreria C core (bionica); le estensioni a `init`, ai file eseguibili e ai daemon core nativi; le API SELinux a livello di framework; i cambiamenti ai servizi del framework core per l'uso di SELinux. In questo paragrafo vediamo ogni modifica e la sua integrazione nel runtime di Android.

Librerie e strumenti

SELinux utilizza gli attributi estesi per memorizzare i contesti di protezione degli oggetti del file system, pertanto per prima cosa sono state aggiunte alla libreria C di Android le funzioni wrapper per le chiamate di sistema usate per gestire gli attributi estesi (`listxattr()`, `getxattr()`, `setxattr()` e così via), al fine di poter recuperare e impostare le etichette di sicurezza di file e directory.

Per sfruttare le funzionalità SELinux dallo userspace, SEAndroid ha aggiunto una porta compatibile con Android della libreria *libselinux*, nonché un set di comandi per gestire l'etichettatura, la policy di sicurezza e il passaggio tra le modalità Enforcing e Permissive di SELinux. Come gran parte delle utility a riga di comando di Android, gli strumenti SELinux sono implementati nel binario `toolbox` e sono installati come collegamenti simbolici a esso. La Tabella 12.1 riepiloga gli strumenti a riga di comando aggiunti o modificati.

Tabella 12.1 Utility a riga di comando SELinux.

Comando	Descrizione
`chcon`	Cambia il contesto di protezione di un file.
`getenforce`	Ottiene la modalità SELinux corrente.
`getsebool`	Ottiene i valori booleani della policy.
`id`	Visualizza il contesto di protezione di un processo.
`load_policy`	Carica un file di policy.
`ls -Z`	Visualizza il contesto di protezione di un file.
`ps -Z`	Visualizza il contesto di protezione dei processi in esecuzione.
`restorecon`	Ripristina il contesto di protezione di uno o più file.
`runcon`	Esegue un programma nel contesto di protezione specificato.

`setenforce`	Imposta la modalità di applicazione.
`setsebool`	Imposta il valore di una policy booleana.

Inizializzazione del sistema

Come nei sistemi Linux tradizionali, in Android tutti i programmi e i daemon dello userspace sono avviati dal processo `init`, il primo avviato dal kernel (PID=1). Tuttavia, a differenza di altri sistemi basati su Linux, gli script di inizializzazione di Android (`init.rc` e le sue varianti) non vengono interpretati da una shell di uso generale, ma da `init` stesso. Ogni script di inizializzazione contiene comandi incorporati che vengono eseguiti da `init` durante la lettura dello script. SEAndroid estende il linguaggio `init` di Android con una serie di nuovi comandi necessari per inizializzare SELinux e impostare i contesti di protezione di servizi e file, come riepilogato nella Tabella 12.2.

Tabella 12.2 Comandi predefiniti init per il supporto di SELinux.

Comando predefinito init	Descrizione
`seclabel`	Imposta il contesto di protezione di un servizio.
`restorecon`	Ripristina il contesto di protezione di un file o una directory.
`setcon`	Imposta il contesto di protezione del processo `init`.
`setenforce`	Imposta la modalità di applicazione.
`setsebool`	Imposta il valore di una policy booleana.

All'avvio, `init` carica la policy SELinux dal file di policy binario `/sepolicy` e imposta la modalità Enforcing in base al valore della proprietà di sistema `ro.boot.selinux` (che `init` imposta in base al valore del parametro da riga di comando del kernel `androidboot.selinux`). Se il valore della proprietà è `permissive`, SELinux passa alla modalità Permissive; quando è impostato su qualsiasi altro valore o se non è impostato, la modalità configurata è Enforcing.

A seguire, `init` carica ed esegue il parsing del file `init.rc` ed esegue i comandi specificati al suo interno. Il Listato 12.18 mostra un estratto di `init.rc`, incentrato sulle parti responsabili dell'inizializzazione di SELinux.

Listato 12.18 Inizializzazione di SELinux in init.rc.

```
--altro codice--
on early-init
    --altro codice--
    setcon u:r:init:s0(1)
```

```
    start ueventd
--altro codice--
on post-fs-data
    chown system system /data
    chmod 0771 /data
    restorecon /data(2)
--altro codice--
service ueventd /sbin/ueventd
    class core
    critical
    seclabel u:r:ueventd:s0(3)
--altro codice--
on property:selinux.reload_policy=1(4)
    restart ueventd
    restart installd
--altro codice--
```

In questo esempio `init` imposta il suo contesto di protezione utilizzando il comando `setcon` **(1)** prima di avviare i daemon del sistema core. Visto che un processo figlio eredita il contesto di protezione del padre, `init` imposta esplicitamente il contesto di protezione del daemon `ueventd` (il primo da avviare) su `u:r:ueventd:s0` **(3)** utilizzando il comando `seclabel`. Per la maggior parte degli altri servizi nativi, il dominio viene impostato automaticamente dalle regole di transizione dei tipi definite nella policy (Listato 12.10). Il comando `seclabel` è usato solo per impostare i contesti di protezione dei processi avviati nelle primissime fasi del processo di inizializzazione del sistema.

Quando vengono montati file system scrivibili, `init` usa il comando `restorecon` per ripristinare le etichette predefinite dei loro punti di mount, visto che un reset di fabbrica potrebbe avere eliminato le loro etichette. Il Listato 12.18 mostra il comando **(2)** che etichetta il punto di mount della partizione *userdata*, `/data`.

Infine, poiché l'impostazione della proprietà di sistema `selinux.reload_policy` a `1` **(4)** potrebbe provocare un nuovo caricamento della policy, `init` riavvia i daemon `ueventd` e `installd` quando questa proprietà è impostata, in modo che la nuova policy possa avere effetto.

Etichettatura dei file

Gli oggetti SELinux persistenti, come i file, dispongono di un contesto di protezione persistente che viene in genere salvato nell'attributo esteso di un file. In Android, il contesto di protezione iniziale di tutti i file è

definito in un file di testo chiamato `file_contexts`, simile a quello mostrato nel Listato 12.19.

Listato 12.19 Contenuti del file file_contexts.

```
/                           u:object_r:rootfs:s0(1)
/adb_keys                   u:object_r:rootfs:s0
/default.prop               u:object_r:rootfs:s0
/fstab\..*                  u:object_r:rootfs:s0
--altro codice--
/dev(/.*)?                  u:object_r:device:s0(2)
/dev/akm8973.*              u:object_r:akm_device:s0
/dev/accelerometer          u:object_r:accelerometer_device:s0
--altro codice--
/system(/.*)?               u:object_r:system_file:s0(3)
/system/bin/ash             u:object_r:shell_exec:s0
/system/bin/mksh            u:object_r:shell_exec:s0
--altro codice--
/data(/.*)?                 u:object_r:system_data_file:s0(4)
/data/backup(/.*)?          u:object_r:backup_data_file:s0
/data/secure/backup(/.*)?   u:object_r:backup_data_file:s0
--altro codice--
```

Come potete vedere, il file contiene un elenco di percorsi (che a volte usano caratteri jolly) con i contesti di protezione associati, ognuno su una riga separata. Il file `file_contexts` viene consultato spesso durante il processo di avvio e generazione di Android. Per esempio, visto che i file system in memoria come il file system root di Android (montato in */*) e il file system del dispositivo (montato in */dev*) non sono persistenti, tutti i file sono di solito associati allo stesso contesto di protezione specificato nel file `genfs_contexts` o assegnato utilizzando l'opzione di mounting `context=`. Per assegnare singoli contesti di protezione a file specifici di tali file system, `init` usa il comando `restorecon` per cercare il contesto di protezione di ogni file in `file_contexts` (**(1)** per il file system root e **(2)** come predefinito per il file system del dispositivo) e impostarlo di conseguenza. Durante il building di Android dal sorgente, il comando `make_ext4fs` consulta anche `file_contexts` per impostare i contesti iniziali dei file delle immagini nella partizione *system* (montata in */system* **(3)**) e nella partizione `userdata` (montata in */data* **(4)**). I contesti di protezione dei punti di mount delle partizioni dati sono anch'essi ripristinati a ogni boot (Listato 12.18) per garantire che siano in uno stato coerente. Infine, il sistema operativo di recovery di Android include una copia di `file_contexts` usata per impostare le corrette etichette dei file creati dal recovery durante gli aggiornamenti del sistema. Questo garantisce che il sistema resti in uno stato di

etichettatura sicura tra un aggiornamento e l'altro ed evita di dover riapplicare tutte le etichette dopo ogni aggiornamento.

Etichettatura delle proprietà di sistema

Android utilizza le proprietà di sistema globali, visibili a tutti i processi, per vari scopi, tra cui la comunicazione dello stato dell'hardware, l'avvio o l'arresto dei servizi di sistema, l'attivazione della crittografia del disco e persino il ricaricamento della policy SELinux. L'accesso alle proprietà di sistema di sola lettura non presenta restrizioni, ma visto che la modifica dei valori delle proprietà chiave di lettura/scrittura altera il comportamento del sistema, l'accesso in scrittura a queste proprietà è limitato e consentito solo ai processi di sistema in esecuzione con UID privilegiati, come *system* e *radio*. SEAndroid migliora questo controllo di accesso basato su UID aggiungendo regole MAC che comandano l'accesso in scrittura alle proprietà di sistema in base al dominio del processo che tenta di modificarle. Affinché questa strategia funzioni, le proprietà di sistema (che non sono oggetti SELinux nativi) devono essere associate a contesti di protezione. Per farlo i contesti di protezione delle proprietà vengono elencati in un file `property_contexts` con le stesse modalità con cui `file_contexts` specifica le etichette di sicurezza dei file. Il file viene caricato in memoria da `property_service` (parte di `init`); la tabella di ricerca dei contesti di protezione risultante viene usata per determinare se un processo dovrebbe poter accedere a una proprietà specifica in base ai contesti di protezione sia del processo (soggetto) sia della proprietà (oggetto). La policy SELinux definisce una nuova classe di oggetti `property_service`, con un unico permesso `set` usato per specificare le regole di accesso, come mostrato nel Listato 12.20.

Listato 12.20 Regole di accesso alle proprietà di sistema in vold.te.

```
type vold, domain;
--altro codice--
allow vold vold_prop:property_service set;(1)
allow vold powerctl_prop:property_service set;(2)
allow vold ctl_default_prop:property_service set;(3)
--altro codice--
```

In questo listato, il dominio `vold` può impostare le proprietà di sistema di tipo `vold_prop` **(1)**, `powerctl_prop` **(2)** e `ctl_default_prop` **(3)**.

Questi tipi sono associati alle proprietà effettive in base al nome della proprietà in `property_contexts`, come mostrato nel Listato 12.21.

Listato 12.21 Associazione dei nomi delle proprietà ai relativi contesti di protezione in property_contexts.

```
--altro codice--
vold.                   u:object_r:vold_prop:s0(1)
sys.powerctl            u:object_r:powerctl_prop:s0(2)
ctl.                    u:object_r:ctl_default_prop:s0(3)
--altro codice--
```

L'effetto di questa policy è che `vold` può impostare i valori di tutte le proprietà il cui nome inizia con `vold.` **(1)**, `sys.powerctl` **(2)** o `ctl.` **(3)**.

Etichettatura dei processi applicativi

Nel Capitolo 2 abbiamo visto che tutti i processi delle app in Android sono sottoposti a forking dal processo *zygote* per ridurre l'uso della memoria e migliorare il tempo di avvio delle applicazioni. Anche il processo *system_server*, che viene eseguito con l'utente `system` e ospita quasi tutti i servizi di sistema, viene sottoposto a forking da *zygote*, seppure con un'interfaccia leggermente diversa.

Il processo *zygote*, eseguito come root, è responsabile dell'impostazione delle credenziali DAC di ogni processo app (UID, GID e GID supplementari), nonché delle relative capability e dei limiti delle risorse. Per supportare SELinux, *zygote* è stato esteso per controllare il contesto di protezione dei suoi client (implementati nella classe `ZygoteConnection`) e impostare il contesto di protezione di ogni processo app sottoposto a forking. Il contesto di protezione è determinato in base alle regole di assegnazione specificate nel file di configurazione `seapp_contexts`, in base all'UID dell'app, al nome del package, a un flag che contrassegna il processo server di sistema e a un attributo stringa specifico per SELinux chiamato `seinfo`. Il file di configurazione `seapp_contexts` contiene regole di assegnazione del contesto di protezione (una per riga) che consistono di attributi dei selettori di input e attributi di output. Per ottenere una corrispondenza con una regola, tutti i selettori di input devono

corrispondere (AND logico). Il Listato 12.22 mostra il contenuto del file `seapp_contexts` nella policy SELinux Android di riferimento (versione 4.4.3).

NOTA

`seapp_contexts`, come tutti i file nella policy di riferimento, si trova nella directory *external/sepolicy/* del source tree di Android. Fate riferimento ai commenti nel file per un elenco completo dei selettori di input, delle regole di precedenza per la corrispondenza dei selettori e degli output.

Listato 12.22 Contenuti del file seapp_contexts.

```
isSystemServer=true domain=system (1)
user=system domain=system_app type=system_data_file(2)
user=bluetooth domain=bluetooth type=bluetooth_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=_app domain=untrusted_app type=app_data_file levelFrom=none(3)
user=_app seinfo=platform domain=platform_app type=platform_app_data_file(4)
user=_app seinfo=shared domain=shared_app type=platform_app_data_file(5)
user=_app seinfo=media domain=media_app type=platform_app_data_file
user=_app seinfo=release domain=release_app type=platform_app_data_file
user=_isolated domain=isolated_app(6)
user=shell domain=shell type=shell_data_file
```

La prima riga **(1)** in questo listato specifica il dominio del server di sistema (`system`), perché il selettore `isSystemServer` (utilizzato una sola volta) è impostato su `true`. Visto che Android usa un identificativo utente SELinux, un ruolo e un livello di sicurezza fissi, il contesto di protezione risultante diventa `u:r:system:s0`.

La seconda regola di assegnazione **(2)** crea una corrispondenza tra il selettore `user` e il nome utente del processo target, derivato dallo UID. Se un processo viene eseguito con uno degli utenti Linux Android predefiniti (`system`, `radio`, `nfc` e così via, come definito in `android_filesystem_config.h`), il nome associato viene utilizzato per la corrispondenza con il selettore `user`. Ai servizi isolati viene assegnata la stringa del nome utente `_isolated`, mentre agli altri processi viene associata la stringa del nome utente `_app`. Di conseguenza, alle app di sistema che corrispondono a questo selettore viene assegnato il dominio `system_app`.

L'attributo `type` specifica il tipo di oggetto assegnato ai file di proprietà del processo target: in questo caso il tipo è `system_data_file`, pertanto il contesto di protezione dei file di sistema diventa `u:object_r:system_data_file:s0`.

La regola **(3)** corrisponde a tutte le app eseguite con uno UID non di sistema e assegna i relativi processi al dominio `untrusted_app`. Alla directory dei dati privata di ogni app non attendibile viene assegnato in maniera

ricorsiva il tipo di oggetto `app_data_file`, generando il contesto di protezione `u:object_r:app_data_file:s0`. Il contesto di protezione della directory dati viene impostato dal daemon `installd` durante la sua creazione come parte del processo di installazione dell'app (fate riferimento al Capitolo 3).

Le regole **(4)** e **(5)** usano il selettore `seinfo` per distinguere le app non di sistema e assegnarle a domini diversi: ai processi app corrispondenti a `seinfo=platform` viene assegnato il dominio `platform_app`, a quelli che corrispondono a `seinfo=shared` il dominio `shared_app`. Come vedremo nel prossimo paragrafo, l'attributo `seinfo` di un'app è determinato dal suo certificato di firma, quindi in effetti le regole **(4)** e **(5)** usano il certificato di firma di ogni app come selettore del dominio di processo.

Infine, la regola **(6)** assegna il dominio `isolated_app` a tutti i servizi isolati (i servizi isolati sono eseguiti con uno UID separato dallo UID dell'app che li ospita e non possono accedere ad alcun servizio di sistema).

Middleware MAC

L'attributo `seinfo` introdotto nel paragrafo precedente fa parte della funzionalità di SEAndroid chiamata *Middleware MAC* (MMAC, uno schema di controllo di accesso di livello superiore separato dal MAC a livello di kernel (implementato nel modulo LSM di SELinux).

MMAC è stato progettato per fornire restrizioni MAC superiori al modello di permessi di Android, che opera a livello del framework e non può essere facilmente associato al MAC predefinito a livello del kernel. L'implementazione originale include una funzione MAC per la fase di installazione, che limita i permessi che si possono concedere a ogni package in base al nome di questo e al suo certificato di firma, indipendentemente dalla decisione di grant dei permessi presa dall'utente. In pratica, anche se un utente decide di concedere a un'app tutti i permessi richiesti, l'installazione può ancora essere bloccata da MMAC se la policy non consente il grant di determinate autorizzazioni.

L'implementazione MMAC di SEAndroid include anche una funzionalità MMAC di intent che usa una policy per controllare quali

intent possono essere scambiati tra le applicazioni. Un'altra funzionalità di SEAndroid è MMAC per i content provider, che definisce una policy per l'accesso ai dati del content provider. Ad ogni modo, l'implementazione MMAC originale di SEAndroid è stata integrata solo in parte in Android mainline, e l'unica funzionalità supportata è l'assegnazione di `seinfo` in base al certificato di firma dell'app.

NOTA

A partire dalla versione 4.3, Android dispone di una funzionalità sperimentale, detta *intent firewall*, che limita gli intent che possono essere inviati e ricevuti con le regole in stile "firewall". Questa funzionalità è simile a MMAC per gli intent di SEAndroid, ma non è integrata con l'implementazione di SELinux.

Il file di configurazione MMAC è chiamato `mac_permission.xml` e risiede nella directory */system/etc/security/* del dispositivo. Il Listato 12.23 mostra il template utilizzato per generare questo file, tipicamente memorizzato come *external/sepolicy/mac_permission.xml* nel source tree di Android.

Listato 12.23 Template per il file mac_permission.xml.

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

    <!-- Chiave di sviluppo della piattaforma in AOSP -->
    <signer signature="@PLATFORM" >(1)
      <seinfo value="platform" />
    </signer>

    <!-- Chiave di sviluppo del supporto in AOSP -->
    <signer signature="@MEDIA" >(2)
      <seinfo value="media" />
    </signer>

    <!-- Chiave di sviluppo condivisa in AOSP -->
    <signer signature="@SHARED" >(3)
      <seinfo value="shared" />
    </signer>

    <!-- Chiave di sviluppo della release in AOSP -->
    <signer signature="@RELEASE" >(4)
      <seinfo value="release" />
    </signer>

    <!-- Tutte le altre chiavi -->
    <default>(5)
      <seinfo value="default" />
    </default>

</policy>
```

Qui le macro `@PLATFORM` **(1)**, `@MEDIA` **(2)**, `@SHARED` **(3)** e `@RELEASE` **(4)** rappresentano i quattro certificati di firma della piattaforma usati in Android (*platform*, *media*, *shared* e *release*) e sono sostituite dai rispettivi certificati,

codificati come stringhe esadecimali, durante la generazione della policy SELinux.

Durante la scansione di ogni package installato, il servizio `PackageManagerService` di sistema confronta il suo certificato di firma con il contenuto del file `mac_permission.xml` e assegna il valore `seinfo` specificato al package se rileva una corrispondenza. Se non viene trovata alcuna corrispondenza, assegna il valore `default seinfo` come specificato dal tag `<default>` **(5)**.

File di policy del dispositivo

La policy SELinux di Android è costituita da un file di policy binario e da quattro file di configurazione di supporto, usati per l'etichettatura di processi, app, proprietà di sistema e file, nonché per l'inizializzazione di MMAC. La Tabella 12.3 mostra la posizione di questi file sul dispositivo e offre una breve descrizione del loro scopo e contenuto.

Tabella 12.3 File di policy SELinux di Android.

File di policy	Descrizione
`/sepolicy`	Policy del kernel binario.
`/file_contexts`	Contesti di protezione dei file, utilizzati per i file system di etichettatura.
`/property_contexts`	Contesti di protezione delle proprietà di sistema.
`/seapp_contexts`	Usato per derivare i contesti di protezione di file e processi applicativi.
`/system/etc/security/mac_permissions.xml`	Mappa i certificati di firma delle app ai valori `seinfo`.

NOTA

Le release di Android con SELinux precedenti alla versione 4.4.3 supportavano l'overriding dei file di policy predefiniti mostrati nella Tabella 12.3 con le relative controparti memorizzate nelle directory */data/security/current/* e */data/system/* (per il file di configurazione MMAC), al fine di consentire aggiornamenti online delle policy senza un aggiornamento OTA completo. Android 4.4.3 ha però rimosso questa funzionalità perché poteva creare discrepanze tra le etichette di sicurezza impostate nel file system e quelle referenziate dalla nuova policy. I file di policy vengono ora caricati dalle posizioni predefinite di sola lettura indicate nella Tabella 12.3.

Registrazione degli eventi delle policy

Le negazioni d'accesso e i grant di accesso corrispondenti alle regole `auditallow` vengono registrati nel buffer del log del kernel e possono essere

visualizzati con `dmesg`, come mostrato nel Listato 12.24.

Listato 12.24 Negazioni di accesso SELinux nel buffer del log del kernel.

```
# dmesg |grep 'avc:'
--altro codice--
<5>[18743.725707] type=1400 audit(1402061801.158:256): avc:  denied  { getattr
} for  pid=9574 comm="zygote" path="socket:[8692]" dev="sockfs" ino=8692
scontext=u:r:untrusted_app:s0 tcontext=u:r:zygote:s0 tclass=unix_stream_socket
--altro codice--
```

Qui il log di audit mostra che a un'applicazione di terze parti (contesto di protezione di origine `u:r:untrusted_app:s0`) è stato negato l'accesso al permesso *getattr* sul socket di dominio Unix *zygote* (contesto target `u:r:zygote:s0`, classe di oggetti `unix_stream_socket`).

Policy SELinux di Android 4.4

Android 4.2 è stata la prima release a contenere codice SELinux, ma SELinux fu disattivato nella fase di compilazione delle build da rilasciare. Android 4.3 ha abilitato SELinux in tutte le build, ma la sua modalità predefinita era impostata su Permissive. Inoltre, tutti i domini erano impostati singolarmente sulla modalità Permissive ed erano basati sul dominio `unconfined`, che concedeva loro accesso completo (nei confini del DAC) anche se la modalità SELinux globale era di tipo Enforcing.

Android 4.4 è stata la prima versione fornita con SELinux nella modalità Enforcing e includeva domini di enforcing per i daemon del sistema core. In questo paragrafo è disponibile una panoramica della policy SELinux di Android nella versione 4.4; vengono inoltre presentati alcuni dei principali domini che compongono la policy.

Informazioni generali sulle policy

Il codice sorgente della policy SELinux base di Android è ospitato nella directory *external/sepolicy/* del source tree di Android. Oltre ai file già presentati in questo capitolo (`access_vectors`, `file_contexts`, `mac_permissions.xml` e così via), il sorgente della policy contiene perlopiù istruzioni di *type enforcement* (TE) e regole suddivise tra più file `.te`, in genere uno per ogni dominio definito. Questi file vengono combinati per produrre il file di policy binario `sepolicy`, incluso nel root dell'immagine di boot come */sepolicy*. Potete esaminarlo utilizzando gli strumenti standard di SELinux come `seinfo`, `sesearch`, `sedispol` e così via. Per esempio, possiamo utilizzare il comando `seinfo` per ottenere un riepilogo del numero di oggetti e regole della policy, come mostrato nel Listato 12.25.

Listato 12.25 Interrogazione di un file di policy binario con il comando seinfo.

```
$ seinfo sepolicy

Statistics for policy file: sepolicy
Policy Version & Type: v.26 (binary, mls)

   Classes:            84    Permissions:       249
   Sensitivities:       1    Categories:       1024
   Types:             267    Attributes:         21
```

```
   Users:               1    Roles:              2
   Booleans:            1    Cond. Expr.:        1
   Allow:            1140    Neverallow:         0
   Auditallow:          0    Dontaudit:         36
   Type_trans:        132    Type_change:        0
   Type_member:         0    Role allow:         0
   Role_trans:          0    Range_trans:        0
   Constraints:        63    Validatetrans:      0
   Initial SIDs:       27    Fs_use:            14
   Genfscon:           10    Portcon:            0
   Netifcon:            0    Nodecon:            0
   Permissives:        42    Polcap:             2
```

Come potete vedere, la policy è piuttosto complessa: definisce `84` classi, `267` tipi e `1140` regole allow.

Potete ottenere ulteriori informazioni sugli oggetti della policy specificando opzioni di filtro nel comando `seinfo`. Per esempio, visto che tutti domini sono associati all'attributo `domain`, il comando del Listato 12.26 elenca tutti i domini definiti nella policy.

Listato 12.26 Recupero di un elenco di tutti i domini definiti con il comando seinfo.

```
$ seinfo -adomain -x sepolicy
   domain
      nfc
      platform_app
      media_app
      clatd
      netd
      sdcardd
      zygote
--altro codice--
```

Potete cercare le regole della policy utilizzano il comando `sesearch`. Per esempio, tutte le regole `allow` con il dominio `zygote` come origine possono essere visualizzate tramite il comando mostrato nel Listato 12.27.

Listato 12.27 Ricerca delle regole di policy con i comandi sesearch.

```
$ sesearch --allow -s zygote -d sepolicy
Found 40 semantic av rules:
   allow zygote zygote_exec : file { read execute execute_no_trans entrypoint open } ;
   allow zygote init : process sigchld ;
   allow zygote rootfs : file { ioctl read getattr lock open } ;
   allow zygote rootfs : dir { ioctl read getattr mounton search open } ;
   allow zygote tmpfs : filesystem mount ;
   allow zygote tmpfs : dir { write create setattr mounton add_name search } ;
--altro codice--
```

NOTA

Per i dettagli sulla creazione e la personalizzazione della policy di SELinux, consultate il documento *Validating Security-Enhanced Linux in Android* (http://bit.ly/1wbgJAM).

Applicazione dei domini

Anche se SELinux è implementato nella modalità Enforcing in Android 4.4, solo i domini assegnati ad alcuni daemon del core sono attualmente in modalità Enforcing, nello specifico `installd` (responsabile della creazione delle directory dati delle applicazioni), `netd` (responsabile della gestione di connessioni di rete e route), `vold` (responsabile del mounting della memoria esterna e dei contenitori sicuri) e `zygote`. Tutti questi daemon vengono eseguiti come root o ricevono capability speciali in quanto devono eseguire operazioni di amministrazione, come cambiare la proprietà di una directory (`installd`), manipolare le regole di routine e filtraggio dei pacchetti (`netd`), montare i file system (`vold`) e cambiare le credenziali del processo (`zygote`), per conto di altri processi.

Avendo privilegi elevati, questi daemon sono stati il bersaglio di vari exploit di escalation dei privilegi, che hanno consentito a processi non privilegiati di ottenere l'accesso root a un dispositivo. Di conseguenza, l'uso di una policy MAC restrittiva per i domini associati a questi daemon di sistema è un passo importante per rafforzare il modello di sicurezza sandboxing di Android e prevenire simili exploit in futuro.

Vediamo ora le regole di type enforcement definite per il dominio `installd` (in `instald.te`) per capire come SELinux limita i contenuti a cui i daemon di sistema possono accedere (Listato 12.28).

Listato 12.28 Policy di applicazione del tipo installd (da installd.te).

```
type installd, domain;
type installd_exec, exec_type, file_type;

init_daemon_domain(installd)(1)
relabelto_domain(installd)(2)
typeattribute installd mlstrustedsubject;(3)
allow installd self:capability { chown dac_override fowner fsetid setgid setuid };(4)
--altro codice--
allow installd dalvikcache_data_file:file create_file_perms;(5)
allow installd data_file_type:dir create_dir_perms;(6)
allow installd data_file_type:dir { relabelfrom relabelto };(7)
allow installd data_file_type:{ file_class_set } { getattr unlink };(8)
allow installd apk_data_file:file r_file_perms;(9)
--altro codice--
allow installd system_file:file x_file_perms;(10)
--altro codice--
```

In questo listato il daemon `installd` viene automaticamente portato in un dominio dedicato (denominato anch'esso `installd`) nella fase di avvio **(1)** grazie alla macro `init_daemon_domain()`. Viene quindi concesso il permesso `relabelto` affinché sia possibile impostare le etichette di sicurezza dei file e

delle directory creati **(2)**. A seguire il dominio viene associato all'attributo `mlstrustedsubject` **(3)**, che consente di aggirare le regole di accesso MLS; `installd` deve impostare il proprietario dei file e delle directory che crea sull'applicazione proprietaria, pertanto riceve `chown`, `dac_override` e altre capability pertinenti alla proprietà dei file **(4)**.

Come parte del processo di installazione delle app, `installd` attiva anche il processo di ottimizzazione DEX, che crea file ODEX nella directory */data/dalvik-cache/* (contesto di protezione `u:object_r:dalvikcache_data_file:s0`): ecco perché il daemon del programma di installazione riceve il permesso di creare file in tale directory **(5)**. A seguire, visto che `installd` crea directory dati private per le applicazioni nella directory */data/*, riceve il permesso di creare e rietichettare le directory (**(6)** e **(7)**); riceve poi gli attributi ed elimina i file **(8)** in */data/* (associata all'attributo `data_file_type`). `installd` deve inoltre leggere i file APK scaricati al fine di eseguire l'ottimizzazione DEX; per questo ottiene l'accesso ai file APK salvati in */data/app/* **(9)**, una directory associata al tipo `apk_data_file` (contesto di protezione `u:object_r:apk_data_file:s0`).

Per finire, `installd` ottiene il permesso di eseguire comandi di sistema (contesto di protezione `u:object_r:system_file:s0`) **(10)** al fine di avviare il processo di ottimizzazione DEX. Nel Listato 12.28 ne abbiamo omesse alcune, ma le restanti regole di policy seguono tutte lo stesso principio, concedendo a `installd` la quantità minima di privilegi necessaria per completare l'installazione del package. Di conseguenza, anche se il daemon è compromesso e un programma dannoso viene eseguito con i privilegi di `installd`, questo può avere accesso solo a un numero limitato di file e directory e si vedrà negare tutti i permessi non consentiti esplicitamente dalla policy MAC.

NOTA

Anche se Android 4.4 dispone solo di quattro domini di enforcing, con l'evoluzione della piattaforma e il perfezionamento della policy SELinux di base, è probabile che tutti i domini alla fine saranno implementati in questa modalità. Attualmente, per esempio, nella policy base della diramazione master dell'*Android Open Source Project* (AOSP), tutti i domini sono impostati come Enforcing nelle build di release e i domini Permissive sono usati solo nelle build di sviluppo.

Anche se un dominio è nella modalità Enforcing, può ottenere accesso senza restrizioni se è derivato da un dominio di base a cui sono concessi tutti o la maggior parte dei permessi di accesso. Nella policy SELinux di Android, tale dominio è `unconfineddomain`, di cui parleremo nel prossimo paragrafo.

Domini unconfined

La policy SELinux di Android contiene un dominio base (detto anche *template*) chiamato `unconfineddomain`, a cui vengono concessi quasi tutti i privilegi di sistema e che viene utilizzato come padre per gli altri domini della policy. In Android 4.4, `unconfineddomain` è definito come mostrato nel Listato 12.29.

Listato 12.29 Definizione di dominio unconfineddomain in Android 4.4.

```
allow unconfineddomain self:capability_class_set *;(1)
allow unconfineddomain kernel:security ~load_policy;(2)
allow unconfineddomain kernel:system *;
allow unconfineddomain self:memprotect *;
allow unconfineddomain domain:process *;(3)
allow unconfineddomain domain:fd *;
allow unconfineddomain domain:dir r_dir_perms;
allow unconfineddomain domain:lnk_file r_file_perms;
allow unconfineddomain domain:{ fifo_file file } rw_file_perms;
allow unconfineddomain domain:socket_class_set *;
allow unconfineddomain domain:ipc_class_set *;
allow unconfineddomain domain:key *;
allow unconfineddomain fs_type:filesystem *;
allow unconfineddomain {fs_type dev_type file_type}:{ dir blk_file lnk_file sock_file fifo_file }
~relabelto;
allow unconfineddomain {fs_type dev_type file_type}:{ chr_file file } ~{entrypoint relabelto};
allow unconfineddomain node_type:node *;
allow unconfineddomain node_type:{ tcp_socket udp_socket rawip_socket } node_bind;
allow unconfineddomain netif_type:netif *;
allow unconfineddomain port_type:socket_class_set name_bind;
allow unconfineddomain port_type:{ tcp_socket dccp_socket } name_connect;
allow unconfineddomain domain:peer recv;
allow unconfineddomain domain:binder { call transfer set_context_mgr };
allow unconfineddomain property_type:property_service set;
```

Come potete vedere, il dominio `unconfineddomain` ottiene tutte le capability del kernel **(1)**, accesso completo al server di protezione SELinux **(2)** (tranne per il caricamento della policy MAC), tutti i permessi relativi ai processi **(3)** e così via. Altri domini "ereditano" i permessi di questo dominio tramite la macro `unconfined_domain()`, che assegna l'attributo `unconfineddomain` al dominio passato come argomento. Nella policy SELinux di Android 4.4, anche tutti i domini Permissive sono "unconfined", e ottengono pertanto accesso senza restrizioni (entro i limiti del DAC).

NOTA

Anche se `unconfineddomain` esiste ancora nella diramazione master di AOSP, è stato considerevolmente limitato e non è più utilizzato come dominio senza restrizioni, ma come policy di base per i daemon di sistema e altri componenti Android privilegiati. Visto che sempre più domini stanno passando alla modalità Enforcing, con l'adeguamento delle relative policy, è probabile che in futuro `unconfineddomain` venga rimosso.

Domini delle app

Abbiamo già visto che SEAndroid assegna diversi domini ai processi applicativi in base al loro UID di processo o al loro certificato di firma. A questi domini applicativi vengono assegnati permessi comuni ereditando l'`appdomain` di base con la macro `app_domain()` che, come definito in `app.te`, include regole che consentono le operazioni comuni richieste da tutte le app di Android. Nel Listato 12.30 è mostrato un estratto del file `app.te`.

Listato 12.30 Estratto della policy appdomain (da app.te).

```
--altro codice--
allow appdomain zygote:fd use;(1)
allow appdomain zygote_tmpfs:file read;(2)
--altro codice--
allow appdomain system:fifo_file rw_file_perms;
allow appdomain system:unix_stream_socket { read write setopt };
binder_call(appdomain, system)(3)

allow appdomain surfaceflinger:unix_stream_socket { read write setopt };
binder_call(appdomain, surfaceflinger)(4)

allow appdomain app_data_file:dir create_dir_perms;
allow appdomain app_data_file:notdevfile_class_set create_file_perms;(5)
--altro codice--
```

Questa policy consente ad `appdomain` di ricevere e utilizzare i descrittori di file da `zygote` **(1)**, leggere le proprietà di sistema gestite da `zygote` **(2)**, comunicare con `system_server` tramite pipe, socket locali o Binder **(3)**, comunicare con il daemon `surfaceflinger` (responsabile del disegno sullo schermo) **(4)** e creare file e directory nella directory dati della sandbox **(5)**. Il resto della policy definisce regole che consentono altri permessi richiesti, come l'accesso alla rete, ai file scaricati e l'accesso di Binder ai servizi di sistema core. Le operazioni che le app in genere non richiedono, come l'accesso al dispositivo a blocchi raw, l'accesso alla memoria del kernel e le transizioni di dominio SELinux, sono esplicitamente vietate dalle regole `neverallow`.

I domini delle app concreti come `untrusted_app` (assegnato a tutte le applicazioni non di sistema in conformità alle regole di assegnazione in `seapp_contexts`, mostrato nel Listato 12.22) estendono `appdomain` e aggiungono altre regole di accesso in base alle necessità delle applicazioni target. Il Listato 12.31 mostra un estratto di `untrusted_app.te`.

Listato 12.31 Estratto della policy di dominio untrusted_app (da untrusted_app.te).

```
type untrusted_app, domain;
permissive untrusted_app;(1)
app_domain(untrusted_app)(2)
net_domain(untrusted_app)(3)
bluetooth_domain(untrusted_app)(4)

allow untrusted_app tun_device:chr_file rw_file_perms;(5)

allow untrusted_app sdcard_internal:dir create_dir_perms;
allow untrusted_app sdcard_internal:file create_file_perms;(6)

allow untrusted_app sdcard_external:dir create_dir_perms;
allow untrusted_app sdcard_external:file create_file_perms;(7)

allow untrusted_app asec_apk_file:dir { getattr };
allow untrusted_app asec_apk_file:file r_file_perms;(8)
--altro codice--
```

In questo file di policy, il dominio `untrusted_app` è impostato sulla modalità Permissive **(1)**, grazie alla quale eredita le policy di `appdomain` **(2)**, `netdomain` **(3)** e `bluetooth_domain` **(4)** tramite le rispettive macro. Il dominio ottiene quindi l'accesso ai dispositivi tunnel (usati per le VPN) **(5)**, alla memoria esterna (schede SD, **(6)** e **(7)**) e ai container di applicazioni crittografati **(8)**. Il resto delle regole (non mostrato) concede l'accesso a socket, pseudoterminali e ad altre risorse del sistema operativo necessarie.

Tutti gli altri domini delle app (`isolated_app`, `media_app`, `platform_app`, `release_app` e `shared_app` nella versione 4.4) ereditano sempre da `appdomain` e aggiungono altre regole `allow`, sia direttamente sia estendendo domini aggiuntivi. In Android 4.4, tutti i domini delle app sono impostati nella modalità Permissive.

NOTA

La policy SELinux nella diramazione master di AOSP semplifica la gerarchia dei domini delle app rimuovendo i domini dedicati `media_app`, `shared_app` e `release_app` e unendoli al dominio `untrusted_app`. Inoltre, solo il dominio `system_app` non presenta restrizioni (unconfined).

Riepilogo

A partire dalla versione 4.3, Android ha integrato SELinux per rafforzare il modello sandbox predefinito utilizzando il controllo di accesso obbligatorio (MAC) disponibile nel kernel di Linux. A differenza del controllo di accesso discrezionale (DAC) predefinito, MAC offre un modello specifico di permessi e oggetti, nonché una policy di sicurezza flessibile che non può essere sostituita o modificata da processi dannosi (a condizione che il kernel non sia compromesso).

Android 4.4 è la prima versione a portare SELinux nella modalità Enforcing nelle build di release; tuttavia, tutti i domini (tranne alcuni daemon core con privilegi elevati) sono impostati nella modalità Permissive per mantenere la compatibilità con le applicazioni esistenti. La policy base SELinux di Android continua a essere perfezionata a ogni release; è probabile che le release future impostino la maggior parte dei domini nella modalità Enforcing e rimuovano il dominio “unconfined”, attualmente ereditato da quasi tutti i domini associati a servizi privilegiati.

Aggiornamenti di sistema e accesso root

Nei capitoli precedenti abbiamo introdotto il modello di sicurezza di Android e spiegato come l'integrazione di SELinux ha rafforzato il sistema operativo. In questo capitolo ci occupiamo invece dei metodi che possono essere utilizzati per aggirare proprio il modello di sicurezza.

Per eseguire un aggiornamento completo del sistema operativo o per ripristinare il dispositivo alle impostazioni di fabbrica, è necessario aggirare la sandbox di sicurezza e ottenere l'accesso completo a un dispositivo, perché anche i componenti di Android con più privilegi non hanno accesso completo a tutte le partizioni di sistema e ai dispositivi di archiviazione. Inoltre, anche se il pieno accesso amministrativo (root) in fase di esecuzione va chiaramente contro la struttura di sicurezza di Android, l'esecuzione con privilegi root può essere utile al fine di implementare funzionalità non offerte da Android, come per esempio l'aggiunta di regole del firewall personalizzate o il backup completo del device (partizioni di sistema comprese). In verità, l'ampia disponibilità di build personalizzate (spesso chiamate *ROM*) e di applicazioni che consentono agli utenti di estendere o sostituire le funzionalità del sistema operativo utilizzando l'accesso root (comunemente noto come *app di root*) è stata una delle ragioni del grande successo di Android.

In questo capitolo ci occuperemo del design del bootloader e del sistema operativo di recovery di Android, mostrando come possono essere utilizzati per sostituire il software di sistema di un dispositivo. Vedremo poi com'è implementato l'accesso root sulle build di engineering e come le build di produzione possono essere modificate per consentire l'esecuzione di codice con privilegi di superuser installando

un’applicazione “superuser”. Infine, vedremo come le distribuzioni Android personalizzate implementano e controllano l’accesso root.

Bootloader

Un *bootloader* è un programma di basso livello eseguito all'accensione di un dispositivo; il suo scopo è inizializzare l'hardware, trovare e avviare il sistema operativo principale.

Come abbiamo visto brevemente nel Capitolo 10, i bootloader di Android sono di solito bloccati e consentono l'avvio o l'installazione solo di un'immagine del sistema operativo firmata dal produttore del dispositivo. Questo è un passo importante per stabilire un percorso di boot verificato, perché assicura che solo il software di sistema attendibile e non modificato possa essere installato su un device. Tuttavia, mentre la maggior parte degli utenti non è interessata a modificare il sistema operativo di base dei propri dispositivi, l'installazione di una build Android di terze parti è un'opportunità interessante, e può anche essere l'unico modo per eseguire una versione più recente di Android su dispositivi che hanno smesso di ricevere gli aggiornamenti del sistema operativo dal loro produttore. Ecco perché alcuni tra i dispositivi più recenti offrono un mezzo per sbloccare il bootloader e installare build Android di terze parti.

NOTA

Anche se i bootloader Android sono perlopiù a sorgente chiuso, quelli di quasi tutti i dispositivi ARM basati su SoC Qualcomm derivano dal bootloader *Little Kernel* (LK) (Code Aurora Forum, http://bit.ly/1u8AHaX), che è open source (http://bit.ly/1wLWdF8).

Nei prossimi paragrafi parleremo dell'interazione con i bootloader Android e di come è possibile sbloccare il bootloader sui dispositivi Nexus; descriveremo poi il protocollo fastboot utilizzato per aggiornare i dispositivi tramite il bootloader.

Sblocco del bootloader

I bootloader dei dispositivi Nexus possono essere sbloccati impartendo il comando `oem unlock` quando il dispositivo è nella modalità fastboot (di cui parliamo nel prossimo paragrafo). Pertanto, per sbloccare un dispositivo, è necessario prima avviarlo nella modalità fastboot, sia con il comando

`adb reboot bootloader` (se il device permette già l'accesso ADB) sia premendo una combinazione di tasti speciale durante il boot. Per esempio, tenendo contemporaneamente premuti i tasti Volume giù, Volume su e il tasto di accensione su un dispositivo Nexus 5 in fase di accensione si interrompe il normale processo di avvio e si apre la schermata di fastboot mostrata nella Figura 13.1.

Il bootloader ha una semplice interfaccia utente che può essere gestita con i tasti del volume e di accensione; consente agli utenti di continuare il processo di boot, riavviare il dispositivo in modalità fastboot o recovery e spegnere il dispositivo.

Il collegamento del dispositivo a un computer host tramite un cavo USB permette di inviare al dispositivo comandi aggiuntivi utilizzando lo strumento a riga di comando `fastboot` (parte dell'SDK di Android). Il comando `fastboot oem unlock` provoca la visualizzazione della schermata di conferma mostrata nella Figura 13.2.

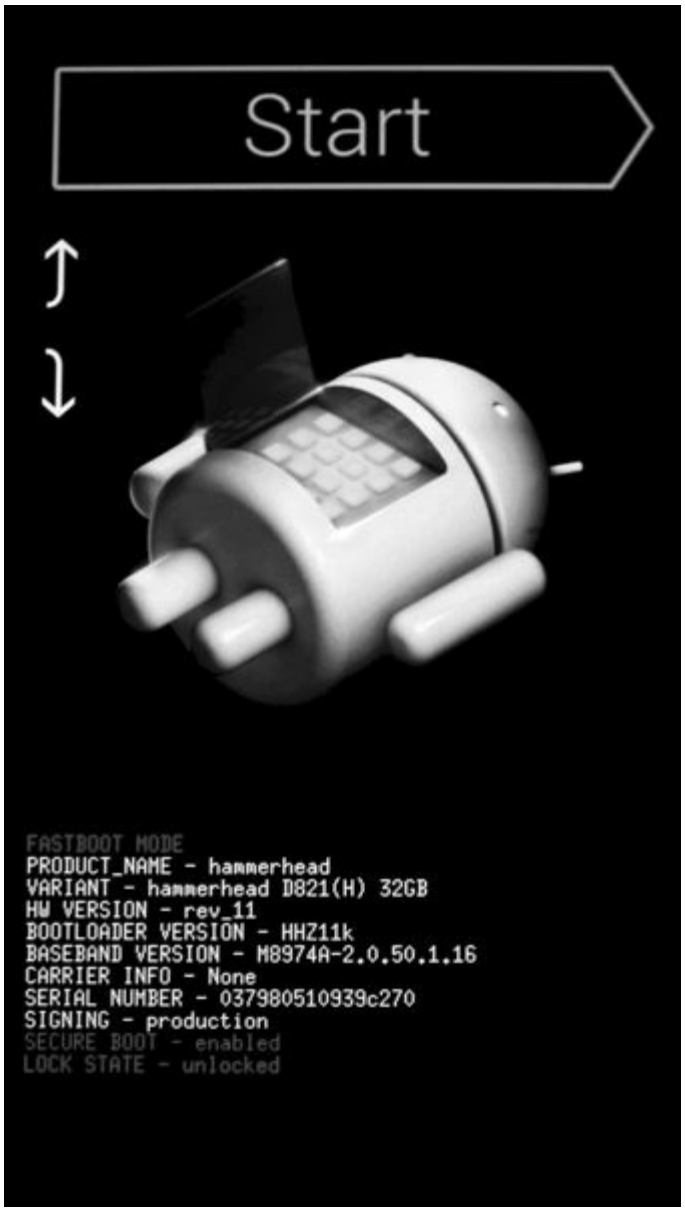

Figura 13.1 Schermata del bootloader di Nexus 5.

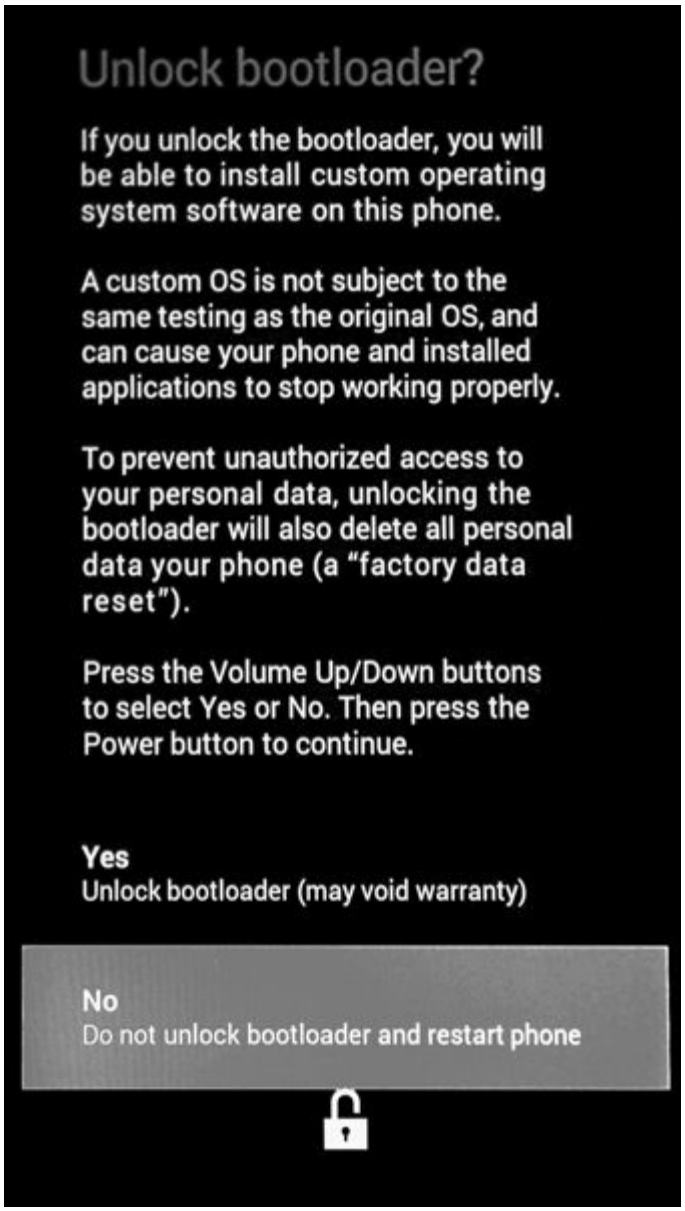

Figura 13.2 Schermata di sblocco del bootloader di Nexus 5.

La schermata di conferma avverte che lo sblocco del bootloader permette l'installazione di build del sistema operativo di terze parti non testate e provoca la cancellazione di tutti i dati utente. Visto che una build di terze parti potrebbe non seguire il modello di sicurezza di Android e potrebbe consentire l'accesso illimitato ai dati, la cancellazione di tutti i dati utente è una misura di sicurezza importante, che garantisce che non possano essere estratti dopo aver sbloccato il bootloader.

Il bootloader può essere nuovamente bloccato con il comando `fastboot oem lock`. Il blocco lo riporta al suo stato originale, dove il caricamento o l'avvio di immagini del sistema operativo di terze parti non è più possibile. Tuttavia, oltre a un flag bloccato/sbloccato, alcuni bootloader

conservano un flag aggiuntivo *tampered* (manomesso) che viene impostato al primo sblocco. Questo flag consente al bootloader di rilevare se è mai stato bloccato e di vietare alcune operazioni o di visualizzare un avviso anche se è in uno stato bloccato.

Modalità fastboot

Per quanto il protocollo e il comando `fastboot` possano essere usati per sbloccare il bootloader, il loro scopo originale era facilitare la cancellazione o la sovrascrittura delle partizioni del dispositivo inviando immagini di partizione al bootloader, che vengono scritte nel dispositivo a blocchi specificato. Questo è particolarmente utile durante il porting di Android su un nuovo dispositivo (si parla anche *di bring-up del device*) o il ripristino di un dispositivo allo stato di fabbrica mediante immagini di partizione fornite dal produttore.

Layout delle partizioni di Android

I dispositivi Android in genere dispongono di diverse partizioni, a cui fastboot fa riferimento per nome (piuttosto che con il corrispondente file di dispositivo Linux). Una lista delle partizioni con i relativi nomi può essere ottenuta elencando i file nella directory *by-name/* corrispondente al SoC del dispositivo in */dev/block/platform/*. Per esempio, visto che Nexus 5 è basato sul SoC Qualcomm (che include un processore baseband Mobile Station Modem, o MSM), la directory corrispondente è chiamata *msm_sdcc.1/*, come mostrato nel Listato 13.1 (timestamp omessi).

Listato 13.1 Elenco delle partizioni su un Nexus 5.

```
# ls -l /dev/block/platform/msm_sdcc.1/by-name
lrwxrwxrwx root     root              DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx root     root              aboot -> /dev/block/mmcblk0p6(1)
lrwxrwxrwx root     root              abootb -> /dev/block/mmcblk0p11
lrwxrwxrwx root     root              boot -> /dev/block/mmcblk0p19(2)
lrwxrwxrwx root     root              cache -> /dev/block/mmcblk0p27(3)
lrwxrwxrwx root     root              crypto -> /dev/block/mmcblk0p26
lrwxrwxrwx root     root              fsc -> /dev/block/mmcblk0p22
lrwxrwxrwx root     root              fsg -> /dev/block/mmcblk0p21
lrwxrwxrwx root     root              grow -> /dev/block/mmcblk0p29
lrwxrwxrwx root     root              imgdata -> /dev/block/mmcblk0p17
lrwxrwxrwx root     root              laf -> /dev/block/mmcblk0p18
lrwxrwxrwx root     root              metadata -> /dev/block/mmcblk0p14
lrwxrwxrwx root     root              misc -> /dev/block/mmcblk0p15(4)
```

```
lrwxrwxrwx root     root              modem -> /dev/block/mmcblk0p1(5)
lrwxrwxrwx root     root              modemst1 -> /dev/block/mmcblk0p12
lrwxrwxrwx root     root              modemst2 -> /dev/block/mmcblk0p13
lrwxrwxrwx root     root              pad -> /dev/block/mmcblk0p7
lrwxrwxrwx root     root              persist -> /dev/block/mmcblk0p16
lrwxrwxrwx root     root              recovery -> /dev/block/mmcblk0p20(6)
lrwxrwxrwx root     root              rpm -> /dev/block/mmcblk0p3
lrwxrwxrwx root     root              rpmb -> /dev/block/mmcblk0p10
lrwxrwxrwx root     root              sbl1 -> /dev/block/mmcblk0p2(7)
lrwxrwxrwx root     root              sbl1b -> /dev/block/mmcblk0p8
lrwxrwxrwx root     root              sdi -> /dev/block/mmcblk0p5
lrwxrwxrwx root     root              ssd -> /dev/block/mmcblk0p23
lrwxrwxrwx root     root              system -> /dev/block/mmcblk0p25(8)
lrwxrwxrwx root     root              tz -> /dev/block/mmcblk0p4
lrwxrwxrwx root     root              tzb -> /dev/block/mmcblk0p9
lrwxrwxrwx root     root              userdata -> /dev/block/mmcblk0p28(9)
```

Come potete osservare, Nexus 5 dispone di 29 partizioni, la maggior parte delle quali memorizza dati proprietari e specifici del dispositivo, come il bootloader di Android in *aboot* **(1)**, il software baseband in *modem* **(5)** e il bootloader della seconda fase in *sbl1* **(7)**. Il sistema operativo Android è ospitato nella partizione *boot* **(2)**, che memorizza il kernel e l'immagine del disco RAM *rootfs*, e nella partizione *system* **(8)**, che invece ospita tutti gli altri file di sistema. I file utente sono salvati nella partizione *userdata* **(9)**, mentre i file temporanei, come le immagini OTA scaricate e i log e i comandi del sistema operativo di recovery, sono memorizzati nella partizione *cache* **(3)**. Infine, l'immagine del sistema operativo di recovery risiede nella partizione *recovery* **(6)**.

Protocollo fastboot

Il protocollo fastboot opera tramite USB ed è guidato dall'host: in pratica, la comunicazione viene avviata dall'host, che usa il trasferimento di massa USB per inviare al bootloader dati e comandi basati su testo. Il client USB (bootloader) risponde con una stringa di stato come `OKAY` o `FAIL`, con un messaggio informativo che inizia con `INFO`, oppure con `DATA`, che indica che il bootloader è pronto ad accettare i dati inviati dall'host. Una volta ricevuti tutti i dati, il bootloader risponde con uno dei messaggi `OKAY`, `FAIL` o `INFO` descrivendo lo stato finale del comando.

Comandi fastboot

L'utility a riga di comando `fastboot` implementa il protocollo fastboot e consente di ottenere un elenco dei dispositivi connessi che supportano

fastboot (utilizzando il comando `devices`), di ottenere informazioni sul bootloader (con il comando `getvar`), di riavviare il dispositivo in varie modalità (con `continue`, `reboot` e `reboot-bootloader`) e di cancellare (con `erase`) o formattare (con `format`) una partizione.

Il comando `fastboot` supporta vari modi per scrivere un'immagine del disco in una partizione. È possibile eseguire il flashing di una singola partizione denominata con il comando `flash partition image-filename`; il flashing di più immagini di partizione contenute in un file ZIP può invece essere eseguito con `update ZIP-filename`.

Il comando `flashall` effettua automaticamente il flashing del contenuto dei file `boot.img`, `system.img` e `recovery.img` nella sua directory di lavoro per le partizioni *boot*, *system* e *recovery* del dispositivo. Infine, `flash:raw boot kernel ramdisk` crea automaticamente un'immagine di boot dal kernel e dal disco RAM specificati, effettuandone il flashing nella partizione *boot*. Oltre al flashing delle immagini di partizione, `fastboot` può essere usato anche per il boot di un'immagine senza la relativa scrittura sul disco: a tal fine è sufficiente utilizzare i comandi `boot boot-image` o `boot kernel ramdisk`.

I comandi che modificano le partizioni del dispositivo, come le diverse varianti di `flash`, e i comandi che effettuano il boot di kernel personalizzati, come il comando `boot`, non sono consentiti se il bootloader è bloccato.

Il Listato 13.2 mostra una sessione `fastboot` di esempio.

Listato 13.2 Sessione fastboot di esempio.

```
$ fastboot devices1
004fcac161ca52c5        fastboot
$ fastboot getvar version-bootloader2
version-bootloader: MAKOZ10o
finished. total time: 0.001s
$ fastboot getvar version-baseband3
version-baseband: M9615A-CEFWMAZM-2.0.1700.98
finished. total time: 0.001s
$ fastboot boot custom-recovery.img4
downloading 'boot.img'...
OKAY [  0.577s]
booting...
FAILED (remote: not supported in locked device)
finished. total time: 0.579s
```

Qui il primo comando **(1)** elenca i numeri di serie dei dispositivi connessi all'host che sono attualmente nella modalità fastboot. I comandi

ai punti **(2)** e **(3)** ottengono rispettivamente le stringhe di versione del bootloader e del baseband. Infine, il comando al punto **(4)** prova ad avviare un'immagine di recovery personalizzata, ma non riesce perché il bootloader è attualmente bloccato.

Recovery

Il *sistema operativo di recovery*, detto anche *console di recovery* o semplicemente *recovery*, è un sistema operativo minimale utilizzato per le attività che non possono essere eseguite direttamente da Android, come il ripristino delle impostazioni di fabbrica (cancellazione della partizione *userdata*) o l'applicazione di aggiornamenti OTA.

Analogamente alla modalità fastboot del bootloader, il sistema operativo di recovery può essere avviato premendo una combinazione di tasti durante l'avvio del dispositivo, oppure tramite ADB utilizzando il comando `adb reboot recovery`. Alcuni bootloader offrono anche un'interfaccia a menu (Figura 13.1) utilizzabile per avviare il recovery. Nei prossimi paragrafi esamineremo il recovery Android "stock" fornito con i dispositivi Nexus e incluso in AOSP, poi introdurremo i recovery personalizzati, che offrono funzionalità superiori ma necessitano di un bootloader sbloccato per l'installazione o il boot.

Recovery stock

Il recovery stock di Android implementa le funzionalità minime necessarie per soddisfare i requisiti della sezione "Updatable Software" dell'*Android Compatibility Definition Document* (CDD), che richiede che le implementazioni del dispositivo includano un meccanismo che sostituisca il software di sistema nella sua interezza e che il meccanismo di aggiornamento utilizzato supporti gli aggiornamenti senza cancellare i dati utente (https://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cdd.pdf).

Detto questo, il documento CDD non specifica il meccanismo di aggiornamento concreto da utilizzare, quindi sono possibili diverse strategie: il recovery stock implementa sia gli aggiornamenti OTA sia gli aggiornamenti in tethering. Per quelli OTA, il sistema operativo principale scarica il file di aggiornamento e chiede al recovery di

applicarlo; nel caso degli aggiornamenti in tethering, gli utenti scaricano il package di aggiornamento sul loro PC e lo inviano al recovery con il comando `adb sideload otafile.zip`. Il processo di aggiornamento effettivo per entrambi i metodi è lo stesso; cambia solo il modo in cui si ottiene il package OTA.

Il recovery stock presenta un'interfaccia a menu semplice (Figura 13.3) gestita con i tasti hardware del dispositivo, di solito quello di accensione e quelli del volume. Il menu è tuttavia nascosto per impostazione predefinita; per attivarlo occorre premere una combinazione di tasti dedicata. Sui dispositivi Nexus il menu di recovery di solito può essere visualizzato tenendo premuti contemporaneamente i tasti di accensione e Volume giù per qualche secondo.

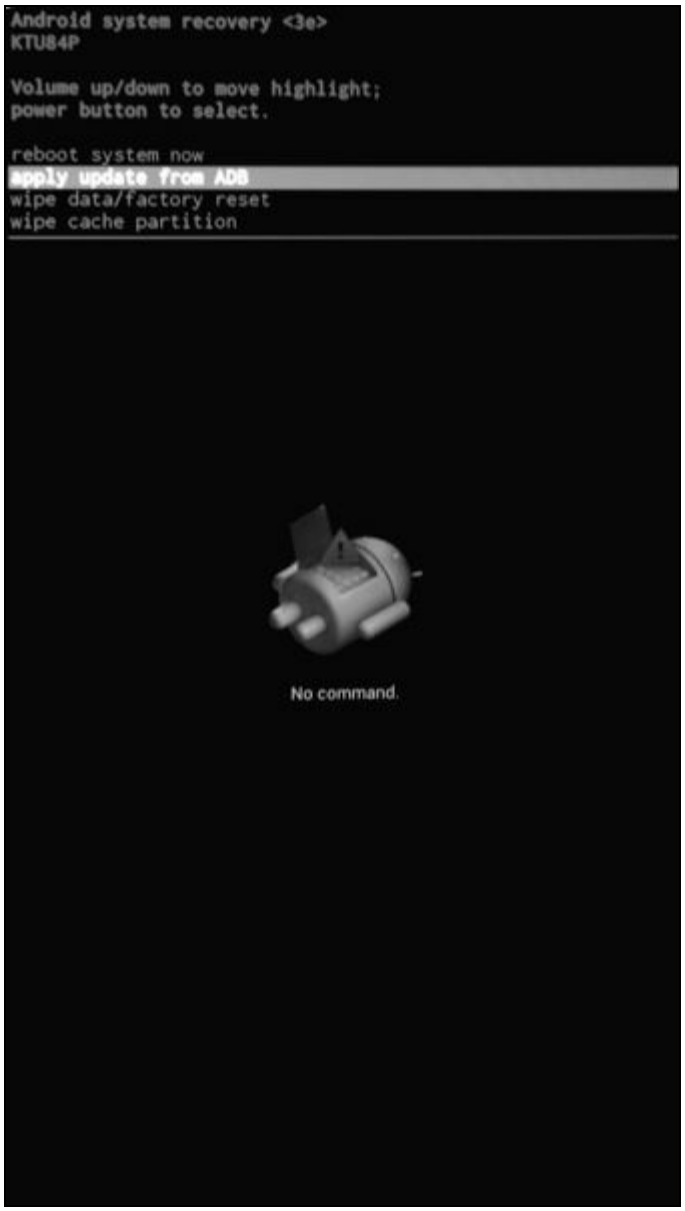

Figura 13.3 Menu del recovery stock.

Il menu di recovery del sistema contiene quattro opzioni: *reboot system now*, *apply update from ADB*, *wipe data/factory reset* e *wipe cache partition*. L'opzione *apply update from ADB* avvia il server ADB sul dispositivo e consente l'aggiornamento in tethering (*sideloading*). Tuttavia, come è facile notare, non è disponibile un'opzione per applicare un aggiornamento OTA, visto che nel momento in cui l'utente sceglie di applicarne uno dal sistema operativo principale (Figura 13.4), questo viene implementato automaticamente senza ulteriori interventi da parte dell'utente. Android ottiene questo risultato inviando comandi di controllo al recovery, eseguiti automaticamente all'avvio del recovery stesso (i meccanismi di controllo del recovery sono descritti nel prossimo paragrafo).

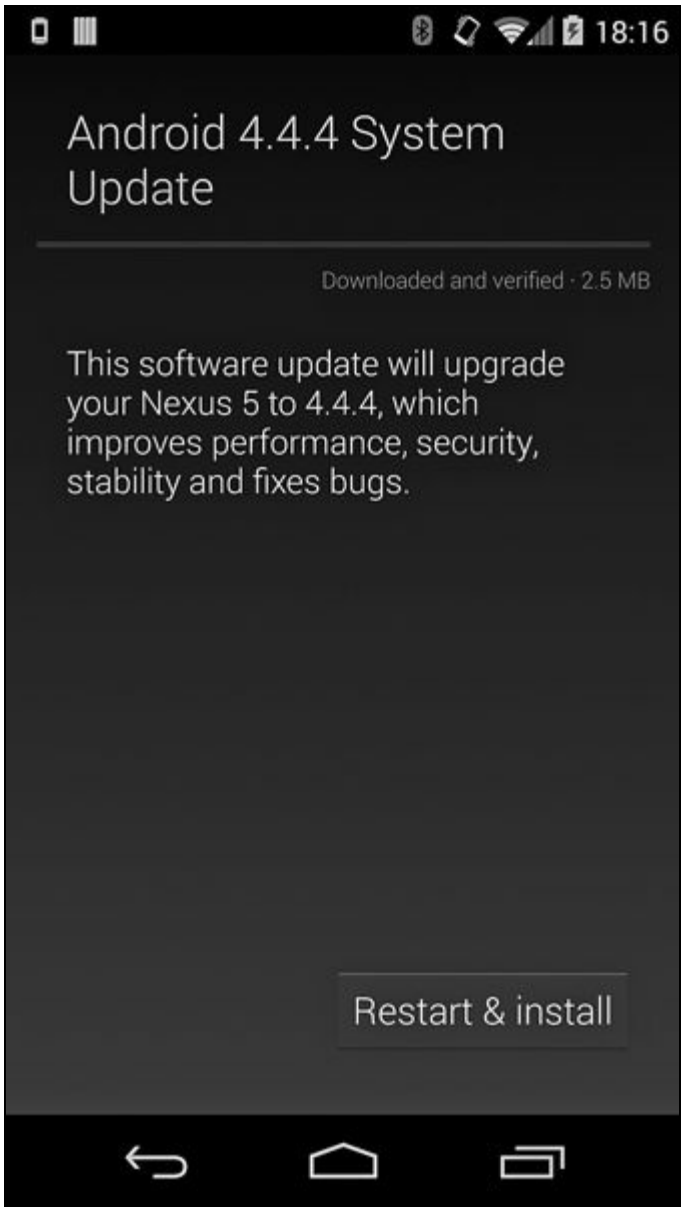

Figura 13.4 Applicazione di un aggiornamento del sistema dal sistema operativo principale.

Controllo del recovery

Il sistema operativo principale controlla il recovery tramite l'API `android.os.RecoverySystem`, che comunica con il recovery scrivendo stringhe di opzione, ciascuna su una riga diversa, nel file */cache/recovery/command*. Il contenuto del file `command` viene letto dal binario `recovery` (che si trova in */sbin/recovery* nel sistema operativo di recovery), avviato automaticamente da `init.rc` al boot del recovery. Le opzioni modificano il comportamento del binario `recovery` e provocano la cancellazione della partizione specificata, l'applicazione di un aggiornamento OTA o semplicemente il riavvio. La Tabella 13.1 mostra le opzioni supportate dal binario `recovery` stock.

Per garantire che i comandi specificati vengano sempre portati a termine, il binario `recovery` copia i suoi argomenti nel blocco di controllo del bootloader (BCB, *Bootloader Control Block*), ospitato nella partizione *misc* (**(4)** nel Listato 13.1). BCB è utilizzato per comunicare al bootloader lo stato attuale del processo di recovery. Il formato del BCB è specificato nella struttura `bootloader_message`, mostrata nel Listato 13.3.

Tabella 13.1 Opzioni per il binario di stock recovery.

Opzione di recovery	Descrizione
`--send_intent=<string>`	Salvare e comunicare al sistema operativo principale l'azione di intent specificata al termine dell'azione stessa.
`--update_package=<OTA package path>`	Verificare e installare il package OTA specificato.
`--wipe_data`	Cancellare le partizioni *userdata* e *cache* e riavviare il sistema.
`--wipe_cache`	Cancellare la partizione *cache* e riavviare il sistema.
`--show_text`	Messaggio da visualizzare.
`--just_exit`	Uscire e riavviare il sistema.
`--locale`	Impostazioni locali da usare per finestre e messaggi del recovery.
`--stages`	Impostare la fase corrente del processo di recovery.

Listato 13.3 Definizione della struttura del formato BCB.

```
struct bootloader_message {
    char command[32]; (1)
    char status[32]; (2)
    char recovery[768]; (3)
    char stage[32]; (4)
    char reserved[224]; (5)
};
```

Se un dispositivo viene riavviato o spento durante il processo di recovery, al successivo avvio il bootloader esamina il BCB e avvia nuovamente il recovery se il BCB contiene il comando `boot-recovery`. Se il processo viene completato con successo, il binario `recovery` cancella il BCB prima di uscire (impostandone tutti i byte a zero), affinché al riavvio successivo il bootloader avvii il sistema operativo Android principale.

Nel Listato 13.3, il comando al punto **(1)** è quello inviato al bootloader (solitamente `boot-recovery`), **(2)** è un file di stato scritto dal bootloader dopo aver eseguito un'azione specifica per la piattaforma, **(3)** contiene le opzioni per il binario `recovery` (`--update_package`, `--wipe-data` e così via), mentre **(4)** è una stringa che descrive la fase di installazione dei package OTA che richiedono più riavvii, per esempio *2/3* se l'installazione richiede tre riavvii. L'ultimo campo **(5)** è riservato, e al momento non è utilizzato.

Sideloading di un package OTA

Oltre al download da parte del sistema operativo principale, un package OTA può essere passato direttamente al recovery da un PC host. Per abilitare questa modalità di aggiornamento, l'utente deve scegliere l'opzione *apply update from ADB* nel menu di recovery. Viene così avviata una versione limitata del daemon ADB standard, che supporta unicamente il comando `sideload`. L'esecuzione di `adb sideload OTA-package-file` sull'host trasferisce il file OTA a */tmp/update.zip* sul dispositivo e lo installa (fate riferimento al paragrafo "Applicazione dell'aggiornamento").

Verifica della firma OTA

Come abbiamo visto nel Capitolo 3, i package OTA sono firmati a livello di codice; la firma è applicata all'intero file (a differenza dei file JAR e APK, che includono una firma separata per ogni file nell'archivio). Quando il processo OTA viene avviato dal sistema operativo Android principale, il package OTA (file ZIP) viene verificato con il metodo `verifyPackage()` della classe `RecoverySystem`. Questo metodo riceve

come parametri sia il percorso del package OTA sia un file ZIP contenente un elenco di certificati X.509 autorizzati a firmare gli aggiornamenti OTA. Se il package OTA è firmato con la chiave privata corrispondente a uno dei certificati nel file ZIP, è considerato valido e il sistema viene riavviato nella modalità di recovery per applicarlo. Se il file ZIP dei certificati non è specificato, viene utilizzato il file predefinito di sistema */system/etc/security/otacerts.zip*.

Il recovery verifica il package OTA che deve essere applicato indipendentemente dal sistema operativo principale al fine di garantire che tale package non sia stato sostituito prima dell'avvio del recovery. La verifica viene eseguita con un set di chiavi pubbliche integrate nell'immagine di recovery. Durante la creazione del recovery, queste chiavi vengono estratte dal set specificato di certificati di firma OTA, convertite nel formato mincrypt con lo strumento `DumpPublicKey` e scritte nel file */res/keys*. Se viene utilizzato RSA come algoritmo di firma, le chiavi sono strutture `RSAPublicKey` di mincrypt, serializzate come valori letterali C (ovvero come apparirebbero in un file sorgente C) e facoltativamente precedute da un identificatore di versione che specifica l'hash usato durante la firma del package OTA e l'esponente pubblico della chiave RSA per la chiave. Il file `keys` potrebbe essere simile a quello del Listato 13.4.

Listato 13.4 Contenuti del file /res/keys nel sistema operativo di recovery.

```
{64,0xc926ad21,{1795090719,...,3599964420},{3437017481,...,1175080310}},(1)
v2 {64,0x8d5069fb,{393856717,...,2415439245},{197742251,...,1715989778}},(2)
--altro codice--
```

La prima riga **(1)** è una chiave serializzata della versione 1 (implicita se non è specificato un identificatore di versione), che ha un esponente pubblico *e=3* e può essere utilizzata per verificare le firme create con SHA-1; la seconda riga **(2)** contiene una chiave della versione 2 con esponente pubblico *e=65537*, anch'essa utilizzata con le firme SHA-1. Gli algoritmi di firma attualmente supportati sono RSA a 2048 bit con SHA-1 (versioni 1 e 2 della chiave) o SHA-256 (versioni 3 e 4 della chiave), ECDSA con SHA-256 (versione della chiave 5, disponibile nella

diramazione master di AOSP) ed EC a 256 bit basato sulla curva NIST P-256.

Avvio del processo di aggiornamento del sistema

Se la firma del package OTA viene correttamente verificata, il recovery applica l'aggiornamento del sistema eseguendo il comando di aggiornamento incluso nel file OTA. Il comando di aggiornamento è salvato nella directory *META-INF/com/google/android/* dell'immagine di recovery come `update-binary` **(1)**, come mostrato nel Listato 13.5.

Listato 13.5 Contenuto di un package OTA per l'aggiornamento del sistema.

```
.
|-- META-INF/
|   |-- CERT.RSA
|   |-- CERT.SF
|   |-- com/
|   |   |-- android/
|   |   |   |-- metadata
|   |   |   '-- otacert
|   |   '-- google/
|   |       '-- android/
|   |           |-- update-binary(1)
|   |           '-- updater-script(2)
|   '-- MANIFEST.MF
|-- patch/
|   |-- boot.img.p
|   '-- system/
|-- radio.img.p
|-- recovery/
|   |-- etc/
|   |   '-- install-recovery.sh
|   '-- recovery-from-boot.p
'-- system/
    |-- etc/
    |   |-- permissions/
    |   |   '-- com.google.android.ble.xml
    |   '-- security/
    |       '-- cacerts/
    |-- framework/
    '-- lib/
```

Il recovery estrae `update-binary` dal file OTA in */tmp/update_binary* e lo avvia, passandogli tre parametri: la versione dell'API di recovery (attualmente la versione 3), il descrittore di file di un pipe che `update-binary` usa per comunicare l'avanzamento e i messaggi al recovery, e il percorso del package OTA. Il processo `update-binary` a sua volta estrae lo script di aggiornamento, incluso con il nome *META-INF/com/google/android/updater-script* **(2)** nel package OTA e lo valuta. Lo script di aggiornamento è scritto in un linguaggio dedicato chiamato *edify* (dalla versione 1.6; le versioni precedenti utilizzavano una variante

chiamata *amend*). Il linguaggio edify supporta semplici strutture di controllo come `if` ed `else`, ed è estensibile tramite funzioni che possono agire anche come strutture di controllo (decidendo quale argomento valutare). Lo script di aggiornamento include una sequenza di chiamate di funzione che attivano le operazioni necessarie per applicare l'aggiornamento.

Applicazione dell'aggiornamento

L'implementazione di edify definisce e registra varie funzioni usate per copiare, eliminare e applicare patch ai file, formattare e montare i volumi, impostare permessi dei file ed etichette SELinux e altro ancora. La Tabella 13.2 offre un riepilogo delle funzioni edify più utilizzate.

Tabella 13.2 Riepilogo delle funzioni edify più importanti.

Nome della funzione	Descrizione
`abort`	Interrompe il processo di installazione con un messaggio di errore.
`apply_patch`	Applica in modo sicuro una patch binaria. Garantisce che il file su cui è stata applicata la patch abbia il valore hash previsto prima di sostituire l'originale. Può anche correggere le partizioni del disco.
`apply_patch_check`	Verifica se un file ha il valore hash specificato.
`assert`	Controlla se una condizione è vera.
`delete/delete_recursive`	Elimina un file/tutti i file in una directory.
`file_getprop`	Recupera una proprietà di sistema dal file di proprietà specificato.
`format`	Formatta un volume con il file system specificato.
`getprop`	Recupera una proprietà di sistema.
`mount`	Monta un volume nel percorso specificato.
`package_extract_dir`	Estrae la directory ZIP specificata in un percorso del file system.
`package_extract_file`	Estrae il file ZIP specificato in un percorso del file system o lo restituisce come blob.
`run_program`	Esegue il programma specificato in un sottoprocesso e attende che finisca.
`set_metadata/set_metadata_recursive`	Imposta il proprietario, il gruppo, i bit dei permessi, le capability dei file e l'etichetta SELinux del file/di tutti i file in una directory.
`show_progress`	Segnala l'avanzamento al processo padre.
`symlink`	Crea un collegamento simbolico a un target, eliminando prima i file di collegamento simbolico esistenti.
`ui_print`	Invia un messaggio al processo padre.
`umount`	Smonta un volume montato.
`write_raw_image`	Scrive un'immagine raw nella partizione del disco specificata.

Nel Listato 13.6 è mostrato il contenuto (abbreviato) di un tipico script edify per l'aggiornamento del sistema.

Listato 13.6 Contenuto di updater-script in un package OTA per l'aggiornamento dell'intero sistema.

```
mount("ext4", "EMMC", "/dev/block/platform/msm_sdcc.1/by-name/system", "/system");
file_getprop("/system/build.prop", "ro.build.fingerprint") == "google/...:user/release-keys" ||
    file_getprop("/system/build.prop", "ro.build.fingerprint") == "google/...:user/release-keys"
||
    abort("Package expects build fingerprint of google/...:user/release-keys; this device has " +
    getprop("ro.build.fingerprint") + ".");
getprop("ro.product.device") == "hammerhead" ||
    abort("This package is for \"hammerhead\" devices; this is a \"" +
    getprop("ro.product.device") + "\".");(1)
--altro codice--
apply_patch_check("/system/app/BasicDreams.apk", "f687...", "fdc5...") ||
    abort("\"/system/app/BasicDreams.apk\" has unexpected contents.");(2)
set_progress(0.000063);
--altro codice--
apply_patch_check("EMMC:/dev/block/platform/msm_sdcc.1/by-name/boot:8835072:21...:8908800:a3...")
|| abort("\"EMMC:/dev/block/...\" has unexpected contents.");(3)
--altro codice--
ui_print("Removing unneeded files...");
delete("/system/etc/permissions/com.google.android.ble.xml",
       --altro codice--
       "/system/recovery.img");(4)
ui_print("Patching system files...");
apply_patch("/system/app/BasicDreams.apk", "-",
            f69d..., 32445,
            fdc5..., package_extract_file("patch/system/app/BasicDreams.apk.p"));(5)
--altro codice--
ui_print("Patching boot image...");
apply_patch("EMMC:/dev/block/platform/msm_sdcc.1/by-name/boot:8835072:2109...:8908800:a3bd...",
            "-", a3bd..., 8908800,
            2109..., package_extract_file("patch/boot.img.p"));(6)
--altro codice--
delete("/system/recovery-from-boot.p",
       "/system/etc/install-recovery.sh");
ui_print("Unpacking new recovery...");
package_extract_dir("recovery", "/system");(7)
ui_print("Symlinks and permissions...");
set_metadata_recursive("/system", "uid", 0, "gid", 0, "dmode", 0755, "fmode", 0644,
                       "capabilities", 0x0, "selabel", "u:object_r:system_file:s0");(8)
--altro codice--
ui_print("Patching radio...");
apply_patch("EMMC:/dev/block/platform/msm_sdcc.1/by-name/modem:43058688:7493...:46499328:52a...",
            "-", 52a5..., 46499328,
            7493..., package_extract_file("radio.img.p"));(9)
--altro codice--
unmount("/system");(10)
```

Copia e applicazione di patch ai file

Lo script di aggiornamento prima monta la partizione *system*, poi verifica se il modello del dispositivo e la sua build attuale sono quelli previsti **(1)**. Questa verifica è indispensabile perché il tentativo di installare un aggiornamento di sistema su una build incompatibile può lasciare il dispositivo in uno stato inutilizzabile (in questo caso spesso si parla di *soft brick*, poiché di solito è possibile ripristinare il dispositivo

effettuando un nuovo flashing di tutte le partizioni con una build funzionante; un *hard brick*, invece, non permette il ripristino).

Un aggiornamento OTA di solito non contiene i file di sistema completi, ma solo patch binarie rispetto alla versione precedente di ogni file modificato (realizzate con `bsdiff`; fate riferimento a Colin Percival, "Binary diff/patch utility", http://www.daemonology.net/bsdiff/); l'applicazione di un aggiornamento può quindi avere successo soltanto se ogni file a cui si applica la patch è identico al file usato per produrre la patch. Per ottenere questa garanzia, lo script di aggiornamento verifica che il valore hash di ogni file a cui applicare la patch sia quello previsto utilizzando la funzione `apply_patch_check` **(2)**.

Oltre ai file di sistema, il processo di aggiornamento corregge anche le partizioni che non contengono un file system, come *boot* e *modem*. Per garantire che l'applicazione di patch a tali partizioni abbia successo, lo script di aggiornamento controlla anche il contenuto delle partizioni target e interrompe l'operazione se lo stato non è quello previsto **(3)**. Dopo aver verificato tutte le partizioni e i file di sistema, lo script elimina i file non necessari e quelli che saranno sostituiti completamente (anziché corretti con la patch) **(4)**. Si procede quindi con l'applicazione di patch a tutti i file di sistema **(5)** e a tutte le partizioni **(6)**, nonché alla rimozione delle patch di recovery precedenti e all'inserimento del nuovo recovery in */system/* **(7)**.

Impostazione di proprietà, permessi ed etichette di protezione dei file

Il passo successivo richiede di impostare l'utente, il proprietario, i permessi e le capability di tutti i file e directory creati o modificati utilizzando la funzione `set_metadata_recursive` **(8)**. Dalla versione 4.3 Android supporta SELinux (Capitolo 12), pertanto tutti i file devono essere adeguatamente etichettati affinché le regole di accesso siano efficaci: ecco perché la funzione `set_metadata_recursive` è stata estesa per impostare l'etichetta

di sicurezza SELinux (l'ultimo parametro, `u:object_r:system_file:s0` in **(8)**) di file e directory.

Completamento dell'aggiornamento

A seguire, lo script di aggiornamento applica le patch al software baseband **(9)**, generalmente archiviato nella partizione *modem*. L'ultimo passo dello script è l'unmounting della partizione di sistema **(10)**.

Dopo l'uscita dal processo `update-binary`, il recovery cancella la partizione della cache, se è stato avviato con l'opzione `-wipe_cache`, e copia i log di esecuzione in */cache/recovery/*, affinché siano accessibili dal sistema operativo principale. Infine, se non vengono segnalati errori, il recovery cancella il BCB e riavvia il sistema operativo principale.

Se il processo di aggiornamento viene interrotto a causa di un errore, il recovery lo segnala all'utente, invitandolo a riavviare il dispositivo per riprovare. Visto che il BCB non è stato cancellato, il dispositivo si riavvia automaticamente nella modalità di recovery, e il processo di aggiornamento viene ricominciato da capo.

Aggiornamento del recovery

Esaminando nei dettagli lo script di aggiornamento nel Listato 13.6, è possibile notare che, oltre ad applicare patch alle partizioni *boot* **(6)** e *modem* **(9)** e a decomprimere una patch per la partizione *recovery* **(7)** (che ospita il sistema operativo di recovery), questa patch decompressa non viene applicata. La scelta è legata al tipo di design. Visto che un aggiornamento può essere interrotto in qualsiasi momento, il processo di aggiornamento deve essere riavviato dallo stesso stato in cui si trovava all'ultima accensione del dispositivo. Se per esempio si verifica un'interruzione dell'energia elettrica durante la scrittura della partizione *recovery*, l'aggiornamento cambierebbe lo stato iniziale e potrebbe lasciare il sistema in uno stato inutilizzabile. Per questo motivo, il sistema operativo di recovery viene aggiornato dal sistema operativo principale

solo quando l'aggiornamento di quest'ultimo è stato completato e il sistema stesso è stato avviato correttamente.

L'aggiornamento viene attivato dal servizio `flash_recovery` nel file `init.rc` di Android, mostrato nel Listato 13.7.

Listato 13.17 Definizione del servizio flash_recovery service in init.rc.

```
--altro codice--
service flash_recovery /system/etc/install-recovery.sh(1)
    class main
    oneshot
--altro codice--
```

Come potete vedere, questo servizio avvia semplicemente lo script della shell `/system/etc/install-recovery.sh` **(1)**. Lo script della shell, insieme a un file di patch per la partizione di recovery, viene copiato dallo script di aggiornamento OTA (**(7)** nel Listato 13.6) se il recovery richiede un aggiornamento. Il contenuto di `install-recovery.sh` è simile a quello mostrato nel Listato 13.8.

Listato 13.8 Contenuto di install-recovery.sh.

```
#!/system/bin/sh
if ! applypatch -c EMMC:/dev/block/platform/msm_sdcc.1/by-name/recovery:9506816:3e90...; then(1)
  log -t recovery "Installing new recovery image"
  applypatch -b /system/etc/recovery-resource.dat \
      EMMC:/dev/block/platform/msm_sdcc.1/by-name/boot:8908800:a3bd... \
      EMMC:/dev/block/platform/msm_sdcc.1/by-name/recovery \
      3e90... 9506816 a3bd...:/system/recovery-from-boot.p(2)
else
  log -t recovery "Recovery image already installed"(3)
fi
```

Lo script utilizza il comando `applypatch` per verificare se il sistema operativo di recovery deve essere modificato controllando il valore hash della partizione *recovery* **(1)**. Se l'hash della partizione *recovery* del dispositivo corrisponde a quello della versione con cui è stata creata la patch, lo script applica la patch **(2)**. Se il recovery è già stato aggiornato o ha un hash sconosciuto, lo script registra un messaggio ed esce **(3)**.

Recovery personalizzati

Un recovery personalizzato è una build del sistema operativo di recovery creata da una terza parte (non dal produttore del dispositivo). Essendo creato da terzi, non è firmato con le chiavi del produttore, pertanto il bootloader del dispositivo deve essere sbloccato per eseguirne

il boot o il flashing. Un recovery personalizzato può essere avviato senza installarlo sul dispositivo con `fastboot boot custom-recovery.img`, ma è possibile anche eseguirne il flashing permanente con il comando `fastboot flash recovery custom-recovery.img`.

Un recovery personalizzato fornisce funzionalità avanzate che in genere non sono disponibili nei recovery stock, come il backup e il ripristino delle partizioni complete, una shell di root con un set completo di utility di gestione del dispositivo, il supporto per il mounting di dispositivi USB esterni e così via. Può anche disabilitare la verifica della firma dei package OTA, che permette l'installazione di build del sistema operativo di terze parti o di modifiche quali le personalizzazioni del framework o del tema.

Sono disponibili svariati recovery personalizzati, ma attualmente quello più ricco di funzionalità e mantenuto in maniera di gran lunga più attiva è Team Win Recovery Project (http://teamw.in/project/twrp2/). È basato sul recovery stock AOSP ed è anche un progetto open source (https://github.com/TeamWin/Team-Win-Recovery-Project/). TWRP dispone di un'interfaccia touch screen, molto simile a quella nativa di Android, in grado di supportare i temi. Supporta inoltre i backup delle partizioni crittografate, l'installazione di aggiornamenti di sistema da dispositivi USB e il backup/ripristino su dispositivi esterni; dispone anche di un file manager integrato. La schermata iniziale di TWRP versione 2.7 è mostrata nella Figura 13.5.

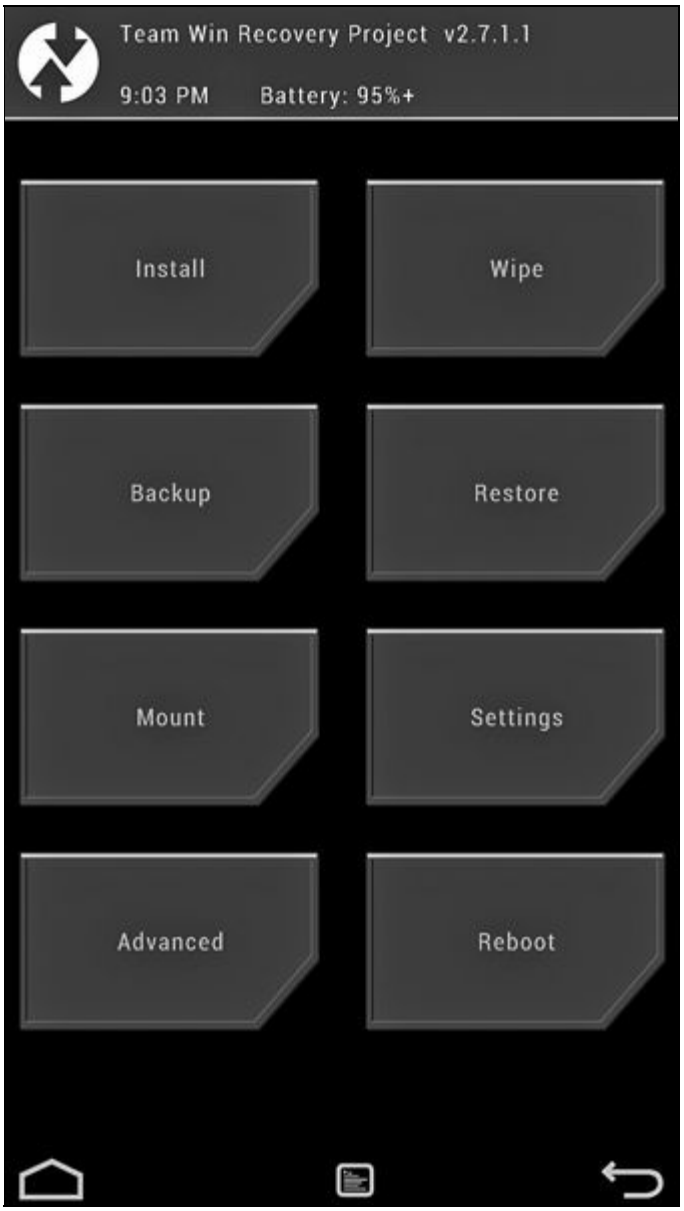

Figura 13.5 Schermata di avvio del recovery TWRP.

Analogamente al recovery stock AOSP, i recovery personalizzati possono essere controllati dal sistema operativo principale. Oltre al passaggio di parametri tramite il file */cache/recovery/command*, i recovery personalizzati di solito consentono di attivare alcune (o tutte) le loro funzionalità estese dal sistema operativo principale. Per esempio, TWRP supporta un linguaggio di scripting minimo che descrive le azioni di recovery che dovrebbero essere eseguite al boot del recovery: in questo modo le app Android possono accodare comandi di recovery utilizzando una comoda interfaccia grafica. A titolo di esempio, la richiesta di un backup compresso delle partizioni *boot*, *userdata* e *system* genera lo script mostrato nel Listato 13.9.

Listato 13.9 Esempio di script di backup TWRP.

```
# cat /cache/recovery/openrecoveryscript
backup DSBOM 2014-12-14--01-54-59
```

ATTENZIONE

Il flashing permanente di un recovery personalizzato che presenta un'opzione per ignorare le firme dei package OTA potrebbe consentire la sostituzione del software di sistema del dispositivo e l'installazione di backdoor che consentono l'accesso ai dispositivi. Si sconsiglia quindi di eseguire tale flashing su un dispositivo che si usa tutti i giorni e che contiene dati personali o informazioni sensibili.

Accesso root

Il modello di sicurezza di Android applica il principio detto del *least privilege* e isola i processi di sistema e delle app eseguendo ogni processo con un utente dedicato. Tuttavia, Android è anche basato su un kernel Linux che implementa un DAC stile Unix standard (tranne quando è abilitato SELinux, come spiegato nel Capitolo 12).

Uno dei grandi difetti di DAC è il fatto che a un utente del sistema, chiamato tipicamente *root* (UID=0) e detto anche *superuser*, viene concesso il potere assoluto sul sistema. L'utente root può leggere, scrivere e modificare i bit dei permessi di un file o di una directory, interrompere qualunque processo, montare e smontare i volumi e così via. Anche se tali permessi senza vincoli sono necessari per la gestione di un sistema Linux tradizionale, la disponibilità dell'accesso superuser su un dispositivo Android consente di aggirare efficacemente la sandbox di Android e di leggere o scrivere i file privati di qualunque applicazione.

L'accesso root permette anche di cambiare la configurazione del sistema attraverso la modifica di partizioni progettate per essere di sola lettura, l'avvio o l'arresto di servizi di sistema e la rimozione o la disabilitazione delle applicazioni di sistema core. Queste operazioni possono incidere negativamente sulla stabilità di un dispositivo, o addirittura renderlo inutilizzabile: ecco perché l'accesso root in genere non è consentito sui dispositivi di produzione.

Inoltre, Android tenta di limitare il numero di processi di sistema in esecuzione come root, perché un errore di programmazione in un processo del genere può aprire la porta ad attacchi di escalation dei privilegi, che a loro volta possono far sì che le applicazioni di terze parti ottengano l'accesso root. Con la distribuzione di SELinux nella modalità Enforcing, i processi sono limitati dalla policy di sicurezza globale; di conseguenza, la compromissione di un processo root non concede necessariamente accesso senza restrizioni a un dispositivo, ma può ancora permettere l'accesso ai dati sensibili o la modifica del comportamento del sistema. Inoltre, anche un processo vincolato da

SELinux può sfruttare una vulnerabilità del kernel per aggirare la policy di sicurezza o comunque ottenere accesso root senza restrizioni.

Detto questo, l'accesso root può essere molto comodo per il debug o il reverse engineering delle applicazioni sui dispositivi di sviluppo; inoltre, sebbene concedere tale accesso alle applicazioni di terze parti comprometta il modello di sicurezza di Android, consente anche varie personalizzazioni del sistema che non sono normalmente disponibili per l'esecuzione sui dispositivi di produzione.

Visto che uno dei punti di forza di Android è sempre stata la facilità di personalizzazione, la domanda di una flessibilità superiore tramite la modifica del sistema operativo core (operazione detta anche *modding*) è sempre stata alta, soprattutto nei primi anni. Oltre alla personalizzazione del sistema, l'accesso root su un dispositivo Android consente l'implementazione di applicazioni che non sono possibili senza la modifica del framework e l'aggiunta di servizi di sistema, quali firewall, backup completo del device, condivisione di rete e così via.

Nei prossimi paragrafi vedremo come è implementato l'accesso root nelle build Android di sviluppo (engineering) e in quelle Android personalizzate (ROM), nonché come è possibile aggiungerlo alle build di produzione. Mostreremo poi come le app che necessitano dell'accesso superuser (tipicamente chiamate *app root*) possono richiedere e utilizzare i privilegi root per eseguire i processi come root.

Accesso root sulle build di engineering

Il sistema di build di Android può produrre diverse varianti di build che differiscono per il numero di applicazioni e utility incluse, nonché per i valori di svariate proprietà di sistema che ne modificano il comportamento. Alcune di queste varianti consentono l'accesso root dalla shell di Android, come vedremo nei prossimi paragrafi.

Avvio di ADB come root

I dispositivi commerciali usano la variante di build *user* (impostata come valore della proprietà di sistema `ro.build.type`), che non include la

diagnostica e gli strumenti di sviluppo, per impostazione predefinita disabilita il daemon ADB, non permette il debug delle applicazioni per cui l'attributo `debuggable` non è impostato esplicitamente su `true` nei relativi manifest e vieta l'accesso root tramite la shell. La variante *userdebug* è molto simile a *user*, ma include alcuni moduli aggiuntivi (con tag di modulo *debug*), consente il debug di tutte le applicazioni e abilita ADB per impostazione predefinita.

Le build di engineering, o *eng*, includono gran parte dei moduli disponibili, consentono il debug, abilitano ADB di default e impostano la proprietà di sistema `ro.secure` a 0, cambiando il comportamento del daemon ADB in esecuzione su un dispositivo. Con l'impostazione 1 (modalità protetta), il processo `adbd`, inizialmente in esecuzione come root, elimina dal suo set di bounding tutte le capability tranne `CAP_SETUID` e `CAP_SETGID` (richieste per implementare l'utility `run-as`). Aggiunge poi diversi GID supplementari richiesti per accedere a interfacce di rete, memoria esterna e log di sistema, e infine cambia i suoi UID e GID in `AID_SHELL` (UID=2000). D'altro canto, se `ro.secure` è impostato a 0 (l'impostazione predefinita per le build di engineering), il daemon `adbd` continua a essere eseguito come root e dispone del set di bounding delle capability completo. Il Listato 13.10 mostra i process ID e le capability per il processo `adbd` in una build *user*.

Listato 13.10 Dettagli del processo adbd in una build utente.

```
$ getprop ro.build.type
user
$ getprop ro.secure
1
$ ps|grep adb
shell     200   1     4588   220   ffffffff 00000000 S /sbin/adbd
$ cat /proc/200/status
Name: adbd
State:      S (sleeping)
Tgid: 200
Pid:  200
Ppid: 1
TracerPid: 0
Uid:  2000  2000  2000  2000(1)
Gid:  2000  2000  2000  2000(2)
FDSize:     32
Groups:     1003 1004 1007 1011 1015 1028 3001 3002 3003 3006(3)
--altro codice--
CapInh:     0000000000000000
CapPrm:     0000000000000000
CapEff:     0000000000000000
CapBnd:     fffffff0000000c0(4)
--altro codice--
```

Come è facile osservare, l'UID **(1)** e il GID **(2)** del processo sono entrambi impostati a `2000` (`AID_SHELL`); inoltre, al processo `adbd` sono aggiunti diversi GID supplementari **(3)**. Per finire, il set di bounding delle capability del processo, che determina quali processi figlio di capability sono consentiti, è impostato a 0x0000000c0 (`CAP_SETUID|CAP_SETGID`) **(4)**. Questa impostazione garantisce che, nelle build *user*, i processi avviati dalla shell di Android siano limitati alle capability `CAP_SETUID` e `CAP_SETGID`, anche se per il binario eseguito è impostato il bit SUID o se le capability del file concedono privilegi aggiuntivi.

Al contrario, su una build *eng* o *userdebug*, il daemon ADB può essere eseguito come root, come mostrato nel Listato 13.11.

Listato 13.11 Dettagli del processo adbd in una build di engineering.

```
# getprop ro.build.type
userdebug(1)
# getprop ro.secure
1(2)
# ps|grep adb
root      19979 1     4656   264   ffffffff 0001fd1c S /sbin/adbd
root@maguro:/ # cat /proc/19979/status
Name: adbd
State:      S (sleeping)
Tgid: 19979
Pid:  19979
Ppid: 1
TracerPid:  0
Uid:  0     0     0     0(3)
Gid:  0     0     0     0(4)
FDSize:     256
Groups:(5)
--altro codice--
CapInh:     0000000000000000
CapPrm:     ffffffffffffffff(6)
CapEff:     ffffffffffffffff(7)
CapBnd:     ffffffffffffffff(8)
--altro codice--
```

Qui il processo `adbd` viene eseguito con UID **(3)** e GID **(4)** `0` (root), non dispone di gruppi supplementari **(5)** e possiede il set completo di capability Linux (**(6)**, **(7)** e **(8)**). Tuttavia, come è facile notare al punto **(2)**, la proprietà di sistema `ro.secure` è impostata a `1`, pertanto `adbd` non dovrebbe essere eseguito come root.

Anche se il daemon ADB elimina i suoi privilegi root nelle build *userdebug* (come in questo esempio al punto **(1)**), è possibile riavviarlo manualmente in modalità non protetta inviando il comando `adb root` da un host, come mostrato nel Listato 13.12.

Listato 13.12 Riavvio di adbd come root nelle build di debug utente.

```
$ adb shell id
uid=2000(shell) gid=2000(shell)(1)
groups=1003(graphics),1004(input),1007(log),1009(mount),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001
 context=u:r:shell:s0
$ adb root (2)
restarting adbd as root
$ adb shell ps|grep adb
root      2734  1     4644   216   ffffffff 0001fbec R /sbin/adbd(3)
$ adb shell id
uid=0(root) gid=0(root) context=u:r:shell:s0(4)
```

Qui il daemon `adbd` è inizialmente in esecuzione come `shell` (UID=2000); anche tutte le shell avviate dall'host hanno UID=2000 e GID=2000 **(1)**. L'invio del comando `adb root` **(2)** (che internamente imposta la proprietà di sistema `service.adb.root` a 1) provoca il riavvio del daemon ADB come root **(3)**; tutte le shell avviate successivamente avranno quindi UID e GUID=0 **(4)**.

NOTA

Visto che in questo particolare dispositivo è abilitato SELinux, anche cambiando UID e GID della shell il contesto di protezione (etichetta di sicurezza) rimane lo stesso, `u:r:shell:s0`, sia in **(1)** sia in **(4)**. Di conseguenza, anche dopo aver ottenuto una shell root tramite ADB, tutti i processi avviati dalla shell sono ancora vincolati dai permessi concessi al dominio `shell` (tranne qualora la policy MAC consenta la transizione a un altro dominio; fate riferimento al Capitolo 12 per i dettagli). In pratica, a partire da Android 4.4, il dominio `shell` è "unconfined"; durante l'esecuzione come root, i processi in questo dominio ottengono un controllo quasi completo sul dispositivo.

Uso del comando su

Nelle build *userdebug*, l'accesso root può essere ottenuto anche senza riavviare ADB come root: è sufficiente eseguire utilizzare il comando `su` (abbreviazione di *substitute user*, *switch user* o *superuser*), che viene installato con il bit SUID impostato e consente ai processi chiamanti di ottenere una shell root o di eseguire un comando con l'UID specificato (compreso UID=0). L'implementazione predefinita di `su` è molto semplice e può essere usata solo dagli utenti `root` e `shell`, come mostrato nel Listato 13.13.

Listato 13.13 Implementazione di su predefinita per le build di debug utente.

```
int main(int argc, char **argv)
{
    --altro codice--
    myuid = getuid();
    if (myuid != AID_ROOT && myuid != AID_SHELL) {(1)
        fprintf(stderr,"su: uid %d not allowed to su\n", myuid);
        return 1;
```

```
    }

    if(argc < 2) {
        uid = gid = 0;(2)
    } else {
       --altro codice--
    }

    if(setgid(gid) || setuid(uid)) {(3)
        fprintf(stderr,"su: permission denied\n");
        return 1;
    }

    --altro codice--

    execlp("/system/bin/sh", "sh", NULL);(4)

    fprintf(stderr, "su: exec failed\n");
    return 1;
}
```

Per prima cosa, la funzione principale controlla se l'UID del chiamante è `AID_ROOT` (0) o `AID_SHELL` (2000) **(1)**, uscendo se la chiamata proviene da un utente con un UID diverso. Imposta quindi UID e GID del processo a `0` (**(2)** e **(3)**) e avvia la shell di Android **(4)**. Eventuali comandi eseguiti da questa shell ereditano per impostazione predefinita i suoi privilegi, consentendo così l'accesso superuser al dispositivo.

Accesso root sulle build di produzione

Come è stato spiegato nel paragrafo “Accesso root sulle build di engineering”, i dispositivi Android commerciali sono di solito basati sulla variante di build *user*: questo significa che il daemon ADB è in esecuzione con l’utente `shell` e che sul dispositivo non è installato il comando `su`.

Questa è una configurazione sicura, che consente alla maggior parte degli utenti di svolgere le operazioni di configurazione e personalizzazione del dispositivo utilizzando gli strumenti forniti dalla piattaforma, o con applicazioni di terze parti come launcher, tastiere o client VPN personalizzati. Non sono tuttavia consentite le operazioni che modificano l’aspetto e il funzionamento né la configurazione core di Android, così come è vietato l’accesso di basso livello al sistema operativo Linux sottostante. Tali operazioni possono essere svolte solo eseguendo determinati comandi con privilegi root: è proprio per questo che molti utenti esperti cercano di consentire l’accesso root sui loro dispositivi.

L’operazione per ottenere l’accesso root su un dispositivo Android è comunemente detta *rooting*; può essere piuttosto semplice sui device che dispongono di un bootloader sbloccabile o quasi impossibile su quelli che non consentono lo sblocco del bootloader e adottano misure supplementari per prevenire le modifiche alla partizione di sistema. Nei prossimi paragrafi vedremo il tipico processo di rooting e introdurremo alcune delle più diffuse app “superuser” che abilitano e gestiscono l’accesso root.

Rooting mediante modifica dell’immagine di boot o di sistema

Su alcuni dispositivi Android, se il bootloader è sbloccato, è possibile trasformare facilmente una build *user* in una di engineering o *userdebug* effettuando il flashing di una nuova immagine di boot (spesso chiamata

kernel o *kernel personalizzato*); con questa operazione vengono cambiati i valori delle proprietà di sistema `ro.secure` e `ro.debuggable`. La modifica di queste proprietà permette l'esecuzione del daemon ADB come root e abilita l'accesso root tramite la shell di Android, come descritto nel paragrafo "Accesso root sulle build di engineering". Tuttavia, quasi tutte le build *user* correnti di Android disabilitano questo comportamento in fase di compilazione (non definendo la macro `ALLOW_ADBD_ROOT`), e i valori delle proprietà di sistema `ro.secure` e `ro.debuggable` vengono ignorati dal daemon `adbd`.

Un altro modo per consentire l'accesso root consiste nel decomprimere l'immagine di sistema, aggiungere un binario `su` SUID o un'utility simile e sovrascrivere la partizione *system* con la nuova immagine di sistema. In questo modo si ottiene accesso root non solo dalla shell, ma anche dalle applicazioni di terze parti. Tuttavia, alcuni miglioramenti alla protezione introdotti in Android 4.3 e versioni successive non consentono alle app di eseguire programmi SUID, eliminando tutte le capability dal set di bounding dei processi generati da Zygote e montando la partizione *system* con il flag `nosetuid` (http://source.android.com/devices/tech/security/enhancements43.html).

Inoltre, nelle versioni di Android in cui SELinux è nella modalità Enforcing, l'esecuzione di un processo con privilegi root cambia il contesto di protezione, anche se tale processo è tuttora limitato dalla policy MAC. Per questi motivi, l'abilitazione dell'accesso root su una versione recente di Android non è un'operazione semplice: non basta cambiare qualche proprietà di sistema o copiare un binario SUID nel dispositivo. Naturalmente, la sostituzione dell'immagine *boot* o *system* consente di disabilitare SELinux e tornare a una sicurezza ridotta, abbassando il livello di protezione del dispositivo e consentendo l'accesso root. Tuttavia, un approccio così radicale non è dissimile dalla sostituzione dell'intero sistema operativo e potrebbe impedire la ricezione di aggiornamenti di sistema dal produttore del dispositivo. Questa situazione è raramente desiderabile, e sono stati sviluppati diversi metodi

di rooting che provano a coesistere con il sistema operativo stock del dispositivo.

Rooting mediante flashing di un package OTA

Un package OTA può aggiungere o modificare i file di sistema senza sostituire l'intera immagine del sistema operativo, ed è quindi un buon candidato per l'aggiunta dell'accesso root a un dispositivo. Le app superuser più popolari sono distribuite combinando un package OTA, che deve essere installato una sola volta, con un'applicazione di gestione, che può essere aggiornata online.

SuperSU

Utilizzeremo il package OTA SuperSU per dimostrare come funziona questo metodo (Jorrit "Chainfire" Jongma, "CF-Root download page", http://download.chainfire.eu/supersu/) e l'app (Jorrit "Chainfire" Jongma, "Google Play Apps: SuperSU", http://bit.ly/1rerQ5D). SuperSU è attualmente l'applicazione superuser più popolare ed è mantenuta attivamente al passo con le più recenti modifiche alla piattaforma Android. Il package OTA SuperSU ha una struttura simile a quella di un package di aggiornamento dell'intero sistema e contiene un numero minimo di file, come mostrato nel Listato 13.14.

Listato 13.14 Contenuto del package OTA SuperSU.

```
.
|-- arm/(1)
|   |-- chattr
|   |-- chattr.pie
|   '-- su
|-- common/
|   |-- 99SuperSUDaemon(2)
|   |-- install-recovery.sh(3)
|   '-- Superuser.apk(4)
|-- META-INF/
|   |-- CERT.RSA
|   |-- CERT.SF
|   |-- com/
|   |   '-- google/
|   |       '-- android/
|   |           |-- update-binary(5)
|   |           '-- updater-script(6)
|   '-- MANIFEST.MF
'-- x86/(7)
    |-- chattr
    |-- chattr.pie
    '-- su
```

Il package contiene alcuni file binari nativi compilati per le piattaforme ARM **(1)** e x86 **(7)**, script per avviare e installare il daemon SuperSU (**(2)** e **(3)**), il file APK dell'applicazione GUI di gestione **(4)** e due script di aggiornamento (**(5)** e **(6)**) che applicano il package OTA.

Per comprendere il modo in cui SuperSU abilita l'accesso root dobbiamo prima esaminare il suo processo di installazione; per farlo, analizziamo il contenuto dello script `update-binary` **(5)**, mostrato nel Listato 13.15 (SuperSU usa un normale script della shell anziché un binario nativo, pertanto `updater-script` è semplicemente un segnaposto).

Listato 13.15 Script di installazione OTA SuperSU.

```
#!/sbin/sh
--altro codice--
ui_print "- Mounting /system, /data and rootfs"(1)
mount /system
mount /data
mount -o rw,remount /system
--altro codice--
mount -o rw,remount /
--altro codice--
ui_print "- Extracting files"(2)
cd /tmp
mkdir supersu
cd supersu
unzip -o "$ZIP"
--altro codice--
ui_print "- Placing files"
mkdir /system/bin/.ext
cp $BIN/su /system/xbin/daemonsu(3)
cp $BIN/su /system/xbin/su
--altro codice--
cp $COM/Superuser.apk /system/app/Superuser.apk(4)
cp $COM/install-recovery.sh /system/etc/install-recovery.sh(5)
cp $COM/99SuperSUDaemon /system/etc/init.d/99SuperSUDaemon
echo 1 > /system/etc/.installed_su_daemon
--altro codice--
ui_print "- Setting permissions"
set_perm 0 0 0777 /system/bin/.ext(6)
set_perm 0 0 $SUMOD /system/bin/.ext/.su
set_perm 0 0 $SUMOD /system/xbin/su
--altro codice--
set_perm 0 0 0755 /system/xbin/daemonsu
--altro codice--
ch_con /system/bin/.ext/.su(7)
ch_con /system/xbin/su
--altro codice--
ch_con /system/xbin/daemonsu
--altro codice--
ui_print "- Post-installation script"
/system/xbin/su --install(8)

ui_print "- Unmounting /system and /data"(9)
umount /system
umount /data

ui_print "- Done !"
exit 0
```

Per prima cosa lo script di aggiornamento monta il file system `rootfs` e le partizioni *system* e *userdata* nella modalità di lettura/scrittura **(1)**, poi estrae **(2)** e copia i file inclusi nelle posizioni previste sul file system. I binari nativi `su` e `daemonsu` **(3)** vengono copiati in */system/xbin/*, la posizione consueta per i binari nativi extra (ovvero non necessari per l'esecuzione del sistema operativo Android). L'applicazione di gestione dell'accesso root viene copiata in */system/app/* **(4)** ed è installata automaticamente dal package manager al riavvio del dispositivo. A seguire, lo script di aggiornamento copia lo script `install-recovery.sh` in */system/etc/* **(5)**.

NOTA

Come abbiamo visto nel paragrafo "Aggiornamento del recovery", questo script è normalmente utilizzato per aggiornare l'immagine di recovery dal sistema operativo principale. È quindi probabile che vi stiate chiedendo perché l'installazione di SuperSU tenta di aggiornare il recovery del device. SuperSU usa questo script per avviare alcuni dei suoi componenti in fase di boot; ne parleremo tra poco.

Il prossimo passaggio del processo di installazione del package OTA è l'impostazione dei permessi **(6)** e delle etichette di sicurezza SELinux **(7)** per i binari installati (`ch_con` è una funzione della shell che chiama l'utility SELinux `chcon` e imposta l'etichetta `u:object_r:system_file:s0`). Infine, lo script chiama il comando `su` con l'opzione `--install` **(8)** per eseguire un'inizializzazione post-installazione ed effettua l'unmounting di */system* e */data* **(9)**. All'uscita dallo script, il recovery riavvia il dispositivo con il sistema operativo principale.

Inizializzazione di SuperSU

Per capire la modalità di inizializzazione di SuperSU possiamo esaminare il contenuto dello script `install-recovery.sh` (Listato 13.16, commenti omessi), che viene eseguito automaticamente da `init` nella fase di boot.

Listato 13.16 Contenuto dello script install-recovery.sh di SuperSU.

```
#!/system/bin/sh
/system/xbin/daemonsu --auto-daemon &(1)

/system/etc/install-recovery-2.sh(2)
```

Lo script esegue per prima cosa il binario `daemonsu` **(1)**, che avvia un processo daemon con privilegi root. Nel passo successivo si esegue lo script `install-recovery-2.sh` **(2)**, che può essere utilizzato per eseguire un'inizializzazione supplementare necessaria per altre applicazioni root. L'uso di un daemon per consentire alle app di eseguire codice con privilegi root è necessario in Android 4.3 e versioni successive, perché tutte le app (sottoposte a forking da *zygote*) presentano un set di bounding delle capability azzerato, che impedisce loro di eseguire operazioni privilegiate anche se tentano di avviare un processo come root. Inoltre, a partire da Android 4.4, SELinux è nella modalità Enforcing, pertanto qualunque processo avviato da un'applicazione eredita il suo contesto di protezione (generalmente `untrusted_app`) ed è pertanto soggetto alle stesse restrizioni MAC dell'applicazione stessa.

SuperSU aggira queste restrizioni alla sicurezza facendo in modo che le app usino i binari `su` per eseguire comandi come root, ovvero inviando tali comandi tramite un socket di dominio Unix al daemon `daemonsu` che alla fine esegue i comandi ricevuti come root nel contesto SELinux `u:r:init:s0`. I processi in gioco sono illustrati nel Listato 13.17.

Listato 13.17 Processi avviati quando un'app richiede l'accesso root tramite SuperSU.

```
$ ps -Z
LABEL                          USER      PID   PPID  NAME
u:r:init:s0                    root      1     0     /init(1)
--altro codice--
u:r:zygote:s0                  root      187   1     zygote(2)
--altro codice--
u:r:init:s0                    root      209   1     daemonsu:mount:master(3)
u:r:init:s0                    root      210   209   daemonsu:master(4)
--altro codice--
u:r:init:s0                    root      3969  210   daemonsu:10292(5)
--altro codice--
u:r:untrusted_app:s0           u0_a292   13637 187   com.example.app(6)
u:r:untrusted_app:s0           u0_a209   15256 187   eu.chainfire.supersu(7)
--altro codice--
u:r:untrusted_app:s0           u0_a292   16831 13637 su(8)
u:r:init:s0                    root      16835 3969  /system/bin/sleep(9)
```

Qui l'app `com.example.app` **(6)** (il cui processo padre è `zygote` **(2)**) richiede l'accesso root passando un comando al binario `su` con la relativa opzione `-c`. Come potete osservare, il processo `su` **(8)** viene eseguito con lo stesso utente (`u0_a292`, UID=10292) e nello stesso dominio SELinux (`untrusted_app`) dell'app richiedente. Tuttavia, il processo **(9)** del comando che l'app ha

richiesto di eseguire come root (`sleep` in questo esempio) viene in realtà eseguito come root nel dominio SELinux `init` (contesto di protezione `u:r:init:s0`). Se tracciamo il suo PID padre (`PPID`, nella quarta colonna), possiamo osservare che il processo `sleep` è avviato dal processo `daemonsu:10292` **(5)**, un'istanza di `daemonsu` dedicata alla nostra app di esempio (con UID=10292). Il processo `daemonsu:10292` **(5)** eredita il suo dominio SELinux `init` dall'istanza `daemonsu:master` **(4)**, che a sua volta è avviata dalla prima interfaccia `daemonsu` **(3)**. Questa è l'istanza avviata tramite lo script `install-recovery.sh` (Listato 13.16) e viene eseguita nel dominio del relativo padre, il processo `init` **(1)** (PID=1).

Il processo `eu.chainfire.supersu` **(7)** appartiene all'applicazione di gestione SuperSU, che mostra la finestra di grant dell'accesso root mostrata nella Figura 13.6.

L'accesso superuser può essere concesso una sola volta, per un periodo prestabilito o a tempo indefinito. SuperSU mantiene una whitelist interna delle app a cui è stato concesso l'accesso root e non visualizza la finestra di grant se l'applicazione richiedente è già nella whitelist.

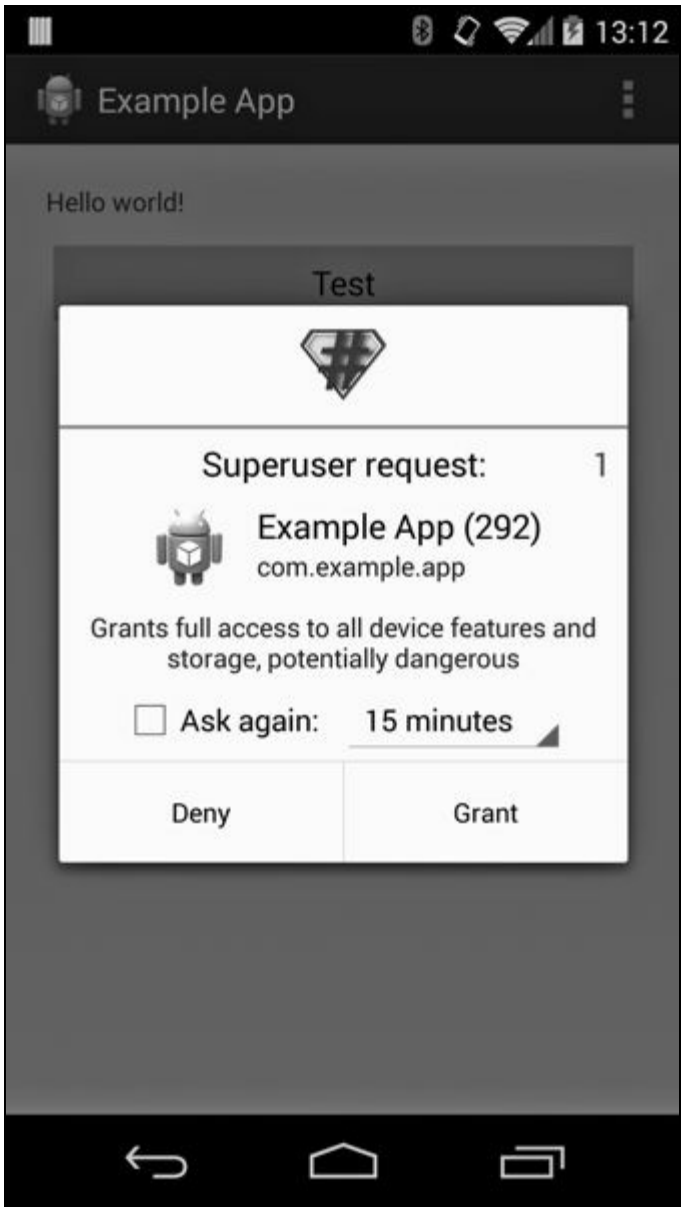

Figura 13.6 Finestra di grant della richiesta di accesso root per SuperSU.

NOTA

SuperSU dispone di una libreria associata, *libsuperuser* (Jorrit "Chainfire" Jongma, libsuperuser, https://github.com/Chainfire/libsuperuser/), che facilita la scrittura di app root fornendo i wrapper Java per i diversi schemi di chiamata del binario `su`. L'autore di SuperSU fornisce inoltre una guida completa alla scrittura di app root chiamata *How-To SU* (Jorrit "Chainfire" Jongma, libsuperuser, https://github.com/Chainfire/libsuperuser/).

Accesso root sulle ROM personalizzate

Le ROM personalizzate che consentono l'accesso root non devono passare attraverso `install-recovery.sh` per avviare il loro daemon superuser (equivalente a `daemonsu` di SuperSU), perché possono personalizzare il processo di avvio a piacere. Per esempio, la famosa distribuzione open

source di Android CyanogenMod avvia il suo daemon `su` da `init.superuser.rc`, come mostrato nel Listato 13.18.

Listato 13.18 Script di avvio per il daemon su in CyanogenMod.

```
service su_daemon /system/xbin/su --daemon(1)
    oneshot

on property:persist.sys.root_access=0(2)
    stop su_daemon

on property:persist.sys.root_access=2(3)
    stop su_daemon

on property:persist.sys.root_access=1(4)
    start su_daemon

on property:persist.sys.root_access=3(5)
    start su_daemon
```

Questo script `init` definisce il servizio `su_daemon` **(1)**, che può essere avviato o arrestato cambiando il valore della proprietà di sistema persistente `persist.sys.root_access` (da **(2)** a **(5)**). Il valore di questa proprietà determina anche se l'accesso root deve essere concesso solo alle applicazioni, solo alle shell ADB o a entrambe. L'accesso root è disabilitato per impostazione predefinita e può essere configurato con le opzioni di sviluppo di CyanogenMod, come mostrato nella Figura 13.7.

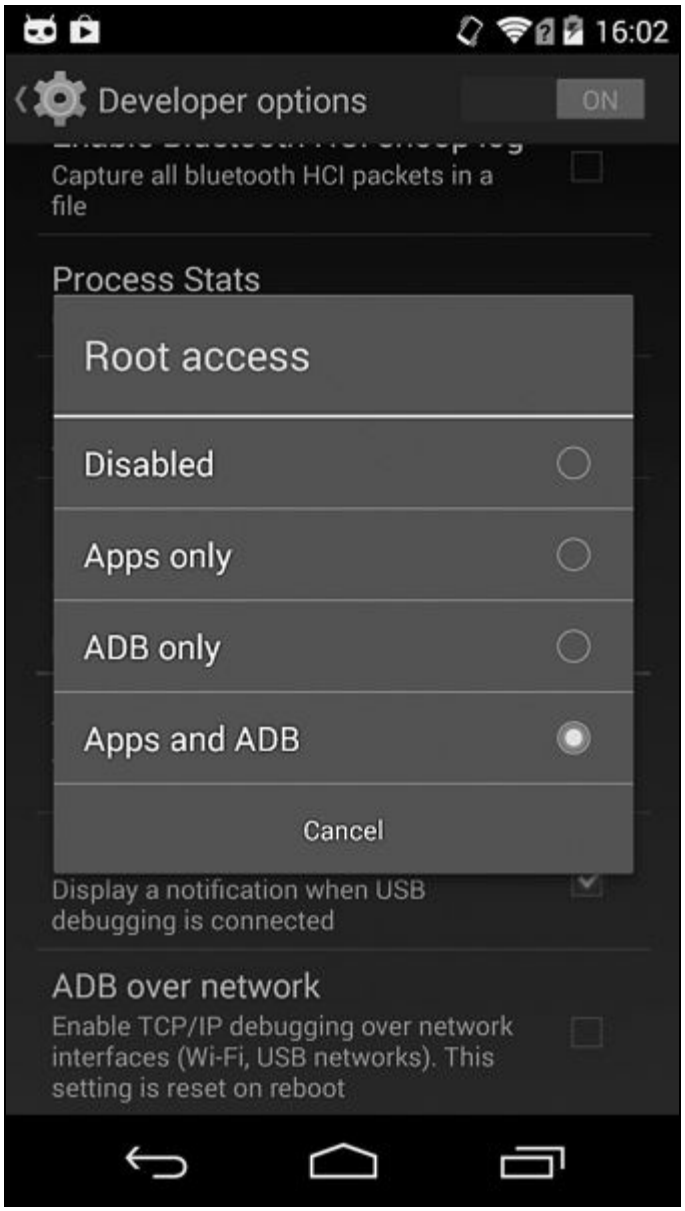

Figura 13.7 Opzioni di accesso root di CyanogenMod.

ATTENZIONE

Anche se SuperSU e le ROM personalizzate che consentono l'accesso root adottano misure specifiche per regolamentare le applicazioni che possono eseguire comandi come root (solitamente aggiungendole a una whitelist), una falla nell'implementazione potrebbe consentire alle app di aggirare queste misure e ottenere accesso root senza la conferma dell'utente. Di conseguenza, l'accesso root dovrebbe essere disabilitato sui dispositivi di uso quotidiano e impiegato solo quando necessario per lo sviluppo e il debug.

Rooting tramite exploit

Sui dispositivi di produzione che non dispongono di un bootloader sbloccabile, l'accesso root può essere ottenuto sfruttando una vulnerabilità di escalation dei privilegi, che consente a un'app o a un processo della shell di avviare una shell root (detta anche *soft root*) e di

modificare il sistema. Gli exploit sono tipicamente inseriti negli script o nelle app "one-click", che tentano di rendere persistente l'accesso root installando un binario `su` o modificando la configurazione del sistema. Per esempio, l'exploit towelroot (distribuito come app di Android) sfrutta una vulnerabilità nel kernel di Linux (CVE-2014-3153) per ottenere l'accesso root, e installa SuperSU per renderlo persistente. L'accesso root può essere reso permanente anche sovrascrivendo la partizione *recovery* con un recovery personalizzato, permettendo così l'installazione di software arbitrario (comprese le applicazioni superuser). Tuttavia, alcuni dispositivi dispongono di protezioni supplementari che impediscono le modifiche alle partizioni *boot*, *system* e *recovery*; in tal caso, l'accesso root permanente potrebbe essere impossibile.

NOTA

Consultate il Capitolo 3 di *Android Hacker's Handbook* (Wiley, 2014) per una descrizione dettagliata delle principali vulnerabilità di escalation dei privilegi utilizzate per ottenere l'accesso root nelle diverse versioni di Android. Il Capitolo 12 dello stesso libro introduce le più importanti tecniche di attenuazione degli exploit implementate in Android al fine di prevenire gli attacchi di escalation dei privilegi e in generale per rafforzare il sistema.

Riepilogo

Per consentire l'aggiornamento del software di sistema o il ripristino di un dispositivo allo stato di fabbrica, i device Android permettono un accesso di basso livello e senza restrizioni alla loro memoria attraverso il bootloader. Il bootloader in genere implementa un protocollo di gestione, solitamente fastboot, che consente il trasferimento e il flashing delle immagini delle partizioni da una macchina host. I bootloader sui dispositivi di produzione sono di solito bloccati e consentono il flashing solo di immagini firmate; tuttavia, la maggior parte di essi può essere sbloccata, permettendo così il flashing di immagini di terze parti.

Android usa una partizione dedicata per memorizzare un secondo sistema operativo minimale, detto recovery, impiegato per applicare package di aggiornamento OTA o per cancellare tutti i dati sul dispositivo. Analogamente ai bootloader, i recovery dei dispositivi di produzione permettono solamente di applicare i package OTA firmati dal produttore del dispositivo. Se il bootloader è sbloccato, è possibile avviare o installare in via definitiva un recovery personalizzato, che consente l'installazione di aggiornamenti firmati da terze parti o di evitare del tutto la verifica della firma.

Le build di engineering o debug di Android consentono l'accesso root dalla shell di Android, ma tale accesso è normalmente disabilitato sui dispositivi di produzione. L'accesso root su tali dispositivi può essere abilitato installando un package OTA di terze parti che include un daemon "superuser" e un'applicazione associata che permette l'accesso root controllato alle applicazioni. Le build Android di terze parti (ROM) di solito consentono l'accesso root immediato, anche se è possibile disabilitarlo dall'interfaccia delle impostazioni di sistema.