UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

CORSO DI LAUREA TRIENNALE IN INFORMATICA

ANALISI DI SICUREZZA DEL SISTEMA ANDROID

Relatore: Prof. Danilo BRUSCHI
Correlatore: Dott. Alessandro REINA

Tesi di Laurea di:
Mattia Giovanni CAMPANA
Matricola 730232

Anno Accademico 2009–10

A mia madre, la persona a cui devo tutto.

Indice

Capitolo 1

Introduzione

Con il passare del tempo, l'evoluzione tecnologica che ha accompagnato lo sviluppo dei normali PC, ha coinvolto i dispositivi così detti "mobili". Evoluzione che li ha trasformati da semplici agende elettroniche a veri e propri terminali ricchi di funzionalità, discreta potenza di calcolo ma soprattutto di connettività. Quest'ultima caratteristica li ha resi estremamente versatili soprattutto per applicazioni di tipo aziendale e di produttività personale.

Negli ultimi anni gli smartphone (e di recente anche i Tablet) hanno ricoperto sempre più un ruolo dominante nel panorama dei dispositivi mobili per la loro principale caratteristica di possedere dimensioni ridotte e per la capacità di offrire servizi di vario genere (geolocalizzazione, ricezione/invio di email, acquisizione di immagini e filmati, navigazione internet, etc.).

In questo contesto sono state realizzate diverse piattaforme, ciascuna con le proprie caratteristiche, per la gestione di tali dispositivi. Si tratta però di sistemi e tecnologie per lo più proprietarie da cui si differenzia solo un ristretto numero di sistemi open source tra cui quello esaminato in questo lavoro di tesi, Android.

Come è possibile notare dalla Figura 1.1 in poco tempo Android ha quasi raggiunto (e superato) la popolarità del suo diretto concorrente, l'iOS di Apple, grazie alla sua caratteristica di essere un sistema open source e quindi adattabile più facilmente a diverse tipologie di dispositivi.

L'aumento di popolarità implica di conseguenza un sempre maggior numero di

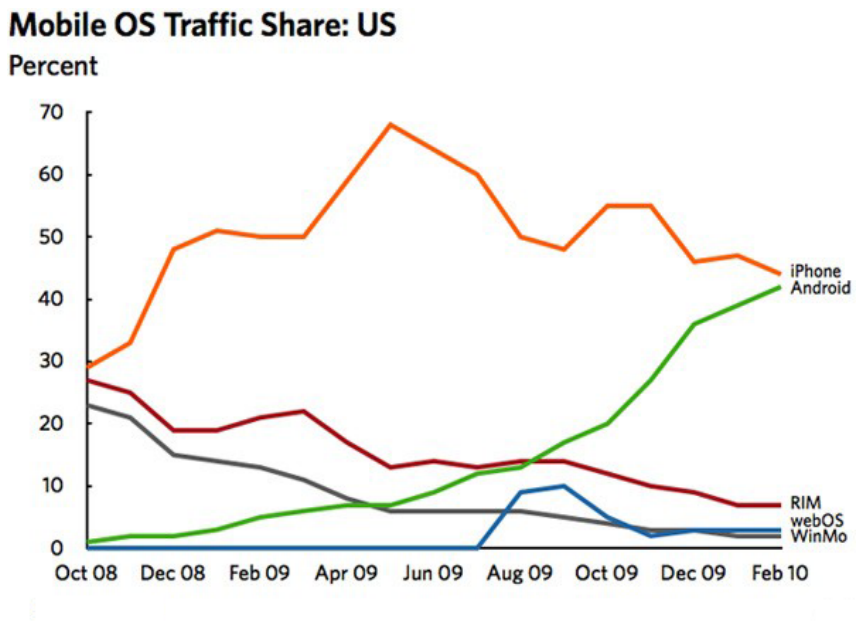

Figura 1.1: Diffusione dei principali sistemi per dispositivi mobili. Fonte: AdMob.

applicazioni disponibili e quindi di sviluppatori; di conseguenza vi è una maggior possibilità di individuare all'interno delle applicazioni stesse la presenza di malware ideati per eseguire operazioni illecite come il furto di informazioni personali a scopi commerciali o fraudolenti.

Per questo motivo la sicurezza dei dati personali ricopre un ruolo fondamentale nella realizzazione di tali sistemi.

L'obiettivo del seguente lavoro di tesi è quello di effettuare un'approfondita analisi del modello di sicurezza implementato nella piattaforma Android, esporre le problematiche riscontrate e le possibili soluzioni.

1.1 Organizzazione del lavoro di tesi

Il presente lavoro di tesi è organizzato come segue:

Nel Capitolo 2 si illustrano i concetti preliminari necessari a comprendere il lavoro. Viene effettuata un'analisi dal punto di vista sistemistico di Android approfondendo ciascun livello dell'architettura del sistema. Vengono inoltre esposte le caratteristiche principali dell'ambiente virtuale in cui sono eseguite le applicazioni e i vantaggi introdotti da tale implementazione.

Nel Capitolo 3 si illustrano le componenti fondamentali di un'applicazione Java di Android e i metodi con cui due applicazioni diverse possono comunicare e scambiare dati tra di loro e il sistema. Viene inoltre esposto il modello di sicurezza implementato nel sistema Android per proteggere i dati sensibili che potrebbero risiedere all'interno di un dispositivo.

Nel Capitolo 4 si espone l'analisi effettuata dell' *Android Permission Framework*, cioè le componenti che interagiscono per controllare che un'applicazione possieda i privilegi necessari per effettuare delle operazioni potenzialmente dannose da un punto di vista della sicurezza. Nel medesimo capitolo viene inoltre esposto il tentativo effettuato di realizzare un software per l'analisi automatica della Virtual Machine utilizzata in Android e i problemi riscontrati.

Inoltre l'*Android Permission Framework* non è l'unica parte del sistema in cui potrebbero risiedere delle vulnerabilità di sicurezza, come dimostrato dall'esistenza di *exploit* noti che sfruttano difetti in applicazioni di sistema.

Al termine di questo capitolo viene analizzato il funzionamento di tali exploit, sfruttati da recenti malware in Android, che effettuano escalation dei privilegi al fine di ottenere i permessi di root.

Il Capitolo 5 conclude il lavoro e mostra possibili sviluppi futuri dell'argomento oggetto di questa tesi segnalando inoltre alcuni progetti preesistenti per la risoluzione di problematiche di sicurezza riscontrate durante l'analisi dell'*Android Permission Framework*.

Capitolo 2

Concetti preliminari

Android è uno stack software per dispositivi mobili che include un sistema operativo, un insieme di librerie e delle applicazioni di sistema per le funzioni base del dispositivo.

Android nacque dall'esigenza di fornire una piattaforma open source per la realizzazione di applicazioni mobili che non avesse il peso di licenze che ne potessero frenare lo sviluppo. Per questo motivo utilizza solo tecnologie open source (prima tra tutte il kernel Linux) e i suoi sorgenti sono scaricabili gratuitamente da chiunque voglia contribuire a migliorarlo, studiarlo o solamente utilizzarlo sui propri dispositivi.

Inizialmente venne sviluppato dalla startup Android Inc., la quale venne acquisita da Google nel 2005 e i principali realizzatori entrarono a far parte del team di progettazione della nuova piattaforma Google.

Nel 2007 nacque la Open Handset Alliance (OHA): un accordo tra le principali aziende di hardware e software nel mondo della telefonia mobile il cui obiettivo è sviluppare degli standard aperti per dispositivi mobili e di cui Android è il progetto principale.

Nello stesso anno venne rilasciata la prima versione del Software Development Kit (SDK) che ha consentito agli sviluppatori di iniziare a testare la piattaforma e di realizzare le prime applicazioni sperimentali che dal 2008 hanno anche potuto essere testate sul primo dispositivo reale, il G1 della T-Mobile.

Nel 2008 venne inoltre rilasciato il codice sorgente di Android con la licenza Open

Source Apache License 2.0 che permette alle varie aziende di utilizzare tale sistema sui propri dispositivi e di realizzare le proprie estensioni (anche proprietarie) senza legami che ne potrebbero limitare l'utilizzo.

Da allora Android si è diffuso su una sempre più vasta gamma di dispositivi diversi e nel corso degli anni è stato più volte aggiornato per apportare migliorie alle componenti interne del sistema e all'interfaccia grafica fino ad arrivare alla versione corrente, la 2.3.3 (Gingerbread) per gli smartphone e ad una versione dedicata appositamente ai computer tablet, la 3.0 (Honeycomb).

In questo capitolo verranno trattati quegli argomenti utili per un'approfondita comprensione del sistema Android. Saranno analizzati principalmente gli aspetti sistemistici di Android tra cui la sua architettura (2.1), la fase di boot (2.2) e l'ambiente virtuale in cui vengono eseguite le applicazioni (2.3).

2.1 Architettura di Android

L'architettura del sistema Android è caratterizzata da una classica struttura a livelli (*layer*) in cui i livelli inferiori offrono servizi a quelli superiori così da permettere un livello di astrazione dell'hardware sempre maggiore.

Nel layer più basso si trova il kernel Linux, il quale fornisce gli strumenti di basso livello per l'astrazione dell'hardware sottostante attraverso la definizione di diversi driver specifici.

Al livello successivo è possibile trovare le librerie native, le quali implementano a basso livello i principali servizi utilizzati dai diversi componenti di Android, ad esempio: funzioni multimediali, grafica 2D e 3D, un DBMS relazionale, etc.

Nello stesso layer viene posto l'**Android Runtime** costituito dalla *Dalvik Virtual Machine (DVM)* e da un set di librerie di base in grado di fornire la maggior parte delle funzionalità disponibili con le librerie standard del linguaggio di programmazione Java.

Tutte le librerie native vengono utilizzate da un insieme di componenti di più alto livello che costituiscono l' **Application Framework** layer il quale rappresenta l'in-

Figura 2.1: Rappresentazione dell'architettura di Android.

sieme di API messe a disposizione dal sistema alle applicazioni (di sistema e di terze parti) che si trovano all'ultimo livello dell'architettura.

2.1.1 Linux Kernel

Android è stato realizzato con l'intento di rendere disponibile una piattaforma completamente open source per i dispositivi mobili. Il kernel Linux è stato scelto come base

per la realizzazione del sistema Android principalmente per le sue caratteristiche di: gestione della memoria e dei processi, il modello di sicurezza basato sui permessi, il supporto alle shared library e per il fatto di essere un preesistente progetto opensource.

Il kernel Linux su cui si basa Android è la versione 2.6 alla quale sono state apportate alcune modifiche per permettere di interagire al meglio con il resto dell'architettura.

Non sono incluse, infatti, molte delle utility standard di un sistema Linux come, ad esempio, il supporto alla libreria standard del linguaggio C e un sistema nativo per la gestione delle finestre.

Oltre a comprendere i driver per gestire l'hardware sottostante (*Display Driver*, *USB Driver*, *Bluetooth Driver*, etc.), sono stati apportati alcuni miglioramenti al kernel unicamente per Android [1] tra i quali il principale è identificabile con il *Binder (IPC) Driver*.

Il *Binder Driver* nasce dalla necessità di permettere ad applicazioni che vengono eseguite in processi diversi di comunicare tra loro e scambiarsi dati in modo sicuro e veloce. Il Binder è pertanto la componente fondamentale che permette la comunicazione tra processi diversi (*Inter Process Communication (IPC)*) ad alte prestazioni grazie all'uso del concetto di memoria condivisa (*shared memory*); non si limita a ricevere i dati da un'applicazione e inoltrarli ad un'altra, bensì gestisce inoltre la comunicazione tra thread dello stesso processo o di processi diversi [2].

La Figura 2.2 illustra per esempio un'applicazione (App A) che vuole comunicare con una seconda applicazione (Service B), entrambi in esecuzione in due processi distinti.

L'App A richiede all'oggetto di alto livello *Context* se il Service B è effettivamente in esecuzione e, in tal caso, *Context* risponderà inviando all'App A un riferimento a Service B.

A questo punto l'App A è in grado di richiamare i metodi messi a disposizione dal Service B. In realtà quando l'App A invoca, come dall'esempio rappresentato in Figura 2.2, il metodo *foo(object)* passandogli come parametro un oggetto generico, la chiamata viene inoltrata al Binder Driver il quale gestirà i dati tramite la *shared memory* e si occuperà di inoltrarli al corretto thread all'interno del Process B. Termi-

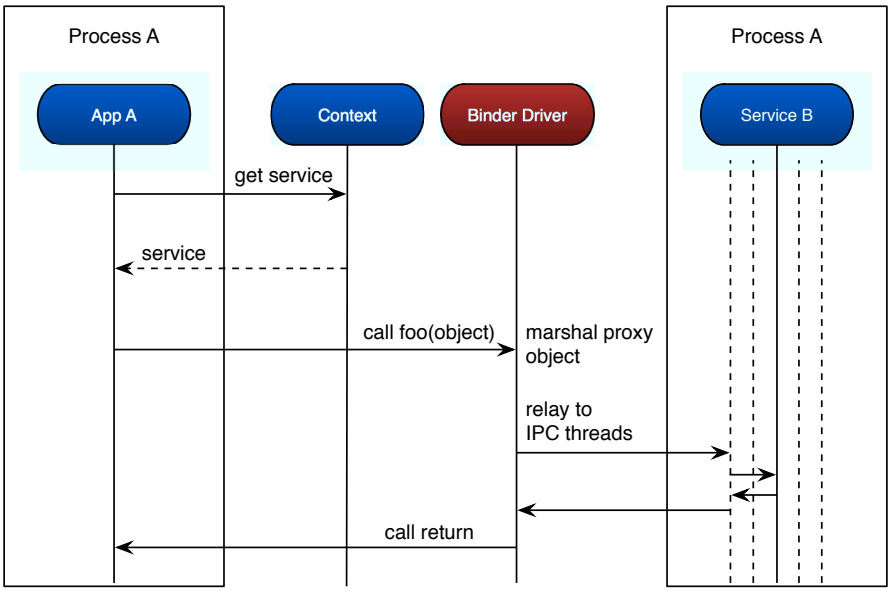

Figura 2.2: Esempio di comunicazione tra processi (e thread) diversi attraverso l'utilizzo del Binder Driver.

nata l'esecuzione del metodo invocato da parte del corrispondente thread del Service B, il risultato di tale esecuzione verrà inviato al Binder Driver il quale provvederà ad inoltrarlo all'App A.

2.1.2 Native Library

Al secondo livello dell'architettura si trovano le *Native Library*, un insieme di librerie sviluppate in C e C++ che implementano a basso livello tutte le funzionalità offerte dall'*Application Framework*.

Le principali librerie sono le seguenti:

Bionic libc : implementazione della libreria standard *libc* ottimizzata per dispositivi basati su Linux embedded come Android.

È stato necessario reimplementare tale libreria per i seguenti motivi [3]:

1. Licenza: la libreria *glibc* è rilasciata sotto licenza GPL la quale vieta di utilizzare codice con tale licenza in progetti con codice proprietario. Per non

obbligare gli sviluppatori a rendere open-source le proprie applicazioni, è stato quindi necessario reimplementare tale libreria e distribuirla secondo la licenza BSD.

2. Dimensione: *Bionic* viene caricata in ogni processo quindi era necessario che avesse dimensioni ridotte; le sue dimensioni, infatti, corrispondono a circa la metà di quelle della controparte rilasciata sotto licenza GPL.

Inoltre *Bionic* supporta importanti servizi di Android quali l'impostazione di parametri del sistema (*setprop* e *getprop*) e operazioni di logging. Non supporta alcune funzionalità POSIX, come ad esempio le eccezioni in C++, perchè non necessarie in Android e pertanto non risulta compatibile con la *GNU libc*.

WebKit : noto *web browser engine* open source che offre pieno supporto alle tecnologie web come CSS, Javascript, DOM e AJAX. Utilizzato anche dai principali web browser in ambiente desktop come Safari e Chrome, WebKit offre il supporto alla visualizzazione di pagine web su dispositivi con display a grandezza ridotta come gli smartphone.

Media Framework : componente in grado di gestire i diversi *codec* per i vari formati di acquisizione e riproduzione audio e video. Basato sulla libreria open source *OpenCore*, permette la gestione dei formati più diffusi tra cui *MPEG4*, *H.264*, *MP3*, *AAC*, *AMR* oltre a quelli per la gestione delle immagini come *JPG* e *PNG*.

SQLite : nota libreria che implementa un *embedded SQL database engine* transazionale. A differenza di altri DBMS, *SQLite* non necessita di un server separato per la gestione dei database poichè ciascuno di essi è rappresentato in un file ordinario il quale viene letto e scritto da tale *engine*. Questa caratteristica, oltre al fatto di essere open source, l'ha reso il candidato ideale a svolgere il ruolo di DBMS in Android poichè ogni applicazione che fa uso di un database possiede il proprio file il quale è leggibile e scrivibile solo dall'applicazione stessa [4].

Surface Manager : server che si occupa di scrivere gli elementi grafici (*Surface*) utilizzati da più applicazioni (eseguite in processi diversi) sullo stesso *frame buffer* il quale verrà di seguito visualizzato sul display del dispositivo.

La comunicazione tra i vari processi e il *Surface Manager* avviene tramite l'utilizzo di Binder così da garantire lo scambio di informazioni in modo sicuro e performante. Gestisce inoltre l'uso combinato di *Surface* 2D e 3D destinati allo stesso device di output utilizzando le librerie *OpenGL ES* e l'accelerazione hardware 2D per la loro rappresentazione [5].

Audio Manager : server che si occupa di gestire i device di output per l'audio analogamente a quanto avviene per il *Surface Manager* con il display del dispositivo. Il compito principale dell'*Audio Manager* è quello di processare diversi audio stream provenienti dalle applicazioni e di dirigerli verso i dispositivi di output appropriati (speaker, auricolari bluetooth, ...).

All'interno del medesimo livello delle *Native Library* è possibile identificare un insieme di librerie native che definiscono le interfacce tra Android e i driver sotto forma di librerie dinamiche che vengono caricate a runtime quando necessario.

Android è stato ideato per essere un sistema utilizzabile su una vasta gamma di dispositivi con caratteristiche hardware diverse; per questo motivo si è reso necessario introdurre tali librerie all'interno dell"architettura allo scopo di separare la logica della piattaforma Android dai driver dei vari componenti hardware i quali vengono sviluppati dai produttori dei dispositivi.

2.1.3 Android Runtime

Al di sopra delle librerie native si trova l'*Android Runtime* che rappresenta l'ambiente virtuale nel quale vengono eseguite le applicazioni Java in Android. Come principale linguaggio di programmazione per le applicazioni, i progettisti del sistema hanno scelto di utilizzare Java al fine di rendere tali applicazioni portabili su qualunque dispositivo utilizzasse Android.

Android, al posto di utilizzare la classica *Java Virtual Machine* per eseguire le applicazioni Java, utilizza una particolare macchina virtuale, la *Dalvik Virtual Machine (DVM)*, ottimizzata appositamente per sfruttare le risorse limitate (quali la potenza della CPU e la poca memoria RAM) dei dispositivi mobili.

Oltre alla *DVM*, in *Android Runtime*, sono presenti anche le *Core Libraries*, scritte in Java, le quali costituiscono i *packages* base su cui l'*Application Framework* di Android è stato realizzato.

Le *Core Libraries* possono essere divise in tre categorie:

dalvik.* : package specifico della *Dalvik Virtual Machine*; contiene classi e metodi per il debugging, per caricare le classi di un'applicazione (*Classloader*) e per la gestione dei file contenente il bytecode interpretabile dalla macchina virtuale.

java.*, javax.* : contengono la maggior parte dei packages standard di Java (ad esempio: `java.io`, `java.sql`, `javax.security`, `javax.xml`,...) ad esclusione dei packages per la gestione delle interfacce grafiche (GUI) `java.awt` e `javax.swing`.

org.apache.http.* : package che contiene interfacce e classi per la comunicazione tramite il protocollo HTTP.

2.1.4 Application Framework

L'*Application Framework (AF)* è costituito da componenti di alto livello che utilizzano le librerie contenute nelle *Core Library* e nelle *Native Library*. Si tratta di un insieme di API (Application Programming Interface) e componenti scritti in Java che vengono utilizzati dalle applicazioni direttamente o indirettamente per poter accedere alle funzionalità offerte dal sistema e ai dispositivi hardware.

Tutte le applicazioni in Android utilizzano lo stesso *Application Framework* così da permettere agli sviluppatori di estendere, modificare o sostituire le funzionalità offerte dalle applicazioni di sistema con quelle proposte dalle proprie applicazioni.

Di seguito sono elencati i principali componenti presenti all'interno dell'*Application Framework*:

Activity Manager : l'*Activity* è un componente che è possibile associare al concetto di schermata con il quale interagisce l'utente. La responsabilità dell'*Activity Manager* è quello di gestire il ciclo di vita di ciascuna *Activity* e di organizzare le varie schermate di un'applicazione in uno stack a seconda dell'ordine di

visualizzazione delle stesse sul display del dispositivo come verrà approfondito nel Capitolo 3.1.

Package Manager : componente responsabile dell'accesso alle varie informazioni relative ai package che contengono le applicazioni (file .apk) [6]. Il *Package Manager* si occupa, ad esempio, di accordare o meno l'utilizzo di una risorsa hardware da parte di un'applicazione a seconda dei permessi concessi a quest'ultima in fase di installazione sul dispositivo; per questo motivo il *Package Manager* risulta essere una delle componenti fondamentali del modello di sicurezza di Android.

Window Manager : gestisce la visualizzazione delle finestre delle applicazioni; costituisce l'astrazione ad alto livello del *Surface Manager* presente nel *Native Library Layer*.

Resource Manager : si occupa dell'ottimizzazione delle risorse utilizzate da ciascuna applicazione quali immagini o file di configurazione.

Ad esempio: se un'applicazione utilizza un documento XML come risorsa, quest'ultimo non viene salvato sul dispositivo come file testuale ma come file binario (Binary XML) ottimizzato rispetto a quella che è la principale operazione possibile su tali file, il *parsing*.

Un altro aspetto fondamentale dell'ottimizzazione delle risorse riguarda la possibilità per l'applicazione di accedere a queste ultime attraverso delle costanti generate in fase di compilazione dell'applicazione stessa. In pratica tutte le risorse, come per esempio gli oggetti che compongono l'interfaccia grafica di un'applicazione, vengono referenziate da delle costanti contenute in una particolare classe Java, la classe *R*, attraverso la quale l'applicazione può utilizzare tali risorse.

L'*Application Framwork*, inoltre, mette a disposizione delle applicazioni una serie di API per l'accesso alle componenti hardware presenti nel dispositivo quali, ad esempio, il *Location Manager* per la geolocalizzazione che si basa su diverse metodologie

(ricevitore GPS, infrastruttura cellulare dell'operatore telefonico, Wifi), il *Wifi Manager* per la connessione dati attraverso la tecnologia Wireless, *Bluetooth Manager*, lo *USB Manager*, il *Sensor Manager*, ...

2.1.5 Interazione tra i livelli

Ciascun layer nasconde il funzionamento interno dei propri servizi, quest'ultimi forniti attraverso un'interfaccia, in modo da consentire ai livelli superiori di richiedere l'esecuzione di un compito senza la necessità di conoscere l'implementazione dello stesso.

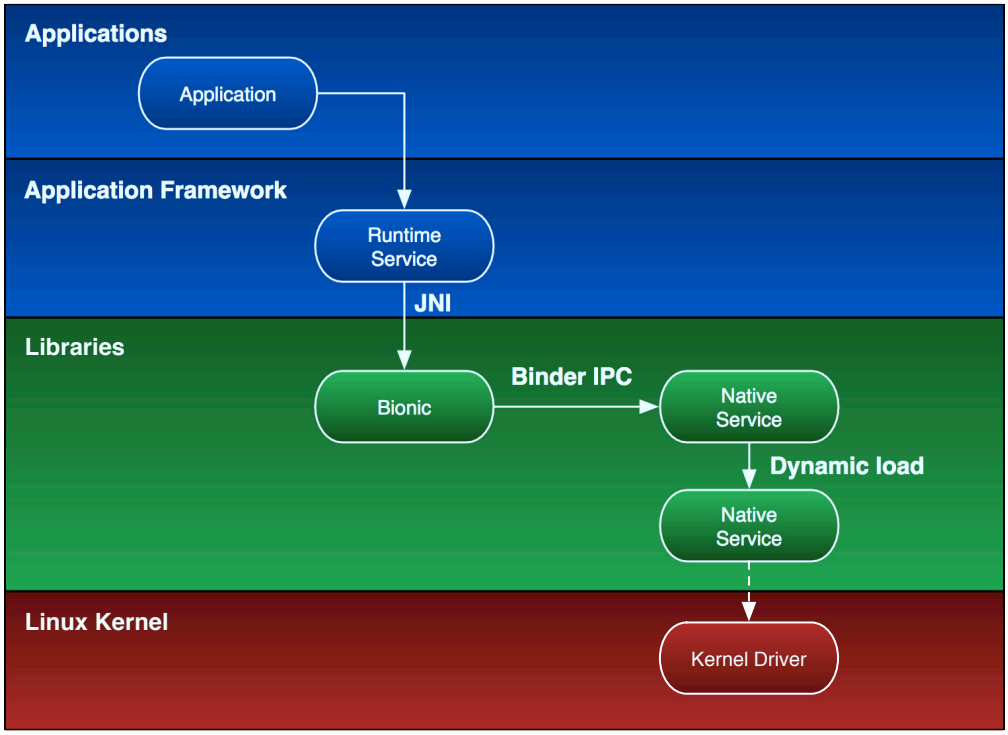

Figura 2.3: Interazione tra i livelli dell'architettura Android.

Scenario tipico nel quale si verfica l'interazione tra tutti i livelli dell'architettura Android è quello in cui un'applicazione necessita di utilizzare un componente hardware installato sul dispositivo.

In questo caso l'applicazione invia una richiesta tramite un *Binder* al particolare *Runtime Service* dell'*Application Framework* il quale, attraverso le *Java Native Interface (JNI)*, inoltra la richiesta all'implementazione nativa del *Service*. A livello di *Native Library* il *Service* nativo carica dinamicamente la libreria (il driver in user-space) il quale è in grado di comunicare con la periferica attraverso il kernel Linux.

Per esempio: un'applicazione che necessita l'utilizzo del ricevitore Gps per la geolocalizzazione invia attraverso un *Binder* tale richiesta al *Location Manager* il quale richiama il *GpsLocationProvider*, cioè un'interfaccia ad alto livello di un generico dispositivo Gps. Il *GpsLocationProvider* inoltra la richiesta dell'applicazione alla rispettiva implementazione nativa che si occupa di caricare in memoria il driver del Gps (`libgps.so`) il quale comunica con il ricevitore Gps attraverso il kernel Linux.

2.2 Sequenza di boot

La sequenza di boot del sistema Android non si differenzia molto, almeno in principio, da quella di un comune sistema operativo basato su Linux.

Il primo software avviato all'accensione del dispositivo è il *bootloader* che, oltre a caricare in memoria il kernel Linux e avviarlo, mette a disposizione la possibilità di eseguire operazioni di recovery, aggiornamento o debugging del sistema a seconda della combinazione di tasti fisici premuti all'avvio (specifici per ciascun dispositivo).

Una volta caricato in RAM, il kernel esegue le tipiche operazioni di inizializzazione quali: inizializzazione della memoria e della *process table*, avvio dei driver e demoni di basso livello, montaggio del file system di root e, in ultimo, l'avvio del primo processo in user-space *init*.

A questo punto l'avvio di Android si differenzia da quello di un sistema Linux-based.

Il primo compito del processo *init* in Android è quello di avviare i seguenti demoni:

usbd : demone USB per gestire le connessioni tramite l'interfaccia di Input/Output USB.

adbd : *Android Debug Bridge Daemon*, rimane in ascolto sull'interfaccia USB di connessioni provenienti dal client *adb*, normalmente installato sul computer dell'utente. Consente di eseguire una shell utente, con permessi dell'utente *shell*, non protetta da password al fine di permettere operazioni (limitate) di gestione del dispositivo quali l'installazione e la rimozione di applicazioni utente, la visualizzazione dei processi e dei messaggi di log [7].

debuggerd : *Debugger Daemon*, si occupa di elaborare le richieste di debug (dump della memoria, visualizzazione processi, etc.).

rild : *Radio Interface Layer Daemon*, gestisce le comunicazioni con l'hardware utilizzato dal dispositivo per interfacciarsi con la rete cellulare (GPRS, UMTS, HSPA, etc.).

Successivamente *init* avvia uno dei processi più importanti dell'intero sistema: *Zygote*. In Android ciascuna applicazione viene eseguita all'interno di un'istanza diversa della *Dalvik Virtual Machine (DVM)*; la responsabilità di *Zygote* (la prima istanza della DVM) è quella di creare un nuovo processo figlio (*fork*) e di avviare in esso una diversa istanza della Virtual Machine ad ogni richiesta pervenutagli dal sistema via socket.

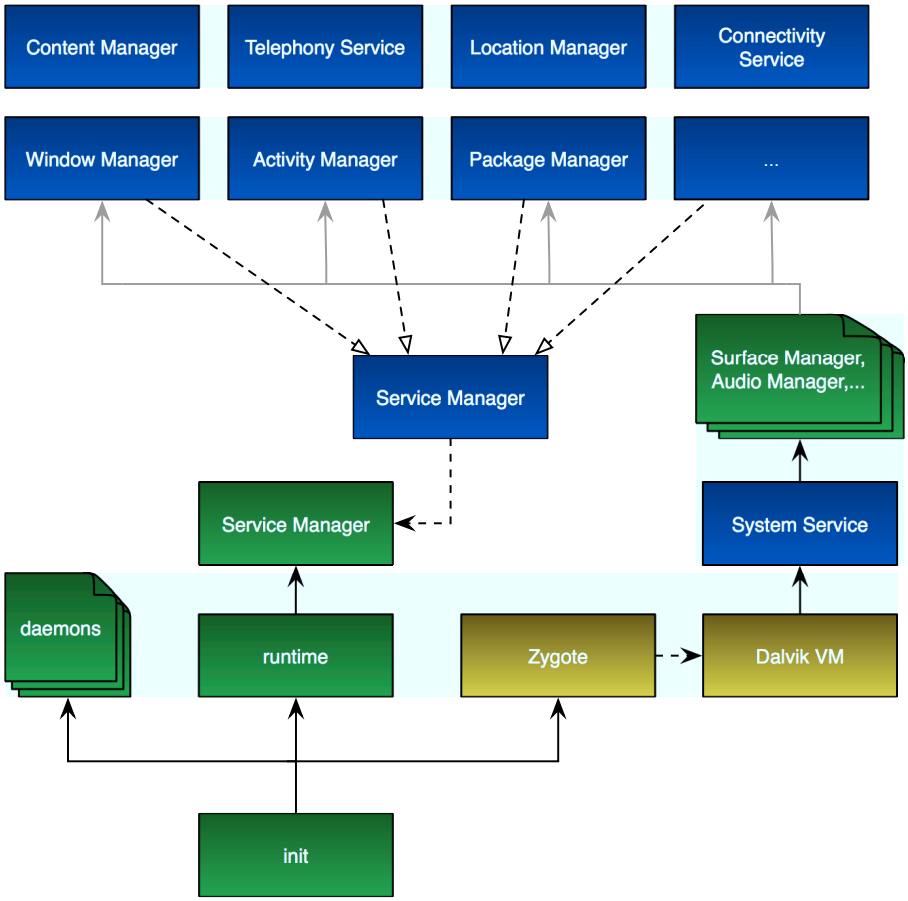

Figura 2.4: Sequenza di boot del sistema Android.

Come ultima operazione, *init* avvia il processo *runtime* che, a sua volta, avvia ed inizializza il *Service Manager*, il quale ha il compito di gestire la registrazione di ciascun *Service* (sia esso nativo o di alto livello) che verrà creato per svolgere specifici compiti in modo tale che il sistema conosca a quale componente inoltrare le richieste ricevute dalle applicazioni attraverso i *Binder*.

A questo punto *runtime* richiede a *Zygote* di creare una nuova istanza della *DVM* e di avviare il *System Service*. Tale *Service* Java avvia prima di tutto i server nativi di sistema come *Surface Manager*, *Audio Manager* e, successivamente, i *Service* dell'*Application Framework*.

Una volta che i *Service* di alto livello sono stati avviati e registrati al *Service Manager* Java, la sequenza di boot di Android può considerarsi terminata e possono essere eseguite le applicazioni di sistema e quelle utente.

2.3 Dalvik Virtual Machine

Le applicazioni Android vengono sviluppate principalmente utilizzando il linguaggio di programmazione Java così da risultare portabili su dispositivi con caratteristiche hardware differenti. Le applicazioni Java non vengono compilate in codice macchina ma in un linguaggio intermedio, chiamato *bytecode*, interpretabile da una Virtual Machine la quale si occupa di tradurlo in codice macchina durante la fase di esecuzione dell'applicazione.

I progettisti di Android hanno scelto di non utilizzare una tipica *Java Virtual Machine (JVM)* optando invece per un'alternativa più performante, la *Dalvik Virtual Machine (DVM)*.

Android è caratterizzato principalmente dalle seguenti proprietà:

1. è stato progettato per essere eseguito su dispositivi con limitate capacità computazionali e poca memoria RAM come gli *smartphone*.

2. ogni applicazione viene eseguita all'interno di un processo diverso in cui è in esecuzione un'istanza della Virtual Machine.

La Dalvik Virtual Machine è ottimizzata per essere eseguita su dispositivi con risorse hardware limitate demandando al sistema operativo la gestione della memoria, l'isolamento dei processi, il supporto ai thread, i controlli di sicurezza e, inoltre, consente l'esecuzione contemporanea di più istanze della Virtual Machine stessa. Tali caratteristiche hanno permesso alla Dalvik Virtual Machine di divenire il componente principale del sistema Android.

2.3.1 Dalvik Executable

Il bytecode Java di un'applicazione è contenuto in uno o più file *.class* e, nel caso di più classi, in fase di compilazione vengono generati tanti file *.class* quante sono le classi totali.

Per esempio, nel codice del Listato 2.1 viene definita una generica interfaccia `myInterface` e il metodo `myMethod`; tale metodo viene successivamente implementato nella classe `myClass` e invocato da una seconda classe `mySecondClass`.

Listing 2.1: Esempio di codice Java.

```
public interface myInterface {
  public String myMethod(String s, Object o);
}

public class myClass implements myInterface {
  public String myMethod(String s, Object o) {
    .... ;
  }
}

public class mySecondClass {
  public void useMyMethod(myInterface interface) {
    interface.myMethod(....);
  }
}
```

In fase di compilazione del codice vengono generati tre file *.class* che possono essere rappresentati come nello schema di Figura 2.5. In tale schema è possibile notare la ripetizione di alcune informazioni in tutti e tre i file come, ad esempio, il prototipo del metodo `myMethod` e della stringha di caratteri `SourceFile` utilizzata esclusivamente dalla Virtual Machine.

Questa ridondanza di informazioni non è accettabile in un contesto di ottimizzazione come quello in cui deve operare la Dalvik Virtual Machine; per tale motivo la DVM non interpeta il comune Java bytecode ma un diverso codice intermedio, il *Dalvik bytecode*, ottimizzato per il riutilizzo delle informazioni in esso contenute e per ridurre il più possibile il numero di istruzioni da eseguire [8].

Il processo di compilazione di una tipica applicazione Java in Android è composto principalmente da due fasi tramite l'ausilio dell'Android SDK:

1. La prima fase consiste nella compilazione del codice in Java bytecode raccolto in uno o più file *.class* attraverso il compilatore Java *javac*.

2. Successivamente i file *.class* vengono passati in input al programma *dx* il quale effettua la traduzione da Java bytecode a Dalvik bytecode incluso in un unico file

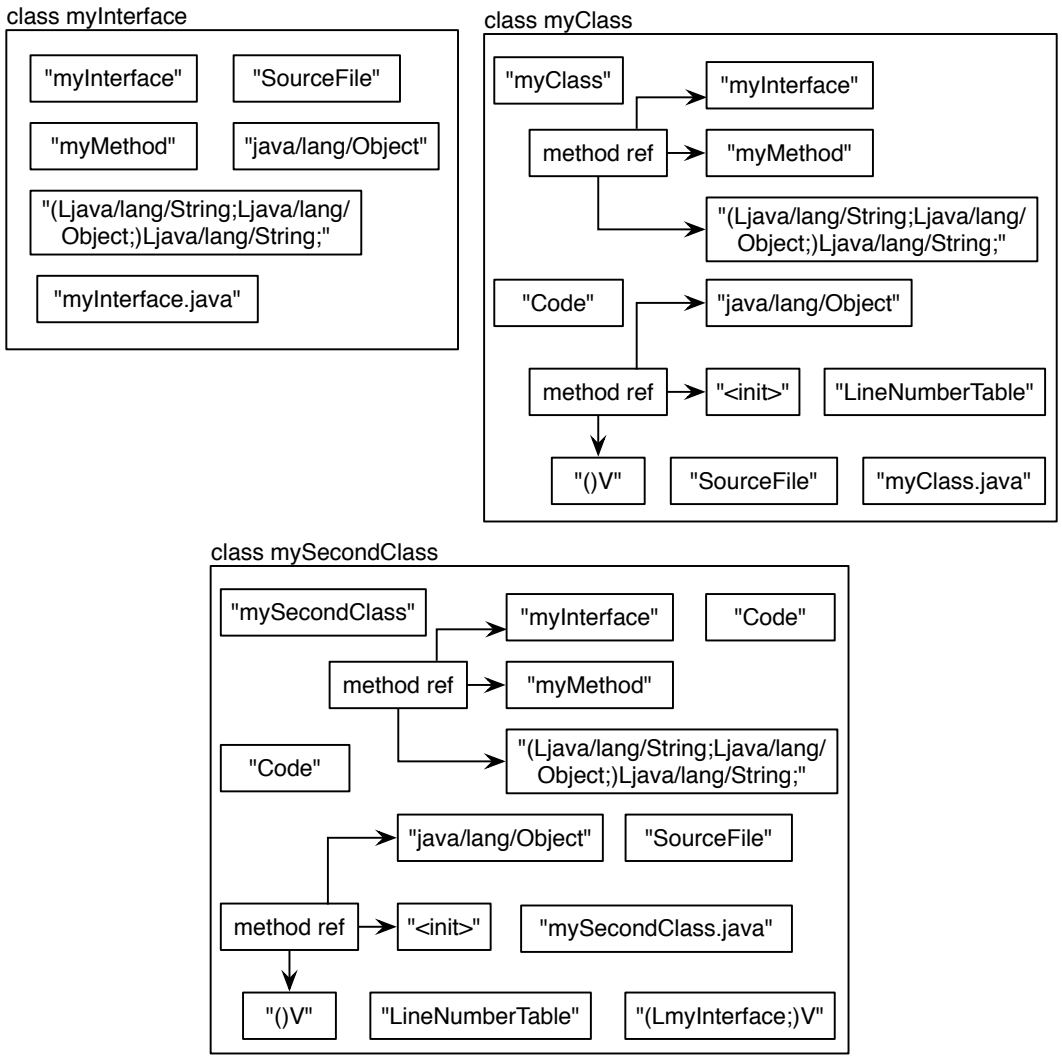

Figura 2.5: Rappresentazione grafica dei file .class generati a eguito della compilazione in Java bytecode del Listato 2.1.

classes.dex anche nel caso in cui siano presenti più classi. Il file *.dex* (insieme ad altri file utili all'applicazione come immagini, file XML, etc.) viene inserito in un archivio compresso *.apk (Android Package)* col quale l'applicazione può essere distribuita e installata sul dispositivo.

Il file *.dex* (*Dalvik Executable*) contenente il Dalvik bytecode è così strutturato:

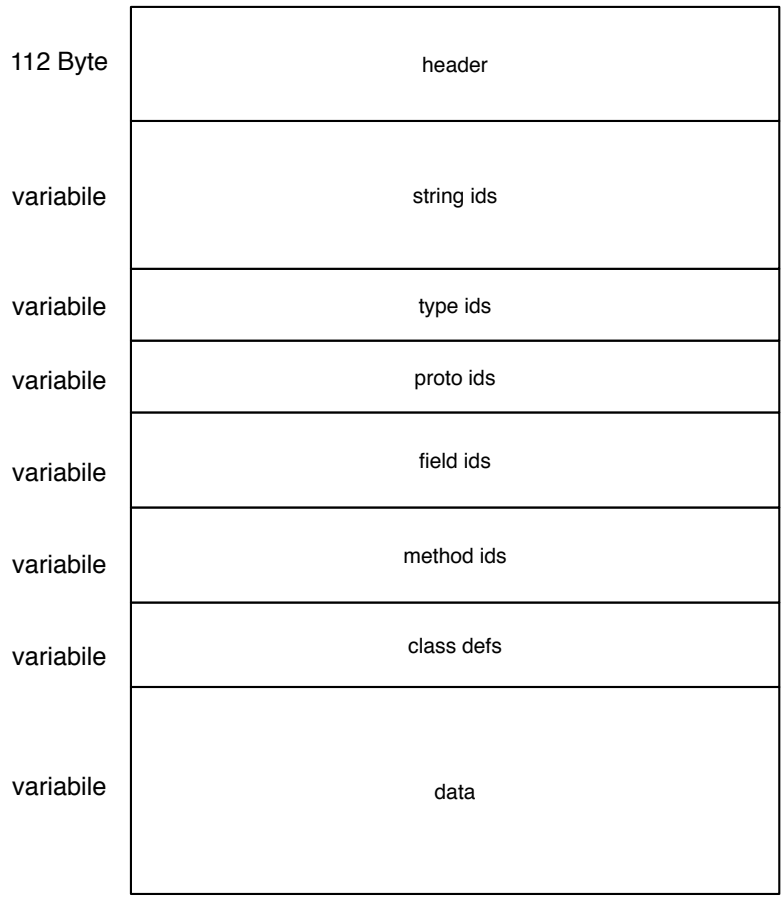

Figura 2.6: Rappresentazione della struttura del file .dex.

header : sezione di dimensione fissa (112 Byte). Contiene le informazioni generali riguardanti il file *.dex* e la sua struttura. Le prime informazioni contenute nell'header sono le seguenti:

- I primi 8 Byte sono dedicati ad ospitare il *magic number* : una stringa di caratteri, `dex\n 035\0` (`0x64 0x65 0x78 0x0a 0x30 0x33 0x35 0x00` in esadecimale), che caratterizza la tipologia del file *.dex*.
- Per rilevare se il file è stato accidentalmente corrotto vengono controllati i successivi 4 Byte che rappresentano il *checksum* calcolato alla creazione del file utilizzando l'algoritmo *adler32*.

- Un campo di 20 Byte che contiene la *signature* calcolata con l'algoritmo *SHA-1* e utilizzata per identificate in maniera univoca ogni file *.dex*.
- I successivi 8 Byte contengono i campi *file size* e *header size* che corrispondono rispettivamente alla dimensione dell'intero file e a quella dell'header.

Il resto dell'*header* contiene le dimensioni delle successive sezioni e gli offset in Byte per tenere traccia della loro posizione all'interno del file *.dex*.

string ids : contiene i puntatori a tutte le stringhe utilizzate all'interno del file comprese quelle per uso interno quali, ad esempio, `java/lang/Object`, `SourceFile` o `LineNumberTable`.

type ids : riferimenti ai tipi di dato utilizzati come array, classi e tipi di dato primitivi.

proto ids : lista contenente i riferimenti ai prototipi dei metodi usati ordinati secondo il tipo di dato restituito e gli argomenti.

field ids : lista degli attributi (*fields*) di ciascuna classe contenuta all'interno del file.

method ids : identificatori dei metodi utilizzati dal file.

class defs : lista di identificatori delle classi utilizzate nel file.

data : in questa sezione sono contenuti tutti i dati referenziati dai puntatori presenti nelle sezioni precedenti del file *.dex*.

Compilando il codice del precedente listato in *Dalvik bytecode*, il contenuto del file *.dex* risultante può essere quindi schematizzato come in Figura 2.7 nella quale è possibile notare la differenza con lo schema di Figura 2.5 dovuta alla mancanza di ridondanza delle informazioni a seguito dell'ottimizzazione.

Uno dei principali vantaggi del processo di ottimizzazione del bytecode interpretato dalla Dalvik Virtual Machine è sicuramente quello del risparmio di memoria occupata. Secondo alcuni dati rilevati dal team di sviluppo di Android, l'utilizzo di file *.dex* permette di dimezzare lo spazio occupato rispetto ai file *.class* [9].

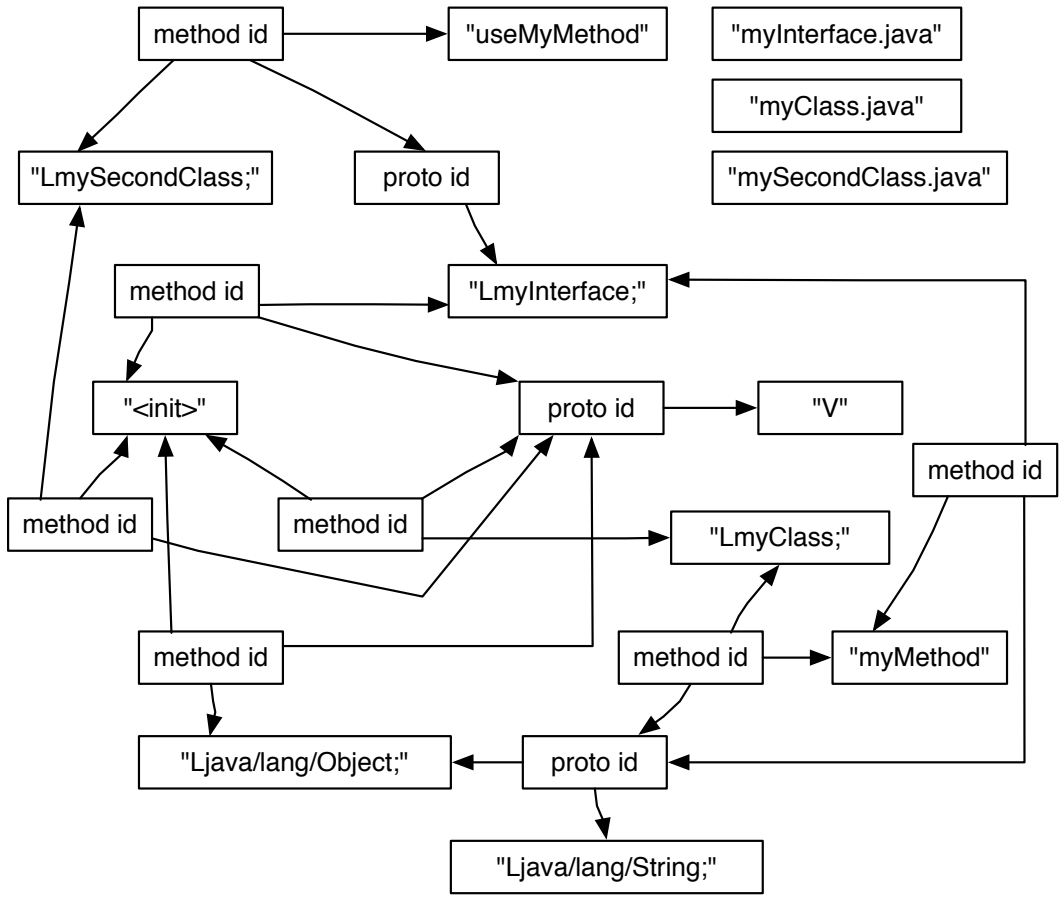

Figura 2.7: Rappresentazione grafica del file .dex generato a seguito della compilazione in Dalvik bytecode.

Tale ottimizzazione ha la caratteristica di non duplicare le informazioni al fine di non occupare inutilmente spazio in memoria; va considerato che questo approccio potrebbe però avere ripercussioni nel caso in cui due applicazioni condividano una stessa porzione di memoria.

Analogamente alla Java Virtual Machine, anche la Dalvik Virtual Machine, possiede un componente dedito all'eliminazione di oggetti allocati in memoria nel momento in cui non fossero più necessari: il *Garbage Collector (GC)*.

Dal momento che ciascuna applicazione viene eseguita all'interno di un'istanza diversa della Virtual Machine, sono presenti diversi Garbage Collector che operano

sugli *heap* (area di memoria assegnata a ciascun processo per l'allocazione dinamica di dati) dei singoli processi indipendentemente l'uno dall'altro.

Codice	**Jar non compresso**	**Jar compresso**	**dex non compresso**
Libreria di sistema	21,445,320 (100%)	10,662,048 (50%)	10,311,972 (48%)
Web Browser App	470,312 (100%)	232,065 (49%)	209,248 (44%)
Alarm Clock App	119,200 (100%)	61,658 (52%)	53,020 (44%)

Tabella 2.1: Dati (in Byte) rilevati per comparare la dimensione dei file .class rispetto ai file dex.

Normalmente il Garbage Collector marca gli oggetti destinati all'eliminazione con un bit (*mark bit*); tali oggetti verranno rimossi definitivamente dall'heap non appena possibile (ed eventualmente necessario) nel momento in cui gli stessi risultassero non più utilizzati dal programma [10]. In caso di processi che condividono la stessa porzione di memoria i mark bit vengono salvati insieme agli oggetti nell'heap condiviso; potrebbe quindi verificarsi che il Garbage Collector di una Virtual Machine (GC A) elimini un oggetto non più richiesto da un'applicazione (App A) ma ancora necessario ad una seconda applicazione (App B) rendendo così tale oggetto non più condiviso [9].

Diversamente, la DVM implementa un Garbage Collector che pone i mark bit in un'area di memoria diversa da quella condivisa; tali bit indicano gli oggetti condivisi in modo tale da prevenire che vengano eliminati dal Garbage Collector durante la scansione dell'heap condiviso.

Capitolo 3

Scenario del lavoro

Nella prima parte di questo capitolo vengono illustrati i componenti fondamentali di un'applicazione Java di Android e i metodi con cui due applicazioni diverse possono comunicare e scambiare dati tra loro e il sistema. Tali concetti risultano fondamentali per una corretta comprensione del modello di sicurezza implementato in Android esposto al termine di questo capitolo.

3.1 Activity

L'*Activity* può essere considerata a tutti gli effetti il principale componente di un'applicazione Java in Android. Rappresenta, infatti, il componente con il quale avviene l'interazione dell'utente con l'applicazione e può essere associata al concetto di "finestra" in ambiente desktop. L'interfaccia grafica di un'applicazione viene realizzata attraverso la definizione di una o più Activity descritte da estensioni dell'omonima classe presente nel package *android.app*.

Il compito di questo tipo di componente è quello di contenere le diverse schermate dell'applicazione e, attraverso la composizione delle *View*[1], gestire le azioni dell'utente.

[1] La classe android.view.View rappresenta l'elemento base per la costruzione di interfacce grafiche di un'applicazione Java in Android [11].

3.1.1 Back stack e il concetto di task

Le Activity di ciascuna applicazione vengono organizzate dal sistema in una struttura a stack (chiamato *"back stack"*), classica struttura dati la cui modalità di accesso segue la politica *LIFO* (*Last In First Out*), in cui le varie Activity possono essere immaginate disposte una sopra l'altra nell'ordine in cui ciascuna di esse viene richiamata dall'utente.

A seconda della posizione occupata sullo stack e della visibilità o meno sul display, ciascuna Activity può trovarsi in uno dei quattro stati elencati nella Tabella 3.1.

Stato	Descrizione
ACTIVE	L'Activity si trova in cima allo stack, è visibile e riceve gli eventi da parte dell'utente.
PAUSED	Activity non attive ma ancora visibili per la trasparenza di quelle superiori o perchè queste non occupano l'intero spazio a disposizione. Essa non è quindi sensibile agli eventi da parte dell'utente e viene eliminata dal sistema unicamente in situazioni di estrema necessità.
STOPPED	Activity non attive nè visibili. Non è sensibile agli eventi dell'utente ed è tra le prime candidate a essere eliminata.
INACTIVE	L'Activity si trova in questo stato quando viene eliminata o prima di essere creata.

Tabella 3.1: Possibili stati di una Activity

Quando, ad esempio, una Activity A avvia una seconda Activity B, quest'ultima viene inserita (operazione *"push"*) in cima allo stack e impostata come Activity corrente (stato ACTIVE). L'Activity A rimane all'interno dello stack, impostata nello stato di STOPPED e con lo stato salvato a prima della chiamata all'Activity B. Premendo il tasto BACK del dispositivo, l'Activity corrente viene rimossa dalla cima dello stack (operazione *"pop"*), eliminata e di conseguenza viene ripristinato lo stato dell'Activity precedentemente bloccata.

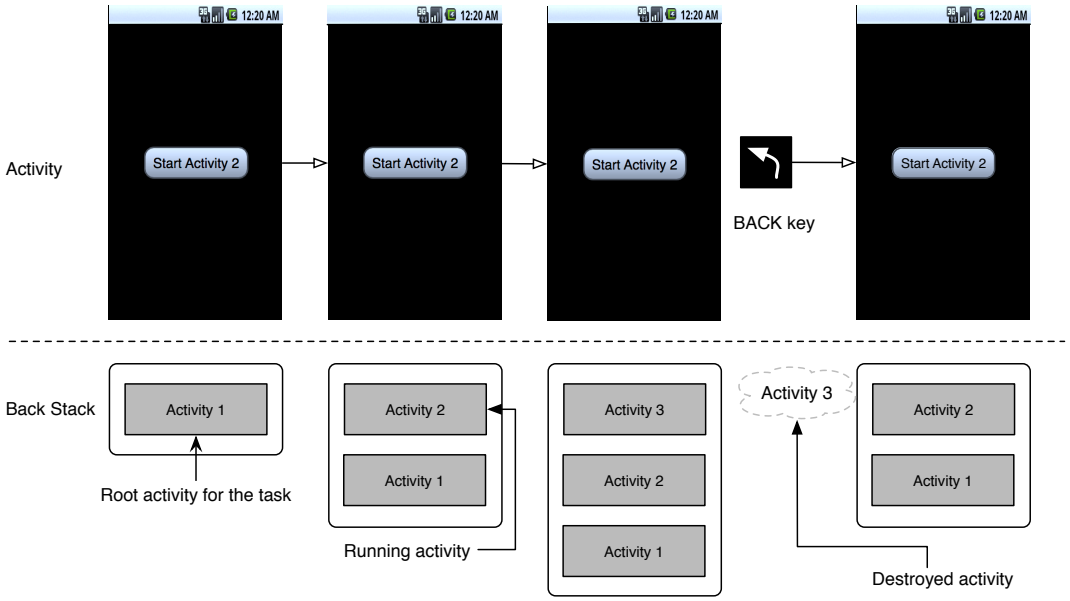

Figura 3.1: Gestione del back stack e distruzione dell'Activity corrente

Il susseguirsi di Activity legate all'esecuzione di un'applicazione viene indicato con il termine *task*, il quale puo' essere immaginato a tutti gli effetti come un elemento, al quale è associato un *back stack*. Il task puo' essere in background o foreground in base alle necessità del momento, costituendo così, una delle componenti di base per la realizzazione del multitasking in Android. Quando tutte le Activity vengono rimosse dallo stack, il task al quale appartengono cessa di esistere.

Per esempio, supponendo che il task corrente (Task A) contenga tre Activity all'interno del proprio back stack e che l'utente, premendo il tasto HOME del dispositivo, avvii un'altra applicazione, quando appare la home screen il Task A viene posto in background (tutte le Activity contenute in esso vengono bloccate) e, nel momento in cui viene avviata una seconda applicazione, il sistema crea un nuovo task (Task B) con il proprio stack di Activity. A questo punto, se l'utente torna alla home screen e riavvia la prima applicazione, il Task A torna in foreground al posto del Task B che viene posto, invece, in background; di conseguenza l'Activity che si trova in cima allo stack del Task A viene visualizzata di nuovo sul display.

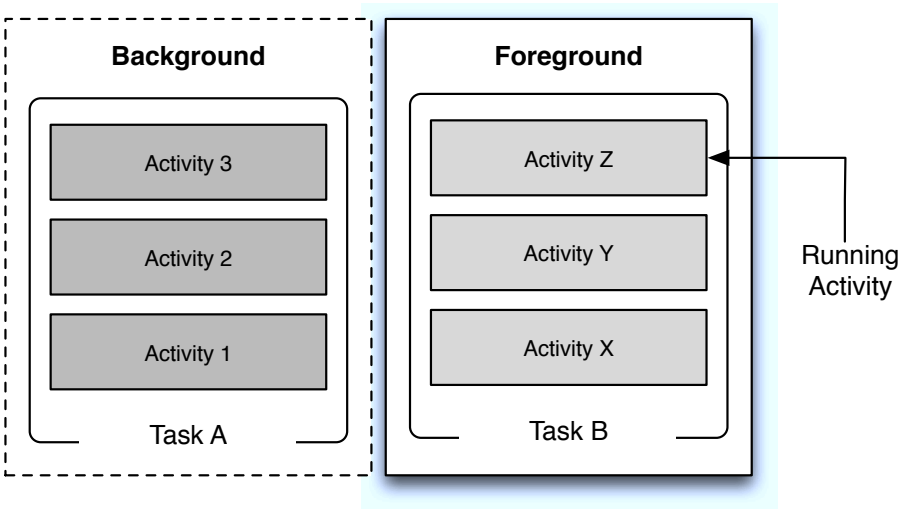

Figura 3.2: Rappresentazione del Task A in background in attesa di essere ripristinato nel momento in cui il Task B viene visualizzato sul display.

3.1.2 Activity Lifecycle

Esistono diversi metodi di *callback* che una Activity può ricevere dal sistema durante la sua attività di cui, i principali, determinano di fatto il ciclo di vita dell'Activity stessa:

- `protected void onCreate(Bundle savedInstaceState)`

  Si tratta del metodo invocato in corrispondenza della creazione dell'Activity e, nella maggior parte dei casi, contiene le principali operazioni di inizializzazione. L'argomento di tipo `android.os.Bundle` fornisce un modo per ottenere il riferimento ad un eventuale stato dell'Activity prima di essere eliminata dall'Activity Manager; è possibile immaginarlo come un contenitore di coppie chiave-valore in cui è possibile salvare le informazioni nel caso in cui il sistema terminasse l'activity [12].

- `protected void onStart()`

  Se il metodo `onCreate()` è terminato con successo, l'Activity è presente in memoria e viene inizializzata per essere visualizzata sul display.

- `protected void onResume()`

In questo metodo è possibile inserire eventuali inizializzazioni di risorse necessarie alla corretta esecuzione dell'Activity. Terminato il metodo `onResume()`, l'Activity si trova nello stato ACTIVE (in cima allo stack) e permette pertanto l'interazione con l'utente.

- `protected void onPause()`

  Questo metodo viene comunemente invocato premento il tasto BACK del dispositivo, il quale rimuove l'Activity corrente dallo stack e procede con l'eventuale ripristino o la nuova creazione dell'Activity da visualizzare.

  Il sistema, per permettere il prima possibile la visualizzazione della nuova Activity, pone immediatamente l'Activity da eliminare nello stato PAUSED.

- `protected void onRestart()`

  Con il seguente metodo il sistema si preoccupa di ripristinare la precedente Activity. La logica è molto simile a quella del metodo `onCreate()`, con la sola differenza che quest'ultimo viene invocato da un'Activity appena istanziata. L'Activity da ripristinare, sarà quindi visualizzata richiamando i metodi `onStart()` ed `onResume()`.

- `protected void onStop()`

  Con questo metodo, l'Activity Manager, dopo il ripristino dell'Activity tramite il metodo `onRestart()`, procede con la transizione dell'Activity precedente dallo stato PAUSED allo stato STOPPED.

- `protected void onDestroy()`

  Questo metodo viene invocato per eliminare definitivamente l'Activity mettendola nello stato INACTIVE.

I metodi `onPause()`, `onStop()` e `onDestroy()` sono gli unici in cui il sistema ha la possibilità di terminare il processo (*kill*). I processi in background hanno priorità inferiore rispetto a quelli in foreground e per questo motivo il sistema li ordina

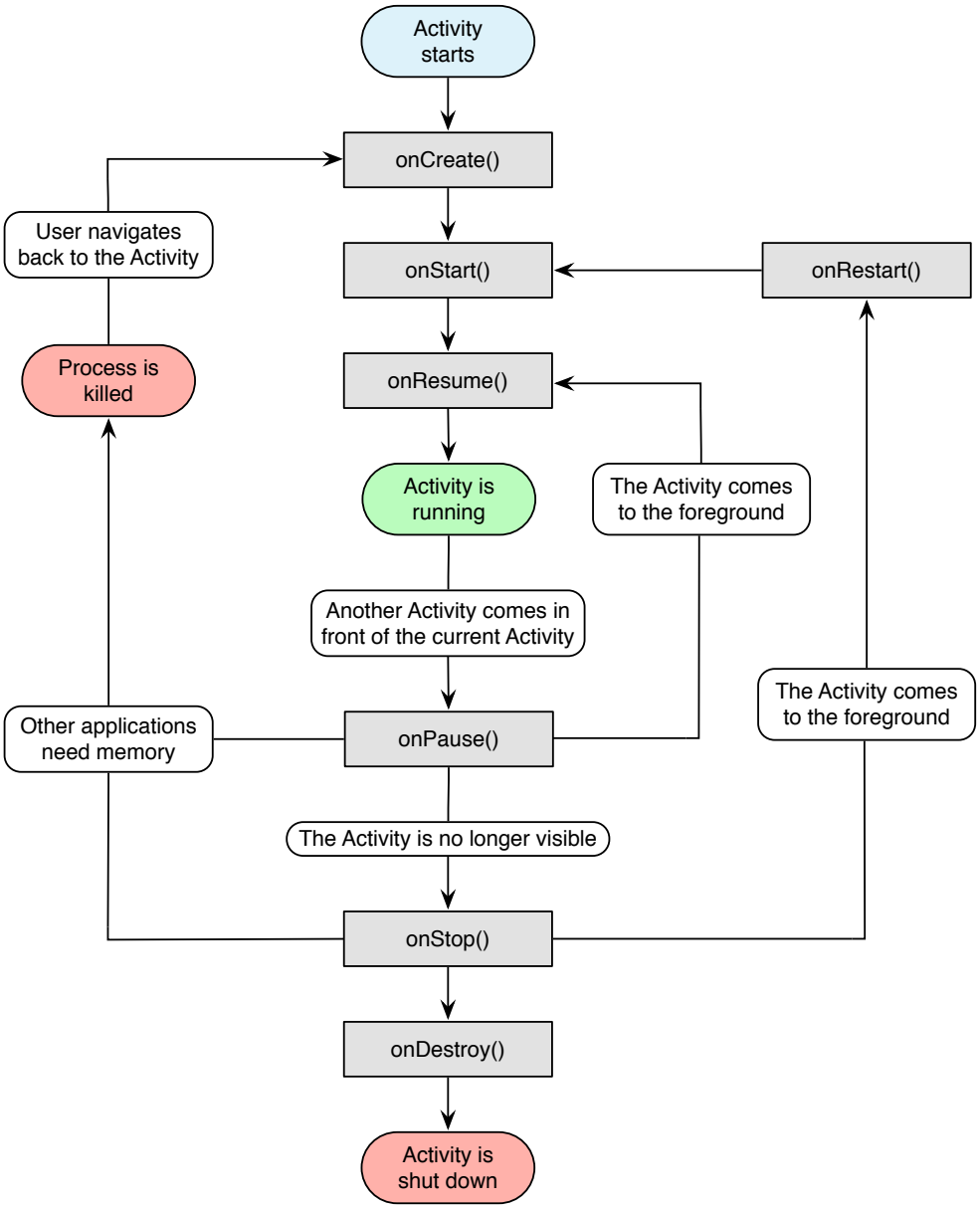

Figura 3.3: Lifecycle di un'Activity.

in base al tempo trascorso dall'ultimo utilizzo attraverso una lista di tipo *Least Recently Used (LRU)* con la quale sarà in grado di determinare quale processo terminare in caso di necessità.

Per evitare la perdita di informazioni il framework di Android mette a disposizione un particolare metodo di *callback*, `onSaveInstanceState(Bundle)`, il quale viene richiamato dal sistema prima di porre l'Activity in uno stato in cui potrebbe essere distrutta dal sistema stesso (ad esempio, per necessità di spazio in memoria).

Un esempio tipico di utilizzo della callback `onSaveInstanceState(Bundle)` è il caso in cui si ha la necessità di gestire l'orientamento del display; in questo, come avviene in tutti gli altri casi di cambio di configurazione del device a runtime, Android si limita a riavviare semplicemente l'Activity utilizzando il metodo `onDestroy()` seguito dal metodo `onCreate()` preservando così lo stato dell'Activity utilizzando la *callback* sopra citata.

3.2 Services

Un Service è un componente definito da un'istanza dell'omonima classe presente nel package *android.app* che permette di svolgere operazioni in background senza la necessità di interazione con l'utente. I Services, a differenza di altri componenti quali le Activity, sono preservati dal sistema che non li elimina se non in situazioni estreme o nel caso in cui non siano utilizzati da nessun altro componente.

In Android analogamente al modello *client-server*, un componente, quale ad esempio l'Activity, assume il ruolo di *client* che richiede ad un particolare Service (in questo caso il *server*) di svolgere alcuni compiti in background per suo conto. Tale richiesta si verifica, ad esempio, quando l'Activity avvia un Service per il download di un file dalla rete. Completato il download, l'applicazione viene chiusa ma il Service può rimanere in esecuzione in background anche mentre l'utente utilizza una seconda applicazione.

Android mette a disposizione due tipologie di Services: *Started* e *Bound*. I primi sono i Services privati di una particolare applicazione, mentre i secondi sono quelli che espongono un'interfaccia di comunicazione utilizzabile da applicazioni diverse tramite meccanismi di *Remote Procedure Calls (RPC)*. Nel primo caso è possibile avviare e fermare il Service, nel secondo, invece, è possibile ottenere un riferimento alle funzionalità dello stesso attraverso la descrizione di una particolare interfaccia. Ciò significa che è possibile ottenere un riferimento (*binding*) ad un Service precedentemente av-

viato da un'altra applicazione. Una stessa implementazione di Service potrà quindi permettere il suo avvio a particolari applicazioni e solamente il binding ad altre [13].

3.2.1 Started Service

Uno Started Service può essere avviato da un'applicazione tramite l'invocazione del metodo `startService()`, definito all'interno della classe *android.content.Context*, il quale chiede al sistema di verificare se il Service in questione è in esecuzione. In caso negativo il framework invoca il metodo di callback della classe Service `onCreate()`. Questo metodo, oltre ad avviare il Service in questione, viene utilizzato anche per svolgere operazioni di inizializzazione dello stesso. Avviato il Service, il sistema è in grado di invocare il metodo `onStartCommand()` inoltrando l'Intent (componente di Inter Process Communication approfondito nel Capitolo 3.3.1) inviatogli dal componente chiamante e contenente i dati necessari al Service per svolgere il proprio compito.

Terminata l'esecuzione di `onStartCommand()`, il Service rimane nello stato RUNNING, a meno di azioni da parte dell'ambiente, fino a quando un componente invoca il metodo `Context.stopService()` oppure fino a quando il Service stesso richiama il proprio metodo `stopSelf()` che porta il sistema ad invocare la callback `onDestroy()` per eliminare definitivamente il Service.

Da notare che, quando un Service viene avviato, esso possiede un ciclo di vita del tutto indipendente dal componente che l'ha richiamato e quindi puo' rimanere in esecuzione in background per un tempo indefinito anche quando il client è stato eliminato. Il metodo `onCreate()` viene invocato una sola volta durante il *lifecycle* del Service; il metodo `onStartCommand()`, al contrario, viene richiamato in corrispondenza di ogni invocazione del metodo `startService()`, ovvero ad ogni richiesta effettuata dal componente chiamante.

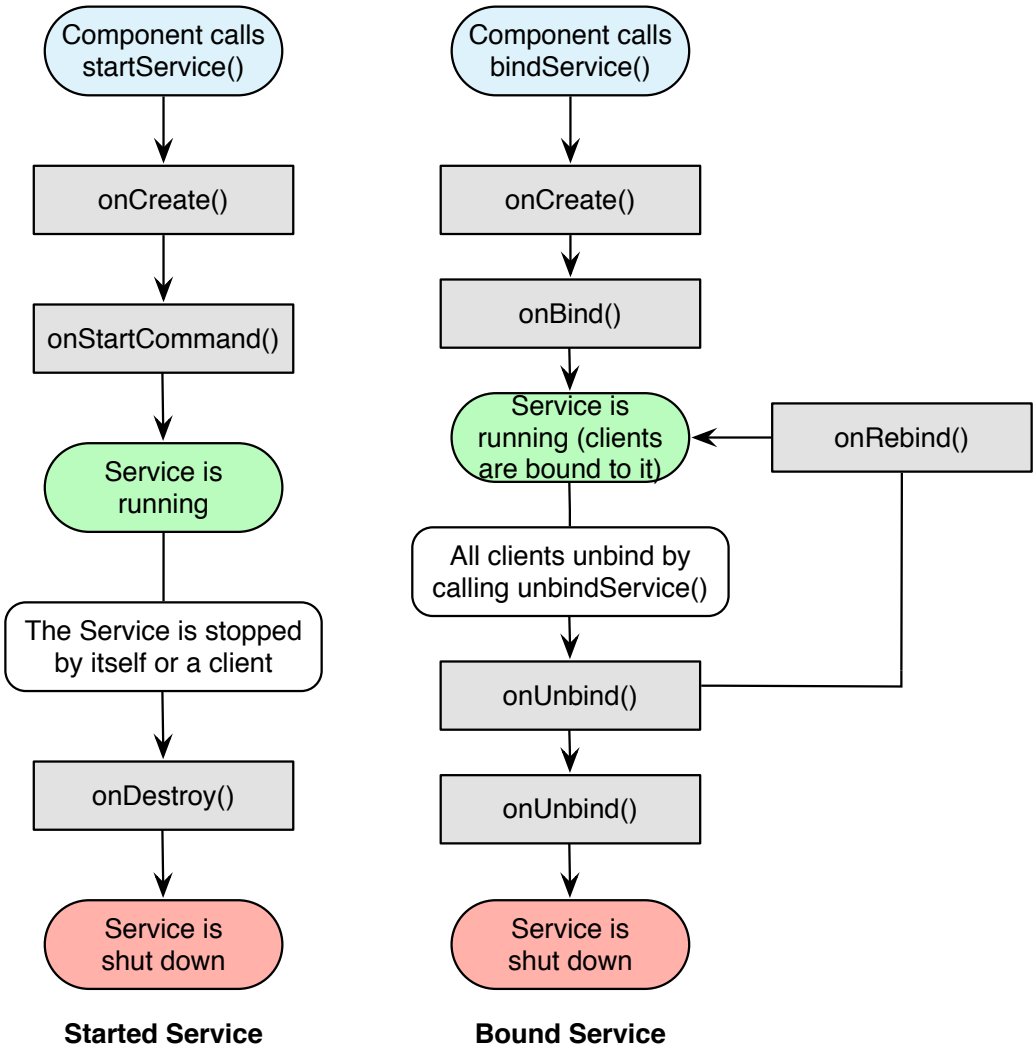

Figura 3.4: A sinistra: il ciclo di vita di uno Started Service. A destra: il ciclo di vita di un Bound Service.

3.2.2 Bound Service

Il concetto di Bound Service nasce dalla necessità di realizzare un componente in grado di eseguire delle operazioni in background ed accessibili da più applicazioni diverse tramite l'utilizzo di meccanismi di *Remote Procedure Calls (RPC)*.

Come è possibile notare dalla Figura 3.4 il Bound Service, a causa della sua diversa natura, ha un ciclo di vita leggermente diverso da quello di uno Started Service.

La modalità di interazione da parte di applicazioni con questo tipo di Service prevede l'invocazione del metodo `Context.bindService()` passandogli come parametro il riferimento all'interfaccia dalla quale sarà possibile accedere alle funzionalità offerte dal Service stesso.

Se il Service non è stato creato precedentemente, il sistema richiama il metodo di callback `onCreate()` come avviene per gli Started Service e, successivamente, il metodo `onBind()`, il quale restituisce il riferimento all'interfaccia di tipo `IBinder` che il client potrà utilizzare per invocare i metodi implementati dal Bound Service. Analogamente agli Started Service, il metodo `onCreate()` viene eseguito solo al momento di creazione del Service mentre il metodo `onBind()` viene invocato ad ogni chiamata del metodo `bindService()` della classe `Context`.

A questo punto il Service rimane nello stato RUNNING e può servire contemporaneamente un numero qualsiasi di client. Quando un'applicazione non intende più utilizzare il Bound Service invoca il metodo `Context.unbindService()`.

Qualora il Service rimanesse senza applicazioni da servire, il framework di Android richiamerebbe il metodo di callback `onUnBind()`, il cui valore di ritorno booleano indica se si desidera o meno l'invocazione del metodo `onRebind()`. Questi ultimi due metodi sono necessari per permettere al Service di liberare eventuali risorse allocate in caso di assenza di client per poi ripristinarle durante l'esecuzione della callback `onRebind()`.

In caso di esplicita richiesta di conclusione del Service o in caso di necessità da parte del sistema, viene invocato il metodo di callback `onDestroy()` per eliminare tale componente dalla memoria.

3.3 Inter Process Communication

3.3.1 Intents

L'Intent è l'oggetto alla base del sistema di *Inter Process Communication* offerto da Android; permette lo scambio di messaggi tra componenti della stessa applicazione o tra applicazioni diverse utilizzando informazioni note a runtime come, ad esempio, la presenza o meno di un particolare Service in esecuzione nel momento di invio dell'Intent.

L'oggetto Intent contiene la descrizione astratta di un'operazione da eseguire; in caso di invio in modalità broadcast rappresenta la comunicazione di un evento avvenuto e che deve essere comunicato alle applicazioni che implementano un particolare componente dell'Application Framework, il *Broadcast Receiver*. Viene tipicamente utilizzato per visualizzare l'interfaccia grafica di un'applicazione, inviare messaggi tra componenti e avviare un Service.

Nella loro forma più semplice, gli Intent (in questo caso si parla di *Intent espliciti*), trovano applicazione nel caso in cui si intenda far comunicare Activity della stessa applicazione già note in fase di sviluppo.

La piattaforma Android permette inoltre l'utilizzo di Intent per la comunicazione tra componenti di applicazioni diverse, la cui conoscenza è determinata solo in fase di esecuzione, sulla base delle informazioni contenute negli *AndroidManifest.xml* delle applicazioni stesse [13]. In questo caso, l'Intent non specifica il nome di un componente di destinazione ma solo il tipo di azione da eseguire che verrà impostata col metodo `setAction()` della classe `android.content.Intent`; è necessario che nel sistema sia installata un'applicazione che abbia dichiarato nel proprio *AndroidManifest.xml* la disponibilità a gestire quel particolare Intent, nel caso in cui il sistema non trovi alcuna applicazione, verrà sollevata un'eccezione del tipo `android.content.ActivityNotFoundException`.

3.3.2 Binders

In Java, due oggetti si intendono remoti se sono in esecuzione in due istanze diverse della Java Virtual Machine [14]. In Android il concetto è analogo: ogni applicazione viene eseguita all'interno di una propria istanza della Dalvik Virtual Machine e il *Binder* viene tipicamente utilizzato per la comunicazione tra un componente (o un'applicazione) ed un Bound Service.

Le operazioni che un componente remoto è in grado di eseguire vengono descritte da un'interfaccia, realizzata tramite un apposito linguaggio chiamato *Android Interface Definition Language (AIDL)* dalla quale, attraverso la compilazione con l'apposito compilatore messo a disposizione dall'Android SDK, verrà generata un'interfaccia di tipo `IBinder` che verrà inviata al client nel momento del *binding* [15].

Binder è un kernel device driver che utilizza la caratteristica della *shared memory* di Linux per ottenere un efficiente e sicuro sistema di Inter Process Communication come precedentemente spiegato nel Capitolo 2.1.1. Un server utilizza tipicamente una sottoclasse di `android.os.Binder` ed implementa il metodo `onTransact()`; il client riceve da esso il riferimento ad una interfaccia di tipo `android.os.IBinder` e invoca il metodo `transact()`.

Entrambi i metodi, `onTransact()` e `transact()`, utilizzano istanze della classe `android.os.Parcel` per lo scambio efficiente dei dati.

Un oggetto `android.os.Parcel` rappresenta a tutti gli effetti un buffer, nel quale è possibile inserire dati e riferimenti ad oggetti, che può essere inviato attraverso un IBinder; l'implementazione nativa di `Parcel` si occuperà successivamente di formattare adeguatamente i dati in modo da permettere al Binder device di gestirli correttamente.

Possedere un'interfaccia `IBinder` permette al client di invocare i rispettivi metodi messi a disposizione dal server (ad esempio: la chiamata `transact()` si traduce nella corrispondente chiamata `onTransact()` lato server); non è tuttavia garantito che il server esegua la richiesta del chiamante. Per esempio, ogni applicazione può ottenere un riferimento all'interfaccia messa a disposizione da Zygote e richiamare il metodo per lanciare un'applicazione come un utente diverso; in questo caso però,

Zygote ignorerà le eventuali richieste pervenute da processi non autorizzati [16].

La sicurezza delle comunicazioni che avvengono tramite l'usilio dei Binder viene garantita dal sistema attraverso due tecniche: *Identity Checking* e *Reference Security*.

Identity Checking

Quando un `IBinder` viene richiamata da un client, l'identità del chiamante viene messa a disposizione in modo sicuro dal kernel.

Android associa l'identità del client (rappresentata dall'application UID e PID assegnato correntemente al processo) col thread nel quale viene gestita la richiesta. Questo permette al ricevente di utilizzare il metodo `checkCallingPermission( String permission)` oppure il metodo `checkCallingPermissionOrSelf (String permission)`, entrambi della classe `Context`, per determinare i privilegi in possesso dal chiamante.

Il Service ha anche la capacità di accedere all'identità del client attraverso i metodi statici `getCallingUid()` e `getCallingPid()` della classe Binder, i quali restituiscono rispettivamente lo User ID e il Process ID del processo che ha effettuato la chiamata.

Reference Security

I riferimenti ad un `Binder` possono essere inviati tramite un'interfaccia `IBinder` utilizzando i metodi `writeStrongBinder()` e `readStrongBinder()` della classe `Parcel` i quali includono alcuni controlli di sicurezza.

Quando viene effettuata la lettura di un `Binder` da un oggetto `Parcel` il binder driver del kernel assicura al ricevente che il componente che l'ha precedentemente scritto abbia effettivamente un riferimento a tale `Binder`. Questo meccanismo è stato ideato per evitare che un client indovini il valore numerico utilizzato dal server per rappresentare una particolare istanza del Binder; in assenza di tale meccanismo il chiamante potrebbe cercare di eseguire metodi del server senza averne il permesso.

I `Binders` sono globalmente univoci così da evitare la possibilità che qualcuno possa creare un `Binder` fittizio facendo credere ad un server che si tratti di un `Binder` legittimo.

3.4 Security Model

Diversamente dai tradizionali sistemi operativi in cui le applicazioni avviate da un utente vengono tutte eseguite con lo stesso UID (a meno di particolari casi come l'utilizzo di *setuid()* e *setgid()* in ambiente UNIX), in Android ciascuna applicazione viene isolata dalle altre. Il sottolivello Linux del sistema Android si occupa di svolgere questo compito di isolamento (*sandboxing*): ciascuna applicazione viene eseguita in un processo diverso con UID e GUID univoci ed è isolata dalle altre applicazioni e dal resto del sistema. Il concetto fondamentale del modello di sicurezza implementato in Android consiste nel fatto che nessuna applicazione, di default, possiede il permesso di svolgere operazioni che potrebbero avere un impatto sulle altre applicazioni, sul sistema operativo o l'utente. Il kernel è il solo responsabile dell'operazione di *sandboxing* la quale viene applicata ad ogni tipo di applicazione: Java, native o ibride.

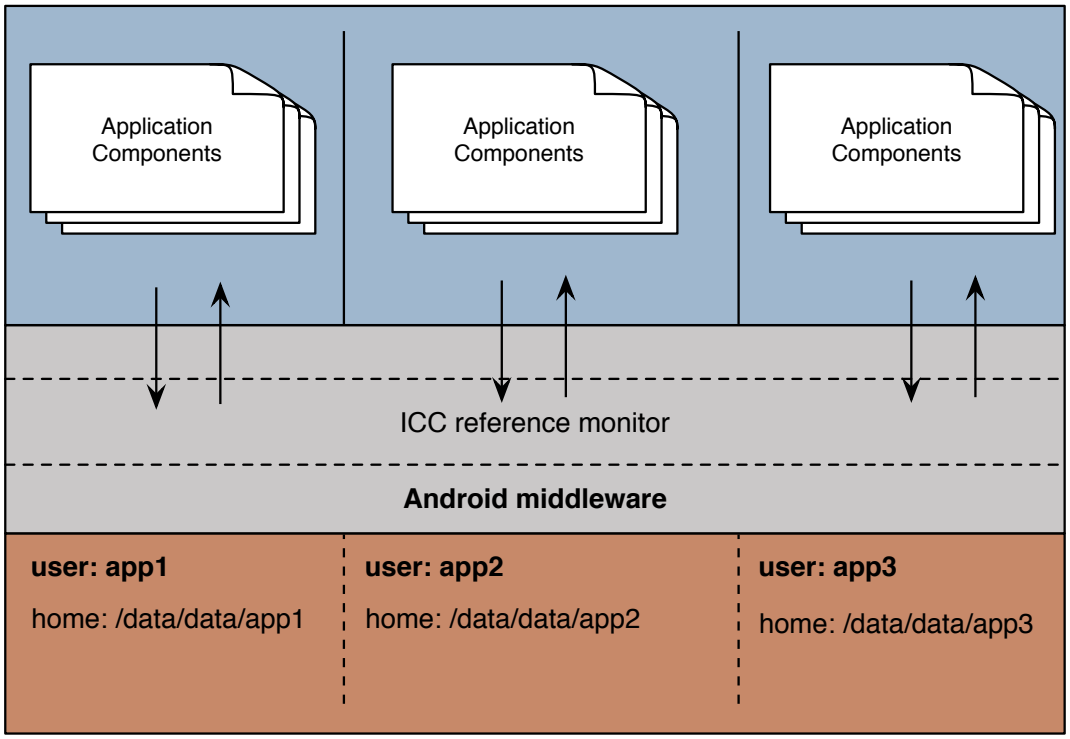

Figura 3.5: Rappresentazione del sandboxing delle applicazioni in Android.

Oltre ad utilizzare la tecnica del *sandboxing*, Android protegge le applicazioni anche a livello di comunicazione tra componenti (*Inter Component Communication - ICC*) utilizzando il concetto di *permesso*. Un *permesso* è una stringa (*label*) che un'applicazione deve possedere per far si che una sua componente possa utilizzare risorse sensibili (quali il GPS, accesso alla rete, rubrica telefonica, ecc...) oppure comunicare con altre componenti (attraverso Intent per esempio).

I permessi di cui un'applicazione necessita sono contenuti all'interno di un particolare file in formato XML, *AndroidManifest.xml*. Completata la fase di installazione non è possibile, per un'applicazione, richiedere i permessi a runtime o, per l'utente, disabilitare dei permessi indesiderabili.

Un *reference monitor* provvede al controllo di accesso ai componenti che un'applicazione cerca di utilizzare. Ciascun accesso ad una componente viene identificato da un *access permission label*. Quando un componente è intenzionato a stabilire una comunicazione con un secondo componente, il *reference monitor* controlla i permessi accordati all'applicazione di cui il primo componente fa parte; se l' *access permission label* del secondo componente è tra quelli contenuti nell'*AndroidManifest.xml* dell'applicazione viene consentita la comunicazioni tra i componenti.

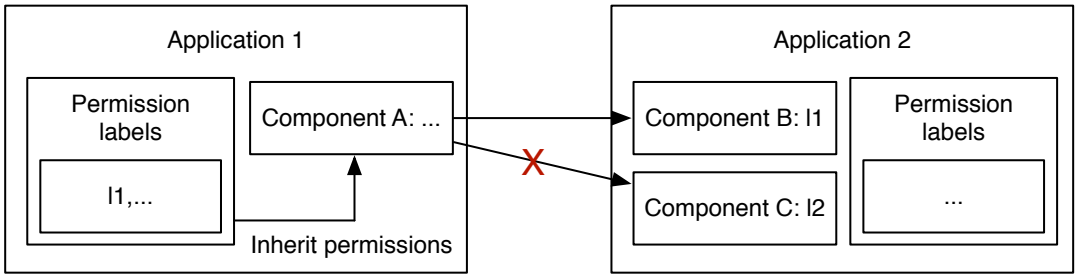

Figura 3.6: Controllo dei permessi attraverso l'Inter Component Communication.

3.4.1 Android Manifest e permessi

In un'applicazione Android (file *.apk*) il file *AndroidManifest.xml*, o più brevemente *Manifest*, assume un ruolo fondamentale tanto da obbligare il sistema ad interrompere l'installazione dell'applicazione stessa nel caso in cui questo file non fosse presente

nella root directory dell'applicazione. Oltre a contenere informazioni sulla struttura dell'applicazione, come la dichiarazione delle componenti da cui è costituita oppure gli Intents che è in grado di gestire, il *Manifest* costituisce il file in cui risiedono tutti i permessi che un'applicazione richiede al sistema per svolgere il proprio compito.

Ciascun permesso possiede una label univoca per poterlo identificare e deve essere specificato all'interno del tag `<uses-permission>` in *AndroidManifest.xml*. Esistono diversi permessi messi a disposizione dal sistema ma ciascuna applicazione può comunque proteggere i proprio componenti definendo nuovi permessi dichiarandoli nel *Manifest* utilizzando il tag `<permission>` [17].

Nel *Manifest* è possibile anche utilizzare il tag `<permission-group>`, il quale dichiara una *label* per un *set* di permessi. Si noti che i permessi che fanno parte di un gruppo non verranno visualizzati all'utente in fase di installazione ma verrà presentato solo il nome e la descrizione del gruppo specificati dallo sviluppatore.

3.4.2 Certificazione delle applicazioni

In Android tutte le applicazioni devono essere certificate affinchè possano essere installate ed eseguite sul dispositivo.

La procedura con la quale viene generata la *signature* del file *.apk*, cioè la firma con cui viene identificato univocamente uno sviluppatore, viene eseguita utilizzando un certificato con chiave privata in possesso dello stesso sviluppatore; la creazione di tale certificato non richiede l'intervento di una *Certification Authority (CA)* ma viene auto generata dallo sviluppatore stesso in fase di rilascio dell'applicazione.

Questo meccanismo di certificazione è necessario al sistema per i seguenti motivi:

Aggiornamenti

Per aggiornare un'applicazione, il sistema confronta la signature dell'upgrade con quello dell'applicazione installata; solo se le due firme risultano identiche allora Android acconsentirà all'installazione dell'aggiornamento.

Modularità

Android permette alle applicazioni certificate con la stessa signature di essere

eseguite all'interno dello stesso processo UNIX e di essere trattate come una singola applicazione. In questo modo uno sviluppatore ha la possibilità di realizzare diversi moduli per la proprio applicazione e di aggiornarli singolarmente.

Condivisione dati tramite permessi

Applicazioni diverse ma con la stessa signature e che utilizzano permessi *signature-based* hanno la garanzia dal sistema di poter condividere in maniera sicura codice e dati tra loro [18].

3.4.3 Installazione di un'applicazione

In fase di installazione l'utente viene informato dal sistema sui permessi richiesti da un'applicazione in base al contenuto del file *AndroidManifest.xml*. Qualora l'utente non accettasse anche solo uno dei permessi richiesti, il sistema non provvederebbe all'installazione di tale applicazione. Dopo aver installato un'applicazione, il sistema non permette di disabilitare dei permessi precedentemente concessi, l'unica possibilità per l'utente è di disinstallare l'applicazione in questione. Durante il controllo dei permessi, l' *Installer* determina se garantire o meno il permesso in esame in base all'identità di chi ha certificato l'applicazione [19].

Dopo aver esaminato i permessi, vengono generati UID e GID univoci coi quali, da ora in poi, verrà identificata l'applicazione all'interno del sistema. A questo punto il sistema crea una entry nella directory `/data/data/` (con permessi di lettura e scrittura per l'applicazione) in cui verranno inseriti, nel caso esistano, gli *assets*[2] utilizzati dall'applicazione stessa. Successivamente, il file *.apk* viene copiato nella directory `/data/app/` in cui verrà conservato il file *AndroidManifest.xml* così da poterlo interrogare a runtime nel momento in cui l'applicazione richieda l'accesso ad una risorsa sensibile. Il file *.apk* in `/data/app/` viene impostato con utente e gruppo proprietario *system* così da impedire all'applicazione di aggiungere permessi al proprio *AndroidManifest.xml* in runtime.

[2]Con il termine *assets* vengono indicati quei file presenti all'interno del file *.apk* a cui un'applicazione puo' accedere in fase di esecuzione [20].

Capitolo 4

Problematiche di sicurezza

Nel seguente capitolo verrà in principio esposta l'analisi dei sorgenti dell'Application Framework svolta durante questo lavoro di tesi per comprendere come viene eseguito il controllo dei permessi da parte del sistema Android nel momento in cui un'applicazione richiede l'utilizzo di una risorsa (dati o dispositivi hardware) ritenuta "sensibile".

Di seguito verranno descritti i tentativi svolti per cercare di aggirare tale sistema di controllo dei permessi e la tecnica utilizzata per testare la Dalvik Virtual Machine a fronte dell'esecuzione di un file dex appositamente corrotto al fine di verificare la presenza di eventuali errori nella programmazione della stessa.

Inoltre saranno analizzati i principali exlpoit conosciuti in ambiente Android utilizzati da alcuni malware per trasmettere informazioni sensibili presenti sui dispositivi a server remoti.

4.1 Android Permission Framework

Con il termine *Android Permission Framework* si intende un insieme di componenti e classi dell'Application Framework che interagiscono tra loro nel momento in cui un'applicazione richiede al sistema di effettuare un'operazione che necessita di particolari permessi.

Uno dei principali componenti di tale framework è l'oggetto `Context` definito dalla classe `android.app.ContextImpl`. L'oggetto `Context` rappresenta, come suggerisce il nome, il contesto in cui il componente di un'applicazione viene eseguito: opera come interfaccia tra l'applicazione ed il sistema Android mettendo a disposizione dell'applicazione un insieme di metodi di uso generale che permettono, ad esempio, la gestione degli Intent, di ottenere la lista dei file privati dell'applicazione, di invocare un secondo componente, etc.

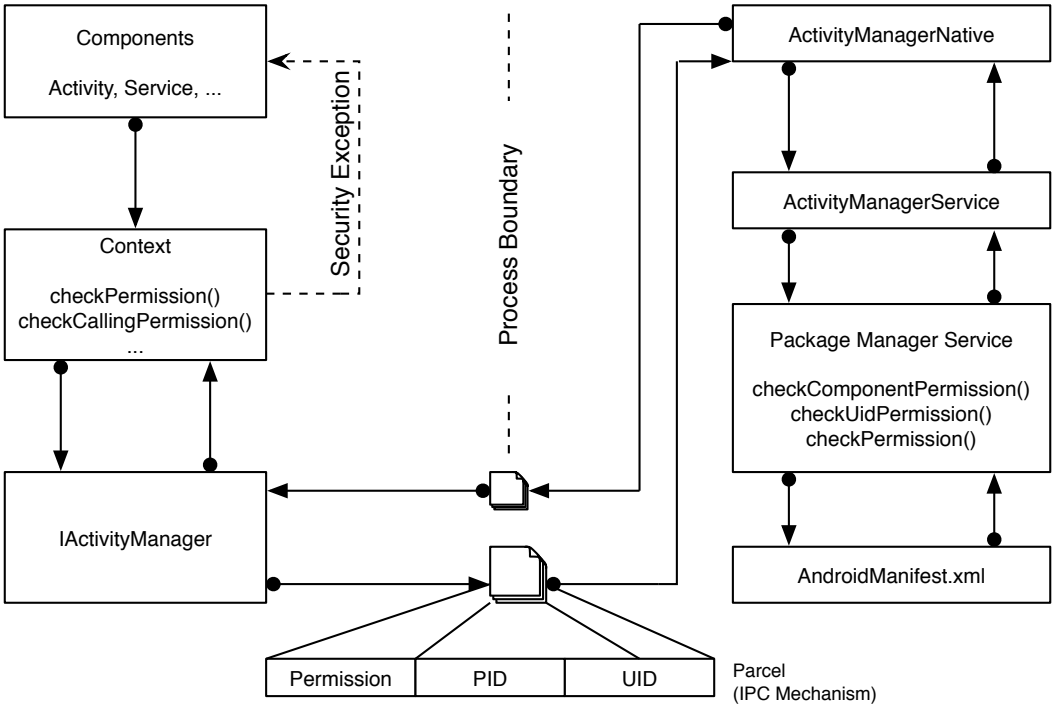

Figura 4.1: Componenti che interagiscono per il controllo dei permessi.

Ogni volta che un'applicazione invia un Intent l'oggetto `Context` controlla l'esistenza di permessi associati a quel particolare Intent e, in tal caso, se tali permessi sono stati concessi all'applicazione che ha sollevato la richiesta.

Per svolgere tali controlli, l'oggetto `Context` crea un `Parcel` contenente il permesso da verificare sotto forma di stringa di caratteri, il Process ID (PID) e lo User ID (UID) del processo in cui l'applicazione chiamante è in esecuzione.

Tale Parcel viene inserito all'interno di un Binder ed inviato all'istanza della classe `ActivityManagerNative` del package `com.android.server.am` che si occupa di estrarre le informazioni in esso contenute e passarle come argomento al metodo `checkPermission()` dell'`ActivityManagerService`.

Come indicato dai commenti presenti all'interno dei sorgenti della classe `ActivityManagerService`, il metodo `checkPermission()` rappresenta l'unica via accedibile dall'esterno dell'Activity Manager per il controllo dei permessi; da qui in avanti, infatti, verranno richiamati metodi privati o utilizzabili solo dall'interno del framework.

Le informazioni contenute all'interno dell'oggetto Parcel, ricevuto dall'Activity Manager Service, vengono passate come argomento al metodo privato `checkComponentPermission()`, il quale eseguirà una serie di controlli: se l'UID del chiamante corrisponde a quello dell'utente *root* (0) oppure a quello dell'utente *system* (1000) il metodo restituisce la costante `PERMISSION_GRANTED` senza alcun controllo aggiuntivo, in caso contrario il controllo passa al Package Manager.

Nel caso in cui il processo chiamante è in esecuzione con permessi diversi da quelli di *root* o *system*, l'Activity Manager richiama il metodo `checkUidPermission()` della classe `com.android.server.PackageManagerService` passandogli come parametri il permesso richiesto e l'UID del processo chiamante.

Tale metodo richiede al Package Manager di eseguire il *parsing* del file *AndroidManifest.xml* contenuto nel file *.apk* corrispondente all'UID passatogli come parametro e di riempire la struttura dati (*HashTable*) `grantedPermission` con i permessi estratti dal file XML. Se il permesso passatogli in input è presente all'interno di `grantedPermission` viene restituita la costante `PERMISSION_GRANTED`; in caso negativo il framework solleva una `SecurityException` non consentendo l'operazione richiesta in quanto il permesso necessario risulta mancante.

La Figura 4.1 rappresenta i componenti che interagiscono per il controllo dei permessi in casi generici quali, ad esempio, l'invio di un Intent da parte di un'applicazione per utilizzare un componente che fa parte di una seconda applicazione.

In altri casi, invece, l'applicazione si connette ad un Service (*binding*) per l'utilizzo di una risorsa hardware che necessita particolari permessi. Vi è quindi la presenza

intermedia di tale Service che riceve le informazioni del processo chiamante attraverso l'oggetto `Context` e le inoltra al `PackageManagerService` per verificare che il client possieda i permessi per l'utilizzo della risorsa hardware da esso gestita.

Ad esempio, come è possibile notare dal Listato 4.1, viene utilizzato un oggetto `LocationManager` per ottenere le coordinate attraverso il ricevitore GPS del dispositivo, mentre l'oggetto `LocationListener` si occupa, ad intervalli di tempo prestabiliti (3 secondi nell'esempio), di richiedere l'aggiornamento di tali coordinate e metterle a disposizione dell'applicazione.

Listing 4.1: Esempio di utilizzo di una risorsa sensibile come il GPS.

```
LocationManager lm;
LocationListener locationListener;

lm=(LocationManager)getSystemService(Context.LOCATION_SERVICE);
locationListener=new MyLocationListener();

lm.requestLocationUpdates(
LocationManager.GPS_PROVIDER, 3000, 0, locationListener);
```

Quando viene invocato il metodo `requestLocationupdates` dell'oggetto `LocationManager`, il `LocationManagerService` riceve un Binder contenente la richiesta di aggiornamento delle coordinate GPS, il permesso necessario per tale operazione e le informazioni del processo chiamante. Il Service inoltra tali informazioni al `PackageManagerService` che esegue i controlli precedentemente descritti per verificare che il processo possieda il permesso di accedere alle informazioni del GPS (permesso `ACCESS_FINE_LOCATION`) e, in caso contrario, solleva un'eccezione del tipo `SecurityException` che termina l'esecuzione dell'applicazione.

Dall'analisi svolta della versione 2.2 (Froyo) del sistema Android, all'interno del framework dei permessi, non si sono riscontrati evidenti difetti che potrebbero permettere ad un'applicazione di ottenere dei permessi a runtime non richiesti al sistema in fase di installazione.

Sono stati effettuati inoltre dei test allo scopo di aggirare il controllo dei permessi del framework di Android le cui principali tipologie sono le seguenti: *generazione di bytecode dinamico* e *utilizzo delle Java Native interface*.

Generazione di bytecode dinamico

Lo scopo di questa tipologia di test è quello di eseguire un'applicazione Java che non utilizza le tipiche componenti dell'Application Framework (Capitolo 2.1.4) così da tentare di aggirare i controlli imposti da tale framework.

Tali test consistono nel creare una classe che utilizza le librerie standard di Java contenute nelle Core Library (specificate nel Capitolo 2.1.3) come, ad esempio, il package `java.net`.

La classe creata viene successivamente compilata in bytecode interpretabile dalla Dalvik Virtual Machine attraverso il tool *dx* incluso all'interno dell'Android SDK e compresso in un file *.jar*.

Il file *.jar* viene incluso all'interno di un'applicazione *.apk* la quale, nel momento del primo avvio, si occupa di copiarlo in un filesystem da essa scrivibile come, ad esempio, all'interno della directory */sdcard* del dispositivo.

A questo punto l'applicazione, utilizzando l'oggetto `DexClassLoader` del package `dalvik.system`, chiede alla Dalvik Virtual Machine di caricare il bytecode contenuto nel file *classes.dex* del pacchetto *.jar* precedentemente salvato.

Caricato in memoria il bytecode, l'applicazione accede a tale classe tramite la caratteristica della *Reflection* di Java per mandare in esecuzione i metodi in essa contenuti.

La *Reflection* è la caratteristica del linguaggio di programmazione Java che permette di esaminare e manipolare le proprietà interne di un programma; può anche essere utilizzata, come in questo caso, per istanziare oggetti di classi non definite all'interno della propria applicazione ma comunque caricate in memoria dalla Virtual Machine [21].

Il seguente listato rappresenta l'utilizzo della *Reflection* Java per eseguire un metodo della classe definita all'interno del file *.jar* che, in questo caso, cerca di scaricare un file dalla rete senza aver definito alcun permesso all'interno del file *Android Manifest.xml*.

Listing 4.2: Esempio di utilizzo di una risorsa sensibile come il GPS.

```
String jarFile = "/sdcard/testDownload.jar";

DexClassLoader classLoader = new DexClassLoader(
jarFile, "/sdcard", null, getClass().getClassLoader());

Class cLoad = classLoader.loadClass("testDownload");
Method mLoad = cLoad.getMethod("download",null);
mLoad.invoke(cLoad.newInstance(), null);
```

Da quanto rilevato dalle prove eseguite con tale tipologia di test, la deduzione che ne è derivata e non supportata dalla documentazione ufficiale è la seguente: il controllo dei permessi non viene effettuato solo nel caso di utilizzo dei componenti dell'Application Framework ma anche per quanto riguarda l'uso delle Core Library.

Tali prove, infatti, hanno come risultato il blocco dell'esecuzione del bytecode a seguito di un'eccezione causata dalla mancanza di permessi; come controprova, la richiesta dello specifico permesso all'interno dell'*Android Manifest.xml* permette all'applicazione secondaria di essere eseguita correttamente.

Utilizzo delle Java Native Interface

Questa tipologia di test è caratterizzata dall'utilizzo delle *Java Native Interface (JNI)*: caratteristica del linguaggio di programmazione Java che permette alle applicazioni scritte con tale linguaggio di utilizzare codice nativo e, quindi, sviluppato in C e C++ [22].

In Android lo sviluppo di codice nativo è reso possibile tramite l'ausilio dell'*Android NDK (Native Development Kit)* il quale utilizza un *cross-compiler* per la compilazione in codice macchina dell'architettura ARM.

Tramite il comando `ndk-build` il codice scritto in C/C++ viene compilato in una libreria dinamica che è possibile caricare dal codice Java attraverso il metodo `System.loadLibrary()`. Una volta caricata la libreria, è possibile utilizzare le funzioni in essa definite specificando nel codice Java dei metodi con lo stesso prototipo e l'attributo `native`.

I test effettuati consistono nel tentativo di caricare una delle librerie (i driver) presenti nel sistema per gestire i dispositivi hardware come, ad esempio, la *libgps.so* per la gestione del ricevitore GPS. Lo scopo finale di tali test è quello di provare ad utilizzare direttamente l'hardware attraverso il codice nativo aggirando così le restrizioni imposte dall'Application Framework di Android.

Il principale problema riscontrato in questi test è che tutte le librerie dinamiche incluse nel sistema all'interno della directory */system/lib/* sono di proprietà dell'utente *root* e il solo permesso accordato al gruppo *others* è quello di lettura. In questo caso quindi risulta di fondamentale importanza il sistema dei permessi Unix poichè la mancanza del permesso di esecuzione al gruppo *others* non permette alle applicazioni, in qualsiasi linguaggio siano esse scritte, di utilizzare le funzioni contenute in tali librerie dinamiche.

Il problema principale del framework di Android consiste nel basso livello di dettaglio (*granularità*) degli stessi permessi messi a disposizione delle applicazioni e l'effettivo uso che queste ultime ne fanno.

Per l'utente è impossibile controllare l'effettivo utilizzo di una risorsa da parte di un'applicazione oppure revocare un permesso precedentemente accordato in fase di installazione; l'unica possibilità è quella di disinstallare l'applicazione.

Ad esempio, un'applicazione malevola si finge una normale applicazione di messaggistica istantanea offrendo anche la possibilità di sincronizzare i contatti della propria rubrica telefonica con la lista degli amici. Per svolgere il proprio compito, l'applicazione richiede l'utilizzo dei permessi `android.permission.NETWORK` e `android.permission.READ_CONTACTS`.

In Android non esiste alcun controllo da parte del sistema che vieti a tale applicazione di leggere le informazioni contenute nella rubrica telefonica al di fuori della sin-

cronizzazione oppure che avvisi l'utente nel momento in cui l'applicazione acceda ad informazioni sensibili, come i contatti, chiedendogli di permettere o meno l'esecuzione di tale operazione.

I permessi accordati ad un'applicazione in fase di installazione vengono sempre concessi dal sistema senza richiedere l'intervento dell'utente.

4.2 Fuzzing della Dalvik Virtual Machine

Oltre all'analisi del modello di sicurezza implementato in Android, in questo lavoro di tesi si sono poste le basi per lo sviluppo di un tool per il *fuzzing* della Dalvik Virtual Machine.

Con il termine *fuzzing* si intende una tecnica di *software testing*, nella sua forma più semplice, che prevede la generazione di casi di test in modo pseudo-casuale. Tale approccio consiste nel collegare l'applicazione da analizzare ad una fonte di dati casuali e, nel caso in cui uno o più di questi dati provochino l'arresto imprevisto del programma o facciano in modo che l'esecuzione entri in un ciclo infinito (o almeno supposto infinito), può essere considerato come sintomo della presenza di errori di programmazione all'interno dell'applicazione stessa.

Tale tipologia di test rientra nella categoria del *black-box testing* poichè l'applicazione da testare viene trattata come una "scatola nera" di cui non si dispone alcuna informazione riguardante il comportamento interno [23].

Per eseguire il *fuzz testing* è stato modificato uno dei principali stumenti dell'Android SDK (Software Development Kit), *dx*. Tale tool ha il compito di tradurre il Java bytecode, generato in fase di compilazione di un'applicazione Android, in Dalvik bytecode interpretabile appunto dalla Dalvik Virtual Machine e la generazione del file *.dex* che lo contiene.

Le modifiche apportate consistono nel sostituire, a seconda dell'opzione ricevuta in input, una o più informazioni contenute all'interno delle varie sezioni del file *.dex* approfondite nel Capitolo 2.3.1.

Per esempio, un'opzione corrisponde a sostituire la dimensione della `stringIds Section` contenuta all'interno dell'header con un valore generato in modo pseudo-

casuale, una seconda opzione invece corrisponde a sostituire il contenuto di una stringa (contenuto nella sezione `Data`) con caratteri ASCII sempre generati in maniera pseudo-casuale, etc.

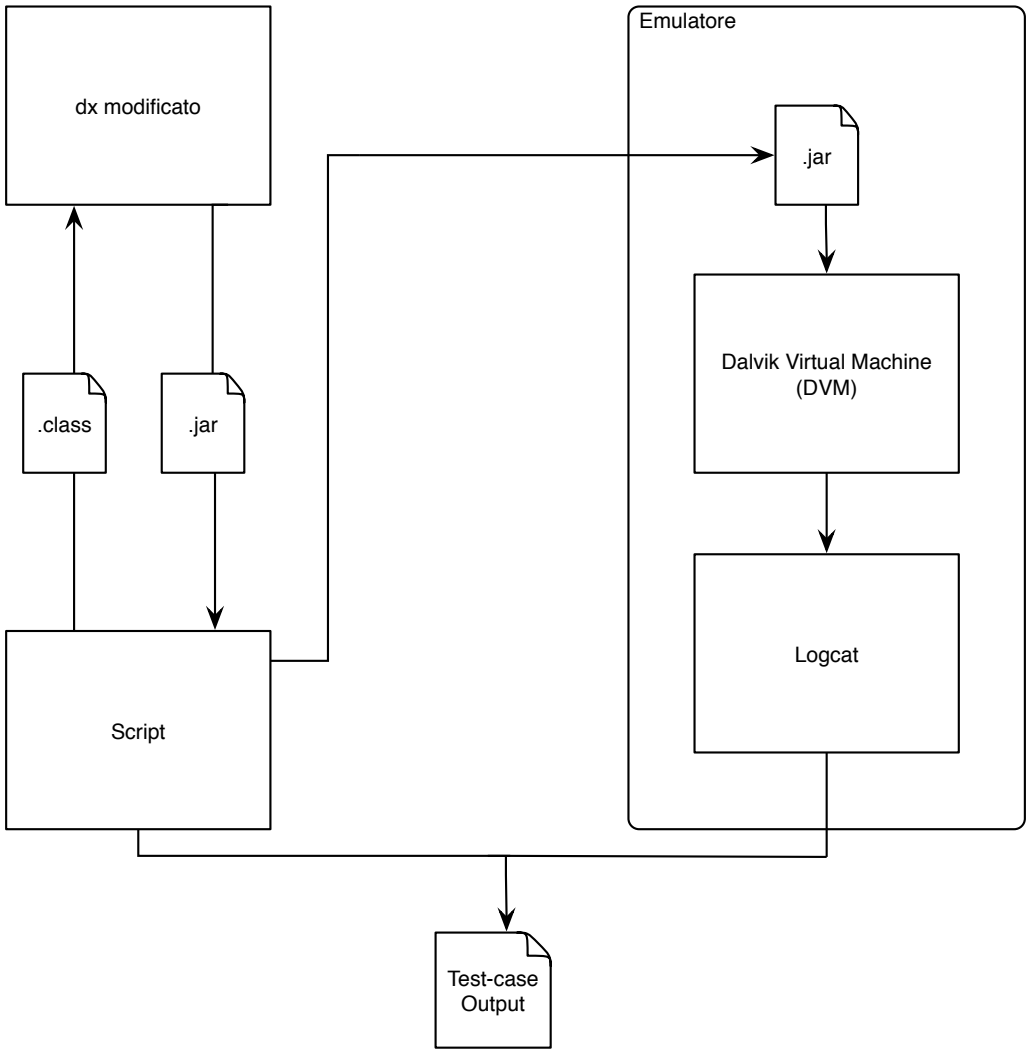

Figura 4.2: Rappresentazione del funzionamento del tool sviluppato per il fuzzing della DVM.

Una seconda componente del tool per il fuzzing della DVM consiste in uno script il quale ha il compito di richiamare il programma *dx* modificato per ciascuna tipologia di *test case* da eseguire. Ad ogni invocazione del software, lo script passa a quest'ultimo il file *.class* generato dalla compilazione del codice Java utilizzato per i test. L'archi-

vio *.jar* (contenente il file *.dex* modificato) generato da *dx* viene caricato all'interno dell'emulatore, rilasciato con l'Android SDK, dallo script attraverso il comando `adb push`. A questo punto lo script si occupa di richiamare la Dalvik Virtual Machine attraverso il comando `adb dalvikvm -cp` passandogli come parametri il percorso dell'archivio *jar* e il nome della classe Java da eseguire.

Infine, l'output della Dalvik Virtual Machine viene intercettato e trascritto in un file di testo associato al corrispondente *test case* dallo script attraverso l'ausilio del comando `adb logcat`, il quale si occupa di stampare i messaggi di log del sistema [24].

Il problema riscontrato durante il lavoro svolto riguarda la presenza di una fase di controllo e una di ottimizzazione dei file *.dex* eseguite entrambe dalla Dalvik Virtual Machine durante il primo avvio di un'applicazione. Tale processo consiste nell'esecuzione dell'applicazione di sistema *dexopt* la quale ha il compito di estrarre il file *classes.dex* contenuto nell'archivio compresso (*.jar* o *.apk*) e, dopo aver effettuato una verifica sul contenuto di tale file, di eseguire una serie di ottimizzazioni quali, ad esempio, la sostituzione degli indici nella sezione `field ids` con l'offset della posizione in memoria in cui verrà posizionato il rispettivo dato in fase di esecuzione dell'applicazione.

Se il processo *dexopt* termina con successo viene creato un file *.odex (Optimized Dex)* all'interno della directory */data/dalvik-cache/* e verrà utilizzato dalla Dalvik Virtual Machine per un'esecuzione più performante dell'applicazione alla quale si riferisce.

Durante la fase di controllo, invece, *dexopt* verifica le informazioni contenute in ciascuna sezione del file *.dex* interrompendo tale processo nel caso in cui siano presenti istruzioni "illegali" quali, ad esempio, riferimenti a stringhe o metodi che non sono effettivamente presenti nella sezione `field ids`, oppure la presenza nell'header di informazioni non corrette (dimensione o offset) riferite alle altre sezioni del file.

Al fine di rendere il tool e le modifiche apportate al programma *dx* realmente utilizzabili per una corretta esecuzione del *fuzz testing* della Dalvik Virtual Machine, risulta necessario lo sviluppo di una tecnica per aggirare il procedimento di verifica e ottimizzazione del file *dex*.

Essendo tali controlli parte integrante della Dalvik Virtual Machine, non è risultato semplice realizzare una tecnica per bypassarli la quale viene demandata ad evoluzioni future del tool sviluppato in questo lavoro di tesi.

4.3 Analisi degli exploit noti

Con il termine *exploit* si indica un codice ideato per sfruttare una vulnerabilità, tipicamente un difetto di programmazione, presente in un software e per far eseguire a quest'ultimo delle operazioni per il quale non era stato originariamente sviluppato [28].

Gli exploit vengono realizzati principalmente per tre scopi:

privilege escalation : permettere ad un utente registrato in un sistema informatico di ottenere maggiori privilegi rispetto a quelli consentiti dal proprio account.

denial of service (DoS) : attacco ideato allo scopo di interrompere l'erogazione di un servizio da parte di un software o, in generale, di un sistema informatico.

esecuzione di codice arbitrario : inclusione nel flusso di esecuzione di un software di codice che non appartiene a quest'ultimo e quindi permettere a tale software di eseguire delle operazioni per le quali non era stato progettato.

I principali exploit conosciuti utilizzati in ambiente Android sono *exploid* e *rageagainstthecage*, entrambi reperibili in rete insieme ai rispettivi sorgenti sul sito del proprio sviluppatore, Sebastian Krahmer [25].

Entrambi gli exploit effettuano una privilege escalation per ottenere una shell con i permessi dell'utente *root*; infatti sono stati originariamente sviluppati per permettere agli utenti di effettuare il *rooting* (accesso come utente root) dei propri dispositivi Android così da poter installare delle versioni personalizzate del sistema (definite *custom ROM*).

exploid

Il seguente exploit sfrutta una vulnerabilità presente nelle versioni antecedenti o uguale alla 1.4.1 del software di sistema *udev* [26]. Tale software non è presente in Android sotto forma di eseguibile come tipicamente avviene nei sistemi Linux ma la maggior parte del suo codice è stato inserito all'interno di *init*, il primo processo in user-space e in esecuzione con i permessi di root [27].

Il software udev viene eseguito come *daemon* (quindi in background) che rimane in ascolto dei messaggi di tipo *uevent* inviati dal kernel attraverso *netlink socket* nel momento in cui un dispositivo viene collegato o rimosso dal sistema; il compito di udev è quello di gestire la creazione e rimozione dinamica della rappresentazione virtuale di tali dispositivi.

La vulnerabilità sfruttata da exploid consiste in un mancato controllo dell'input: le versioni di udev affette da tale bug non controllano se la sorgente dei messaggi uevent ricevuti sia effettivamente il kernel [26].

Listing 4.3: Esempio di utilizzo di una risorsa sensibile come il GPS.

```
if ((sock = socket(
   PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT)) < 0)
   die("[-] socket");
close(creat("loading", 0666));
if ((ofd = creat("hotplug", 0644)) < 0)
   die("[-] creat");
if (write(ofd, path , strlen(path)) < 0)
   die("[-] write");
close(ofd);
symlink("/proc/sys/kernel/hotplug", "data");
snprintf(buf, sizeof(buf), "ACTION=add%cDEVPATH=/..%s%c"
                "SUBSYSTEM=firmware%c"
                "FIRMWARE=../../..%s/hotplug%c", 0,
                basedir, 0, 0, basedir, 0);
printf("[+] sending add message ...\n");
if (sendmsg(sock, &msg, 0) < 0)
   die("[-] sendmsg");
close(sock);
```

Quando exploid viene eseguito, effettua una copia di se stesso in una directory scrivibile dal processo.

Conclusa tale operazione invia il messaggio `NETLINK_KOBJECT_UEVENT` tramite socket al demone udev/init per chiedere a quest'ultimo di avviare la copia del proprio eseguibile nel momento in cui verrà gestito il prossimo evento *uevent hotplug* ovvero, quando viene collegato un nuovo dispositivo al sistema.

La copia dell'eseguibile avviato da init controlla quindi se è stato eseguito con i permessi di root e, in questo caso, rimonta (comando *remount*) la partizione */system* (la quale è montata di default in modalità di sola lettura) e copia il programma */system/bin/sh* (la shell) in */system/bin/rootshell* settando i permessi di quest'ultimo a 0477, cioè con il permesso di esecuzione al gruppo *others* e il bit *user ID* impostati così da permetterne l'esecuzione a tutti gli utenti e di essere eseguito sempre con i privilegi di root.

rageagainstthecage

Il seguente exploit permette di eseguire il processo *adbd (Android's Debugging Bridge daemon)* con i permessi di root.

Come precedentemente illustrato nel Capitolo 2.2, adbd è un processo in background che permette l'esecuzione di una shell (con i permessi dell'utente *shell*) attraverso un collegamento con l'adb client tipicamente installato sul computer dell'utente, permettendo operazioni (limitate) di gestione del dispositivo come, ad esempio, l'installazione/rimozione di applicazioni utente, la visualizzazione dei processi attivi e dei messaggi di log provenienti dal sistema.

Una volta avviato, l'exploit ottiene il numero massimo di processi che il sistema permette di eseguire simultaneamente attraverso il valore indicato in `NPROC`. Successivamente cerca il process ID (PID) del demone adb correntemente in esecuzione analizzando il contenuto della directory */proc*.

A questo punto l'attacco puo' avere inizio: viene avviato un ciclo in un secondo processo in cui l'exploit continua a generare processi figli (*fork-bomb*) fino al

raggiungimento del limite imposto da `NPROC`. Raggiunto tale limite, uno dei processi creati viene eliminato (*kill*) e il demone adb riavviato.

Quando in Android viene avviato il demone adb, inizialmente quest'ultimo possiede i privilegi di root e, solo durante l'esecuzione, viene richiamata la funzione `setuid()` per effettuare una sostituzione dei permessi con quelli dell'account *shell*. Nel momento in cui viene effettuata la chiamata alla funzione `setuid()` il limite di processi gestibili dal sistema è stato raggiunto e, per questo motivo, tale funzione fallisce lasciando quindi il demone adb in esecuzione con i permessi di root.

La vulnerabilità qui sfruttata consiste nel mancato controllo da parte dell'adbd del valore di ritorno della funzione `setuid()` continuando quindi la propria esecuzione anche nel caso in cui tale funzione fallisca.

Tali exploit (o varianti di essi) hanno permesso la realizzazione di alcuni *malware* che, ottenendo una shell con i permessi di root a runtime, accedono ad informazioni sensibili presenti all'interno del dispositivo e le inviano a server remoti senza la necessità di richiedere i permessi necessari attraverso l'AndroidManifest.xml e senza che l'utente si accorga di nulla. L'ultimo malware scoperto, *DroidDream*, era presente in cinquanta applicazioni dell'Android Market ufficiale successivamente rimosse da Google.

Le diverse analisi [32] su tale malware effettuate attraverso *disassembler* specifici per i file *.dex* come, ad esempio, *dedexer* o *smali/backsmali*, concordano sul fatto che DroidDream, dopo essere stato avviato inconsciamente dall'utente, scaricava ed installava ad insaputa dell'utente una seconda applicazione la quale si occupava di inviare ad un server remoto informazioni del dispositivo come IMEI (International Mobile Equipment Identity), IMSI (International Mobile Subscriber Identity), modello del dispositivo, nazione, lingua utilizzata, etc.

Capitolo 5

Conclusioni

In questo lavoro di tesi è stato analizzato nel dettaglio l'*Android Permission Framework* il quale si occupa di effettuare i controlli illustrati nel Capitolo 4.1 nel caso in cui un'applicazione tentasse di accedere a risorse ritenute sensibili per la sicurezza dei dati contenuti nel dispositivo.

Nel medesimo capitolo è stato posto in rilievo il principale problema di tale sistema: il basso livello di granularità dei permessi e l'effettivo uso che le applicazioni fanno di essi. In fase di installazione di un'applicazione il sistema informa l'utente su quali permessi l'applicazione stessa necessiti per il suo corretto funzionamento. L'utente ha solo due scelte: accordare tutti i permessi e quindi procedere con l'installazione oppure interrompere l'installazione nel caso in cui non volesse concedere anche uno solo dei permessi richiesti dall'applicazione.

Una volta che l'applicazione è installata nel sistema le verrà sempre concesso l'utilizzo di una risorsa associata a dei permessi concessi in fase di installazione senza informare l'utente. Questo meccanismo implica però che un'applicazione può far credere all'utente di utilizzare una risorsa sensibile per uno scopo lecito ma che, in realtà, la utilizzi per tutt'altri scopi all'insaputa dell'utente.

Per porre rimedio a tale problema sono in fase di sviluppo due progetti che affrontano tale problematica con differenti approcci:

TaintDroid : progetto realizzato da un gruppo di ricercatori statunitensi [29] che consiste in una serie di modifiche apportate alla piattaforma Android per permettere

di “marcare” (*taint*) i dati sensibili a cui un’applicazione accede così da poterne conoscere l’effettivo utilizzo.

Ad esempio, se un’applicazione accede alla rubrica telefonica e legge alcuni contatti, questi ultimi vengono “marcati” con un particolare identificativo. Se l’applicazione cerca di inviare tali dati attraverso un’interfaccia di rete, *TaintDroid* è in grado di identificarli attraverso la marcatura effettuata precedentemente e di bloccarne o meno l’invio a seconda delle impostazioni effettuate dall’utente.

Come si evince dall’esempio, *TaintDroid* svolge quindi operazioni simili a quelle di un firewall per la piattaforma Android causando però un rallentamento dell’ Inter Process Communication pari a circa il 27% del normale [29]. L’overhead aggiunto da *TaintDroid* non è trascurabile e rende quindi tale soluzione non applicabile in contesti reali ma solamente per operazioni di analisi.

Android Permission Extension (Apex) : ideato dai ricercatori pakistani Mohammad Nauman e Sohail Khan, consiste nell’aggiunta di alcuni componenti a livello dell’*Application Framework* per permettere all’utente di abilitare o disattivare a proprio piacimento uno o più permessi precedentemente accordati ad un’applicazione.

Quando il *Package Manager* effettua l’ultima fase del controllo di un permesso viene invocato un secondo componente, l’*Access Manager*. Tale componente ha il compito di verificare la presenza di regole imposte dall’utente per abilitare o meno il permesso in esame condizionando così il valore di ritorno del *Package Manager* [30].

Alla fine del Capitolo 4 è stato descritto il funzionamento dei principali exploit noti che permettono di ottenere una shell di root su un dispositivo Android. E’ stato anche sottolineato come tali exploit (o varianti di essi) siano stati utilizzati per la realizzazione di *malware* in grado di ottenere una shell di root a runtime così da poter accedere ad informazioni sensibili senza averne richiesto il permesso in fase di installazione.

Anche se le vulnerabilità che gli exploit sfruttano sono state risolte dalla versione 2.2.1 (Froyo) di Android risultano essere ancora funzionanti su una vasta gamma di dispositivi.

Il problema risiede nel fatto che gli aggiornamenti di Android non provengono da Google stessa ma dai produttori dei diversi dispositivi i quali risultano, fino ad oggi, estremamente lenti nel rilascio delle nuove versioni del sistema. Google rilascia i sorgenti della nuova versione di Android i quali vengono prelevati dai produttori di dispositivi che si occupano di apportare le proprie modifiche e aggiunte (modifiche all'interfaccia grafica, aggiunta di driver per hardware specifici, etc.) per poi inviare l'aggiornamento a ciascun dispositivo via OTA (Over the Air).

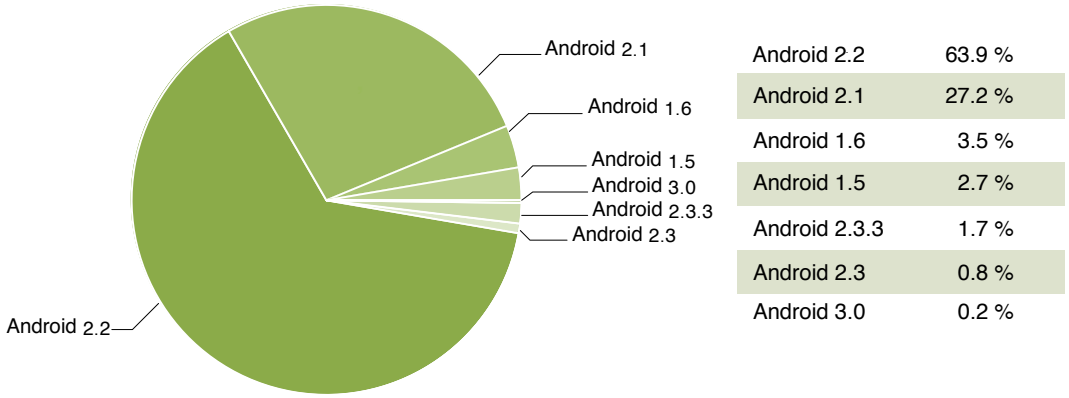

Figura 5.1: Frammentazione di Android. Dati pubblicati da Google, Aprile 2011 [31].

Come è possibile notare dalla Figura 5.1 il panorama delle versioni di Android utilizzate è ancora molto frammentato il quale implica che l'utilizzo da parte di malware degli exploit analizzati in questo lavoro di tesi è ancora possibile su una vasta gamma di dispositivi.

Bibliografia

[1] Embedded Linux, *Android Kernel Features*. http://elinux.org/Android_Kernel_Features, scaricato il 11 aprile 2011.

[2] Dianne Hackborn, *OpenBinder*, Ottobre 2007. http://www.angryredplanet.com/~hackbod/openbinder, scaricato il 11 aprile 2011.

[3] Patrick Brady, *Anatomy & Physiology of an Android*, Google I/O 2008. http://http://sites.google.com/site/io/anatomy--physiology-of-an-android, scaricato il 11 aprile 2011.

[4] Dr. Richard Hipp, *About SQLite*, http://www.sqlite.org/about.html, scaricato il 11 aprile 2011.

[5] David Blythe, *OpenGL ES Common/Common-Lite Profile Specification*.

[6] Google Inc., *Android Reference: Package Manager*. http://developer.android.com/reference/android/content/pm/PackageManager.html, scaricato il 11 aprile 2011.

[7] *Android Adb Analyse*, 2009. http://blog.csdn.net/liranke/archive/ 2009/12/13/4999210.aspx, scaricato il 11 aprile 2011.

[8] Google Inc., *Bytecode for the Dalvik VM*.

[9] Dan Bornstein, *Dalvik VM Internals*, Google I/O 2008. http://http://sites.google.com/site/io/dalvik-vm-internals, scaricato il 11 aprile 2011.

[10] Roberto Aloi, *Gestione della Memoria e Garbage Collection in Real-Time Java*, Ottobre 2005.

[11] Google Inc., *Android Reference: View*. `http://developer.android.com/reference/android/view/View.html`, scaricato il 11 aprile 2011.

[12] Google Inc., *Android Reference: Bundle*. `http://developer.android.com/reference/android/os/Bundle.html`, scaricato il 11 aprile 2011.

[13] Massimo Carli. *Android. Guida per lo sviluppatore*. Apogeo, 2010.

[14] Oracle Corporation, *Java Remote Method Invocation*, `http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html`, scaricato il 11 aprile 2011.

[15] Google Inc., *Android Interface Definition Language (AIDL)*. `http://developer.android.com/guide/developing/tools/aidl.html`, scaricato il 11 aprile 2011.

[16] Jesse Burns. *Mobile application security on Android.*, Giugno 2009.

[17] Google Inc., *Android Reference: Manifest permissions*. `http://developer.android.com/reference/android/Manifest.permission.html`, scaricato il 11 aprile 2011.

[18] Google Inc., *Android Reference: Permission protection level*. `http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel`, scaricato il 11 aprile 2011.

[19] Google Inc., *The AndroidManifest.xml File*. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`, scaricato il 11 aprile 2011.

[20] Google Inc., *Application Resources*. `http://developer.android.com/guide/topics/resources/index.html`, scaricato il 11 aprile 2011.

[21] Glen McCluskey, *Using Java Reflection*, Gennaio 1998.

[22] Sheng Liang, *The Java Native Interface Programmer's Guide and Specification*, ADDISON-WESLEY, Maggio 1999.

[23] Roberto Paleari, *Analisi dinamica di codice binario*, 2006.

[24] Google Inc., *Logcat*. `http://developer.android.com/guide/developing/tools/logcat.html`, scaricato il 11 aprile 2011.

[25] Sebastian Krahmer, `http://c-skills.blogspot.com/`, scaricato il 11 aprile 2011.

[26] National Vulnerability Database, *Vulnerability Summary for CVE-2009-1185*, Aprile 2009 `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-1185`, scaricato il 11 aprile 2011.

[27] Benn, *Android Root Source Code: Looking at the C-Skills*, Settembre 2010. `http://intrepidusgroup.com/insight/2010/09/android-root-source-code-looking-at-the-c-skills/`, scaricato il 11 aprile 2011.

[28] Charles P. Pfleeger, Shari Lawrence Pfleeger, *Security in Computing (4th Edition)*, Prentice-Hall, 2007.

[29] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth, *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones*.

[30] Mohammad Nauman, Sohail Khan, *Design and Implementation of a Fine-grained Resource Usage Model for the Android Platform*, Marzo 2010.

[31] Google Inc., *Application Stats on Android Market*, Marzo 2011. `http://developer.android.com/resources/dashboard/platform-versions.html`, scaricato il 11 aprile 2011.

[32] Lookout, Inc., *DroidDream Technical Tear Down*, Marzo 2011. `http://blog.mylookout.com/droiddream/`, scaricato il 11 aprile 2011.