# Breve panoramica sull'uso di SELinux in Android

CERT-AGID

**Sommario**

SELinux è stato introdotto in Android con la versione 4.3[1]. Come per gli altri utilizzi del kernel Linux, anche in Android SELinux *non* sostituisce il meccanismo di autorizzazione classico, il suo principale utilizzo è quello di "contenimento" del danno prodotto da una falla di sicurezza. In questo documento verrà mostrato come SELinux è utilizzato in Android 10 per isolare applicazioni, servizi, file (in senso Unix) e le proprietà di sistema, qualora una malconfigurazione o una vulnerabilità metta l'attaccante in una posizione privilegiata.

## Indice

# 1 Introduzione a SELinux

Il tradizionale meccanismo di autorizzazione di Linux, di tipo DAC[2], soffre di due problemi:

1. **É poco espressivo**. Si tratta di un meccanismo RBAC che non si adatta bene a tutte le situazioni.
2. **Fa affidamento sugli utenti**. In quanto DAC la responsabilità di proteggere determinare risorse spetta al proprietario di queste.

Queste limitazioni hanno portato l'NSA[3] a sviluppare SELinux.

Mentre il secondo punto è stato risolto elegantemente da SELinux, tramite l'adesione di una politica MAC, il primo punto è stato affrontato nel classico modo in cui si affrontano tutti i problemi con specifiche generiche: introducendo livelli di astrazione su livelli di astrazione. Per questo motivo SELinux risulta piuttosto generico e poco chiaro nella sua documentazione.

## 1.1 Soggetti e oggetti

Come ogni meccanismo di autenticazione, SELinux regola le azioni che dei *soggetti* possono compiere su degli *oggetti*.

Anche se, tecnicamente, qualsiasi entità può fare da soggetto, a meno di non aver qualche LSM personale compilato nel kernel, gli unici soggetti sono i **processi**[1].
Gli oggetti sono qualsiasi risorsa con cui un processo può interagire, tipicamente sono file (o loro descrittori), processi, device o socket ma, a patto di avere l'opportuno LSM, tutto può essere un oggetto SELinux. Android ad esempio definisce una sua classe di oggetti denominata `binder`.

Il compito di SELinux è quindi quello di, dato un soggetto, un oggetto ed un'azione, determinare se il soggetto può effettuare l'azione sull'oggetto.

## 1.2 Contesti, domini, label e tipi

Uno dei motivi per cui SELinux è piuttosto complesso da capire è dovuto all'inconsistenza della documentazione: dieci manuali diversi su SELinux daranno dodici nomi diversi delle entità in gioco.

SELinux ha bisogno di un meccanismo per rappresentare i soggetti e gli oggetti, nel tentativo di rimanere il più possibile flessibile, è stato scelto di rappresentare entrambi con il concetto di **contesto**.

É importante riflettere sul fatto che, per quanto riguarda SELinux, si è solo interessati a rappresentare un soggetto od un oggetto dal punto di vista della sicurezza. Un contesto è solo una stringa che rappresenta *il livello di sicurezza* posseduto (richiesto) dal (per accedere al) soggetto (oggetto).

Un contesto ha la forma

```
utente:ruolo:tipo:livello
```

che spiegheremo tra poco.

Per la comprensione della documentazione su SELinux, è importante notare che quando un contesto è applicato ad un oggetto è detto **label**[4] [2].
Quando un contesto è applicato ad un soggetto (leggi: un processo), viene detto **dominio**.

Dei campi presenti in un contesto, solitamente `livello` non viene usato (è usato per implementare MLS e di rimando MCS e quando non usato ha il solito valore in tutti i contesti) mentre `utente` e `ruolo` sono utili principalmente per i soggetti[5] e si limitano a determinare l'insieme dei possibili valori validi per `tipo`[3].
Ne consegue che `tipo` è il campo più "importante" di tutti, al punto che, purtroppo, viene spesso usato al posto del contesto quando ci si riferisce ai meccanismi di SELinux. Capiterà quindi di trovare documenti in cui `tipo` è chiamato domino (o label) o viceversa.

## 1.3 Classi, SLM e permessi

Il meccanismo di autorizzazione legacy di Linux, quello DAC, si basa sul concetto che tutto è un file in Linux e sul fatto che le operazioni che si vogliono fare sui file è leggerli, scriverli ed eseguirli.

[1] Non siamo precisi qui, Linux ragiona in termini di thread ma non ci interessa la differenza in questo contesto.

[2] Perché, per i file, i contesti sono applicati secondo un insieme di regole che devono poter essere rivalutate nel tempo (tipo dopo un update dell'OS che potrebbe aver modificato le regole stesse). A questa operazione si da il nome di "rietichettatura".

[3] Qui si vede la prima incongruenza di SELinux nel rappresentare soggetti e oggetti nello stesso modo.

Per le directory già si vede come questo meccanismo sia poco flessibile, il flag $X$ è infatti riusato per indicare la attraversabilità della directory.

Questi tre permessi sono stati usati anche per l'accesso ai device file, i quali sono essenzialmente un modo per chiamare funzionalità kernel. Ci si è quindi resi conto che questi tre permessi non sono sufficientemente espressivi.
Sebbene questo avesse poca influenza sui processi non privilegiati, per quelli privilegiati già il kernel aveva discretizzato i permessi in un insieme di *capabilities*[6].

Durante lo sviluppo di SELinux non si voleva quindi commettere lo stesso errore di avere un insieme generico di permessi, per cui sono state definite le **classi**. Un classe rappresenta una tipologia di risorsa (da non confondersi con il campo `tipo` del contesto) sulla quale sono possibili delle azioni che richiedono dei privilegi. Ad esempio `file` è una classe, così come `socket`. Ogni classe ha associato un insieme di azioni, ad esempio i file possono essere letti, scritti, aperti e così via.

Notare che un'azione identifica il relativo permesso che è necessario avere per compierla. L'azione `open` della classe file indica anche il permesso omonimo, le azioni che non richiedono permessi non sono oggetto di SELinux ed ogni azione richiede uno ed un solo permesso. Per questo motivo quello che abbiamo chiamato azioni sono in realtà denominati permessi.

Linux ha un insieme di classi di default e relativi permessi ma altri se possono essere aggiunti tramite l'uso di Linux Security Module, che permettono di estendere i check effettuati dal kernel Linux riusando il framework di SELinux.

## 1.4 Policy e controllo d'accesso

SELinux deve determinare se il contesto del soggetto può fare un'azione nel contesto dell'oggetto. Usando la terminologia che si trova in rete, SELinux deve determinare se un processo nel dominio $D$ può accedere alla risorsa di tipo $T$[4].

Per fare ciò i due contesti sono analizzati per vedere se ci sono regole che **permettono** l'accesso. La politica di SELinux è whitelist: di default nessuna interazione tra contesti diversi è permessa, solo tramite delle regole apposite è possibile permettere interazioni.

Il controllo di accesso di SELinux si basa su due meccanismi:

- Regole di accesso, stabiliscono, dato un `tipo` $T$, quali permessi possiede su altri `tipi` $S_1$, $S_2$, ....
- Vincoli, limitano l'applicabilità delle regole di accesso e permettono confronti basati sugli altri campi del contesto.

Notare come le regole di accesso si basino solo sui tipi presenti nei contesti, non considerano utente, ruolo o livello. Questi sono infatti considerati dai vincoli.

SELinux è quindi fondamentalmente un MAC di tipo Type Enforcement, sul quale è possibile emulare MAC di tipo RBAC e MLS/MCS tramite opportuni vincoli.

Le regole di accesso hanno la forma

```
allow tipo_soggetto tipo_oggetto:classe_oggetto { permesso1, ..., permessoN }
```

Dove vengono concessi i permessi `permesso1`, ..., `permessoN`, ad un processo nel dominio `tipo_soggetto`, sulla risorsa di tipologia `classe_oggetto` e tipo (a.k.a. label) `tipo_oggetto`.

Queste regole non permettono, per sintassi, di fare decisioni sulla base degli altri campi dei contesti[7] coinvolti.

Il check fatto dai vincoli si basa sul concetto di *dominanza* tra i livelli e ruoli (anche se questa funzionalità è stata deprecata)[8].

- Un livello $L$ domina un livello $M$ se e solo se $L \geq M$ e ogni categoria di $M$ è anche in $L$.
- Un ruolo $R$ domina ruolo $S$ sse $R \geq S$.

La relazione $\geq$ è totale nei rispettivi domini (livelli e ruoli) ed è definita tramite apposite regole della policy che richiedono all'autore di mettere in ordine tutti gli elementi.

Con questo meccanismo d'accesso è facile creare MAC di varie tipologie:

[4] Abbiamo usato il termine "tipo" perché di solito viene usato questo, ma ovviamente il termine giusto è *label*. I due termini sono usati l'uno al posto dell'altro frequentemente.

**RBAC** Un unico tipo, un unico livello, un'unica categoria, più utenti e più ruoli. Le policy sono implementate principalmente con i vincoli.

**MLS** Un unico tipo, più livelli, un'unica categoria, un unico utente ed un unico ruolo. Le policy sono implementate principalmente con i vincoli.

**MCS** Un unico tipo, un unico livello, più categorie, un unico utente ed un unico ruolo. Le policy sono implementate principalmente con i vincoli.

**TE** Più tipi, un unico livello, un unica categoria, un unico utente ed un unico ruolo. Le policy sono implementate principalmente tramite le regole di accesso.

Ovviamente nella realtà un sistema non sarà mai esattamente un'istanza pura di queste tipologie, anche se molti sistemi si avvicinano al tipo TE.

### 1.5 Modalità *permissive* e *booleans*

SELinux può messo nella modalità *permissive* in cui si limita a riportare le violazioni di policy (ma non a farle rispettare). Ovviamente il cambio di stato può essere attivato solo se permesso dalla policy e dalla configurazione kernel [10] e consiste nello scrivere il carattere "0" nel file `/sys/fs/selinux/enforce`.

Infine, alcune regole delle policy sono soggetti a vincoli booleani che possono essere attivati o disattivati al volo tramite l'interfaccia `/sys/fs/selinux/booleans`. Questo, supposto che sia permesso, dà la possibilità agli amministratori di spegnere temporaneamente alcune parti della policy caricata.

Non sarà affrontato il discorso dei SID e dell'Access Vector Cache (AVC) per questioni di brevità. Tuttavia questi particolari implementativi risultano piuttosto semplici una volta compreso l'algoritmo di decisione e possono essere trovati, ad esempio, in [10].

Possiamo solo aggiungere che ogni classe di oggetti può definire fino a 32 permessi, in modo che le regole di accesso possano essere rappresentate con una matrice di interi a 32 bit. Nelle righe e nelle colonne si trovano i tipi e ogni cella $(i, j)$ contiene l'intero a 32 bit che indica quali permessi ha il tipo $i$ sul tipo $j$. C'è quindi una matrice per ogni classe di oggetti. I tipi devono essere mappati su un valore numerico che funga da indice di riga e di colonna, questo valore numerico è detto *sid* (Security ID). La matrice è detta `AVC`. Il calcolo delle matrici è compito del componente kernel di SELinux detto `security server`.

### 1.6 Attributi estesi e file system

Risulta importante notare che il contesto (a.k.a. label) dei file è salvato negli attributi estesi (più precisamente nell'attributo `security.selinux`), per cui tutti i file system che non supportano attributi estesi (come *yaff2*, il vecchio FS di Android) non permettono di etichettare i file. In questo caso un contesto di default, definito da policy, viene assegnato ai file.

## 2 Metodologia e ambiente di test

Per verificare l'impiego di SELinux in Android abbiamo deciso di adottare un approccio "Reverse engineering supportato dalla documentazione". Con questo si intende una metodologia che consiste nello scrutinare il codice sorgente della versione 10 di Android[5] al fine di determinare come SELinux venga impiegato. Data la *vastità* e complessità del progetto Android[6] verifichiamo ogni conclusione tratta contro la documentazione disponibile in rete.

Così facendo diamo priorità al codice, unica fonte di verità, rispetto a quanto riportato nella documentazione o già analizzato.

Le parti di Android che sono protette da SELinux ma non vi interagiscono direttamente, saranno invece descritte come riportato nella documentazione o in altre sedi, a meno che questa descrizione non sia mancante e si renda necessario uno studio dei sorgenti.

[5] Quella preinstallata di default con Android Studio al momento della scrittura di questo documento.
[6] Che va ben al di là delle 50 ore previste.

Al fine di poter verificare la configurazione SELinux, è utile disporre di un ambiente di test; per questo scopo è possibile usare un cellulare Android roottato[7] a patto che con il root sia inclusa anche la relativa policy SELinux[9].
Un'alternativa meno costosa, e adatta anche a chi preferisce altri sistemi operativi, è quella di usare l'emulatore di Android. Android Studio e relativi SDK installano un'immagine x86 di Android pensata per il debug di applicazioni. In essa è incluso `su` ed è fornita della relativa policy SELinux per permettergli di mettere SELinux in modo *permissive*.

Può far comodo installare strumenti come *toybox* e *busybox*, con i quali si assume familiarità.

# 3 Tutto inizia da `init`

Come ogni buon sistema ad imitazione di Unix, anche in Android il primo processo lanciato dal kernel è `init`. La sua implementazione si trova in `/platform/system/core/init`[8], nel file `main.c` è presente l'entrypoint di `init` come mostrato in figura 1.

Figura 1: Funzione `main` di `init`.

```
int main(int argc, char** argv) {
    #if __has_feature(address_sanitizer)
        __asan_set_error_report_callback(AsanReportCallback);
    #endif
        // Boost prio which will be restored later
        setpriority(PRIO_PROCESS, 0, -20);
        if (!strcmp(basename(argv[0]), "ueventd")) {
            return ueventd_main(argc, argv);
        }

        if (argc > 1) {
            if (!strcmp(argv[1], "subcontext")) {
                android::base::InitLogging(argv, &android::base::KernelLogger);
                const BuiltinFunctionMap& function_map = GetBuiltinFunctionMap();

                return SubcontextMain(argc, argv, &function_map);
            }

            if (!strcmp(argv[1], "selinux_setup")) {
                return SetupSelinux(argv);
            }

            if (!strcmp(argv[1], "second_stage")) {
                return SecondStageMain(argc, argv);
            }
        }

        return FirstStageMain(argc, argv);
    }
```

Risulta evidente dal codice sopra che il setup di SELinux avviene se ad `init` viene passato il parametro `selinux_setup`, ma quando e come viene passato questo parametro? Dentro `/platform/system/core/init/README.md` troviamo la risposta (vedi figura 2): la sequenza di boot è divisa in tre stadi, in ognuno dei quali `init` è invocato. Il kernel monta il FS di root secondo quanto configurato (con o senza un passaggio da un initramfs) e poi invoca il primo stadio di `init`. Questo stadio si occupa di preparare un ambiente minimale creando vari punti di mount, dopodiché il secondo stadio viene invocato per l'abilitazione di SELinux.

Questo stadio si occupa del caricamento della policy di sistema ed è interamente compiuto dalla funzione `SetupSelinux` in `/platform/system/core/init/selinux.cpp`.

L'analisi del codice C++ è piuttosto semplice, un commento ad inizio file spiega l'algoritmo per il caricamento. La policy di SELinux può essere *monolitica* o *divisa*.

[7] Con questo neologismo, che ci risparmia battute sulla tastiera, si intende un cellulare sul quale è stato installato `su`.

[8] Non saranno forniti link al sorgente Android. Si assume familiarità con il reperimento del sorgente e la gestione dei repository. Facciamo solo notare che Android è diviso in vari moduli che compongono la gerarchia di directoty completa del progetto, questi moduli possono essere trovati replicati in altre piattaforme (tipo GitHub) con nomi che sono stati però normalizzati.

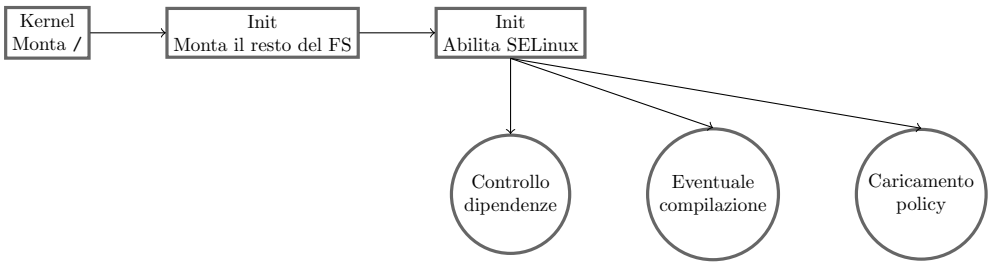
Figura 2: Esecuzione multi stadio di `init`.

```
The early init boot sequence is broken up into three stages: first stage init, SELinux setup, and
second stage init.

First stage init is responsible for setting up the bare minimum requirements to load the rest of the
system. Specifically this includes mounting /dev, /proc, mounting 'early mount' partitions (which
needs to include all partitions that contain system code, for example system and vendor), and moving
the system.img mount to / for devices with a ramdisk.

[...]

Once first stage init finishes it execs /system/bin/init with the "selinux_setup" argument. This
phase is where SELinux is optionally compiled and loaded onto the system. selinux.cpp contains more
information on the specifics of this process.

Lastly once that phase finishes, it execs /system/bin/init again with the "second_stage"
argument. At this point the main phase of init runs and continues the boot process via the init.rc
scripts.
```

Una policy monolitica è composta da un unico file (`/selinux`) che è caricato da `init` nella sua interezza. Una policy divisa è composta da due parti: una parte riguardante i componenti standard di Android (platform) ed una parte personalizzata dal produttore. La prima si trova in `/system/etc/selinux/-plat_sepolicy.cil`, la seconda si trova nella directory del produttore (che dipende da vari parametri di build[9]). Quest'ultima parte non sarà considerata in quanto riguarda i componenti aggiuntivi installati dal produttore[10].
Per le policy divise `init` non si limita a caricare due o più file anziché uno, controlla anche che "le dipendenze" della policy del produttore siano "aggiornate". Questa infatti dipende dalle regole dichiarate nella policy platform, per garantire che non vi siano inconsistenze la policy platform è fornita con il proprio SHA256 e la policy del produttore è fornita con lo SHA256 della policy platform che usa come dipendenza. Qualora i due non corrispondessero, `init` ricompila la policy del produttore (si veda il sorgente per maggiori dettagli).

Possiamo quindi riassumere il caricamento della policy SELinux come mostrato in figura 3.

Figura 3: Caricamento della policy SELinux.




# 4 I sorgenti della policy

Il processo `init` carica la policy compilata, ma dove sono i sorgenti?
É facile trovare i sorgenti della policy con una rapida ricerca su Google[12] o nel repository[13].

Sempre in [12] è indicata la divisione dei sorgenti di policy, per quanto riguarda la policy platform:

**public** I sorgenti della policy che sono pubblici e possono essere presi a riferimento. Solo i tipi, attributi e macro definiti qui possono essere utilizzati dai produttori.

**private** I sorgenti della policy che contengono le definizioni interne e soggette a cambiamento senza nota.

Il numero di regole definite da Android è decisamente elevato, per questo motivo non sarà fatta una mappatura dei vari tipi e delle reciproche interazioni. Ci limitiamo a determinare alcune informazioni di

[9] Nella versione più semplice si trova in `/etc/selinux/vendor_sepolicy.cil`.
[10] Tutto quel bloatware che aziende di bassa lega usano per infiocchettare l'UI.

carattere generale, lasciando l'analisi della policy alle sezioni successive e solo riguardo quando necessario per comprendere l'isolamento dei componenti.

### 4.1 Utenti SELinux di Android

Iniziamo con il determinare gli utenti SELinux definiti in Android. Lo statement `user` ha la forma seguente.

```
user <nome> roles ...
```

Possiamo quindi usare un'apposita regex per cercare questo tipo di stringa nei sorgenti pubblici e privati della policy. Notare che la ricerca della sola stringa "user" comporta vari falsi positivi.

```
$ grep -Pr "\\buser\\s+.*?\\s+roles" sepolicy/public sepolicy/private/
sepolicy/private/users:user u roles { r } level s0 range s0 - mls_systemhigh;
$
```

Come mostrato dal risultato sopra, è presente un unico utente SELinux, il quale ha accesso a tutti i livelli MLS (vedremo che questo è importante). L'utilizzo di un unico utente trova giustificazione nel fatto, come vedremo, che in Android SELinux è stato inserito in un sistema che già faceva ampio uso del meccanismo DAC legacy di Linux.

### 4.2 Ruoli SELinux di Android

In modo analogo possiamo recuperare i ruoli definiti dalla policy platform.

```
$ grep -Pr "^[^\\s]*role[^s].*?;" sepolicy/public sepolicy/private
sepolicy/public/roles:role r types domain;
sepolicy/private/roles_decl:role r;
$
```

Notiamo che è presente un unico ruolo `r`, al quale è permesso accedere all'unico tipo (a.k.a. dominio, in questo caso) `domain`. `domain` è in realtà un attributo, la sua definizione si trova in `./sepolicy/public` (dove il percorso si intende relativo al repository delle policy).

```
# All types used for processes.
attribute domain;
```

Cercando nei sorgenti della policy la stringa "`, domain;`" è possibile vedere come tutti i tipi assegnati ai soggetti (a.k.a. i processi) finiscono sotto questo attributo[11].

Quindi neanche l'unico ruolo presente in Android è in qualche modo limitato.
Possiamo dire che Android non una un modello RBAC, almeno **non direttamente**. Vedremo infatti che SELinux è solo uno dei meccanismi di isolamento tra le app, in particolare il meccanismo legacy DAC è ancora attivo e, a meno che non richiesto (e permesso), ogni app ha un proprio utente Linux generato automaticamente da Android[12].

## 5 Isolamento delle app

Le app Android hanno come riferimento la VM Dalvik[11], questa è una JVM per dispositivi con poca memoria. Le app Android, indipendentemente dal linguaggio in cui sono scritte[13] si portano con sé le caratteristiche delle applicazioni Java, in particolare la necessità di far partire la VM per essere interpretate e tutto il meccanismo di class loading[14].

Quando Android si è presentato per la prima volta nel mercato, queste operazioni erano onerose per uno smartphone e quindi Google ha cercato di sfruttare il più possibile il meccanismo di CoW[15] di Linux. All'avvio un processo chiamato *Zygote* è lanciato da `init`, questo processo carica l'interprete Dalvik e le librerie più comuni del framework di Android. Terminati questi compiti crea un socket e vi si mette in ascolto.

[11] In SELinux, un attributo è un alias per un insieme di tipi
[12] Da bionic per la precisione, come vedremo.
[13] Ci si riferisce a Java e Kotlin principalmente, è possibile fare app che fanno ampio uso dell'NDK e che quindi escono dal controllo della VM.
[14] Android utilizza il suo formato, dex, per il packaging delle classi compilate
[15] Copy-on-write, si veda un qualsiasi manuale base su Linux.

Uno dei comandi inviabili a Zygote è quello dell'avvio di una nuova applicazione (inviato dall'*ActivityManager*), quando questo succede, Zygote fa un **fork** di sé stesso e avvia la nuova applicazione nel processo figlio. Questo ha già l'interprete e le librerie caricate, velocizzando l'avvio dell'app[16].

## 5.1 Contesto di Zygote

Come ci si può aspettare, e come è facile confermare con un semplice comando (vedi figura 4), Zygote è lanciato da `init`. Risulta naturale chiederci in quale contesto SELinux (leggi: dominio) sia eseguito Zygote.

Figura 4: Il padre del processo `zygote` ha PID 1 (i.e. è `init`).

```
generic_x86_arm:/data/busybox # ps -e | grep zygote
root          288     1 1838208 123860 do_sys_poll      0 S zygote
webview_zygote 834   288 1773816 60436 do_sys_poll      0 S webview_zygote
generic_x86_arm:/data/busybox #
```

Zygote è lanciato con un init-script (vi sono tre varianti dello script, a seconda del bitness dell'architettura) che si trova in `/platform/system/core/rootdir/init.zygote*.rc` e nel quale è possibile vedere (figura 5) che Zygote è lanciato con `/system/bin/app_process64`.

Figura 5: L'avvio di Zygote.

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server --socket-name
    =zygote
   class main
   priority -20
   user root
   group root readproc reserved_disk
   socket zygote stream 660 root system
   socket usap_pool_primary stream 660 root system
   onrestart exec_background - system system -- /system/bin/vdc volume abort_fuse
   onrestart write /sys/power/state on
   onrestart restart audioserver
   onrestart restart cameraserver
   onrestart restart media
   onrestart restart netd
   onrestart restart wificond
   task_profiles ProcessCapacityHigh MaxPerformance
   critical window=${zygote.critical_window.minute:-off} target=zygote-fatal
```

Il programma `app_process` si trova in `/platform/frameworks/base/cmds/app_process/app_main.cpp`, dove una rapida occhiata al codice (figura 6) ci permette di capire che è responsabile per l'avvio non solo di Zygote ma anche del *System server*.

[16] Ironico pensare che tutto questo non serve a niente a fronte di un mancato controllo sui produttori hardware. Android è, e rimane, famoso per essere notevolmente lento.

Figura 6: Il codice di app_process mostra come sia responsabile per l'avvio di Zygote e del System server.

```
// Everything up to '--' or first non '-' arg goes to the vm.
//
// The first argument after the VM args is the "parent dir", which
// is currently unused.
//
// After the parent dir, we expect one or more the following internal
// arguments :
//
// --zygote : Start in zygote mode
// --start-system-server : Start the system server.
// --application : Start in application (stand alone, non zygote) mode.
// --nice-name : The nice name for this process.
//
// For non zygote starts, these arguments will be followed by
// the main class name. All remaining arguments are passed to
// the main method of this class.
//
// For zygote starts, all remaining arguments are passed to the zygote.
// main function.
//
[...]
++i; // Skip unused "parent dir" argument.
while (i < argc) {
    const char* arg = argv[i++];
    if (strcmp(arg, "--zygote") == 0) {
        zygote = true;
        niceName = ZYGOTE_NICE_NAME;
    } else if (strcmp(arg, "--start-system-server") == 0) {
        startSystemServer = true;
    } else if (strcmp(arg, "--application") == 0) {
        application = true;
    } else if (strncmp(arg, "--nice-name=", 12) == 0) {
        niceName.setTo(arg + 12);
    } else if (strncmp(arg, "--", 2) != 0) {
        className.setTo(arg);
        break;
    } else {
        --i;
        break;
    }
}
```

Ma ancora non abbiamo risposta alla domanda originale: in quale contesto è eseguito Zygote? Osservando il file /platform/system/sepolicy/private/zygote.te è possibile vedere la regola init_daemon_domain(zygote), questa risulta essere una macro (definita in /platform/system/sepolicy/public/te_macros) così composta:

```
####################################
# init_daemon_domain(domain)
# Set up a transition from init to the daemon domain
# upon executing its binary.
define(`init_daemon_domain', `
domain_auto_trans(init, $1_exec, $1)
')
```

Si tratta quindi di una macro per la semplificazione della definizione delle regole di transizione di contesto. In questo caso, ma è un modello generale, quando viene eseguito un file etichettato come zygote_exec viene fatta una transizione al dominio zygote (se provenienti dal dominio init).

Possiamo confermare che questo è il meccanismo di transizione usato ispezionando l'etichetta di app_process:

```
generic_x86_arm:/data/busybox # ls -lZ /system/bin/app_process*
lrwxr-xr-x 1 root shell u:object_r:system_file:s0 13 2020-10-14 00:35 /system/bin/app_process ->
     app_process32
-rwxr-xr-x 1 root shell u:object_r:zygote_exec:s0 26132 2020-10-14 00:24 /system/bin/app_process32
```

Vedremo più avanti come avviene l'etichettatura dei file in Android, queste etichette (come abbiamo appena visto) risultano fondamentali per la corretta sicurezza del sistema.

Zygote è quindi eseguito nel contesto omonimo `zygote`.
Rimangono adesso quattro domande (per il momento escludiamo i servizi):

- Come Zygote imposta il contesto delle app avviate?
- Come sono etichettati i file delle app?
- Come varia il contesto delle app a seconda dei permessi Android concessi?
- Cosa impedisce ad un'app di inviare comandi a Zygote?

## 5.2 Il contesto del socket di Zygote

Per capire come viene impostato il contesto delle App da Zygote, è necessario cercare nel codice di Zygote la parte relativa all'avvio di una nuova app. Abbiamo visto che Zygote è lanciato dal programma `app_process`, quest'ultimo è un programma nativo, lo è anche Zygote? Nel codice di `app_process` vediamo che Zygote è avviato così:

```
if (zygote) {
    runtime.start("com.android.internal.os.ZygoteInit", args, zygote);
```

La stringa `com.android.internal.os.ZygoteInit` è palesemente un FQN di una classe Java, quindi Zygote è scritto in Java. Possiamo finalmente vedere come la variabile `runtime` sia di tipo `AppRuntime`, una sottoclasse di `AndroidRuntime`. Quest'ultima è definita in `/platform/frameworks/base/core/jni/AndroidRuntime.cpp` e contiene il metodo `startVm` che è dedicato proprio all'avvio della VM Dalvik.

É errore comune dire che sia Zygote ad avviare la VM Dalvik, ma come abbiamo visto, questo non sarebbe possibile in quanto Zygote `è eseguito in una VM Dalvik`. `app_process` avvia la VM, `Zygote` è solo il primo (e normalmente anche unico) processo ad avere una VM appositamente avviata per lui. Tuttavia `app_process` permette di avviare app standalone, ovvero che non riusano la VM di Zygote.

La classe di avvio di Zygote è `/platform/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`, la quale contiene un metodo `main` che, tra le altre cose, si occupa di far partire il server di Zygote, quello che accetta i comandi per l'avvio delle app.

```
zygoteServer = new ZygoteServer(isPrimaryZygote);

if (startSystemServer) {
    Runnable r = forkSystemServer(abiList, zygoteSocketName, zygoteServer);

    // {@code r == null} in the parent (zygote) process, and {@code r != null} in the
    // child (system_server) process.
    if (r != null) {
        r.run();
        return;
    }
}
```

Dal codice sopra si nota come Zygote sia anche responsabile del System Server, come vedremo poi.

La classe `ZygoteServer` è definita in un omonimo file che si trova nella stessa cartella di `ZygoteInit.java`, in generale, quando l'identificazione del file sorgente risulta scontata (come in questo caso), ometteremo di indicarlo.

Nel costruttore di `ZygoteServer` viene chiamato `createManagedSocketFromInitSocket` il quale crea il socket del server di Zygote a partire da un file descriptor creato da `init`.

Nell'init script che avvia Zygote è infatti presente il seguente parametro:

```
socket zygote stream 660 root system
```

Nel file `/platform/system/core/init/service_parser.cpp` (`ParseSocket`) è presente il codice di parsing di questo parametro, dove è indicato che il formato è `name type perm [ uid gid context ]`.
Non viene quindi specificato un contesto esplicito per il socket, in questo caso (si veda `/platform/system/core/init/service.cpp`, `ComputeContextFromExecutable`) viene usata la funzione SELinux standard `security_compute_create`[14] con ingresso il contesto di `init`, ovvero il dominio `init_t`, il contesto del file eseguibile, ovvero l'etichetta `zygote_exec_t` e la classe `process`. Questo genera, secondo la policy di transizione di dominio per Zygote vista prima, il tipo `zygote_t`.

Questo meccanismo è quindi pensato per permettere l'accesso al socket `solo` al servizio per cui è destinato, ovviamente eventuali regole possono rilassare queste restrizioni.

Non vale la pena guardare chi ha accesso a socket di tipo `zygote_t` perché in realtà il file è re-etichettato tramite il meccanismo di etichettatura dei file, il tipo file del socket è `zygote_socket`[17].

Il tipo del socket del server di Zygote può essere facilmente confermato con il comando seguente:

```
generic_x86_arm:/ # ls -Z /dev/socket/zygote
u:object_r:zygote_socket:s0 /dev/socket/zygote
```

Osservando il file di policy `/platform/system/sepolicy/private/app.te` (vedremo in seguito gli altri contesti in cui sono eseguite le app) è possibile vedere che le app **non possono** inviare comandi a Zygote.

```
neverallow appdomain zygote_socket:sock_file write;
```

Questo compito è infatti riservato al System Server (che contiene l'ActivityManager), questo può essere confermato leggendo il file `/platform/system/sepolicy/public/domain.te` che contiene:

```
# Only system_server should be able to send commands via the zygote socket
neverallow { domain -zygote -system_server } zygote:unix_stream_socket connectto;
neverallow { domain -system_server } zygote_socket:sock_file write;
```

## 5.3 Il contesto delle app e altre restrizioni

Fino ad ora abbiamo visto come sia etichettato il socket di Zygote, assumendo che un processo possa scriversi, rimane ora da determinare in che modo è impostato il contesto delle app.

Nel file `Zygote.java` è possibile vedere il metodo `acceptCommandPeer`, il quale è responsabile di creare una classe `ZygoteConnection` per la gestione di una nuova connessione[18] al socket di Zygote.

Dentro il file `ZygoteConnection.java` troviamo il metodo `processOneCommand` che ha il compito di eseguire un comando inviato tramite il socket. Questo metodo ritorna un `Runnable` nel processo figlio (ottenuto dal fork) di Zygote, si tratta quindi del metodo che avvia un'app. Nello specifico è la seguente chiamata che avvia una nuova applicazione:

```
pid = Zygote.forkAndSpecialize(parsedArgs.mUid, parsedArgs.mGid, parsedArgs.mGids,
parsedArgs.mRuntimeFlags, rlimits, parsedArgs.mMountExternal, parsedArgs.mSeInfo,
parsedArgs.mNiceName, fdsToClose, fdsToIgnore, parsedArgs.mStartChildZygote,
parsedArgs.mInstructionSet, parsedArgs.mAppDataDir, parsedArgs.mIsTopApp,
parsedArgs.mPkgDataInfoList, parsedArgs.mWhitelistedDataInfoList,
parsedArgs.mBindMountAppDataDirs, parsedArgs.mBindMountAppStorageDirs);
```

Tutti i parametri passati a `forkAndSpecialize`, eccetto tre, sono presi direttamente dal comando inviato sul socket. I tre rimanenti sono:

- `rlimits` Quote di risorse, hard e soft, da applicare al processo figlio. Queste quote sono applicate con `setrlimit`[15], si tratta delle usuali quote Linux.
- `fdsToClose`. Sono i file descriptor da chiudere nel processo figlio. Questo parametro contiene il socket ed il file descriptor del server di Zygote.
- `fdsToIgnore`. Sono i file descriptor da non chiudere (ignorare). Questi descriptor sono usati solo quando viene usato il parametro `invoke-with`, il quale è usato per avviare un'app con un comando wrapper (che fa redirect di stdin, stdout e stderr). L'avvio di un'app tramite un wrapper è riservato per il solo utente root e solo per il debug delle app (si veda `applyInvokeWithSecurityPolicy` nella classe `Zygote`).

Il metodo `forkAndSpecialize` chiama `nativeForkAndSpecialize`, la cui implementazione si trova in `/platform/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`. Questa implementazione ci aiuta a capire il significato dei vari parametri. Questi sono passati a `SpecializeCommon`[19] e possono essere riassunti nella tabella sottostante.

[17] Anticipando, è possibile trovare la relativa regola di etichettatura del file in `/platform/system/sepolicy/private/file_contexts`

[18] Vedere chi e quando chiama `acceptCommandPeer` è piuttosto semplice ed è lasciato come esercizio al lettore.

[19] Il termine specialize indica la configurazione di un fork di Zygote per l'avvio di un'app. Da Android 10, Zygote può mantenere un pool di processi pre-forkati ma non ancora specializzati (vedi USAP).

| | |
|---|---|
| uid | User id da applicare al nuovo processo |
| gid | Group id da applicare al nuovo processo |
| gids | Gruppi aggiuntivi da applicare al nuovo processo |
| runtime_flags | Flag per indicare se il processo è debuggabile, se usa ART, se è profilabile e se usare altre feature non rilevanti per questo documento. |
| rlimits | Quote delle risorse Linux da applicare al nuovo processo (tipo massimo numero di file aperti). |
| permitted_capabilities | Le capability Linux usabili dal, ma non ancora concesse al, nuovo processo (vedi sotto). |
| effective_capabilities | Le capability Linux concesse al nuovo processo. |
| mount_external | Modalità in cui lo storage esterno è montato nel nuovo processo (vedi sotto). |
| managed_se_info | Questo è un tag utilizzato per l'assegnazione del contesto del nuovo processo (vedi sotto) |
| managed_app_data_dir | Percorso della directory dell'app. |
| pkg_data_info_list | Informazioni per montare la directory dell'app (vedi sotto). |
| whitelisted_data_info_list | Lista di applicazioni che possono essere "viste" da quest'app (vedi sotto) |
| mount_data_dirs | Se isolare il nuovo processo dall'accesso alle cartelle di altre app tramite il namespacing di Linux (vedi sotto). |
| mount_storage_dirs | Come sopra ma per le cartelle nello storage esterno. |

In base all'user id, al group id e all'avvio o meno di uno Zygote figlio, sono concesse varie capability[20]. Ad esempio, se il processo avviato ha uid `AID_BLUETOOTH` (vedremo più avanti come sono generati gli utenti Linux in Android), sono concesse le capability per l'accesso alle interfacce di rete e per il risveglio del dispositivo.
Il codice si trova nel metodo `CalculateCapabilities` e può essere riassunto nella seguente tabella.

| | |
|---|---|
| uid == AID_BLUETOOTH | CAP_WAKE_ALARM, CAP_NET_ADMIN, CAP_NET_RAW, CAP_NET_BIND_SERVICE, CAP_SYS_NICE |
| uid == AID_NETWORK_STACK | CAP_NET_ADMIN, CAP_NET_BROADCAST, CAP_NET_BIND_SERVICE, CAP_NET_RAW |
| AID_WAKELOCK in {gid, gids} | CAP_BLOCK_SUSPEND |
| is_child_zygote | CAP_SETUID, CAP_SETGID, CAP_SETPCAP |

Zygote fa l'unmount dello storage esterno prima di avviare un processo figlio, in questo modo le app non autorizzate non possono accedervi. Il mount dello storage esterno è fatto secondo il parametro `mount_external`. La gestione dello spazio esterno, considerato pubblico, non è oggetto di questo documento, si faccia riferimento a [16].

Basti sapere che `/storage` è bindata su un file in `/mnt/runtime/` (si veda il vettore `ExternalStorageViews`).

Il parametro `managed_se_info` è passato alla funzione `selinux_android_setcontext`, definita in `/platform/external/selinux/libselinux/src/android/android_platform.c`. Questa funzione necessita di una spiegazione più approfondita e sarà affrontata tra poco (sarà questa funzione a determinare il contesto dell'applicazione).

Se `mount_data_dirs` è vero, l'app è isolata anche tramite il namespacing di Linux[21] La funzione `ensureInAppMountNamespace` si occupa di creare un nuovo namespace FS tramite `unshare(CLONE_NEWNS)`[17]. La funzione `isolateAppData` spiega come `/data/data` sia montata in un `tmpfs` e in questo siano montate varie directory, tra cui la cartella dell'app (passata in `pkg_data_info_list`), le directory delle app in whitelist (`whitelisted_data_info_list`) e vari file per la gestione degli storage CE e DE[22].

Una simile operazione è effettuata per le directory delle app nello storage esterno se `mount_storage_dirs` è vero.

[20] Questo sono capabilities del kernel Linux. Si assume che il lettore abbia familiarità con queste.
[21] Feature sulla quale si basano i software di containerizzazione.
[22] Che non sono oggetto di questo documento, si veda [18].

### 5.3.1 User, group e app id in Android

Per capire come viene impostato il contesto delle app è prima necessario capire come sono gestiti gli utenti Linux in Android, e come questi siano correlati all'id dell'applicazione.

Linux, il kernel, non è interessato a come siano generati gli user ed i group id. Questi sono trattati come interi opachi, con l'unica eccezione che l'uid e il gid di valore 0 sono associati all'utente e gruppo `root`.

Come è ben risaputo, il database degli utenti è contenuto in `/etc/passwd`[23] ma questo file è vuoto in Android:

```
generic\_x86\_arm:/ # cat /etc/passwd
generic\_x86\_arm:/ #
```

Android associa ad ogni app un unico utente e gruppo, per cui la gestione degli utenti tramite l'usuale file `passwd` non è possibile (o quantomeno non sarebbe conveniente). La gestione degli utenti è invece affidata a Bionic, l'implementazione di Android del runtime C, che implementa le funzioni POSIX di gestione degli utenti (tipo `getpwnam`) in `/platform_bionic/libc/bionic/grp_pwd.cpp`.

In questo file è possibile vedere come gli utenti siano creati dinamicamente, l'uid di un utente Android può essere convertito in una stringa che ha il formato `uN_X` dove $N$ è un numero (tipicamente 0 o 1) e $X$ è l'id di un'app. $N$ è un numero progressivo assegnato ad ogni utilizzatore finale del dispositivo (ovvero una persona fisica, si veda [19]). $X$ descrive l'app, le app installate dall'utente hanno un $X$ della forma `aK` o `iK`, dove $K$ è un numero (ad esempio, `u0_a1234`) mentre le app di sistema hanno un $X$ fisso e specifico (es. `u1_system` per il System Server).

Lo spazio degli user id di Android è diviso in segmenti di grandezza `AID_USER_OFFSET`, ogni utilizzatore ha un segmento a sé dedicato. Entro questo segmento, che è quindi un range numerico, Android alloca gli uid per i processi delle applicazioni.
La lista completa può essere trovata in `/platform/system/core/libcutils/include/private/android_filesystem_config.h`, per quel che ci interessa ogni segmento è ulteriormente diviso come mostra la tabella che segue (i valori sono relativi all'inizio del segmento).

| | |
|---|---|
| [0, AID_APP_START) | Applicazioni di sistema e/o specifiche (adb, shell, ...) |
| [AID_APP_START, AID_ISOLATED_START) | App dell'utente (che non hanno richiesto di essere isolate, vedi dopo) |
| [AID_ISOLATED_START, AID_USER_OFFSET) | App dell'utente (isolate) |

É importante aggiungere che Android supporta un insieme di password file, come mostra la funzione `getpwnam_internal`, ma questi sono generalmente vuoti.

### 5.3.2 Lookup del contesto

Possiamo ora tornare allo studio della funzione `selinux_android_setcontext` dentro la quale troviamo la seguente chiamata:

```
seapp_context_lookup(SEAPP_DOMAIN, uid, isSystemServer, seinfo, pkgname, NULL, ctx)
```

Osservando il codice di questa funzione si nota come dall'uid sia ricavata una stringa che corrisponde a `_app` se l'uid appartiene ad un'app dell'utente (non isolata), `_isolated` se l'app è dell'utente e isolata, oppure al nome del servizio se l'app è un'app di sistema predefinita di Android (es: `radio`, `system`, ...). Questa stringa è usata, insieme al nome del package dell'app, all'etichetta `seinfo` e ad un booleano indicante se l'uid appartiene al system server, per trovare il contesto dell'applicazione appena lanciata nella tabella `seapp_contexts`.

Dal codice è possibile dedurre che questa tabella è caricata dal file `/system/etc/selinux/plat_seapp_contexts`, il cui formato è documentato in `/platform/system/sepolicy/private/seapp_contexts` (oltre che essere intuibile dal codice).

Ogni riga di questo file può servire per definire il contesto (leggi: dominio) di un'applicazione o per stabilire delle proibizioni.
Le righe che iniziano con `neverallow` sono usate per assicurarsi che in nessun caso alcune condizioni siano violate. Queste condizioni fanno sì che, ad esempio, solo il System Server possa avere il dominio

[23] In realtà non esiste un unico database utenti, considerando la flessibilità di PAM.

`system_server`. Altri esempi sono: la separazione forzata tra app isolate e non, tra app di sistema e non, l'assegnazione del dominio `shell` solo alla shell ADB, e così via. Un breve estratto segue.

```
# only the system server can be in system_server domain
neverallow isSystemServer=false domain=system_server
neverallow isSystemServer="" domain=system_server
# system domains should never be assigned outside of system uid
neverallow user=((?!system).)* domain=system_app
neverallow user=((?!system).)* type=system_app_data_file
# any non priv-app with a non-known uid with a specified name should have a specified
# seinfo
neverallow user=_app isPrivApp=false name=.* seinfo=""
neverallow user=_app isPrivApp=false name=.* seinfo=default
# neverallow shared relro to any other domain
# and neverallow any other uid into shared_relro
neverallow user=shared_relro domain=((?!shared_relro).)*
neverallow user=((?!shared_relro).)* domain=shared_relro
# neverallow non-isolated uids into isolated_app domain
# and vice versa
neverallow user=_isolated domain=((?!isolated_app).)*
neverallow user=((?!_isolated).)* domain=isolated_app
# uid shell should always be in shell domain, however non-shell
# uid's can be in shell domain
neverallow user=shell domain=((?!shell).)*
# only the package named com.android.shell can run in the shell domain
neverallow domain=shell name=((?!com\.android\.shell).)*
neverallow user=shell name=((?!com\.android\.shell).)*
# Ephemeral Apps must run in the ephemeral_app domain
neverallow isEphemeralApp=true domain=((?!ephemeral_app).)*
```

Oltre alle regole `neverallow`, vi sono le regole per l'assegnazione del contesto. Queste sono righe composte da una serie di attributi e valori separati da spazi, del tipo:

```
key1=val1 key2=val2 ... keyN=valN
```

Le chiavi `domain` e `type` indicano l'output della riga, rispettivamente il dominio da assegnare al processo dell'app e ai file nelle cartelle private dell'app. Entrambe sono opzionali (ma non contemporaneamente). Oltre a queste due le chiavi `levelFrom` e `level` sono usate per il calcolo del livello MLS/MCS, vedremo tra poco come questo calcolo sia fatto (notare che qui è evidente l'errore tipicamente fatto nell'identificare il dominio/tipo di un oggetto con il suo contesto, infatti il contesto è un dominio/tipo più altre informazioni, tra cui il livello MLS/MCS).
Le altre chiavi sono selettori, ovvero sono usati per il match delle proprietà dell'app. Notare che queste proprietà sono tutte calcolate a partire dai parametri in input a `seapp_context_lookup`, che ricordiamo essere l'UID del processo, il package dell'app, se l'app è il System Server e l'etichetta *seinfo*.

| | |
|---|---|
| isSystemServer | Se l'app è il System Server. Questo parametro è impostato da Zygote ed è vero solo quando il contesto è calcolato per il processo che avvia il System Server. |
| isEphemeralApp | Se l'app è una Instant App[20]. |
| isOwner | Se l'app è dell'utente (fisico) primario. |
| user | Nome utente corrispondente all'UID. Per le app utente è `_app` o `_isolated`, per quelle di sistema è il nome utente come definito nell'apposito file visto precedentemente (es: `system`, `radio`, ...). |
| seinfo | Etichetta arbitraria. |
| name | Nome del package dell'app. |
| path | Percorso della directory privata dell'app. |
| isPrivApp | Se l'applicazione è una di quelle pre-installate in `/system/priv-app`. |
| minTargetSdkVersion | SDK minimo con cui l'app è compatibile. |
| fromRunAs | Se l'app è stata lanciata con il comando `run-as`[24][21] |

Notare che `isPrivApp`, che indica se l'app è privilegiata, `minTargetSdkVersion`, `fromRunAs` e `isEphemeralApp` non si trovano tra i parametri di input di `seapp_context_lookup`. Questi due valori sono

[24] `run-as` è una specie di `sudo` per Android, ma i soli utenti root e shell possono usarlo.

calcolati da `manged_seinfo`, il quale ha la forma
`seinfo:tipo_app:targetSdkVersion=XX:fromRunAs:ephemeralapp`, dove `tipo_app` può essere `privapp` o assente (per le app non particolari). Sia `fromRunAs` che `ephemeralapp` possono essere assenti.
Nel seguito indicheremo indistintamente con `seinfo` sia il valore esteso `managed_seinfo`, sia il valore `seinfo` stesso (non vi è rischio di confusione, dato che `seinfo` in sè è usato solo all'interno di `seapp_context_lookup`).

La chiave di output `level` è usata per impostare il campo `livello` del contesto dell'app (o dei suoi file), deve quindi essere una stringa che contiene un livello SELinux valido (tipo `s0-s0:c0,c1`).
La chiave `levelFrom` è usata per impostare il livello del contesto dell'app o dei suoi file a partire dall'uid del processo. Nel file `/platform/external/selinux/libselinux/src/android/android_platform.c`, sempre dentro `seapp_context_lookup`, è possibile trovare l'algoritmo che fa il calcolo.
La tabella sotto mostra il livello calcolato in base al valore di `levelFrom`, si ricorda che l'app id è l'offset dell'uid del processo dell'app all'interno del segmento di uid riservati per l'utente fisico (o, in parole semplici, $app\_id = uid \ \% \ AID\_USER\_OFFSET$). Per l'utente primario, l'app id corrisponde all'uid del processo.
Inoltre il numero dell'utente fisico è l'id progressivo dato ad ogni utente fisico del dispositivo, è calcolato $userid = uid \ / \ AID\_USER\_OFFSET$.

| | |
|---|---|
| app | `s0:cL,cH` dove `cL` è la rappresentazione decimale del byte basso dell'app id e `cH` è la rappresentazione decimale del byte alto dell'app id sommato a 256. |
| user | `s0:cL,cH` dove `cL` è la rappresentazione decimale del numero dell'utente fisico sommato a 512 e `cH` è la rappresentazione decimale del byte alto del numero dell'utente fisico sommato a 768. |
| all | `s0:cAL,cAH,cUL,cUH` dove `cAL` e `cAH` sono come per il tipo `app` mentre `cUL` e `cUH` sono come per il tipo `user`. |

Ad esempio se il processo di un app ha uid 0x1234, il suo numero dell'utente fisico è 0 e il suo app-id è 0x1234. Nel caso sia usato `levelFrom=all`, le categorie assegnate al contesto sono `s0:c52,c274,c512,c768`, dove 52 è il byte basso di 0x1234 (0x34) e 274 è 256 + 0x12, 512 è 512 + 0 e uguale per 768.

Leggendo il file `/system/etc/selinux/plat_seapp_contexts` di un dispositivo Android 10, si trova la riga

```
user=_app minTargetSdkVersion=30 domain=untrusted_app type=app_data_file levelFrom=all
```

la quale marca le app dell'utente (si faccia riferimento al file del proprio dispositivo per convincersi che questo è effettivamente il caso). É interessante notare come tutte le app finiscano nel contesto `untrusted_app` ma con **categorie diverse** (poichè ogni app ha il suo app id).
Android usa quindi SELinux in modalità MLS/MCS. Questo si può facilmente confermare creando due App minimali ed elencando i contesti delle loro directory (che corrispondono al contesto dei rispettivi processi, tranne che per il tipo).

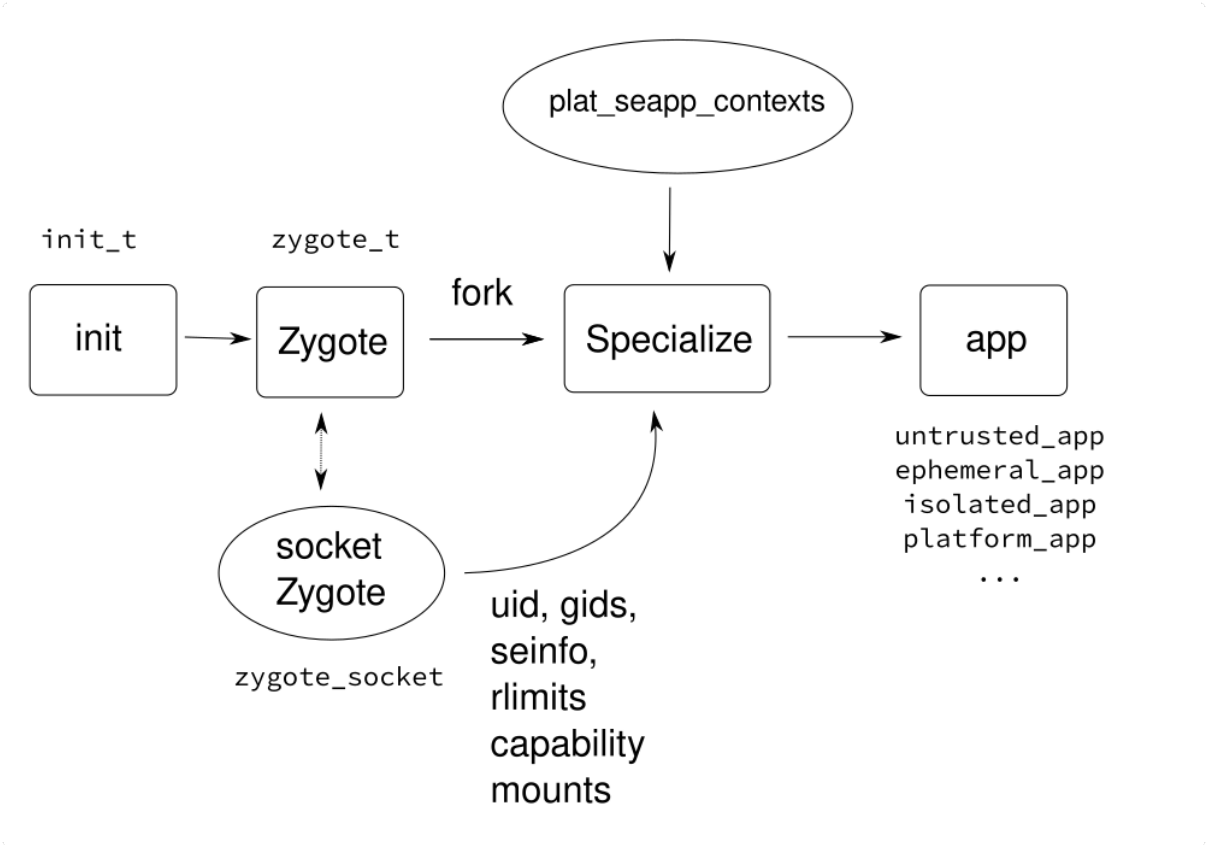
```
generic_x86_arm:/data/data # ls -Z | grep "com.example.myapp"
u:object_r:app_data_file:s0:c158,c256,c512,c768 com.example.myapplication3
u:object_r:app_data_file:s0:c157,c256,c512,c768 com.example.myapplication4
generic_x86_arm:/data/data # ps -eZ | grep "com.example.myapp"
u:r:untrusted_app:s0:c157,c256,c512,c768 u0_a157 ...
u:r:untrusted_app:s0:c158,c256,c512,c768 u0_a158 ...
```

Il contesto delle app è quindi calcolato tramite il file `seapp_contexts`, che nell'immagine finale di Android è salvato, post processato, in `/system/etc/selinux/plat_seapp_contexts`.

L'etichetta `seinfo` è una stringa arbitraria ma controllata dalla configurazione del produttore, vedremo più avanti come è ricavata.

Per il momento siamo arrivati a definire come è calcolato il contesto di un'app a partire dai parametri che Zygote riceve sul suo socket, abbiamo anche visto che le app non possono accedere a questo socket e che sono isolate mediante una serie di meccanismi aggiuntivi (oltre che da SELinux).

Lo schema sotto riassume quanto noto fin ora.




### 5.4 Il contesto dei file delle applicazioni (e non solo)

Abbiamo già visto che il file `plat_seapp_contexts` permette anche di specificare le label da assegnare ai file delle varie app. In generale l'utilizzo di SELinux richiede un meccanismo di relabelling dei file a seguito di update delle policy.

La cartella privata dell'app, sia nella memoria interna che in quella esterna, è etichettata usando il solito file usato per il calcolo del contesto dell'app ma con lo scopo di ottenere la chiave di output `type`. Di questo si occupano le funzioni `selinux_android_setfilecon` e `pkgdir_selabel_lookup`, le quali sono in ultima analisi usate dalle funzioni `selinux_android_restorecon` e `selinux_android_restorecon_pkg`.
Queste ultime due funzioni sono più generiche delle prime menzionate, infatti esse sono usate per calcolare il contesto di un **qualsiasi** file all'interno del file system di Android. Sono ad esempio usate da `init` varie volte per impostare il contesto di vari device, sono usate in vari componenti di Android che creano nuovi file o directory e un wrapper JNI di nome `native_restorecon` esiste per permettere al framework Android di usarle.

Le funzioni `selinux_android_restorecon` e `selinux_android_restorecon_pkg` utilizzano il file `plat_file_contexts` per il labelling dei file. Il file è strutturato riga per riga, ogni riga ha un formato molto semplice: `percorso contesto`, dove `percorso` può contenere GLOB o regex.

Un estratto del file, a titolo informativo, segue.

```
/data/system/ndebugsocket    u:object_r:system_ndebug_socket:s0
/product_property_contexts u:object_r:property_contexts_file:s0
/nonplat_property_contexts u:object_r:property_contexts_file:s0
/vendor_hwservice_contexts u:object_r:hwservice_contexts_file:s0
/system/bin/servicemanager   u:object_r:servicemanager_exec:s0
/system/bin/surfaceflinger   u:object_r:surfaceflinger_exec:s0
/system/bin/mediadrmserver   u:object_r:mediadrmserver_exec:s0
/system/bin/mediaextractor   u:object_r:mediaextractor_exec:s0
/system/bin/otapreopt_slot u:object_r:otapreopt_slot_exec:s0
/system/etc/event-log-tags u:object_r:system_event_log_tags_file:s0
```

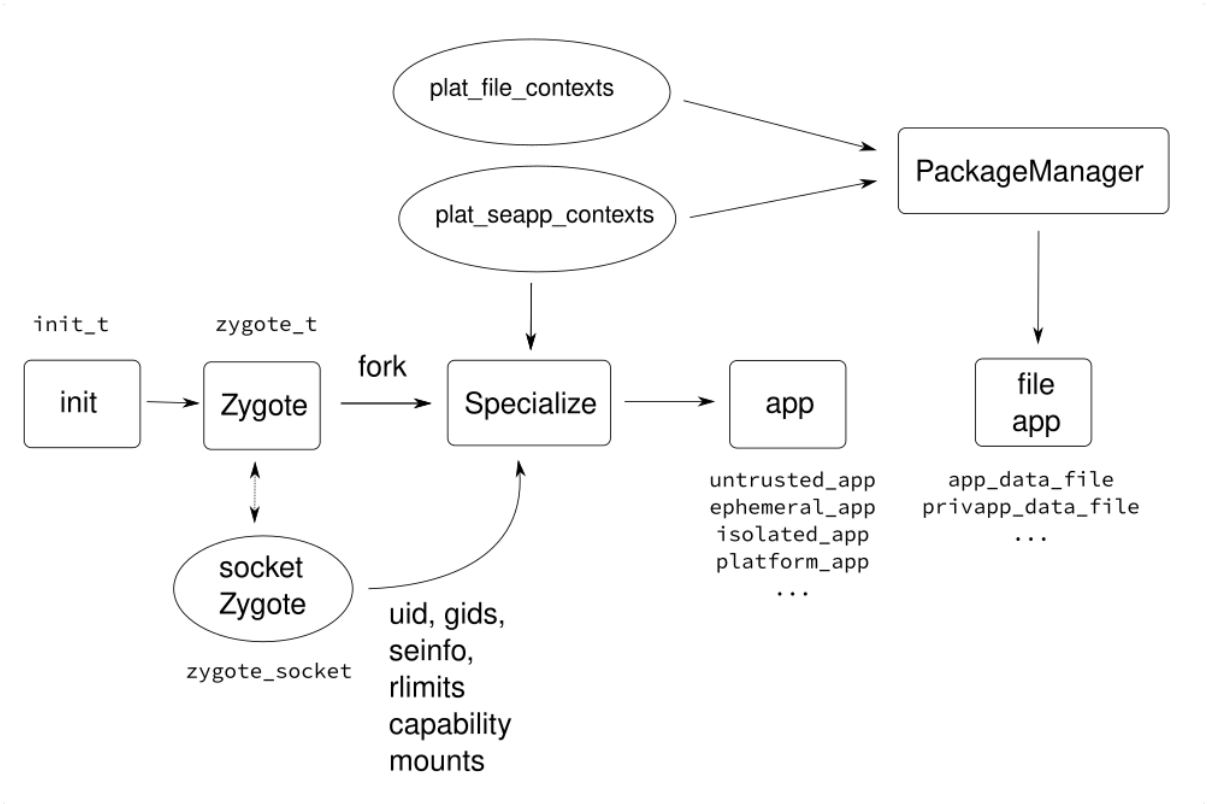
Il codice che definisce queste funzioni e che è stato analizzato per ricavare il comportamento appena descritto si trova in `/platform/external/selinux/libselinux/src/android/android_platform.c`, nello stesso file è presente un commento ed una condizione che indicano come il file `plat_file_contexts`,

usato per il labelling generale dei file, ed il file `plat_seapp_contexts`, usato per il labelling delle app e dei loro file, possano "convivere" senza un accavallarsi di compiti.

```
/*
     * For subdirectories of /data/data or /data/user, we ignore selabel_lookup()
     * and use pkgdir_selabel_lookup() instead. Files within those directories
     * have different labeling rules, based off of /seapp_contexts, and
     * installd is responsible for managing these labels instead of init.
     */
    if (!strncmp(pathname, DATA_DATA_PREFIX, sizeof(DATA_DATA_PREFIX)-1) ||
        !strncmp(pathname, DATA_USER_PREFIX, sizeof(DATA_USER_PREFIX)-1)) {
        if (pkgdir_selabel_lookup(pathname, seinfo, uid, &secontext) < 0)
            goto err;
    }
```

Quindi il file `plat_seapp_contexts` ha la precedenza per i file in `/data/data` e `/data/user`.

Possiamo quindi ora aggiornare la figura della sezione precedente.




Dove abbiamo *ipotizzato* che sia il package manager ad impostare i contesti dei file delle app, è ragionevole aspettarsi che sia questo a creare la directory dell'app (i file in essa ereditino automaticamente il contesto).

Questa ipotesi può essere confermata guardando il codice del metodo `com.android.server.pm.PackageManagerService.prepareAppDataLeafLIF`, dove viene recuperato il valore `seinfo` da usare per il labelling della directory dell'applicazione.

## 5.5 Riepilogo dei meccanismi di isolamento delle app

Quello che abbiamo visto sono le capacità di isolamento di Android di basso livello, in un certo senso è l'interfaccia di isolamento di Android. Prima di vedere come dal manifest si arrivi ai parametri passati sul socket di Zygote, facciamo riepilogo del meccanismo di isolamento delle app.

Ogni app viene eseguito in un processo associato ad un uid unico, l'uid, come abbiamo visto, non è del tutto arbitrario in quanto specifici intervalli sono riservati per le app. Analogo discorso vale per il gid.

Questo permette un isolamento basato sul tradizionale meccanismo DAC di Linux, e proprio perché è discrezionale, SELinux è stato introdotto per rimediare agli errori grossolani di alcuni programmatori

che lasciavano le proprie cartelle e file leggibili a tutti (figurano tra questi i programmatori di app come Skype [22]).
Ogni app è messa in un dominio SELinux in base al file `plat_seapp_contexts`, tuttavia questo file è pensato per dividere le app in base alla loro tipologia (se di sistema, se isolata, eccetera) e, in generale, due diverse app senza particolari richieste finiscono nel solito dominio SELinux (che in Android 10 è `untrusted_app`). Quello che impedisce ad un'app di accedere ai file di un altra non è il dominio in sé, che è solo una *parte* del contesto del processo dell'app, ma è il *livello MLS/MCS*. Questo è calcolato a partire dall'uid del processo, in un modo che lo scompone nelle componenti *app id* e *numero dell'utente fisico* (sempre chiamato user id nei sorgenti). Da ognuna di queste due componenti vengono ricavate due categorie i cui valori numerici, quando sommati, danno il valore numerico del componente stesso.
Sono queste categorie che impediscono, secondo il concetto di *dominanza* di SELinux, alle app di accedere ai file di altre app.

Questo meccanismo di isolamento non è però perfetto, è infatti possibile determinare se un'app è installata provando ad accedere alla sua cartella in `/data/data`. Se l'app è presente si ottiene un errore di accesso, altrimenti un errore di file non trovato.
Per rimediare a questo "side channel attack", Android ha la possibilità di clonare il namespace file system di un app appena avviata, questo permette di fare modifiche (tipo montare o smontare percorsi) senza intaccare la visione che hanno gli altri processi del file system stesso (questo è il meccanismo alla base dei container). Android monta un file system *tmpfs* in `/data/data`, di fatto impedendo all'app di trovare qualsiasi directory in questo percorso. Tuttavia l'app si aspetta di poter accedere alla propria directory, per cui questa è montata (o meglio, si tratta di una operazione di *bind*) in `/data/data/<nome_package>`. Altre operazioni simili sono fatte sullo spazio privato dell'app nello storage esterno.

Queste sono le capacità di isolamento delle app di Android, rimane da risolvere la questione di come si passi dal `Manifest.xml` delle app ai parametri di avvio passati a Zygote.

## 5.6 Dal manifest ai parametri per Zygote

La chiamata al socket di Zygote è fatta da `ZygoteProcess.start` che è a sua volta richiamata da `Process.start`. Quest'ultimo è chiamato dal metodo `com.android.server.am.ProcessList.startProcess` (tramite la catena `AppZygote.getProcess`, `ChildZygoteProcess`, `ZygoteProcess.start`, `ZygoteProcess.startViaZygote`).

In questo metodo è già possibile vedere come sono impostati alcuni parametri, seguendo il codice (in particolare in `startProcessLocked` nella stessa classe) è possibile determinare come i parametri utili per Zygote sono impostati.

| | |
|---|---|
| uid | Unico per app a meno che non sia stato specificato `manifest:android:sharedUserId` nel manifest. In quest'ultimo caso viene riusato l'uid della prima app creata da un manifest con il solito valore di `sharedUserId`, altrimenti l'uid è creato nel range delle app isolate se `manifest.service:android:isolatedProcess` è `true` nel manifest e lo specifico servizio viene avviato. |
| gid | Uguale a uid. Alcune app di sistema possono avere un gid diverso ma per le app utente Android non usa i gruppi del meccanismo DAC legacy di Linux. |
| mountExternal | `MOUNT_EXTERNAL_NONE` di default o per app isolate o instant-app, altrimenti in base ai permessi di accesso allo storage definiti nel manifest. |
| targetSdkVersion | Da `manifest.uses-sdk:android:minSdkVersion` nel manifest. |
| seinfo | Dal file plat_mac_permissions.xml (vedi dopo) |
| isTopApp | Se l'activity dell'app avviata è quella visibile all'utente (non dipende dal manifest). |
| pkgDataInfoMap | Tutte i package delle app che hanno lo stesso valore di `manifest:android:sharedUserId` nel loro manifest. Se il processo è avviato per eseguire un servizio isolato, questo array viene messo a null. |
| whiteListedAppInfoMap | Da tutti i tag `app-data-isolation-whitelisted-app` nei file XML presenti in `/system/etc/permissions`. |
| bindMountAppsData | Se l'app è un'app utente (e il meccanismo di isolamento tramite namespacing è abilitato). |
| bindMountAppStorageDirs | Se l'app richiede l'accesso allo storage esterno ma non con uno dei seguenti modi: `MOUNT_EXTERNAL_ANDROID_WRITABLE`, `MOUNT_EXTERNAL_PASS_THROUGH`, `Zygote.MOUNT_EXTERNAL_INSTALLER` |

Zygote permette anche di specificare le quote e le capability di un processo. Le prime non sono usate per l'avvio delle app utente (non ci risultano proprio mai usate), le seconde sono usate solo internamente a Zygote per l'avvio di altri processi Zygote o del System server (vedi capitoli precedenti).

### 5.6.1 Assegnazione del valore `seinfo`

Il valore `seinfo` passato a Zygote è preso in considerazione come etichetta arbitraria quando viene consultato il file `plat_seapp_contexts` al fine di determinare il dominio dell'app e l'etichette dei suoi file. Questo valore è calcolato in `Policy.getMatchedSeInfo` nel file `/platform/frameworks/base/services/core/java/com/android/server/pm/SELinuxMMAC.java` e, sia dal codice che dai commenti, è facile capire l'algoritmo usato.
Viene letto, una volta tantum, il file `/system/etc/selinux/plat_mac_permissions.xml` il quale contiene la struttura XML mostrata qui sotto.

```
<?xml version="1.0" encoding="iso-8859-1"?><!-- AUTOGENERATED FILE DO NOT MODIFY -->
<policy>
    <signer signature="308204...de">
        <seinfo value="platform"/>
    </signer>
    <signer signature="308204...6e">
        <seinfo value="media"/>
    </signer>
    <signer signature="308206...b1">
        <seinfo value="network_stack"/>
    </signer>
</policy>
```

Questa associa ad ogni certificato di firma il valore di `seinfo` da usare. Le app quindi ricevono un valore in base a chi le ha firmate. Nel codice è possibile vedere che l'algoritmo è leggermente più flessibile:

1. Ogni policy contiene una o più *stanze*. Ogni stanza contiene un insieme di certificati di firma, un valore di `seinfo` di default ed una mappa opzionale che associa ad ogni package uno specifico valore di `seinfo`.

2. Se tutti i certificati con cui è stata firmata l'applicazione da avviare coincidono esattamente con i certificati di una stanza (deve esserci una corrispondenza biunivoca), allora viene usata la mappa

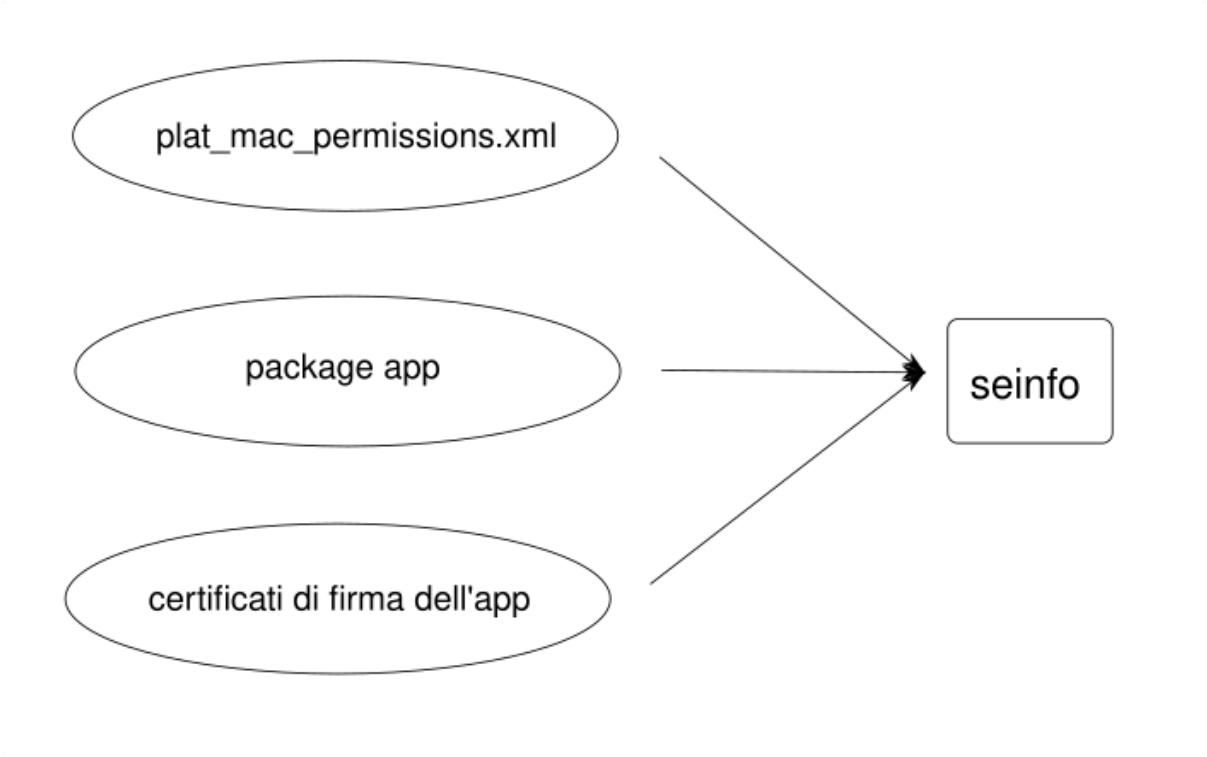
dei package per determinare se c'è un valore specifico di `seinfo`. Se la mappa non contiene il package dell'app da avviare, viene usato il valore globale.

3. Se i certificati dell'app non coincidono con quelli di nessuna stanza, il valore di `seinfo` è uno di default (`default`, appunto).

Notare come gli overload della funzione `SELinuxMMAC.getSeInfo` si occupino di aggiungere alla stringa `seinfo` ottenuta dal file `plat_mac_permissions.xml` le altre parti necessarie a formare il valore `managed_seinfo` (tipo `:isPrivApp` e simili).

Per le app con utente condiviso, il contesto se Linux è preso considerando il più piccolo SDK supportato (si veda `com.android.server.pm.SharedUserSetting.fixSeInfoLocked()`).

Nella classe `SELinuxUtil`, il metodo `assignSeinfoUser` è utilizzato per aggiungere, eventualmente, alla stringa `seinfo` il valore `:ephemeralapp`, ad indicare un'instant-app.




### 5.6.2 Permessi

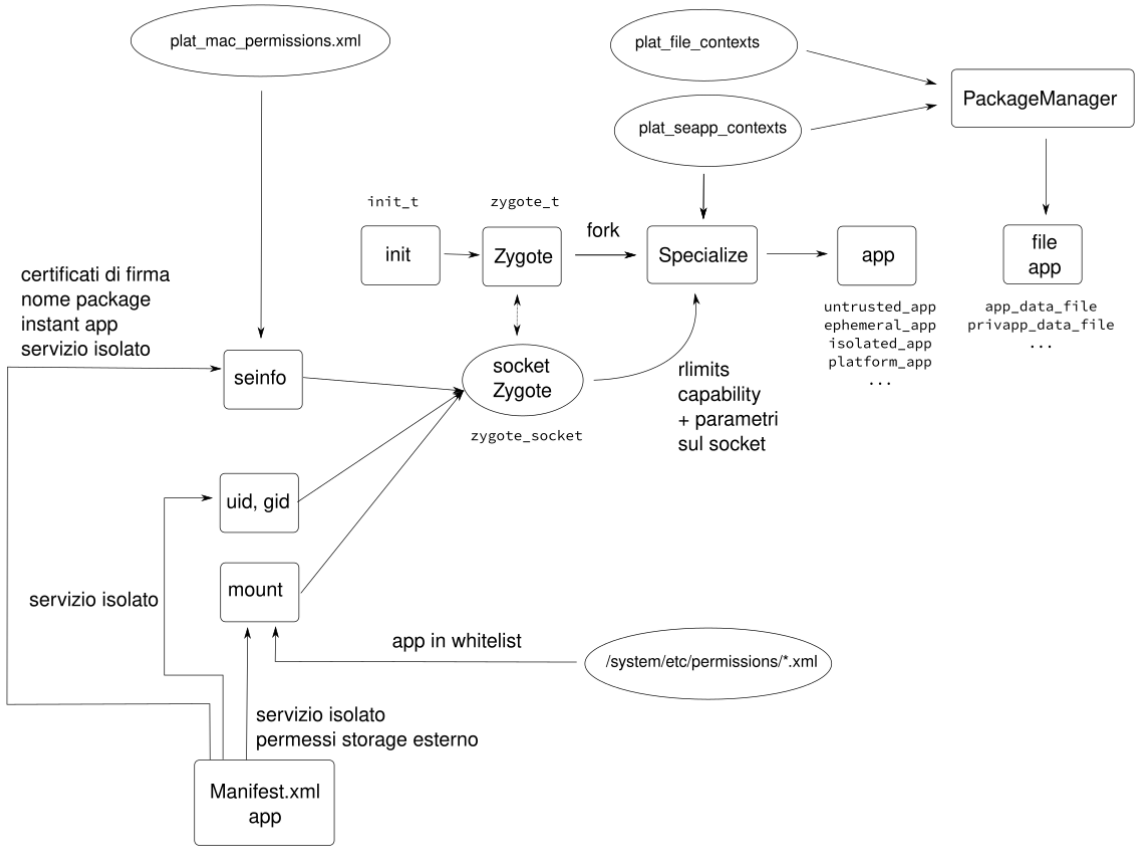
In tutto questo non abbiamo mai menzionato i permessi di Android. Che ruolo ha SELinux in questo? **Nessuno**.

SELinux non è usato per il controllo dei permessi delle app. Questi sono infatti controllati con la classe `PermissionManagerService` che tiene traccia dei permessi di ogni app in esecuzione. Questo è un meccanismo interamente a user space. Va però aggiunto che questa classe si trova nel System Server ed è quindi in un dominio SELinux apposito.

Secondo [22] SELinux è usato per controllo dei permessi delle app durante le installazioni, compito che sarebbe svolto dal `PackageManagerService`. A noi non risulta che il meccanismo descritto nel documento, che fa riferimento a `mac_permissions.xml`, sia usato per il calcolo dei permessi Android (ma per il calcolo del dominio SELinux di un'app e per l'etichetta dei suoi file).

### 5.6.3 Schema finale

Possiamo ora disegnare uno schema definitivo di come siano isolate le app in Android, incluso quindi il ruolo di SELinux. La figura è volutamente semplificata.




Notare che in Android i processi sono avviati con lo scopo di far partire un servizio o un activity, per cui è in questo contesto che vanno considerati i campi del manifesto come quello che indica se un servizio è isolato. Se un'app ha due servizi, uno isolato ed uno no, essa finirà in due domini SELinux diversi a seconda di quale servizio è fatto partire.

# 6 Servizi

Android offre una serie di servizi per l'accesso a funzionalità privilegiate (come l'accesso ai sensori, bluetooth, wifi, posizione GPS, eccetera). Per chi programma in Android, queste funzionalità si ottengono chiedendo il relativo *manager* al *service manager*, tipicamente con una chiamata del tipo `getSystemService(Context.SENSOR_SERVICE)`.

Tutti questi manager sono in realtà servizi, un po' Java un po' nativi, che operano all'interno del *System Server*. Questo è un processo privilegiato forkato direttamente e automaticamente da Zygote.

Non entreremo nei dettagli del System Server, basti sapere che esso contiene, in thread diversi, servizi molto importanti, inclusi quelli della gestione dei permessi e dell'avvio delle app.
Il comando seguente mostra il gran numero di thread del System Server.

```
generic_x86_arm:/data/busybox # ls /proc/$(pidof system_server)/task | wc -l
148
```

Dato che il nostro scopo è quello di analizzare l'utilizzo di SELinux in Android, non sembra avere senso parlare del System Server o dei servizi, il cui dominio è già calcolato nel modo visto precedentemente. Il motivo per cui ci soffermiamo sui servizi è perché in Android le comunicazioni IPC sono gestite, in via preferenziale, dal *binder*. Questo meccanismo è un'implementazione di *OpenBinder*[24].

## 6.1 Il binder ed il meccanismo di invito

La descrizione del binder di Android può essere trovata in [23], forniremo quindi solo una panoramica generale.

Il binder è un componente kernel che espone la sua interfaccia su `/dev/binder`, questa è controllata tramite IOCTL e consente di fare tre operazioni principali:

- Registrazione di un servizio. Detto *bind*. Notifica al kernel la presenza di un nuovo thread che offre un servizio.
- Ricerca di un servizio. Detto *find* o *discovery*. Ottiene un handle per la comunicazione con un servizio in un thread qualsiasi.
- Chiamata di un metodo di un servizio. Il classico servizio RPC, il binder si occupa di fare il controllo di accesso, il marshalling dei dati ed il loro mapping nei due thread.

I servizi accessibili con il binder sono tutti i classici servizi di Android, per molti dei quali è necessario richiedere un permesso nel manifest dell'app. Viene naturale chiederci se sia possibile bypassare i check sui permessi chiamando il binder direttamente.

Sebbene il binder, la componente kernel, non sia a conoscenza dei permessi Android, essa ha comunque un meccanismo di protezione detto *ad invito*. Questo meccanismo consente di avere un servizio privilegiato, detto *Service Manager*, che può o meno fornire l'accesso agli altri servizi.
Il binder kernel forza questa semplice regola:

> Quando il thread $T$ invia una comunicazione al servizio $S$, ad $S$ viene permesso l'accesso ai servizi nel thread $T$.

Si tratta di un meccanismo in cui, per poter accedere ad un servizio, un processo deve essere stato contattato da questo precedentemente.

Questa regola da sola però non basta, non c'è infatti un "inizio", in questo modo nessuno può comunicare con nessuno perché nessuno può iniziare una comunicazione senza che qualcun altro lo abbia fatto prima.
Per ovviare a questo paradosso, il binder usa il Service Manager. Il primo servizio registrato diviene infatti un servizio privilegiato[25] detto Service Manager. Per comunicare con il service manager **non** è necessario nessun invito.

Quindi tutti i thread possono comunicare con il Service Manager, ma devono essere invitati per comunicazione con gli altri servizi.

Il Service Manager è un componente user space ma che ha l'autorità di negare le richieste fatte al binder[25], questo infatti lo consulta per gestire le richieste di *bind* e *find* dei servizi.
Se il Service Manager nega l'accesso ad un servizio, il thread richiedente non ha modo di accedervi, la richiesta di *find* è infatti simile alla syscall *open*, senza l'handle non si può procedere[26].

## 6.2 Il ServiceManager ed il controllo MAC

Abbiamo visto che il Service Manager ha in mano il compito di gestire l'accesso ai servizi registrati in Android (siano essi di sistema o delle app).

A tal file Android ha esteso SELinux con due backend: uno per i servizi ed un per le proprietà di sistema. Il codice si trova in `/platform/external/selinux/libselinux/src/label.c`. Questi backend sono usati con la funzione standard della libreria SELinux `selabel_lookup`, che dato il nome di un oggetto (e altri due parametri che dipendono dal backend) ritorna il contesto di quell'oggetto. Normalmente `selabel_lookup` lavora con i backend standard di Linux (si veda [26]) ma il service manager usa un suo backend apposito. Questo è usato per dare un'etichetta ad ogni servizio, registrato nel binder, tramite il file `/system/etc/selinux/plat_service_contexts`. Ogni riga di questo file ha il formato `chiave contesto` dove `chiave` è una stringa che viene matchata sul nome del servizio che un processo vuole accedere.

Grazie agli hook SELinux implementati nel binder kernel, è possibile definire, nella policy, regole per l'accesso ai servizi in base al dominio del processo chiamante.

Ad esempio la riga che etichetta il servizio bluetooth è la seguente:

```
bluetooth                    u:object_r:bluetooth_service:s0
```

E nella policy, relativamente al dominio `bluetooth`, troviamo:

[25] Il cui handle ha valore fisso pari a zero.
[26] Gli handler del binder sono univoci globalmente, mentre i file descriptor sono unici per processo, ma il kernel tiene comunque traccia di quali handle un processo ha aperto e quali no.

```
allow bluetooth audioserver_service:service_manager find;
allow bluetooth bluetooth_service:service_manager find;
allow bluetooth drmserver_service:service_manager find;
allow bluetooth mediaserver_service:service_manager find;
allow bluetooth radio_service:service_manager find;
allow bluetooth app_api_service:service_manager find;
allow bluetooth system_api_service:service_manager find;
allow bluetooth network_stack_service:service_manager find;
allow bluetooth system_suspend_control_service:service_manager find;
```

Infine, nel file `plat_seapp_contexts` troviamo i criteri affinché un'app sia etichettata con il dominio `bluetooth`:

```
user=bluetooth seinfo=platform domain=bluetooth type=bluetooth_data_file
```

Questa regola indica che solo le app firmate con una chiave "platform" e lanciate con l'utente corrispondente a `AID_BLUETOOTH`. Il processo lanciato con questo uid è il demone *hcid* (`/platform/external/bluez/utils/hcid/main.c`).

Nella cartella `/platform/native/cmds/servicemanager/` si trovano i file `ServiceManager.cpp`, che contiene le chiamate alla funzioni di controllo dei permessi MAC SELinux, e `Access.cpp`, che contiene le chiamate alle funzioni standard della libreria userspace di SELinux.

Questo meccanismo MAC del binder lavora ad un livello molto basso e prende in considerazione il contesto di un processo e di un servizio. Tuttavia le app utente girano sotto un insieme limitato di domini (`untrusted_app`, `ephemeral_app`, `isolated_app`), questi non differenziano tra le app che hanno i permessi Android per accedere ad un servizio e le app che non le hanno.

Nella policy, relativamente all'accesso ai servizi da parte della app, troviamo:

```
# Use the Binder.
binder_use(appdomain)
# Perform binder IPC to binder services.
binder_call(appdomain, binderservicedomain)
# Perform binder IPC to other apps.
binder_call(appdomain, appdomain)
# Perform binder IPC to ephemeral apps.
binder_call(appdomain, ephemeral_app)
```

Dove è chiaro che le app possono accedere al binder ed ai servizi di altre app utente. Possono però anche accedere ai servizi etichettati con `binderservicedomain`, tra questi vi è anche il servizio per il controllo dei permessi.

```
# allow all services to run permission checks
allow binderservicedomain permission_service:service_manager find;}
```

I permessi Android sono quindi forzati a livello di servizio e non da SELinux. Ad esempio il servizio per l'accesso alle statistiche della batteria è etichettato con `batterystats_service`, le due regole di policy mostrate sotto indicano che le app utente possono accedere a questo servizio.

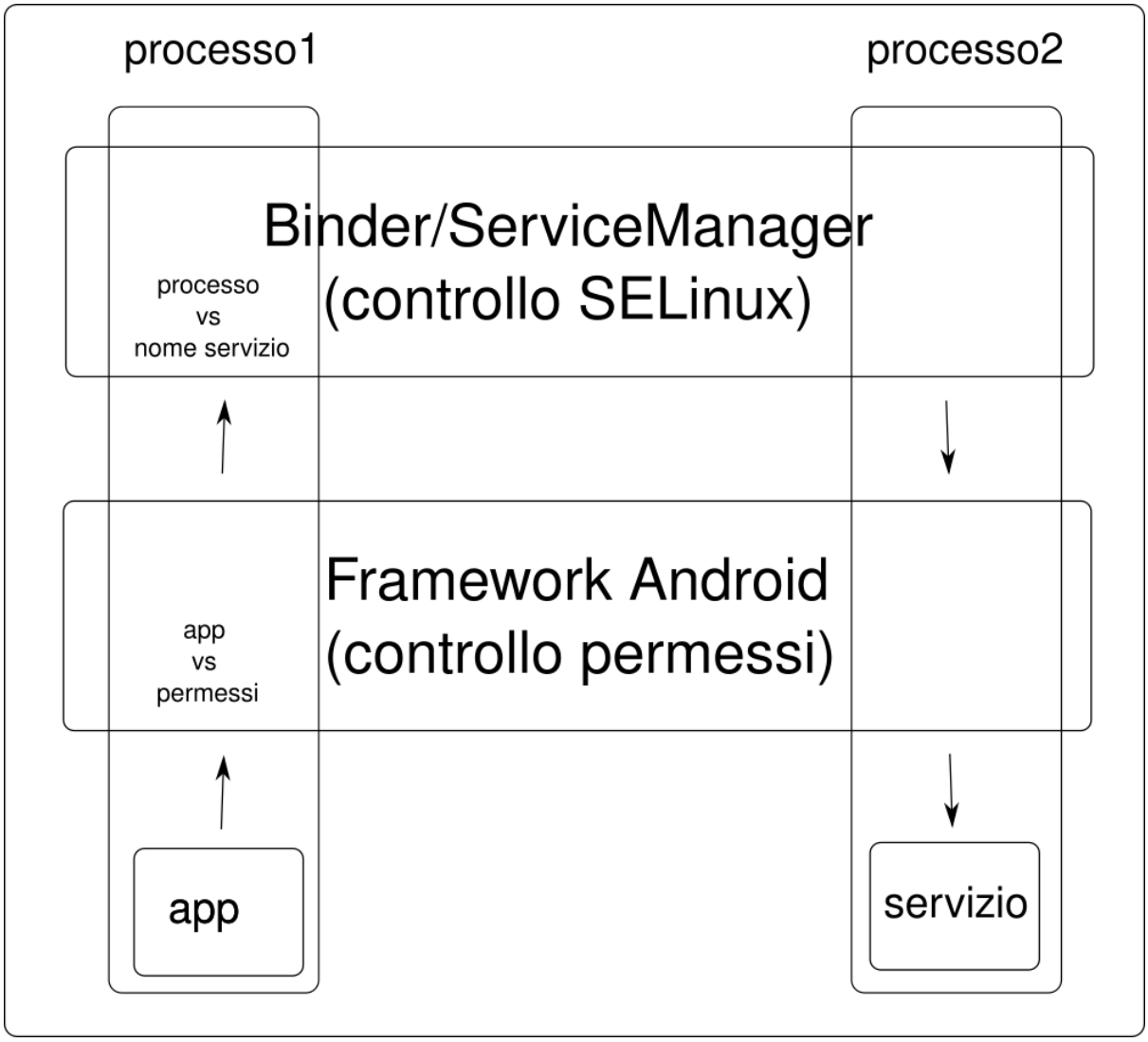
```
type batterystats_service, app_api_service, ephemeral_app_api_service, system_server_service,
     service_manager_type;
allow untrusted_app_all app_api_service:service_manager find;
```

Ma ovviamente il servizio prima controlla che l'app abbia i necessari permessi Android.

Per riepilogare, l'utilizzo di SELinux per il controllo degli accessi ai servizi è un meccanismo che lavora a livello di binder, impedisce la comunicazioni tra un processo non autorizzato ed un servizio privilegiato. Ma è principalmente usato per evitare l'escalation quando un processo di sistema viene compromesso[27], non per l'implementazione dei permessi Android.
Questi rimangono a carico del servizio stesso, che deve fare controlli espliciti (usando l'apposito servizio di controllo dei permessi). Possiamo immaginare che per i servizi delle app utente che richiedono permessi personalizzati, Android li incapsuli in un wrapper che faccia i necessari controlli automaticamente.

Il seguente schema dovrebbe aiutare a fare mente locale sulla situazione.

[27] Alcuni processi che hostano servizi sono lanciati direttamente da `init` e hanno libertà particolari nella policy Linux.




### 6.3 Il System Server

Il System Server è un fork di Zygote privilegiato, contiene molti servizi del framework di Android. La policy contiene infatti queste regole:

```
binder_use(system_server)
binder_call(system_server, appdomain)
binder_call(system_server, binderservicedomain)
binder_call(system_server, dumpstate)
binder_call(system_server, fingerprintd)
binder_call(system_server, gatekeeperd)
binder_call(system_server, gpuservice)
binder_call(system_server, idmap)
binder_call(system_server, installd)
binder_call(system_server, incidentd)
binder_call(system_server, iorapd)
binder_call(system_server, netd)
binder_call(system_server, notify_traceur)
userdebug_or_eng(‘binder_call(system_server, profcollectd)’)
binder_call(system_server, statsd)
binder_call(system_server, storaged)
binder_call(system_server, update_engine)
binder_call(system_server, vold)
binder_call(system_server, wificond)
binder_call(system_server, wpantund)
binder_service(system_server)
```

Che permettono al System Server di chiamare molti servizi, tra cui tutti quelli create dalle app (cfr. `appdomain`).

Il System Server ha delle regole di policy meno restrittive rispetto alle app utente ma per il resto è soggetto alle regole di accesso ai servizi appena viste, in questo aspetto non è speciale.
La sua principale utilità consiste nell'avere una serie di servizi essenziali già pronti quando si avviano le app e di circoscrivere il codice del framework che è privilegiato.

# 7 Proprietà di Android

Le proprietà di sistema (System properties) sono coppie chiave-valore usate da Android per passare dei parametri di configurazione a tutti i componenti del sistema. Sono caricate da file in sola lettura e mappate in memoria.

L'accesso a queste proprietà è gestito tramite un servizio che implementa un meccanismo di accesso MAC del tutto simile a quello usato per i servizi. I controlli sono effettuati in `/platform/system/core/init/property_service.cpp` e le etichette delle proprietà sono prese da `/system/etc/selinux/plat_proprerty_contexts`.

Lo scopo è quello di limitare i danni che un processo privilegiato compromesso può compiere (alcune proprietà se modificate possono portare a LPE).

# 8 Conclusioni

## 8.1 Esempi di exploit mitigati da SELinux

Riportiamo alcuni esempi di exploit per Android mitigati da SELinux e presi da [22].

### 8.1.1 KillingInTheNameOf

Questo exploit utilizza `mprotect` per rimappare in scrittura l'area di memoria in cui *init*, tramite `ashmem`, mappa alcune proprietà di sistema. Tra queste vi è `ro.secure`, se posta a 0 permette di ottenere una shell privilegiata con *adb*.

SELinux è in grado di bloccare questo exploit perché gli altri processi non hanno diritto di scrivere sulle aree `ashmem` di `init`.

### 8.1.2 psneuter

Questo exploit usa un IOCTL di `ashmem` per impostare il tipo di protezione da usare per il mapping delle aree di memoria. Usando un tipo non valido, il mapping fallisce facendo risultare tutte le proprietà (inclusa `ro.secure`) con valore 0.

SELinux non mitiga la possibilità di usare IOCTL per `ashmem` ma il processo root avviato da *adb* rimane confinato dalla policy.

### 8.1.3 Mempodroid

Questo exploit sfrutta una vulnerabilità nella gestione kernel del file system `/proc/<pid>/mem` che può indurre un programma a scrivere la propria memoria. Questo diventa un problema quando il programma ha il bit *setuid* impostato ed i dati scritti possono essere passati da input, come è il caso del programma `run-as` quando `stderr` viene impostato all'fd del proprio `/proc/<pid>/mem`.

Un policy ad hoc per `run-as` può impedire a questo di scrivere su `/proc/<pid>/mem` ed in ogni caso se anche si ottenesse una shell con uid=0, il contesto di questa rimane quello del chiamante.

### 8.1.4 File world-readable

Molte delle vulnerabilità delle app utente, prima dell'introduzione di SELinux in Android, era relative al non corretto uso dei permessi DAC. Molte app creavano file leggibili da chiunque, con il risultato di esporre dati privati al furto.

## 8.2 Il paradosso Google-Apple

Abbiamo visto che Android ha subito un notevole sforzo ingegneristico per includere SELinux, la policy introdotta è in grado di limitare notevolmente i danni di un processo privilegiato compromesso. Addirittura, una shell di root non è sufficiente per l'accesso completo al dispositivo. Infatti, questa rimane eseguita con il contesto dell'app chiamante e, nonostante l'uid privilegiato, la shell non è in grado di accedere ai file delle altre app.

*Possiamo quindi dire che Android ha raggiunto un livello di sicurezza maturo?*
**No.**
Quando si analizza la sicurezza di un sistema è necessario guardarlo da ogni aspetto. Android è molto resistente ad attacchi che mirano al controllo completo del dispositivo. Tutte le misure SELinux sono molto efficaci per il confinamento di processi privilegiati, gli sviluppatori del kernel Linux reagiscono velocemente alla scoperta di falle di sicurezza e adottano una politica di trasparenza (open source, responsible disclosure) che favorisce la creazione di software sicuro. Sebbene, alcune vulnerabilità persistano in modelli obsoleti di smartphone (la più nota è *Dirty cow*[27], usata dallo spyware Exodus[28]), in generale gli aggiornamenti di Android sono tempestivi.

Vale la pena confrontare questo aspetto della sicurezza di Android con l'analogo per iOS. iOS è un colabrodo a riguardo, l'utilizzo di codice nativo genera un enorme superficie di attacco. Il degrado della qualità che i prodotti Apple stanno subendo in questi ultimi anni ha portato alla scoperta di numerosissime falle di sicurezza la cui gravità media è molto elevata. In quasi tutti i security advice di Apple, pubblicati a seguito di aggiornamenti, compaiono vulnerabilità di RCE ed escalation dei privilegi. Spesso addirittura a livello kernel. Al punto che Zerodium, nota azienda che acquista 0-day, ha pubblicato un tweet in cui bloccava l'acquisto di vulnerabilità LPE e simili per iOS, dato l'elevato numero di offerte[29]. A questo va aggiunto che Apple ha cicli di aggiornamento *molto* lenti e poco trasparenti.

In sostanza, i dispositivi Android sono ben protetti da attacchi di alto profilo mentre quelli Apple lo sono a mala pena.

Questo significa che Android è più sicuro di iOS?
No, se proprio fosse possibile e necessario stabilire un vincitore, quello sarebbe iOS.

Android infatti ha un approccio verso la comunità di sviluppatori ed utilizzatori che è "piuttosto rispettoso", c'è la tendenza, tipica dell'open source, a considerare gli utilizzatori (includendo anche i programmatori di app) come tecnici competenti. Nessuno installerebbe mai un'applicazione scaricata tramite un link contenuto in un'e-mail proveniente da un indirizzo forgiato, giusto? Invece è proprio questo il punto debole di Android, la fiducia negli utenti e nei programmatori.

Apple dal lato suo ha un approccio completamente diverso: i nostri utenti sono incapaci e non vogliono pensare o prendere decisioni, per cui lo facciamo noi per loro. Inoltre i programmatori sono stupidi, per cui le loro app devono passare i nostri test e devono pagare per avere un posto nel nostro store (in modo da scremare tutti quelli che non sono professionalmente stabili da permettersi un abbonamento annuale di 100€).

Ironicamente questo approccio funziona, perché pochissimi utenti sono vittime di attacchi di alto livello (le 0-day costano e una volta "bruciate" non possono essere usate per altri soggetti, magari più interessanti/pericolosi); La grande maggioranza di utenti è vittima di truffe, che spesso si concretizzano con l'uso di malware. I malware utilizzano il principio del *minimo sforzo*[30], per cui se ottenere i privilegi per accedere alle altre app è troppo faticoso, ci si concentrerà su altri metodi.
In Android, è molto difficile essere vittime di ransomware o di infostealer, proprio perché è molto difficile accedere ai file di altre applicazioni. Tuttavia è molto facile ingannare l'utente, perché Android ha molte componenti framework estendibili, tra cui anche servizi che non sono privilegiati a livello di sistema operativo ma sono molto molto privilegiati nel framework. Lo stesso framework che gestisce il contenuto delle activity nello schermo, l'interazione con la tastiera ed il tocco.
Stiamo parlando di componenti come i servizi di accessibilità[28] o le tastiere di terze parti.

La tendenza dei malware per Android è quella di usare un servizio di accessibilità per controllare il dispositivo, si veda ad esempio [31], con un risultati efficaci quanto inquietanti.

Nel panorama iOS non sono diffusi malware, non è possibile installare applicazioni da terze fonti e la pubblicazione sull'app store richiede il pagamento di una quota annuale e il superamento dei test di Apple. Tutti elementi che violano il principio del minimo sforzo, il gioco non vale la candela.

Tuttavia, avversari più determinati (leggi APT o governativi), trovano terreno più facile su iOS.

[28] Che hanno accesso al testo digitato dall'utente, possono simulare tocchi sullo schermo e leggere tutte le view delle activity mostrate

In finale, c'è questo paradosso tra Google ed Apple: il primo è sicuro contro attacchi di alto livello (e più rari) ma molto insicuro contro attacchi banali (e più diffusi), mentre il secondo è molto insicuro contro attacchi di alto livello ed è sicuro contro attacchi banali.

# Riferimenti bibliografici


[1] SELinux concepts
https://source.android.com/security/selinux/concepts.

[2] Understanding Linux File Permissions
https://www.linux.com/training-tutorials/understanding-linux-file-permissions/

[3] SELinux Overview
http://selinuxproject.org/page/NB_Overview

[4] SELinux/Labels
https://wiki.gentoo.org/wiki/SELinux/Labels

[5] Understanding SELinux Roles
https://danwalsh.livejournal.com/75683.html

[6] capabilities(7), Linux manual page
https://man7.org/linux/man-pages/man7/capabilities.7.html

[7] Section 7.1. A Closer Look at the Access Decision Algorithm
https://flylib.com/books/en/2.803.1.54/1/

[8] Constraint Statements
https://selinuxproject.org/page/ConstraintStatements

[9] Magisk, the magic mask for Android
https://github.com/topjohnwu/Magisk

[10] SELinux from the inside out
https://www.imperialviolet.org/2009/07/14/selinux.html

[11] Dalvik
https://en.wikipedia.org/wiki/Dalvik_(software)

[12] Policy placement
https://source.android.com/security/selinux/customize#policy-placement

[13] `/platform/system/sepolicy`
https://android.googlesource.com/platform/system/sepolicy/+/master/README

[14] `security_compute_create`
https://linux.die.net/man/3/security_compute_create

[15] `setrlimit`
https://linux.die.net/man/2/setrlimit

[16] Storage
https://source.android.com/devices/storage

[17] Understanding the behavior of unshare CLONE_NEWNS
https://stackoverflow.com/questions/6258696/understanding-the-behavior-of-unshare-clone-newns

[18] File Based Encryption https://source.android.com/security/encryption/file-based

[19] Supporting Multiple Users https://source.android.com/devices/tech/admin/multi-user

[20] Overview of Google Play Instant https://developer.android.com/topic/google-play-instant/overview

[21] Run-as source code https://android.googlesource.com/platform/system/core.git/+/android-4.2.2_r1/run-as/run-as.c#36

[22] Security Enhanced (SE) Android: Bringing Flexible MAC to Android
https://www.ndss-symposium.org/wp-content/uploads/2017/09/02_4.pdf

[23] Android Binder Security Note
On ¿Passing Binder Through Another Binder¡
http://crypto.hyperlink.cz/files/xbinder.pdf

[24] OpenBinder
https://en.wikipedia.org/wiki/OpenBinder

[25] Android IPC: Part 2 - Binder and Service Manager Perspective
https://blog.hacktivesecurity.com/index.php?controller=post&action=view&id_post=48

[26] selabel_lookup
https://linux.die.net/man/3/selabel_lookup

[27] Dirty cow
https://en.wikipedia.org/wiki/Dirty_COW

[28] Exodus, eSurv: storia di uno spyware italiano
https://www.punto-informatico.it/exodus-esurv-spyware-italiano/

[29] Prices for iOS one-click chains (e.g. via Safari) without persistence will likely drop in the near future.
https://twitter.com/zerodium/status/1260541578747064326?lang=en

[30] I malware nella PA.
http://eventipa.formez.it/sites/default/files/allegati_eventi/Malware_nella_PA_LUSINI.pdf

[31] Oscorp, il "solito" malware per Android
https://cert-agid.gov.it/news/oscorp-il-solito-malware-per-android/