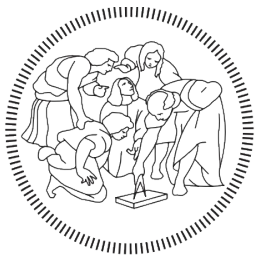

POLITECNICO
MILANO 1863

School of Industrial and Information Engineering
Master of Science in Computer Science and Engineering

SELinux policies for fine-grained protection of Android apps

Supervisor: Prof. Stefano PARABOSCHI
Co-supervisor: Prof. Gerardo PELOSI

Master's thesis by:
Matthew ROSSI, 858880

Academic Year 2017-2018

Abstract

Android supports three different access control techniques: Android Permission Framework, kernel-level Discretionary Access Control and Mandatory Access Control. Mandatory Access Control has been implemented with the use of SELinux and its adaptation to Android (SEAndroid). Currently, the use of the MAC model is limited to the protection of system resources. All the apps that are installed by users fall in a single undifferentiated domain, untrusted app. We implement an extension of the architecture that permits to associate with each app a dedicated MAC policy, contained in a dedicated app policy module, in order to protect app resources even from malware with root privileges. A crucial difference with respect to the support for policy modules already available in some SELinux implementations is the need to constrain the policies in order to guarantee that an app policy is not able to manipulate the system policy. We present the security requirements to be enforced on modules and show that our solution satisfies these requirements. The support for SELinux policies for fine-grained protection of Android apps can also be the basis for the automatic generation of policies, with a stricter enforcement of Android permissions. A prototype has been implemented and tested with an example app to inspect its feature and functionalities.

Sommario

Android sfrutta diverse tecniche di controllo degli accessi: Android Permission Framework, Discretionary Access Control del kernel Linux e Mandatory Access Control. L'implementazione del Mandatory Access Control di Android è realizzata con l'uso di SELinux, in particolare una sua variante, SEAndroid. Attualmente, l'uso del modello MAC è limitato alla protezione delle risorse di sistema, mentre tutte le applicazioni installate dagli utenti vengono eseguite in un singolo dominio, untrusted app.

In questa tesi abbiamo implementato un estensione dell'architettura che permette di associare ad ogni app una politica di sicurezza MAC, con l'obiettivo di proteggere le risorse dell'app perfino da malware con privilegi di root. Una differenza cruciale, rispetto a implementazioni già esistenti di SELinux che supportano la modularità della politica di sicurezza, è la necessità di filtrare tali moduli per salvaguardare la politica di sicurezza di sistema dall'installazione di nuove applicazioni, le quali non possono essere ritenute fidate. Presentiamo quindi i requisiti di sicurezza necessari affinchè la politica di sicurezza possa essere installata e mostriamo che la nostra soluzione soddisfa tali requisiti.

L'introduzione di moduli di sicurezza specifici per la protezione delle applicazioni con SELinux potrà inoltre fungere da base per la generazione automatica delle politiche di sicurezza, garantendo ulteriormente il rispetto degli Android permissions attraverso l'utilizzo di MAC. Un prototipo è stato implementato e testato con un'applicazione esempio per verificarne le caratteristiche e funzionalità.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The wide deployment of mobile operating systems has introduced a number of challenging security requirements.

On one hand, mobile devices are high-value targets, since they offer a direct financial incentive in the use of the credit that can be associated with the device or in the abuse of the available payment services (e.g., Google Wallet, telephone credit and mobile banking) [8]. In addition, mobile devices permit the recovery of large collections of personal information and are the target of choice if an adversary wants to monitor the location and behavior of an individual.

On the other hand, the system presents a high exposure, with users continuously adding new apps to their devices, to support a large variety of functions. The risks are then greater and different from those of classical operating systems [12]. The frequent installation of external code creates an important threat. The design of security solutions for mobile operating systems has to consider a careful balance between, on one side, the need for users to easily extend with unpredictable apps the set of functions of the system and, on the other side, the need for the system to be protected from potentially malicious apps.

Android supports Mandatory Access Control, which extends previous security services relying on the Android Permission Framework and on the kernel-level Discretionary Access Control. This extension has been obtained with the use of SELinux and its adaptation to Android (SEAndroid [15]). The goal of SEAndroid is to build a mandatory access control (MAC) model in Android using SELinux to enforce kernel-level MAC, introducing a set of middleware

MAC extensions to the *Android Permission Framework*. The middleware MAC extension chosen to bridge the gap between SELinux and the Android permission framework is called *install-time MAC* [15] . This mechanism allows to check an app against a MAC policy (i.e. mac permissions.xml).

The integration of this middleware MAC ensures that the policy checks are unbypassable and always applied when apps are installed and when they are loaded during system startup. The current design of SEAndroid aims at protecting core system resources from possible flaws in the implementation of security in the *Android Permission Framework* or at the DAC level. The exploitation of vulnerabilities becomes harder due to the constraints on privilege escalation that are introduced by SELinux. Unfortunately, the current use of SELinux in Android aims at protecting the system components and trusted apps from abuses by third-party apps. All the third-party apps fall within a single *untrusted app* domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on Android permissions and the Linux DAC support. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy.

We implement an extension of the architecture that permits to associate with each app a dedicated MAC policy, contained in a dedicated app policy module, in order to protect app resources even from malware with root privileges. A crucial difference with respect to the support for policy modules already available in some SELinux implementations is the need to constrain the policies in order to guarantee that an app policy is not able to manipulate the system policy.

We present the security requirements that have to be satisfied by the support for modules:

- *No impact on the system policy*
- *No escalation*
- *Flexible internal structure*
- *Protection from external threats*

and show that our solution satisfies these requirements.

The support for SELinux policies for fine-grained protection of Android apps can also be the basis for the automatic generation of policies, with a stricter enforcement of Android permissions.

The *Android Open Source Project* (AOSP) has extensively been modified to implement a prototype of the appPolicyModules methodology [10] and tested with an example app to inspect its feature and functionalities.

The thesis is structured as follows.

Chapter 2 describes *Security-Enhanced Linux* (SELinux), the primary *Mandatory Access Control* (MAC) mechanism built into almost any Linux distributions. A section is dedicated to explain what *Mandatory Access Control* (MAC), while other sections state some of the SELinux key concept needed to understand its potential and the usefulness in the thesis.

Chapter 3 introdues the current implementation of SELinux in Android and the required configuration files to assign security contexts across the system and define the policy. It also shows an excursus of the changes implemented along different Android releases since its introduction in Android 4.3.

Chapter 4 discusses the *appPolicyModules* methodology, indentifies the threat model that it aims to limit, the requirements to be enforced on the policy and how to correctly enforce them.

Chapter 5 describe the implementation of the discussed methodology in a recent Android release. First descrives how to correctly set up a working environment for the *Android Open Source Project* (AOSP) and then analyzes in depth the changes done to Android to implement a fine-grained Mandatory Access Control for third-party apps.

Chapter 6 concludes the thesis and exposes possible future developements.

Chapter 2

SELinux

Security-Enhanced Linux (SELinux) is the primary *Mandatory Access Control* (MAC) mechanism built into almost any Linux distributions. SELinux originally started as the Flux Advanced Security Kernel (FLASK) development by the Utah university Flux team and the US Department of Defence. The development was then enhanced by the NSA and released as open source software.

2.1 Core SELinux Components

Figure 2.1 shows a high level diagram of the SELinux core components that manage enforcement of the policy and comprise the following elements:

- A subject that causes the action on the object (such as a process that reads a file).
- An Object Manager that knows the set of actions that can be performed on the particular resource (such as a file) and can enforce those actions (i.e. allow to write a file if permitted by the policy).
- A Security Server that makes decisions regarding the subjects rights to perform the requested action on the object, based on the security policy rules.
- A Security Policy that describes the rules using the SELinux policy language.

- An Access Vector Cache (AVC) that improves system performance by caching security server decisions.

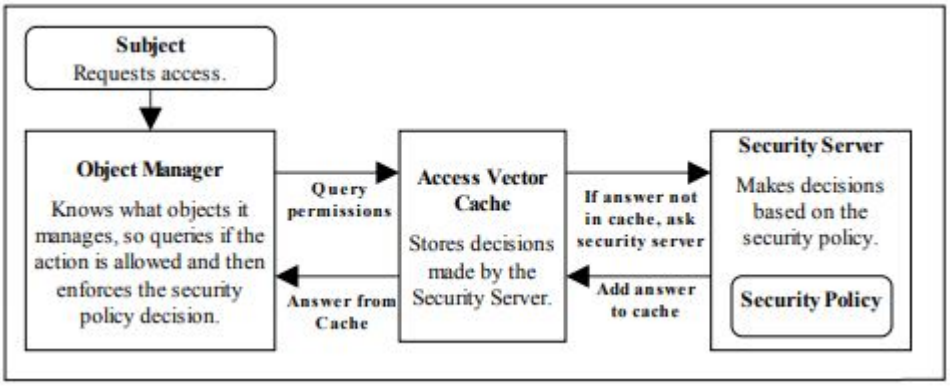

Figure 2.1: High Level Core SELinux Components - Decisions by the Security Server are cached in the AVC to enhance performance of future requests. Note that it is the Object Manager that enforce the policy.

SELinux has been implemented as part of the Linux Security Module (LSM) framework, which recognizes various kernel objects, and sensitive actions performed on them. At the point at which each of these actions would be performed, an LSM hook function is called to determine whether or not the action should be allowed based on the information for it stored in an opaque security object. SELinux provides an implementation for these hooks and management of these security objects, which combine with its own policy, to determine the access decisions.

2.2 Mandatory Access Control (MAC)

Mandatory Access Control (MAC) is a type of access control in which the operating system is used to constrain a subject (such as a user or process) from accessing or performing an operation on an object (such as a file, disk, memory etc.). Each of the subjects and objects have a set of security attributes that can be interrogated by the operating system to check if the requested operation can be performed or not.

Note that the subject (and therefore the user) cannot decide to bypass the policy rules (as long as SELinux is enabled) since a center authority is in charge to decide on all access attempts according to the MAC policy.

This is quite a difference with Linux's familiar *discretionary access control* (DAC) system. In a DAC system, a concept of ownership exists, whereby an owner of a particular resource controls access permissions associated with it. This is generally coarse-grained and subject to unintended privilege escalation.

In a SELinux enabled system, DAC and MAC work side-by-side to enforce access control and the decision making chain is shown in Figure 2.2.

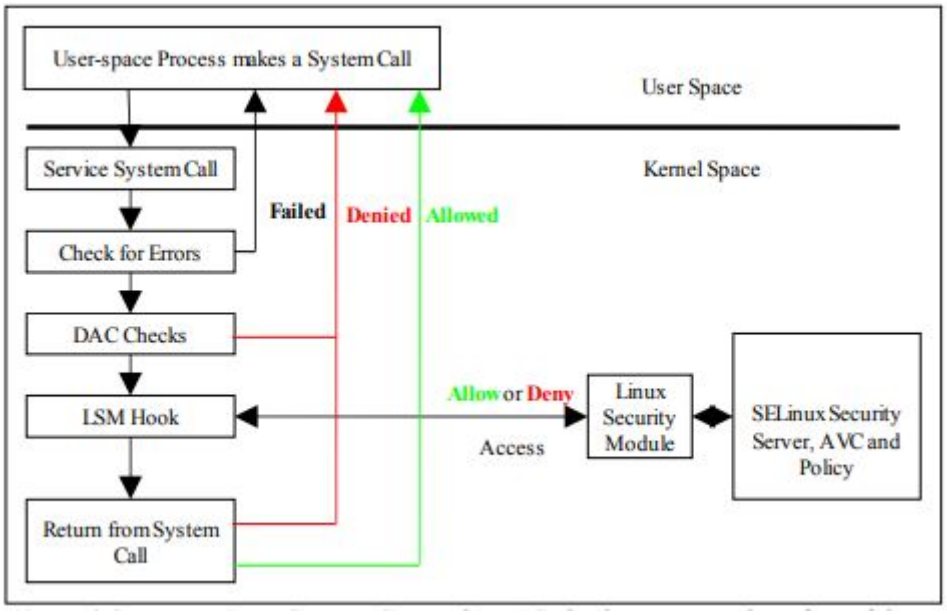

Figure 2.2: Processing a System Call - The DAC checks are carried out first, if they pass then the Security Server is consulted for a decision.

SELinux supports two forms of MAC, Type Enforcement and Multi-Level Security.

Type Enforcement labels each process and object with a security context and the actions that can be performed on objects are controlled by the policy. This is the implementation used for general purpose MAC within SELinux along with *Role Based Access Control*.

Multi-Level Security is an implementation based on the Bell-La Padula (BLP) model and it is used by organizations where different levels of access are required so that restricted information is separated from classified information

to maintain confidentiality. This allows enforcement rules such as 'no write down' and 'no read up' to be implemented in a policy by extending the security context with security levels.

Multi-Category Security (MCS), a variant of Multi-Level Security, is now more generally used to maintain application separation, for example SELinux enabled:

- Virtual Machines use MCS categories to allow each VM to run within its own domain to isolate VMs from each other.
- Android devices use dynamically generated MCS categories so that an app running on behalf of one user cannot read or write files created by the same app running on behalf of another.

2.3 Security Context

SELinux requires a *security context* to be associated with every subject and object that are used by the security server to decide whether access is allowed or not as defined by the policy.

Within SELinux, a security context is represented as a sequence of strings that define the SELinux user, role, type identifier and an optional MLS or MCS security range or level as follows:

`user:role:type[:range]`

Access decisions regarding a subject make use of all the components of the security context, in fact it is possible to explicitly add constraints in the policy on each part of the string.

2.3.1 User

Users in Linux are generally associated to human users (such as Alice and Bob) or operator/system functions (such as admin), while this can be implemented in SELinux, SELinux user names are generally groups or classes of user.

The SELinux user name is the first component of a 'security context' and by convention SELinux user names end in '`_u`', however this is not enforced by any SELinux service (i.e. it is only to identify the user component).

2.3.2 Role-Based Access Control (RBAC)

To further control access to TE domains SELinux makes use of *role-based access control* (RBAC). This feature allows SELinux users to be associated to one or more roles, where each role is then associated to one or more domain types.

The SELinux role name is the second component of a 'security context' and by convention SELinux roles end in '`_r`', however this is not enforced by any SELinux service (i.e. it is only used to identify the role component).

2.3.3 Type Enforcement (TE)

SELinux enforces mandatory access control associating to all subjects and objects a type identifier that can then be used to enforce rules laid down by the policy.

The SELinux type identifier is a simple string that is defined in the policy and then associated to a security context. It is also used in the majority of SELinux language statements and rules to build a policy that will, when loaded into the security server, enforce policy via the object managers.

Because the type identifier (or just 'type') is associated to all subjects and objects, it can sometimes be difficult to distinguish what the type is actually associated with. In the end it comes down to understanding how they are allocated in the policy itself and how they are used by SELinux services.

Basically, if the type identifier is used to reference a subject it is referring to a Linux process or collection of processes (a domain or domain type). If the type identifier is used to reference an object, then it is specifying its object type (i.e. file type).

SELinux type is the third component of a 'security context' and by convention SELinux types end in '`_t`', however this is not enforced by any SELinux service (i.e. it is only used to identify the type component).

2.3.4 Multi-Level Security and Multi-Category Security

SELinux supports *Multi-Level Security* (MLS) and *Multi-Category Security* (MCS) by adding an optional level or range entry to the security context.

Figure 2.3 shows a simple diagram where security levels represent the classification of files within a file server. The security levels are strictly hierarchical and conform to the Bell-La & Padula model (BLP)[7] in which a process, running at the 'Confidential' level, can read or write at their current level but only read lower levels or write higher levels (assuming the process is authorized).

This ensures confidentiality as the process can copy a file up to the secret level but can never re-read that content unless the process steps up to that level, also the process cannot write files to the lower levels as confidential information would then drift downwards.

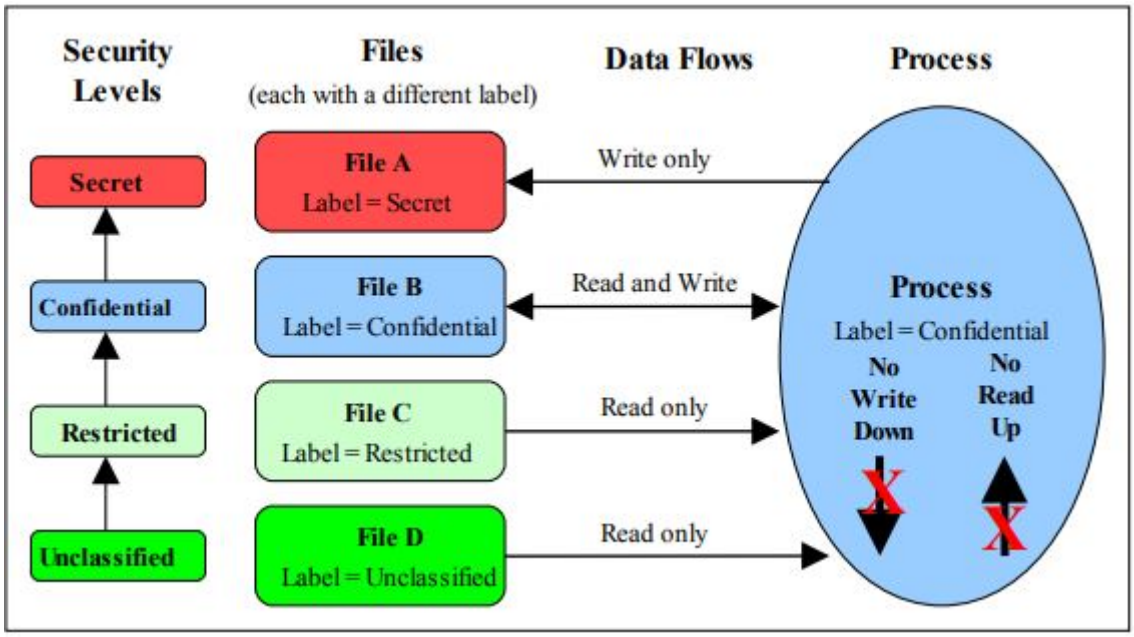

Figure 2.3: Security Levels and Data Flows - This shows how the process can only 'Read Down' and 'Write Up' within an MLS enabled system.

The sections that follow discuss the format of a security level and range, and how these are managed by the constraints mechanism within SELinux using dominance rules.

2.3.4.1 Security Levels

A *security level* (or level) consists of a *sensitivity* and zero or more *category* entries.

```
sensitivity [: category, ... ]
```

A range, the forth and optional component of the security context within an MLS or MCS environment, consists of two security levels for the lower and upper bound, the former identifies the current level or sensitivity and the latter the clearance.

```
sensitivity [: category, ... ] - sensitivity [: category, ... ]
```

Where:

- *sensitivity*: the hierarchical sensitivity level with (traditionally) s0 being the lowest. These values are defined using the sensitivity statement and their hierarchy is defined using the dominance statement. For MLS systems the highest sensitivity is the last one defined in the dominance statement (low to high). Traditionally the maximum for MLS systems is `s15`. For MCS systems there is only one sensitivity defined, and that is `s0`.

- *category*: the optional part of the sensitivity level that consists of zero or more categories and form an unordered and unrelated list of 'compartments'. These values are defined using the category statement, where for example `c0.c3` represents a range and `c0, c3, c7` represent an unordered list. Traditionally the values are between `c0` and `c255`.

- *level*: a combination of the sensitivity and category values that form the actual security level. These values are defined using the level statement.

The lowest level is generally '`s0`' with no categories and the highest is '`s15:c0.c255`'.

2.3.4.2 Dominance rules

As stated above, the security levels hierarchy is managed by dominance rules. These rules are as follows:

- Security Level 1 dominates Security Level 2 if the sensitivity of Security Level 1 is equal to or higher than the sensitivity of Security Level 2 and the categories of Security Level 1 are the same or a superset of the categories of Security Level 2.

- Security Level 1 is dominated by Security Level 2 if the sensitivity of Security Level 1 is equal to or lower than the sensitivity of Security Level 2 and the categories of Security Level 1 are a subset of the categories of Security Level 2.

- Security Level 1 equals Security Level 2 if the sensitivity of Security Level 1 is equal to Security Level 2 and the categories of Security Level 1 and Security Level 2 are the same set.

- Security Level 1 is incomparable to Security Level 2 if the categories of Security Level 1 and Security Level 2 cannot be compared (i.e. neither Security Level dominates the other).

To illustrate the usage of these rules, Table 2.1 lists the security level attributes associated to files. The process that accesses these files is running with a range of `s0-s3:c1.c5` and has access to the files highlighted within the grey box area. As the MLS dominance statement is used to enforce the sensitivity hierarchy, the security levels now follow that sequence (lowest = `s0` to highest = `s3`) with the categories being unordered lists of 'compartments'. To allow the process access to files within its scope and within the dominance rules, the process will be constrained by using the mlsconstrain.

	Category →	c0	c1	c2	c3	c4	c5	c6	c7
s3	Secret	s3:c0					s3:c5	s3:c6	
s2	Confidential		s2:c1	s2:c2	s2:c3	s2:c4			s2:c7
s1	Restricted	s1:c0	s1:c1						s1:c7
s0	Unclassified	s0:c0			s0:c3				s0:c7

Table 2.1: MLS Security Levels - Showing the scope of a process running at a security range of `s0 - s3:c1.c5`.

As illustrated in Figure 2.4 to allow write-up, the source level (`s0:c3` or `s1:c1`) must be dominated by the target level (`s2:c1.c4`) and to allow read-down, the source level (`s2:c1.c4`) must dominate the target level (`s0:c3`).

However, in the real world the SELinux MLS Reference Policy defaults to use `l1 eq l2`, i.e. the levels have to be equal to perform read and write operation.

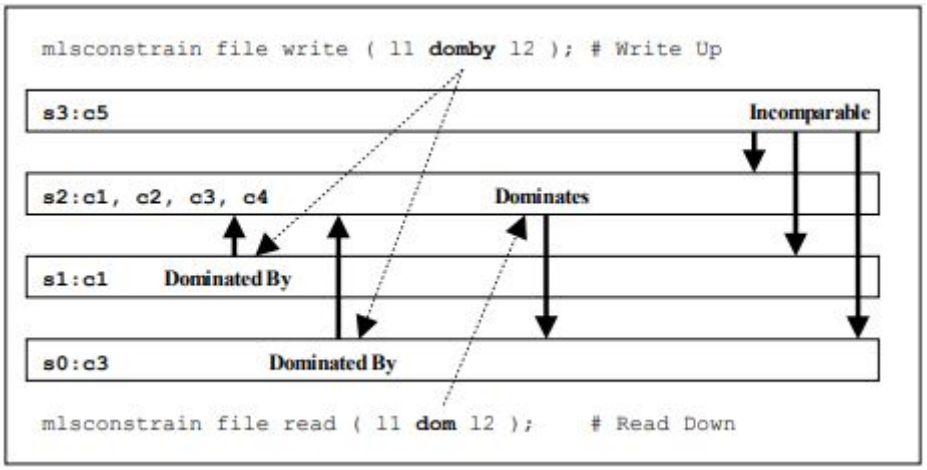

Figure 2.4: Showing the mlsconstrain statements controlling Read Down & Write Up - This ties in with Table 2.1 that shows a process running with a security range of `s0 - s3:c1.c5`.

2.4 Subjects

A subject is an active entity, generally in the form of a process or device, that changes the system state or causes information to flow among objects.

However, within SELinux, a process can also be addressed as an object, because each process has an associated object class called 'process' that defines what permissions the policy is allowed to grant or deny on the active process.

In SELinux subjects can be:

- *Trusted*: generally, these are commands, applications etc. that have been written or modified to support specific SELinux functionality to enforce the security policy (e.g. the kernel, init, and login). However, it can also cover any application that the organization is willing to trust as a part of the overall system. Although, the best policy is to trust nothing until it has been verified that it conforms to the security policy.

- *Untrusted*: everything else

2.4.1 Labeling Subjects

On a running Linux system, processes inherit the security context of the parent process as follows:

- On fork a process inherits the security context of its creator/parent.
- On exec, a process may transition to another security context based on policy statements or, if security-aware, by calling setexeccon if allowed by the policy.
- At any time, a security-aware process may invoke setcon to switch its security context if permitted by policy, although this practice is generally discouraged as exec-based transitions are preferred.

2.5 Objects

Within SELinux an object is a resource such as files, sockets, pipes or network interfaces that are accessed by processes (also known as subjects). These objects are classified according to the resource they provide with access permissions relevant to their purpose (e.g. open, read and receive).

2.5.1 Object Classes and Permissions

At each object is assigned a *class* identifier that defines its purpose (e.g. file, socket) along with a set of *permissions* that describe what services the object can handle (read, write, send etc.).

The objective of the policy is to enable the user of the object (the subject) access to the minimum permissions needed to complete the task according to the *'least privileged'* security principle.

These object classes and their associated permissions are built into object managers by developers and are therefore not generally updated by policy writers. The number of object classes and their permissions can vary depending on the features configured in the Linux release.

2.5.2 Labeling Objects

Within a running SELinux enabled system the labeling of objects is managed by the system without requiring any explicit user interaction (until labeling goes wrong). As processes and objects are created and destroyed, they either:

- Inherit their labels from the parent process or object.
- The policy type, role and range transition statements allow a different label to be.
- SELinux-aware applications can enforce a new label (with the policies approval of course) using the *libselinux API* functions.
- An object manager (OM) can enforce a default label that can either be built into the OM or obtained via a configuration file.
- Use an 'initial security identifier' (or initial SID). These are defined in all base and monolithic policies and are used to either set an initial context during the boot process or set any object that does not already have a valid context.

2.5.2.1 Labeling Extended Attribute Filesystems

The labeling of file systems that implement extended attributes (such as ext3 and ext4) is supported by SELinux using:

- The fs_use_xattr statement within the policy to identify what file systems use extended attributes. This statement is used to inform the security server how to label the filesystem.
- A 'file contexts' file that defines what the initial contexts should be for each file and directory within the filesystem.
- A method to initialize the filesystem with these extended attributes. This is achieved by SELinux utilities such as fixfiles and setfiles or also commands such as chcon, restorecon and restorecond.

Extended attributes containing the SELinux context of a file can be viewed by the ls -Z or getfattr commands.

2.5.3 Allowing a subject access to object

SELinux operates on the principle of default denial, which means that every single access for which it has a hook in the kernel must be explicitly allowed

by the policy. This means a policy file is comprised of a large amount of information regarding types, classes and permissions, but mostly allow rules that allow processes the given access permissions to an object.

The policy rules come in the form:

```
allow subject_types object_types:classes permissions;
```

Where:

- `subject_types`: an identifier for the process or set of processes
- `object_types`: an identifier for the object or set of objects
- `class`: the kind of object being accessed
- `permission`: the operation being performed

And so, an example use of this is:

```
allow unconfined_t ext_gateway_t : process transition;
```

The above allow rule shows that a process can also be an object as it allows processes running in the `unconfined_t` domain, permission to 'transition' the external gateway application to the `ext_gateway_t` domain once it has been executed.

In addition to individually listing domains or types in a rule, one can also refer to a set of domains or types via an attribute. An attribute is simply a name for a set of domains or types. Each domain or type can be associated with any number of attributes. When a rule is written that specifies an attribute name, that name is automatically expanded to the list of domains or types associated with the attribute. For example, the `domain` attribute is associated with all process domains, and the `file_type` attribute is associated with all file types.

2.5.4 Neverallow rule

This rule specifies that an allow rule must not be generated for the operation, even if it has been previously allowed. The neverallow statement is a

compiler enforced action, so the compiler checks if any allow rules have been generated in the policy source, if so it will issue a warning and stop.

For example:

```
neverallow { domain -mmap_low_domain_type } self:memprotect
                                mmap_zero;
```

States that no allow rule may ever grant mmap_zero permissions to any type associated to the domain attribute except those associated to the `mmap_low_domain_type` attribute (as these have been excluded by the negative operator (-).

2.5.5 Type transition

The `type_transition` statement is used to transition a process from one domain to another (a domain transition) or transition an object from one type to another (an object transition).

These transitions can also be achieved using the *libselinux API* functions for SELinux-aware applications.

2.5.5.1 Domain transition

A *domain transition* occurs when a process in one domain starts a new process under a different security context. There are two ways a process can define a domain transition:

- Using a `type_transition` statement, where the exec system call will automatically perform a domain transition. This is the most common method and it is done with the following statement:

```
type_transition unconfined_t secure_services_exec_t :
                        process ext_gateway_t;
```

- SELinux-aware applications can specify the domain of the new process using the libselinux API call setexeccon. To achieve this the SELinux-aware application must also have the setexec permission, for example:

```
allow crond_t self : process setexec;
```

However, before any domain transition can take place the policy must specify that:

- The source domain has permission to transition into the target domain.

```
allow unconfined_t ext_gateway_t : process
   transition;
```

- The application binary file needs to be executable in the source domain.

```
allow unconfined_t secure_services_exec_t : file
   { execute read getattr };
```

- The application binary file needs an entry point into the target domain.

```
allow ext_gateway_t secure_services_exec_t : file
    entrypoint;
```

These are shown in Figure 2.5 where `unconfined_t` forks a child process, that then exec's the new program into a new domain called `ext_gateway_t`. Note that because the `type_transition` statement is being used, the transition is automatically carried out by the SELinux enabled kernel.

2.5.6 Object transition

An *object transition* is where a new object requires a different label to that of its parent (e.g. a file creation). This can be achieved automatically using a type_transition statement as follows:

`type_transition ext_gateway_t in_queue_t:file in_file_t;`

This `type_transition` statement states that when a process running in the `ext_gateway_t` domain (the source domain) wants to create a file object in the directory that is labeled `in_queue_t`, the file should be relabeled `in_file_t` if allowed by the policy. However, to create the file, the following minimum permissions need to be granted:

- The source domain needs permission to add file entries into the directory:

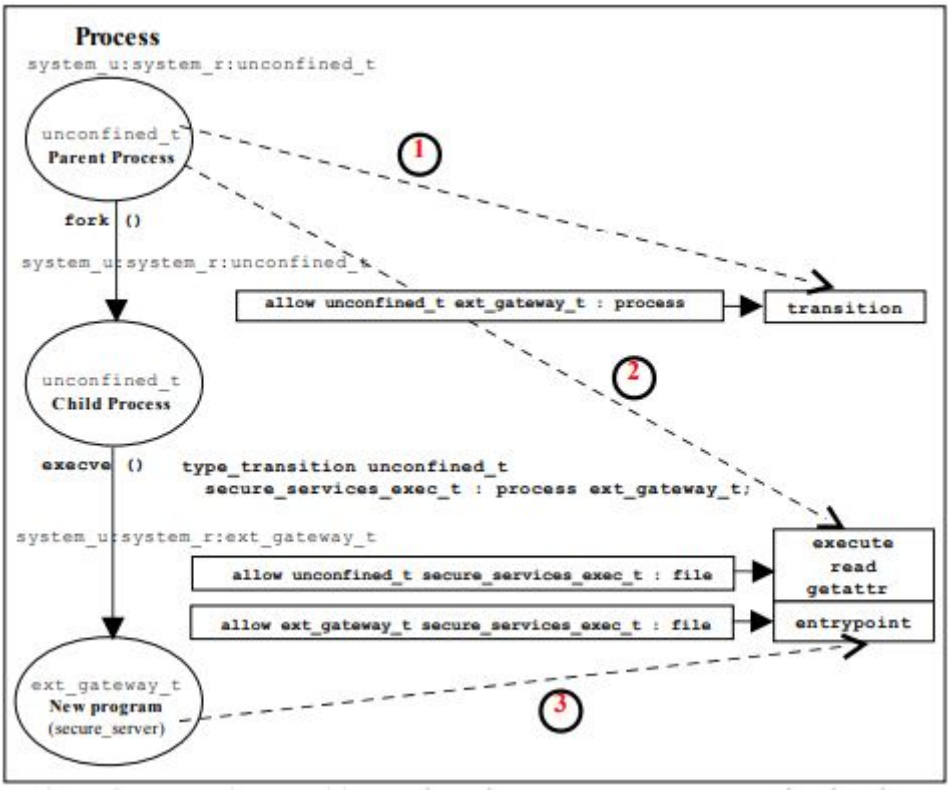

Figure 2.5: Domain Transition: the secure_server is executed within the `unconfined_t` domain and then transitioned to the `ext_gateway_t` domain.

```
allow ext_gateway_t in_queue_t : dir { write
   search add_name };
```

- The source domain needs permission to create file entries:

```
allow ext_gateway_t in_file_t : file { write
   create getattr };
```

- The policy can then ensure (via the SELinux kernel services) that files created in the in_queue are relabeled:

```
type_transition ext_gateway_t in_queue_t : file
   in_file_t;
```

2.6 Bounds

Because multiple threads share the same memory segment, SELinux was unable to check the information flows between these different threads when using setcon in pre 2.6.28 kernels. This meant that if a thread (the parent) should launch another thread (a child) with a different security context, SELinux could not enforce the different permissions.

To resolve this issue the typebounds statement was introduced with kernel support in 2.6.28 that stops a child thread (the '*bounded domain*') having greater privileges than the parent thread (the '*bounding domain*') thus ensuring the child domain will always have equal or less privileges than the parent.

The official SELinux Documentation [4] defines the rule as:

'The typebounds rule was added in version 24 of the policy. This defines a hierarchical relationship between domains where the bounded domain cannot have more permissions than its bounding domain (the parent).'

This definition does not provide a full formalization. For example it is not immediately clear what should be checked when both the source type and the target type are typebounded as stated at [14].

It is useful to emphasize that:

- a type can have at least zero and at most one bounding (parent) type;
- the typebounds rule does not automatically assign any authorization to the bounded type; it just sets the upper bound of the bounded type privileges to those held by the bounding type;
- an exception is raised at runtime if the bounded type make use of an allowed privilege that violates the bounding type.

A usage example is the following typebounds statement and allow rules:

```
typebounds httpd_t httpd_child_t;
allow httpd_t etc_t : file { getattr read };
allow httpd_child_t etc_t : file { read write };
```

State that the parent domain (`httpd_t`) has getattr and read permissions over file and the child domain (`httpd_child_t`) has read and write permissions

over file, but at run-time, this would not be allowed by the kernel because the parent does not have the write permission.

When setcon is used to set a different context on a new thread without an associated typebounds policy statement, then the call will return 'Operation not permitted' and an `SELINUX_ERR` entry will be added to the audit log stating '`op=security_bounded_transition result=denied`' with the old and new context strings.

Should there be a valid typebounds policy statement and the child domain exercises a privilege the parent domain does not have, the operation will be denied and an `SELINUX_ERR` entry will be added to the audit log stating '`op=security_compute_av reason=bounds`' with the context strings and the denied class and permissions.

It is also possible to enforce bounds on user and role, but this can be done only at compile time during policy validation using CIL.

Chapter 3

Security Enhancements for Android

As part of the Android security model, Android uses Security-Enhanced Linux (SELinux) to enforce mandatory access control (MAC) over all processes, even processes running with root/superuser privileges (Linux capabilities). With SELinux, Android can better protect and confine system services, control access to application data and system logs, reduce the effects of malicious software, limit the potential damage of compromised machines and accounts, and protect users from potential flaws in code on mobile devices.

After a brief summary of the improvements done over the years on Android's SELinux implementation, this chapter will focus on the *Android 8.1* release, the latest platform version supported by the *Nexus 6P*, the device used to develop the thesis.

3.1 Updates over the years

The Android security model is based in part on the concept of application sandboxes. Each application runs in its own sandbox.

Prior to Android 4.3, these sandboxes were defined by the creation of a unique Linux UID[1] for each application at time of installation.

In Android 4.3 and higher, SELinux provides a mandatory access control (MAC) umbrella over traditional discretionary access control (DAC) environ-

[1]User identifier

ments as descripted in [15].

In Android 5.0 and later, SELinux is fully enforced, building on the permissive release of Android 4.3 and the partial enforcement of Android 4.4. With this change, Android shifted from enforcement on a limited set of crucial domains (*installd, netd, vold* and *zygote*) to everything (more than 60 domains) providing also a separation between the system and the apps. However, all third-party apps ran within the same SELinux context, so inter-app isolation was primarily enforced by UID DAC.

Android 6.0 hardened the system by reducing the permissiveness of the policy to include better isolation between users, IOCTL[2] filtering, reduced threat of exposed services, further tightening of SELinux domains, and extremely limited `/proc` access.

Android 7.0 updated SELinux configuration to further lock down the application sandbox and reduce attack surface. This release also broke up the monolithic mediaserver stack into smaller processes to reduce the scope of their permissions.

Android 8.0 updated SELinux to work with *Treble*, which separates the lower-level vendor code from the Android system framework. This release updated SELinux policy to allow device manufacturers to update their parts of the policy, build their images, then update those images independent of the platform or vice versa.

3.2 SELinux modes of operation

SELinux can operate in two global modes:

- *Permissive* mode, in which permission denials are logged but not enforced.
- *Enforcing* mode, in which permissions denials are both logged and enforced.

To check in which one our device is running as use the command getenforce.

Android includes SELinux in enforcing mode and a corresponding security policy that works by default across AOSP[3]. In enforcing mode, disallowed

[2]Input/Output ConTroL a system call for device-specific input/output operations
[3]Android Open Source Project https://source.android.com/

actions are prevented and all attempted violations are logged by the kernel to dmesg and logcat. When developing, these errors are the key to refine your software and SELinux policies before enforcing them.

SELinux also supports a *per-domain permissive* mode in which specific domains (processes) can be made permissive while placing the rest of the system in global enforcing mode. A domain is simply a label identifying a process or set of processes in the security policy, where all processes labeled with the same domain are treated identically by the security policy. Per-domain permissive mode enables incremental application of SELinux to an ever-increasing portion of the system and policy development for new services (while keeping the rest of the system enforcing).

To determine the SELinux mode for each domain, you must examine the corresponding files or run the latest version of sepolicy-analyze with the appropriate (-p) flag.

3.2.1 Permissive mode and policy validation

When a device is in permissive mode, denials are logged but not enforced. Permissive mode is important for two reasons:

- Permissive mode ensures that policy bringup does not delay other early device bringup tasks.
- An enforced denial may mask other denials. For example, file access typically entails a directory search, file open, then file read. In enforcing mode, only the directory search denial would occur. Permissive mode ensures all denials are seen.

The simplest way to put a device into permissive mode is using the kernel command line, build a new bootimage, and flash it on the device.

But SELinux enforcement can also be disabled via ADB[4] on userdebug or eng builds. To do so, first switch ADB to root by running adb root and then, disable SELinux enforcement, with: adb shell setenforce 0

SELinux policy related log messages contain avc: and so may easily be found with grep. It is possible to capture the ongoing denial logs with logcat or to capture denial logs from the current boot sequence with dmesg.

[4]Android Debug Bridge

With this output, policy writers can readily identify when system users or components are in violation of SELinux policy. We can then repair this bad behavior, either by changes to the software, SELinux policy, or both.

Specifically, these log messages indicate what processes would fail under enforcing mode and why. Here is an example:

```
avc: denied { connectto } for pid=2671 comm="ping"
    path="/dev/socket/dnsproxyd" scontext=u:r:shell:s0
    tcontext=u:r:netd:s0 tclass=unix_stream_socket
```

Interpret this output like so:

- The `{ connectto }` above represents the action being taken. Together with the tclass at the end (`unix_stream_socket`), it tells you roughly what was being done to what. In this case, something was trying to connect to a unix stream socket.
- The `scontext` (`u:r:shell:s0`) tells you what context initiated the action. In this case this is something running as the shell.
- The `tcontext` (`u:r:netd:s0`) tells you the context of the action's target. In this case, that's a unix_stream_socket owned by netd.
- The `comm="ping"` at the top gives you an additional hint about what was being run at the time the denial was generated. In this case, it's a pretty good hint.

After addressing the majority of denials, move back into enforcing mode and address bugs as they come in. Domains still producing denials or services still under heavy development can be temporarily put into permissive mode, but move them back to enforcing mode as soon as possible.

3.3 SELinux Policy

In Android 8.0, the SELinux policy is split into *platform* and *non-platform* components to allow independent platform and non-platform policy updates while maintaining compatibility. The platform policy is further split into *platform private* and *platform public* parts to export specific types and attributes

to non-platform policy writers. In order to bind platform and non-platform extensive use of attributes and versioning is required and as a result of this step platform private mapping are created.

3.3.1 Compatibility attributes

SELinux policy is an interaction between source and target types for specific object classes and permissions. Every object (processes, files, etc.) affected by SELinux policy may have only one type, but that type may have multiple attributes.

Policy is written mostly in terms of existing types:

```
allow source_type target_type:target_class permission(s);
```

This works because the policy is written with knowledge of all types. However, if the vendor policy and platform policy use specific types, and the label of a specific object changes in only one of those policies, the other may contain policy that gained or lost access previously relied upon. For example:

```
File_contexts:
/sys/A    u:object_r:sysfs:s0
Platform: allow p_domain sysfs:class perm;
Vendor: allow v_domain sysfs:class perm;
```

Could be changed to:

```
File_contexts:
/sys/A    u:object_r:sysfs_A:s0
```

Although the vendor policy would remain the same, the `v_domain` would lose access due to the lack of policy for the new `sysfs_A` type.

By defining a policy in terms of attributes, a type is given to the underlying object such that it has an attribute corresponding to policy for both the platform and vendor code. This done for all types can effectively create an attribute-policy wherein concrete types are never used. In practice, this is required only for the portions of policy that overlap between platform and vendor, which are defined and provided as *platform public policy* that gets built as part of the vendor policy.

Defining public policy as *versioned attributes* satisfies two policy compatibility goals:

- Ensure vendor code continues to work after platform update. Achieved by adding attributes to concrete types for objects corresponding to those on which vendor code relied, preserving access.
- Ability to deprecate policy. Achieved by clearly delineating policy sets into attributes that can be removed as soon as the version to which they correspond no longer is supported. Development can continue in the platform, knowing the old policy is still present in the vendor policy and will be automatically removed when/if it upgrades.

3.3.2 Policy versioning

To meet the goal of not requiring knowledge of specific version changes for policy development, Android 8.0 includes a mapping between platform-public policy types and their attributes.

Such that, type `foo` is mapped to attribute `foo_vN`, where `N` is the version targeted. `vN` corresponds to the `PLATFORM_SEPOLICY_VERSION` build variable and is of the form `MM.NN`, where `MM` corresponds to the platform SDK number and `NN` is a platform sepolicy specific version.

Attributes in public policy are not versioned, but rather exist as an API on which platform and vendor policy can build to keep the interface between the two partitions stable.

Both platform and vendor policy writers can continue to write policy as usual, in fact policy versioning is automatically taken care of during the policy building process.

As part of the build process platform-public policy exported as allow source_foo target_bar:class perm; is included as part of the vendor policy. During compilation (which includes the corresponding version) it is transformed into the policy that will go to the vendor portion of the device, that shown in the transformed *Common Intermediate Language* (CIL) will look like:

```
(allow source_foo_vN target_bar_vN (class (perm)))
```

As vendor policy is never ahead of the platform, it should not be concerned with prior versions. However, platform policy will need to know how far back vendor policy is, include attributes to its types, and set policy corresponding to versioned attributes.

3.3.3 Policy mapping

Automatically creating attributes by adding `_vN` to the end of each type does nothing without mapping of attributes to types across version differences. When using attributes to map to policy versions, a type maps to an attribute or multiple attributes, ensuring objects labeled with the type are accessible via attributes corresponding to their previous types. This is done in the aforementioned mapping files with statements.

Maintaining a goal to hide version information from the policy writer means automatically generating the versioned attributes and assigning them to the appropriate types. In the common case of static types, this is straightforward: `type_foo` maps to `type_foo_v1`.

For more complex changes like collapsing types, creating this mapping is non-trivial. Platform policy maintainers must determine how to create the mapping at transition points for objects, which requires understanding the relationship between objects and their assigned labels and determining when this occurs. For backwards compatibility, this complexity needs to be managed on the platform side, which is the only partition that may be ahead in terms of policy revision.

3.3.4 Platform public policy

The *platform-public policy* is the core of conforming to the Android 8.0 architecture model without simply maintaining the union of platform policies from v1 and v2.

The platform public sepolicy includes everything defined under system/sepolicy/public. The platform can assume the types and attributes defined under public policy are stable APIs for a given platform version. This forms the part of the sepolicy that is exported by platform on which non-platform (i.e. device) policy developers may write additional device-specific policy.

Types are versioned according to the version of the policy that non-platform files are written against, defined by the `PLATFORM_SEPOLICY_VERSION` build variable.

3.3.5 Platform private policy

The *platform private sepolicy* includes everything defined under `system/sepolicy/private`. This part of the policy forms platform-only types, permissions, and attributes required for platform functionality. These are not exported to the vendor or device policy writers. Moreover, these rules are allowed to be modified or may disappear as part of a framework-only update.

3.3.6 Platform private mapping

The *platform private mapping* includes policy statements that map the attributes exposed in platform public policy of the previous platform versions to the concrete types used in current platform public sepolicy. This ensures non-platform policy written on platform public attributes from previous platform public sepolicy version(s) continues to work. The versioning is based on the `PLATFORM_SEPOLICY_VERSION` build variable set in AOSP for a given platform version. A separate mapping file exists for each previous platform version from which this platform is expected to accept vendor policy.

3.3.7 Non-platform policy

The combination of platform-public policy and vendor policy satisfies the Android 8.0 architecture model goal of allowing independent platform and vendor builds.

Vendors are exposed to a subset of platform policy that contains useable types and attributes and rules on those types and attributes which then becomes part of vendor policy. SoC vendor and ODM rely on that to write device-specific policy on top of the stable APIs the public policy delivers in order to use SELinux security context on newly defined services.

Device-specific customization is done through the `BOARD_SEPOLICY_DIRS` variable defined in device-specific Boardconfig.mk file. This global build variable contains a list of directories that specify the order in which to search for additional policy files.

For example, a SoC vendor and an ODM might each add a directory, one for the SoC-specific settings and another for device-specific settings, to generate the final SELinux configurations for a given device:

```
BOARD_SEPOLICY_DIRS += device/SOC/common/sepolicy
BOARD_SEPOLICY_DIRS += device/SoC/DEVICE/sepolicy
```

3.4 SELinux contexts labeling

This section details the Android specific policy configuration files. Together these files are used to assign security contexts to all subjects and objects of the system.

With the introduction of *Treble* all these contexts files have been split into two files:

- a *platform* file, that has no device-specific labels and generally resides in the system partition
- a *non-platform* file, built by combining contexts file found in the directories pointed to by BOARD_SEPOLICY_DIRS in the device's Boardconfig.mk files and resides in the vendor partition

3.4.1 File and genfs contexts

`file_contexts` assigns security contexts to filesystems that support extended attributes and genfs_contexts to filesystems, such as proc or vfat that do not.

`file_contexts` is consulted during the Android building process to generate a system image with a correct assignment of security labels, at the init stage to correctly label dev, sys, and data filesystems, but also at runtime, whenever a system process, that has the permission to do so, requires a restorecon on a file or a directory.

`file_contexts` entries comes in the form:

```
pathname_regexp file_type security_context
```

Where:

- `pathname_regexp` defines the portion of the filesystem the entry is referred to. It can be a specific path or a regular expression to be resolved in order to understand the paths the entry refers to

- `file_type` is the category of filesystem entity we are considering within the entry (i.e directory, file, etc)
- `security_context` is the SELinux security context used to label the files

`genfs_contexts` configuration is loaded as part of the kernel policy and each entry is a genfscon statement that specifies the filesystem name, a partial path and the security context allocated to the filesystem.

3.4.2 Service and property contexts

`service_contexts` assigns labels to Android binder services to control what processes can add (register) and find (lookup) a binder reference for the service. This configuration is read by the servicemanager process during startup.

`property_contexts` assigns labels to Android system properties to control what processes can set them. This configuration is read by the init process during startup.

Both configuration files are extremely straightforward, all it takes for each entry is the identifier of the property or service and its associated `security_context`.

3.4.3 Seapp contexts

This configuration file was introduced specifically for SE Android startup and allows the specification of how to label app processes and `/data/data` directories.

`seapp_contexts` is read by the zygote process on each app launch and by installd during startup.

The entries in this file define how security contexts for apps are determined. Each entry lists input selectors, used to match the app, and outputs which are used to determine the security contexts for matching apps.

There are three types of input selectors:

- boolean: `isSystemServer` (defaults to false), `isEphemeralApp`, `isV2App`, `isOwner`, `isPrivApp`
- string: `user`, `seinfo`, `name`, `path`

- unsigned integer: `minTargetSdkVersion` (defaults to 0)

All specified input selectors in an entry must match and an unspecified string or boolean selector with no default matches any value.

`user` is a selector based upon the UID, `_app` matches any regular app UID, `_isolated` matches any isolated service UID and other values of user are matched against the name associated with the process UID.

`seinfo` matches against the seinfo tag for the app, determined from `mac_permissions.xml` files, name matches against the package name of the app and path matches against the directory path when labeling app directories.

`minTargetSdkVersion` matches applications with a targetSdkVersion greater than or equal to the specified value.

When processes load this configuration file, entries are sorted based on selectivity. Apps are then checked against the entries and the first matching entries will be the one deciding the outputs which are used to determine the security contexts.

Outputs:

- `domain`: determines the label to be used for the app process, entries without a specified domain are ignored for this purpose.
- `type`: specifies the label to be used for the app data directory; entries that don't have a type are ignored for this purpose.
- `levelFrom` and `level`: are used to determine the level (sensitivity and categories) for MLS/MCS.

`levelFrom` selects how the security level is computed and can assume one of the following values:

- none: omits the level (use the one specified by the level keyword)
- app: determines the level from the process UID
- user: determines the level from the user ID.
- all: determines the level from both UID and user ID.

Examples of `seapp_contexts` entries extracted from the default SEAndroid `seapp_contexts` file are:

```
user=_app isV2App=true isEphemeralApp=true domain=
   ephemeral_app type=app_data_file levelFrom=user
user=_app isPrivApp=true domain=priv_app type=
   app_data_file levelFrom=user
user=_app minTargetSdkVersion=26 domain=untrusted_app
    type=app_data_file levelFrom=user
user=_app domain=untrusted_app_25 type=app_data_file
   levelFrom=use
```

3.4.4 MAC permissions

`mac_permissions.xml` assigns a seinfo tag to apps based on their *signature* and optionally their package name. The seinfo tag can then be used as a key in the `seapp_contexts` file to assign a specific label to all apps with that seinfo tag. the middleware MAC policy configuration is read by `system_server` during startup.

A *signature* is a hex encoded X.509 certificate or a tag defined in keys.conf and is required for each signer tag. The signature can either appear as a set of attached cert child tags or as an attribute.

Each signer or package tag is allowed to contain one seinfo tag. This tag represents additional info that each app can use in setting a SELinux security context on the eventual process as well as the apps data directory.

seinfo assignments are made according to the following rules:

- Stanzas with package name refinements will be checked first.
- Stanzas without package name refinements will be checked second.
- The "default" seinfo label is automatically applied to no matching apps.

Examples of valid stanzas are:

- a single certificate protecting seinfo:

```
<signer signature="@PLATFORM" >
    <seinfo value="platform" />
</signer>
```

- multiple certs protecting explicitly named app (all certs must match)

```
<signer>
    <cert signature="@PLATFORM1"/>
    <cert signature="@PLATFORM2"/>
    <package name="com.android.foo">
        <seinfo value="bar" />
    </package>
</signer>
```

3.5 Building the Policy

SELinux policy is built from the combination of core AOSP policy (platform) and device-specific policy (vendor).

The SELinux policy build flow for Android 4.4 through Android 7.0 merged all sepolicy fragments then generated monolithic files in the root directory. This meant that SoC[5] vendors and ODM[6] manufacturers modified boot.img (for non-A/B devices) or system.img (for A/B devices) every time policy was modified.

In Android 8.0 and higher, platform and vendor policy can build separately, so that SOCs and OEMs[7] can update their parts of the policy, build their images (vendor.img, boot.img, etc.) and then update those images independent of platform updates.

However, as modularized SELinux policy files are stored on `/system` and `/vendor` partitions, the init process must mount the system and vendor partitions earlier so it can read SELinux files from those partitions and merge them with core SELinux files in the system directory (before loading them into the kernel).

Policy exists in the following locations:

[5]System on a Chip
[6]Original Design Manufacturer
[7]Original Equipment Manufacturer

Location	Contains
`system/sepolicy/public`	The platform's sepolicy API
`system/sepolicy/private`	Platform implementation details
`BOARD_SEPOLICY_DIRS`	Vendor sepolicy

Table 3.1: AOSP policy directories

The build system takes these policies and depending on the value of `PRODUCT_FULL_TREBLE` produces either a monolithic policy or a split policy between the system and vendor partitions.

This conditional build logic closely mimics the logic with which the init process loads SELinux policy files.

So, for compatibility reasons device manufacturers can decide whether, to support Treble and its clear separation between platform and vendor, or to stick to the older way and built it monolithically.

The steps of the build system in both cases, start with:

1. Converting policies to the SELinux *Common Intermediate Language* (CIL) format, specifically:

2. *Versioning* the part of the vendor policy that uses the stable API defined in the public platform. This is done by using the produced public CIL policy to inform the combined public, vendor and BOARD_SEPOLICY_DIRS policy as to which parts must be turned into attributes in order to be linked to the platform policy.

3. Creating a *mapping* file linking the platform and vendor parts. Initially, this just links the types from the public policy with the corresponding attributes in the vendor policy; later it will also provide the basis for the file maintained in future platform versions, enabling compatibility with vendor policy targeting this platform version.

4. Lastly, if the device, the build system is targeting, is:

    - A *Treble* supported device, the Security-Enhanced CIL compiler (secilc) and the CIL files produced in the previous steps will be loaded on the device as-is.

- A *non-Treble* device the compilation and build of the CIL files is in charge of the build system so that a monolithic binary policy file is produced and loaded on the device.

3.5.1 Policy Build Tools

The `system/sepolicy` directory contains a number of tools related to policy, some of which are used in building and validating the policy and others are available for help in auditing and analyzing policy.

The following tools have been extremely useful to keep the policy and contexts files coherent across the many changes done during the thesis and their source codes helped develop the runtime validation explained in the Implementation section.

- `checkfc`

  A utility for checking the validity of a file_contexts or a property_contexts configuration file. Used as part of the policy build to validate both files. Requires the sepolicy file as an argument in order to check the validity of the security contexts in the file_contexts or property_contexts file.

  Usage 1:

  ```
  checkfc sepolicy file_contexts
  checkfc -p sepolicy property_contexts
  ```

  Also used to compare two file_contexts or file_contexts.bin files. Displays one of subset, equal, superset, or incomparable.

  Usage 2:

  ```
  checkfc -c file_contexts1 file_contexts2
  ```

  Example:

  ```
  $ checkfc -c out/target/product/shamu/system/etc/
     general_file_contexts out/target/product/shamu/
     root/file_contexts.bin subset
  ```

- `checkseapp`

  A utility for merging together the main seapp_contexts configuration and the device-specific one, and simultaneously checking the validity of the configurations. Used as part of the policy build process to merge and validate the configuration.

  Usage:

  ```
  checkseapp -p sepolicy input_seapp_contexts0 [
     input_seapp_contexts1...] -o seapp_contexts
  ```

- `post_process_mac_perms`

  A tool to help modify an existing mac_permissions.xml with additional app certs not already found in that policy. This becomes useful when a directory containing apps is searched and the certs from those apps are added to the policy not already explicitly listed.

  Usage:

  ```
  post_process_mac_perms [-h] -s SEINFO -d DIR -f
     POLICY
  -s SEINFO, --seinfo SEINFO seinfo tag for each
     generated stanza
  -d DIR, --dir DIR Directory to search for apks
  -f POLICY, --file POLICY mac_permissions.xml
     policy file
  ```

- `sepolicy-check`

  A tool for auditing a sepolicy file for any allow rule that grants a given permission.

  Usage:

  ```
  sepolicy-check -s <domain> -t <type> -c <class> -
     p <permission> -P out/target/product/<board>/
     root/sepolicy
  ```

- `sepolicy-analyze`

  A tool for performing various kinds of analysis on a sepolicy file:

– type equivalence (`typecmp`)

```
sepolicy-analyze out/target/product/<board>/
   root/sepolicy typecmp -e
```

Display all type pairs that are "equivalent", i.e. they are identical with respect to allow rules, including indirect allow rules via attributes and default-enabled conditional rules (i.e. default boolean values yield a true conditional expression).

Equivalent types are candidates for being coalesced into a single type.

– duplicate allow rules (`dups`)

```
sepolicy-analyze out/target/product/<board>/
   root/sepolicy dups
```

Displays duplicate allow rules, i.e. pairs of allow rules that grant the same permissions where one allow rule is written directly in terms of individual types and the other is written in terms of attributes associated with those same types. The rule with individual types is a candidate for removal.

– attribute (`attribute`)

```
sepolicy-analyze out/target/product/<board>/
   root/sepolicy attribute <name>
```

Displays the types associated with the specified attribute name.

```
sepolicy-analyze out/target/product/<board>/
   root/sepolicy attribute -r <name>
```

Displays the attributes associated with the specified type name.

– neverallow checking (`neverallow`)

```
sepolicy-analyze out/target/product/<board>/
   root/sepolicy neverallow \ [-w] [-d] [-f
   neverallows.conf] | [-n "neverallow string
   "]
```

Check whether the sepolicy file violates any of the neverallow rules from the neverallows.conf file or a given string. You can use an entire policy.conf file as the neverallows.conf file and sepolicy-analyze will ignore everything except for the neverallows within it. If there are no violations, sepolicy-analyze will exit successfully with no output. Otherwise, sepolicy-analyze will report all violations and exit with a non-zero exit status.

Chapter 4

Mandatory Access Control for Third-Party Apps

The wide deployment of mobile operating systems has introduced a number of challenging security requirements.

On one hand, mobile devices are high-value targets, since they offer a direct financial incentive in the use of the credit that can be associated with the device or in the abuse of the available payment services (e.g., Google Wallet, telephone credit and mobile banking) [8]. In addition, mobile devices permit the recovery of large collections of personal information and are the target of choice if an adversary wants to monitor the location and behavior of an individual.

On the other hand, the system presents a high exposure, with users continuously adding new apps to their devices, to support a large variety of functions. The risks are then greater and different from those of classical operating systems [12]. The frequent installation of external code creates an important threat. The design of security solutions for mobile operating systems has to consider a careful balance between, on one side, the need for users to easily extend with unpredictable apps the set of functions of the system and, on the other side, the need for the system to be protected from potentially malicious apps.

It is to note that the greatest threats derive from apps that are offered through delivery channels that are alternative to the "official" app markets (e.g. [16]), whose number of app installations is increasing rapidly, pointing out the need of wider security layers. Apps in official markets, instead, are ver-

ified by the market owner and the ones detected as misbehaving are promptly removed from the market. The correct management of the app market is crucial, nevertheless it is not able by itself to fully mitigate the security concerns. The mobile operating systems have to provide a line of defense internal to the device against apps that, due to malicious intent or the presence of flaws in system components or other apps, may let an adversary abuse the system.

As explained in the previous chapter the SEAndroid initiative [15] has led to a significant extension of the security services, with the integration of Security Enhanced Linux (SELinux) into the Android operating system. The goal of SEAndroid is to build a mandatory access control (MAC) model in Android using SELinux to enforce kernel-level MAC, introducing a set of middleware MAC extensions to the *Android Permission Framework*. The middleware MAC extension chosen to bridge the gap between SELinux and the Android permission framework is called *install-time MAC* [15] . This mechanism allows to check an app against a MAC policy (i.e. mac permissions.xml).

The integration of this middleware MAC ensures that the policy checks are unbypassable and always applied when apps are installed and when they are loaded during system startup. The current design of SEAndroid aims at protecting core system resources from possible flaws in the implementation of security in the *Android Permission Framework* or at the DAC level. The exploitation of vulnerabilities becomes harder due to the constraints on privilege escalation that are introduced by SELinux. Unfortunately, the current use of SELinux in Android aims at protecting the system components and trusted apps from abuses by third-party apps. All the third-party apps fall within a single *untrusted app* domain and an app interested in getting protection from other apps or from internal vulnerabilities can only rely on Android permissions and the Linux DAC support. This is a significant limitation, since apps can get a concrete benefit from the specification of their own policy.

4.1 Threat model

In Android each app receives a dedicated UID and GID[1] at install-time. These identifiers are used to set the user and the group owner of the resources

[1]Group IDentifier

installed by the app in the default data directory. Since by default, the apps databases, settings and general data, as well as, user data resides there, it is important that only that application has access to that particular folder. This confinement of the data folders permits to enforce a strict isolation from other applications. In Android this isolation is only enforced at DAC level, but this is not enough to protect the app and its own resources by other apps with root privileges. Android, by default, comes with a restricted set of permissions for its user and the installed applications (i.e. no root privileges). Despite this, apps can gain root privileges in two ways:

- with the user intervention which is the case for benign apps that require root privileges to accomplish their job. For example, *Titanium Backup* [6] needs root privileges to backup system and user applications along with their data. In this scenario, the user typically flashes a recovery console on the device which has the permission to write on the system partition and from there she installs an app such as *SuperSU* or *Superuser* in order to gain and manage root privileges. After that, the user can give root privileges to other applications. According to Google, users install non-malicious rooting apps by a ratio of 671 per million in 2014 (increased by 38% compared to the 491 per million in 2013 [11]). Moreover, there are successful community ROMs, such as CyanogenMod with over 10 million installations, that provide root access to the user by default.

  Here, the user is aware of the fact that some apps act as root in the system and have access to everything, however she does not know how these privileges are used and she has to trust the app.

- exploiting a bug in a system component a malware could gain root privileges to freely access the whole system in order to steal personal information or perform fraudulent actions. In this scenario the user is unaware of the fact that an app acts as root. Over the years Android has been attacked by threatening malware apps such as *DroidDreamLight*, which affected 30,000-120,000 users in May 2011 [13] and *towelroot* that exploited the `CVE-2014-3153` bug of Linux kernel, to "root the device" without the need to flash a recovery console, and give root privileges potentially to all apps.

The following methodology provides a solution to both scenarios through the use of *appPolicyModules* [10] defined by the app and attached to the SELinux system policy.

4.2 Requirements

Analyzing the introduction of *appPolicyModules* in the management of per-app security, we need to consider the different cases that emerge from the combination of the system policy and an appPolicyModule. From what presented in the SELinux chapter, we note that every allow rule defined in a policy has a source and a target. These types may be defined in either the system policy or the appPolicyModule. We then have four types of allow rule, depending on the origin of the source and target domains. Each configuration is associated with a specific requirement that must be satisfied by appPolicyModules.

- *No impact on the system policy*: the app must not change the system policy and can only impact on processes and resources associated with the app itself. An appPolicyModule is intended to extend the system policy and to be managed by the same software modules that manage the system policy. Since third-party apps can not be trusted a priori, it is imperative that the provided appPolicyModule must not be able to have an impact on privileges where source and target are system types.

  Without this restriction the appPolicyModule could provide `untrusted_app` write access to the type `system_data_file` and corrupt the system resources, or enhance the privileges of system resources that the app can access, creating unpredictable vulnerabilities.

- *No escalation*: the app cannot specify a policy that provides to its types more privileges than those available to untrusted app. New domains declared in an appPolicyModule must always operate within the boundaries defined by the system policy as acceptable for the execution of apps. When a new application is installed, its domain has to be created "under" the `untrusted_app` domain, so the system policy can flexibly define the maximum allowed privileges for third-party apps.

- *Flexible internal structure*: apps may provide many functionalities and use different services (e.g., geolocalization, social networks). The appPolicyModule has to provide the flexibility of defining multiple domains with different privileges so that the app, according to the functionality in use, may switch to the one that represents the "*least privilege*" domain needed to accomplish the job, in order to limit potential vulnerabilities deriving from internal flaws. Greater flexibility derives from the possibility to freely manage privileges for internal types over internal resources, building a MAC model that remains completely under the control of the app.

  To enhance app security and protect the user even from possible app flaws, the appPolicyModule could specify a switch of context (i.e., it may change the SELinux domain associated with its process) to satisfy the "*least privilege*" principal on different operating modes.

- *Protection from external threats*: users of mobile devices may unconsciously install malware apps from untrusted sources that, exploiting some security vulnerabilities, could compromise the entire system or other apps (e.g., steal user information). To mitigate the risk, an appPolicyModule should provide a common way to isolate the app's critical resources. The use of MAC support of- fers protection even against threats coming from the system itself, like a malicious app that abuses root privileges. The app can protect its resources from other apps, specifying its own types and defining in a flexible way which system components may or may not access its domains. This requirement depends on the ability of the MAC model implemented by SELinux to let app types be protected against system-level elements which assume a rigid hierarchical structure. Indeed, in the SELinux policy model every privilege has to be explicitly authorized and new types are not accessible by system types unless a dedicated rule is introduced in the appPolicyModule.

4.3 Policy module language

We now present the concrete structure of appPolicyModules. A critical design requirement is the compatibility with the SELinux implementation avail-

able today, which facilitates the adoption of the proposed approach. For this purpose we won't use the kernel policy language, but the CIL policy language, in fact as said in Section 3.5 only CIL policy files are installed on the device.

The appPolicyModule can make use of the following CIL statements: *typebounds, type, typeattribute, typeattributeset, allow, neverallow*, and *typetransition*. These statements are the only ones that can be used in the definition of the appPolicyModule. The syntax for these statements is succinctly presented in Table 4.1.

Statement	Syntax
block	`(block block_id cil_statement ...)`
type	`(type type_id)`
typeattribute	`(typeattribute typeattribute_id)`
typeattributeset	`(typeattributeset typeattribute_id (type_id ... | expr ...))`
typebounds	`(typebounds parent_type_id child_type_id)`
typetransition	`(typetransition source_type_id target_type_id class_id [object_name] default_type_id)`
allow	`(allow source_id target_id|self classpermissionset_id ...)`
neverallow	`(neverallow source_id target_id|self classpermissionset_id ...)`

Table 4.1: Simplified CIL syntax [1] used in APMs.

The *typebounds* statement permits to specify that the collection of privileges of the bounded has to fall within the boundaries of another. The evaluation of compatibility takes into account the presence of other *typebounds* statements in the target, considering as correct the use in the target of a type that is bounded by the type appearing in the higher-level rule (as explained in [14]).

It is useful to emphasize that the *typebounds* statement does not assign the authorizations to the bounded domain, it only sets its upper bound. This is a core principle in our scenario, where policy writers are outside of the trust domain of the core system resources.

The *type* statement permits to introduce a new type.

The *typeattribute* statement declares an identifier that can be used to define rules. SELinux policies make extensive use of attributes to provide a structure

to policies. No constraint needs to be introduced on the definition of new attributes.

To avoid name conflicts between types or typeattributes defined in different modules, the module must be wrapped in a *block* statement.

The *block* statement creates a policy namespace, so that any new type or typeattribute definition will be globally referred as `block_id.type_id`. To guarantee unicity of the `namespace_id` the package name of the app is enforced.

The *allow* statement is used to allow permission to the `source_id`, as such we need to check that `source_id` has nothing to do with the system policy. If `source_id` is a type it has to be newly defined in this module and bounded to `untrusted_app`, if it is a typeattribute each type member has to be newly defined in this module and bounded to `untrusted_app`.

The *neverallow* statement restricts what an allow statement can permit in the policy, if violated a compilation error occurs.

Attributes produce an effect on the policy when they are used in the *typeattributeset* statement, which has been presented above.

The *typeattributeset* statement may function as an undirect way to give permissions to types, in fact if the `typeattribute_id` is defined by the system policy permissions are assigned to it and 'entering' that set will transfer those permission to the `type_id`. Since `type_id` got permissions, we need to bound it to `untrusted_app`.

Finally, the *typetransition* statement permits to describe the admissible transitions between types at runtime. We introduce the constraint, that the type defined as first parameter has to be a type defined in the module. The type transition statement is used to perform object and domain transitions.

- An object transition occurs when an object needs to be relabeled (i.e., a file label is changed).

- A domain transition occurs when a process with one type (we call it transition-startpoint) switches to another type (we call it transition-endpoint), enacting different authorizations from the original ones. An app could define different domains with limited authorization and use them when performing specific actions.

4.4 Correctness

We want to show that the appPolicyModules will satisfy the four requirements described in Section 4.2.

With respect to *No impact on the system policy*, we note that the appPolicyModule statements do not have an impact on the system policy, because all the allow and neverallow statements have to specify as source a new type, guaranteed to be outside of the system policy.

The correctness with respect to *No escalation* is guaranteed through the use of the typebounds statements associated with all the type that appear as source in the allow statements or get 'indirect' permissions through typeattributeset statement. It is to note that the neverallow rules do not have to be considered here, because they may only cause the rejection of the appPolicyModule by the compiler, but they cannot lead to the escalation of privileges for the new type. The consideration by the compiler of the typebounds statements indeed verifies that each allow rule in the appPolicyModule that refers to system types has a corresponding allow rule associated with `untrusted_app`.

Flexible internal structure is satisfied by defining at least two new domain types associated with distinct privileges and permissions to transition from one to the other.

Protection from external threats is supported, in fact, without an explicit rule giving permission, a process associated with `untrusted_app` is not authorized to access files associated with a newly defined type.

Chapter 5

Implementation

This chapter explains step-by-step the changes done to the *Android Open Source Project (AOSP)* to implement the *appPolicyModule* methodology explained in chapter 4, and some peculiar modification required to make it work on a *Nexus 6P* device which is our target test device.

5.1 Setting up the build server

As building the Android OS is time consuming and requires quite a system to be carried out successfully in a bunch of minutes, it has beed demanded to a server set up as described in this section.

The Android Open Source Project (AOSP) is traditionally developed and tested on Ubuntu Long Term Support (LTS) releases (Ubuntu 14.04 Trusty is recommended).

The Android build system normally uses ART[1], running on the build machine, to pre-compile system dex files. Since ART is able to run only on Linux, the build system skips this pre-compilation step on non-Linux operating systems, resulting in an Android build with reduced performance.

The *master* branch of Android in the AOSP comes with prebuilt versions of OpenJDK, but for older versions a separate installation of the JDK is required.

For Ubuntu LTS 14.04 there are no available supported OpenJDK 8 packages, but the Ubuntu 15.04 OpenJDK 8 packages can be used.

[1]Android RunTime

Other required packages can be installed with:

```
sudo apt-get install git-core gnupg flex bison gperf
   build-essential zip curl zlib1g-dev gcc-multilib g
   ++-multilib libc6-dev-i386 lib32ncurses5-dev
   x11proto-core-dev libx11-dev lib32z-dev libgl1-mesa
   -dev libxml2-utils xsltproc unzip
```

5.1.1 Downloading the Source

The Android source tree is located in a *Git repository* hosted by Google. The Git repository includes metadata for the Android source, including those related to changes to the source and the date they were made.

Repo is a tool that makes it easier to work with Git in the context of Android.

To install Repo:

1. Create (if not already there) a bin/ directory in the home directory and add it to path:

   ```
   mkdir ~/bin
   PATH=~/bin:$PATH
   ```

2. Download the Repo tool and ensure that it is executable:

   ```
   curl https://storage.googleapis.com/git-repo-
      downloads/repo > ~/bin/repo
   chmod a+x ~/bin/repo
   ```

After installing Repo, I created a local *mirror* on the server to maintain the global state of my AOSP tree, so that clients have a smart way to push changes onto the server and from there build the Android system. This has been very helpful also considering I was working on this from different clients and the download for a full mirror is smaller than the download of two clients, while containing more information.

The first step is to create and *sync* the mirror itself. Notice the `--mirror` flag, which can be specified only when creating a new repo client:

```
mkdir -p /mnt/data/mrossi/aosp/mirror
cd /mnt/data/mrossi/aosp/mirror
repo init -u https://android.googlesource.com/mirror/
   manifest --mirror
repo sync
```

Once the mirror is synced, new clients can be created from it. Note that one of these client is the server itself that needs a AOSP tree in order to carry out the build process.

To check out a *branch* other than "master", specify it with -b. For a list of branches, see Source Code Tags and Builds.

```
mkdir -p /mnt/data/mrossi/aosp/thesis
cd /mnt/data/mrossi/aosp/thesis
repo init -u /mnt/data/mrossi/aosp/mirror/platform/
   manifest.git -b android-8.1.0_r47
repo sync
```

We selected the Android 8.1 release, because it is the last one supported by the target device we are using for testing, which is the *Nexus 6P* device.

To distinguish in a clearer way the code I wrote from the standard release, I decided to create a new branch called thesis on the mirror:

1. Go into the local AOSP source tree and create the branch across all repositories defined in the AOSP manifest.

   ```
   cd /mnt/data/mrossi/aosp/thesis
   repo forall -c "git branch thesis; git push aosp
      thesis"
   ```

2. Go into the manifest directory, create the thesis branch and check it out

   ```
   cd /mnt/data/mrossi/aosp/thesis/repo/manifests
   git branch thesis
   git checkout thesis
   ```

3. Modify the `default.xml` manifest to use thesis as default revision and remove the reference to the Google review server (as shown in Table 5.1).

4. Commit and push the changes

Before

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <remote name="aosp"
            fetch=".."
            review="https://android-review.googlesource.
               com/" />
    <default revision="master"
             remote="aosp"
             sync-j="4" />
    ...
```

After

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
    <remote  name="aosp"
             fetch=".." />
    <default revision="thesis"
             remote="aosp"
             sync-j="4" />
    ...
```

Table 5.1: AOSP manifest - Before and after the modifications performed

```
    git commit -a -m "Changed default revision to
       thesis and removed review server."
    git push origin thesis
```

After this we can verify that everything is set correctly with:

```
cd /mnt/data/mrossi/aosp/thesis
repo init -u /mnt/data/mrossi/aosp/mirror/platform/
   manifest.git -b thesis
repo sync
```

5.1.2 Downloading the binaries

AOSP cannot be used from pure source code only and requires additional hardware-related proprietary libraries to run, such as for hardware graphics acceleration.

Official binaries for the supported devices running tagged AOSP release branches from Google's drivers can be downloaded from https://developers.google.com/android/drivers. These *binaries* add access to additional hardware capabilities with non-open source code. In our case we downloaded the Nexus 6P binaries for Android (`OPM7.181005.003`) which are the binaries associated to `android-8.1.0_r47` release.

Each set of binaries comes as a self-extracting script in a compressed archive. Uncompress each archive, run the included self-extracting script from the root of the source tree, then agree to the terms of the enclosed license. The binaries and their matching makefiles will be installed in the vendor/ hierarchy of the source tree.

To ensure the newly installed binaries are properly taken into account after being extracted, delete the existing output of any previous build using: `make clobber`

5.2 Setting up the clients

5.2.1 Downloading the Source

First install the repo client as described in Section 5.1.1, then download the Android source repository on the server:

1. Create an empty directory to hold your working files.
2. Configure git with name and email address.
3. Run repo init to bring down the latest version of Repo with all its most recent bug fixes. You must specify a URL for the manifest, which specifies where the various repositories included in the Android source will be placed within your working directory.

   ```
   repo init -u mrossi@bender:/mnt/data/mrossi/aosp/
      mirror/platform/manifest.git -b thesis
   ```

The above `repo` command uses ssh to download the thesis branch, where `mrossi` is my server user and `bender` is the server name as stored in the `ssh` configiguration file.

4. Pull down the Android source tree the working directory from the repositories, as specified in the default manifest.

   ```
   repo sync
   ```

5.2.2 IDE

IDEGen automatically generates Android IDE configurations for *IntelliJ IDEA* and *Eclipse*. For personal preference I decided to go for *Android Studio* which is based upon IntelliJ IDEA.

Android is large, thus IDEA needs a lot of memory and some settings need to be tweaked.

- Increase the initial memory and maximum memory heap size allocated by the JVM for running IntelliJ IDEA.

  Add `"-Xms1g -Xmx5g"` to the VM options in *"Help > Edit Custom VM"*

- Increase the maximum size of files (in kilobytes) for which IntelliJ IDEA provides code assistance.

  Add `"idea.max.intellisense.filesize=100000"` in *"Help > Edit custom properties"*

Create a JDK configuration named `"1.8 (No Libraries)"` by adding a new JDK and then removing all of the jar entries under the "Classpath" tab. This will ensure that you only get access to Android's core libraries and not those from your desktop VM.

Finally, we need to run

```
make idegen && development/tools/idegen/idegen.sh
```

from the source root (everytime we sync). This produces a few files including android.ipr that allows us to import sources to Android Studio or IntelliJ. At the first IDE launch sources will need to be indexed, but at the end of this quite long operation, we can easily browse, write and refactor code. If

the project is already open, hit the sync button in Android Studio, and it will automatically detect the updated configuration.
Android Studio will spot that AOSP source has been downloaded from *Git repositories* and suggest to import them as Git projects. This is very helpful since it allows to use the Android Studio built in support for Git.

To address an Android Studio bug that occurs with the AOSP source and stucks the IDE in an indexing loop the following header of the android.iml has been used.

```
<?xml version="1.0" encoding="UTF-8"?>
<module relativePaths="true" type="JAVA_MODULE"
        version="4">
  <component name="FacetManager">
    <facet type="android" name="Android">
      <configuration>
        <option name="ALLOW_USER_CONFIGURATION"
                value="false" />
      </configuration>
    </facet>
    ...
```

5.3 Building Android

As said in chapter 5.1.1 we configured a build server, so all the operations told in this section will be executed on it through `ssh`.

Initialize the environment with the envsetup.sh script.

```
source build/envsetup.sh
```

Choose which target to build with lunch. The exact configuration can be passed as an argument. All build targets take the form `BUILD-BUILDTYPE`, where the `BUILD` is a codename referring to the particular feature combination. The `BUILDTYPE` is one of the following:

- *user*: limited access, suited for production
- *userdebug*: like user but with root access and debuggability, preferred for debugging

- *eng*: development configuration with additional debugging tools

The userdebug build should behave the same as the user build, with the ability to enable additional debugging that normally violates the security model of the platform.

The following command is the one used to configure a full build of the AOSP source tree for the Nexus 6P device.

```
lunch aosp_angler-userdebug
```

The choise to use userdebug instead of eng was to be as close to production as possible while maintaining the possibility to debug and have root access to the device.

To build the Android OS GNU `make` is used, usually followed by `-jN` to do it on parallel tasks. It's common to use a number of tasks `N` that's between one and two times the number of hardware threads on the computer.

5.4 Flash the device

Once the images produced are downloaded from the server, the client is set to flash them on the device.

Fastboot is a *bootloader* mode in which you can flash a device.

All recent devices have factory-reset protection and require a multi-step process to unlock the bootloader.

1. To enable *OEM unlocking* on the device:
   (a) In *Settings*, tap *About phone*, then tap *Build number* seven (7) times to unlock *Developer options*.
   (b) Tap *Developer options* and enable *OEM unlocking* and *USB debugging*.
2. Reboot into the bootloader and use fastboot to unlock it.

   ```
   fastboot flashing unlock
   ```

3. Confirm the unlock onscreen.

Once the device has been OEM unlocked we can flash an entire Android system in a single command; doing so verifies that the system being flashed is compatible with the installed bootloader and radio, writes the boot, recovery, system and partitions together, then reboots the system.

To flash the Nexus 6P device:

1. Press and hold volume down, then press and hold power to boot into `fastboot` mode or using the command command `adb reboot bootloader` that reboots directly the device into the bootloader.

2. After the device is in fastboot mode, run:

   ```
   fastboot flashall -w
   ```

   The `-w` option wipes the `/data` partition on the device.

5.5 Device specific changes

As described shortly in Section 3.5, the SEPolicy on Android comes in two variants: *monolithic* and *split*.

The *monolithic* policy variant is for legacy non-treble devices that contain a single SEPolicy file located at `/sepolicy` and is directly loaded into the kernel SELinux subsystem.

The *split* policy is for supporting treble devices. It splits the SEPolicy across files on `/system/etc/selinux` (the 'plat' portion of the policy) and `/vendor/etc/selinux` (the 'nonplat' portion of the policy). This is necessary to allow the system image to be updated independently of the vendor image, while maintaining contributions from both partitions in the SEPolicy.

Nexus 6P does not support *Treble* so it will put on the device only the monolithic binary policy, which doesn't allow us to perform on device policy build as we wouldn't have the policy sources.

The following changes will force the *SELinux CIL compiler* and *split* policy on the device, and modify kernel and filesystem configurations to allow the early load of system and vendor partitions as originally not required by the Nexus 6P.

5.5.1 Policy build

As shown in Table 5.2, the original make file has a conditional that checks whether `PRODUCT_FULL_TREBLE` is set or not and depending on it the split policy or the monolithic modules are built.

To force the the build system to use the split policy modules all we did was removing the conditional and use always the split SELinux policy.

Before

```
ifeq ($(PRODUCT_FULL_TREBLE),true)
# Use split SELinux policy
LOCAL_REQUIRED_MODULES += \
$(platform_mapping_file) \
    26.0.cil \
    nonplat_sepolicy.cil \
    plat_sepolicy.cil \
plat_and_mapping_sepolicy.cil.sha256 \
    secilc \
    plat_sepolicy_vers.txt
    ...
else
# Use monolithic SELinux policy
LOCAL_REQUIRED_MODULES += sepolicy
endif
```

After

```
# Use split SELinux policy
LOCAL_REQUIRED_MODULES += \
$(platform_mapping_file) \
    26.0.cil \
    nonplat_sepolicy.cil \
    plat_sepolicy.cil \
plat_and_mapping_sepolicy.cil.sha256 \
    secilc \
    plat_sepolicy_vers.txt
    ...
```

Table 5.2: system/sepolicy/Android.mk - Before and after the required changes

5.5.2 Modify vendor image

Nexus 6P vendor image is extracted and installed to the vendor directory from the self-extracting script contained in the device binaries. As originally the policy wasn't split the vendor partition has no policy files at `etc/selinux` so we changed the vendor image.

1. Mount the vendor partition locally.

```
cd aosp
thesis/out/host/linux-x86/bin/simg2img thesis/
   vendor/huawei/angler/proprietary/vendor.img
   vendor.img.raw
mkdir mnt-point
sudo mount -t ext4 -o loop vendor.img.raw mnt-
   point/
```

2. Copy the files into the mounted filesystem.

```
cd thesis/out/target/product/angler/vendor/etc/
   selinux
cp nonplat_sepolicy.cil plat_sepolicy_vers.txt  /
   mnt/data/mrossi/aosp/mnt-point/etc/selinux/
```

3. Create an image of the mounted filesystem.

```
cd /mnt/data/mrossi/aosp
thesis/out/host/linux-x86/bin/make_ext4fs -s -S
   file_contexts -L vendor -l 195M -a vendor
   vendor.img.new mnt-point/
```

   Where the `file_contexts` used is obtain by the concatenation of plat and nonplat file contexts.

4. Substitute our image to the original one, so that our vendor image is used.

```
cp vendor.img.new thesis/vendor/huawei/angler/
   proprietary/vendor.img
```

Newly introduced types in the public platform policy will require to change `nonplat_sepolicy.cil` as a new versioned typeattribute will be added by the build system, so the above commands have to be repeated to avoid any policy build error.

5.5.3 Mount system and vendor early

`init` starts with the kernel security context, but transitions into its own unique `init` domain by re-exec it-self when the policy has been successfully loaded. To make sure `init` can load SELinux policy fragments that are spread across system and vendor partitions devices must enable first stage mount on those partitions.

To enable, Android 8.x and higher supports mounting `/system` and `/vendor` as early as init's first stage (i.e before selinux is initialized). `fstab` entries for early mounted partitions can be specified using *device tree*.

A *device tree* (DT) is a data structure of named nodes and properties that describe non-discoverable hardware. Hardware vendors supply their own DT source files, which Linux then compiles into the *Device Tree Blob* (DTB) file used by the kernel.

As showed in Figure 5.1 DTBs are appended to the kernel image.

To add `fstab` entries into the DT the following operations needs to be done:

1. Extract the appended dtb from Nexus 6P kernel

   To perform this operation split-appended-dtb [5]has been used

   ```
   split-appended-dtb aosp/thesis/device/huawei/
      angler-kernel/Image.gz-dtb
   ```

2. Convert DTB to *device tree source* (DTS)

   ```
   aosp/thesis/prebuilts/misc/linux-x86/dtc/dtc -I
      dtb -O dts -o dts_N.dts dtbdump_N.dtb
   ```

3. Append to the existing DTS the fstab entries as specified in [2]

4. Compile the DTS in DTB

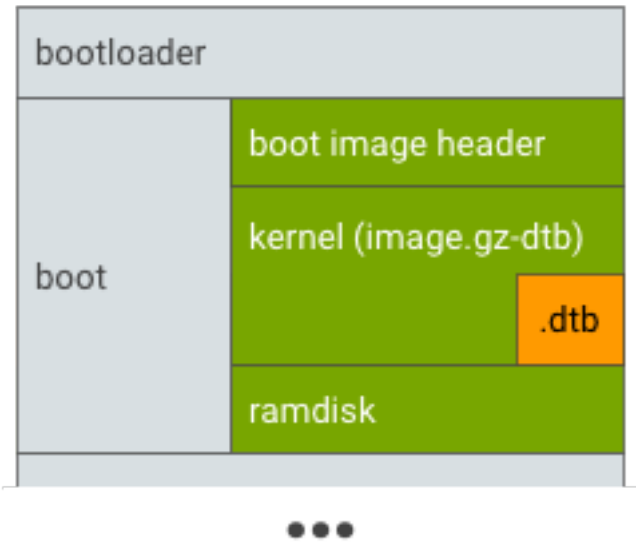

Figure 5.1: Boot partition - Put .dtb in boot partition by appending to image.gz and passing as "kernel" to mkbootimg

```
aosp/thesis/prebuilts/misc/linux-x86/dtc/dtc -O
   dtb -I dts -o dtbdump_N.dtb dts_N.dts
```

5. Append the new DTB to the kernel

   ```
   cat dtbdump_N.dtb >> kernel
   ```

Finally, we must not repeat in the `fstab` file fragments the entries provided via *device tree overlays*. So remove system and vendor partition from `device/huawei/angler/fstab.aosp_angler`.

5.6 Install app

App installation in Android is carried out thanks to a group of services. The image 5.2 shows the installation workflow within an Android OS.

5.6.1 Package Installer

PackageInstaller is the default application for Android to interactively install a normal package, it provides a user interface to manage applications. To

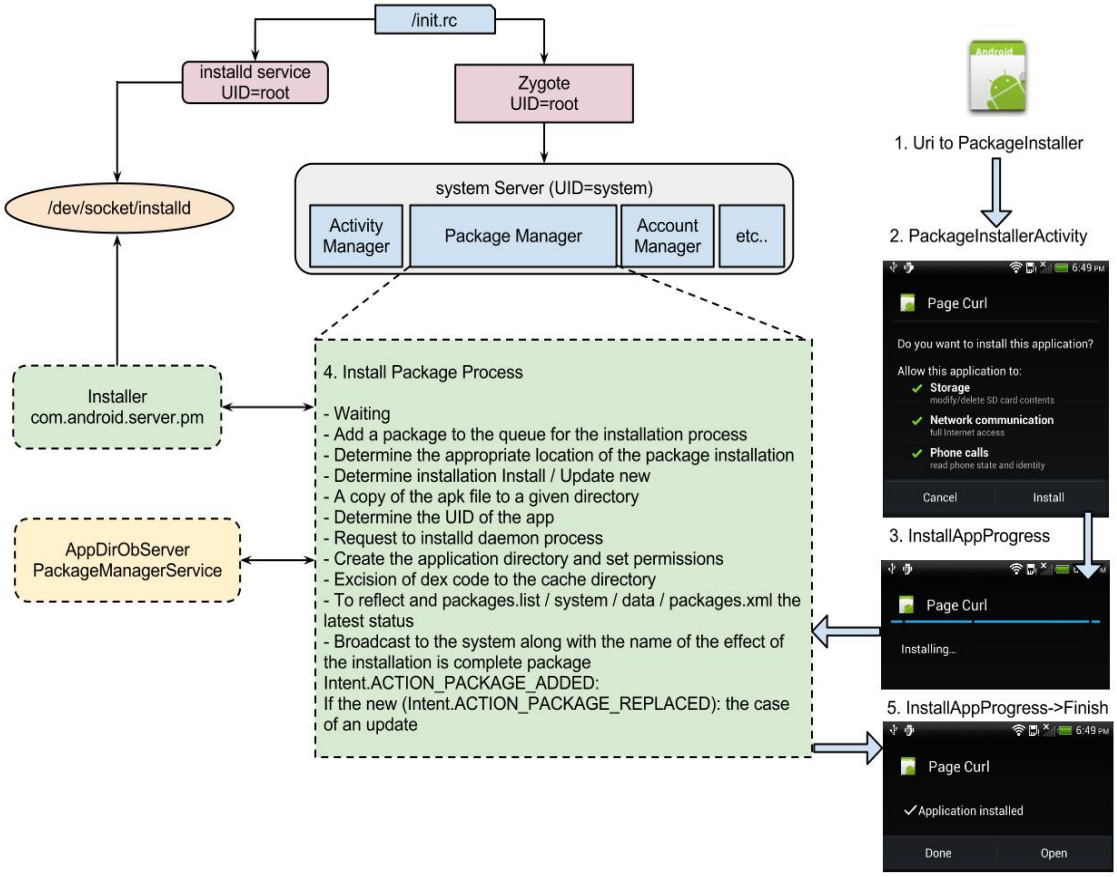

Figure 5.2: App installation work-flow

correctly function Package Installer has its own service the *Package Installer Service* that keeps track of installation progress and actively sends installation requests to the *Package Manager Service*.

During the thesis work this system component has been mostly bypassed installing the test app APK with:

```
adb install path_to_apk
```

This comunicates directly with the Package Manager Service in order to complete the app installation.

5.6.2 Package Manager

Package Manager is an API[2] that actually manages application install, uninstall, and upgrade. When we install the APK[3] file, Package Manager parse the package file and displays confirmation.

Since the third-party app policy files are part of the APK, this is where we extract them and perform the policy checks required to satisfy the requirements stated at Section 4.2, but also validate the contexts files shipped with it.

Once we pass validation, *Package Manager Service* does a call to the install deamon with a JNI method in charge of policy compilation, build and load.

Finally we store the policy files to `data/selinux/<package_name>` directory to make it available to the underlying SELinux library (*libselinux*) and future policy build.

Of course to retain compatibility with 'classic' apps that run under the `unstrusted_app` domain the policy files are not required for a successful installation.

5.6.2.1 SEPolicy

This section explains the module in charge of enforcing the policy requirements.

The *Policy Lexer* and *Parser* implement a simplified version of the *SELinux CIL compiler*, enforces the policy requirements and ensures the contexts used in the contexts files are legitimate.

1. Analyze lexically the given `sepolicy.cil` file producing as output a list of *tokens*. A *token* is constructed as a triplet: type, value and line, where the type can be `COMMENT, OPAREN, CPAREN, SYMBOL, QSTRING` and `UNKNOWN`.

2. Parse the sequence of tokens and based upon the limited CIL grammar we support create an abstract syntax tree that shows the structure of the `sepolicy.cil` file.

[2]Application Programming Interface
[3]Android PacKage

3. Walk through the abstact syntax tree to extrapolate relevant information to the application of the requirements.

   The result of this step are: a list of the *type,* a list of *typeattribute*, a list of the *type* and *typeattribute* bounded to `untrusted_app`, a list of *typeattributeset* statements and a list of *type* and *typeattribute* used as source in at least on allow rule.

4. Apply the requirements specified in Section 4.2

   - *No impact on the system policy*: all elements in the set of a global *typeattribute* and all allow statements sources must be local
   - *No escalation*: all elements in the set of a global *typeattribute* and all allow statements sources must be bounded
   - *Flexible internal structure*: as long as the bound to `untrusted_app` is satisfied the app can provide functionalities and access services, but also change domain between the ones defined by the app.
   - *Protection from external threats*: the app its free to define new security contexts and to assign them to its domain and files. Since in the SELinux policy model every privilege has to be explicitly authorized new types are not accessible by system types unless a dedicated rule is introduced in the `sepolicy.cil` file.

5. Verify contexts files legitimacy:

   - all types used in file_contexts have to be defined in `sepolicy.cil` with the exception of `app_data_file`
   - all domains used in seapp_contexts have to be defined in sepolicy.cil with the exception of `untrusted_app`

5.6.2.2 File contexts

The parsing of the `file_contexts` is performed using regular expressions that verifies the structure of each file entry and restricts the security context that can be associated to app file.

In Android file objects have user `u` and role `object_r`, a type that depends on the file we are considering and security level that depends on how they are created.

System files have only sensitivity `s0` associated to them while application file have also a category depending on levelFrom and level specified in `seapp_context`.

`levelFrom` assume one of the following values:

- none: use the categories specified by the level keyword
- app: `c256,c512`
- user: `c512,c768` (the one we enforce)
- all: `c0,c256,c512,c768`

5.6.2.3 Seapp contexts

`seapp_contexts` was originally meant to both set app domain and file contexts and that's how it still works for 'classic' apps. For policy enhanced apps we decided to use `seapp_contexts` only for the purpose of assigning the domain and to use `file_contexts` for the file labeling.

We took this decision for two reasons:

- file_contexts has regular expression support, which may be helpfull to developers
- file_contexts is way more compact than the seapp_contexts. seapp_context language is aimed at describing an app in a single entry, so a lot of information would have been repeated from on entry to another just to define a path security context

The parsing of the `seapp_contexts` is done by *Seapp Parser* which restricts what kind of app an entry can define.

For example the security enhanced third-party app can't be system_server (`isSystemServer=false`), as well as a preinstalled app (`isPrivApp=flase`) and have to match the `seinfo` definition extracted from the MAC perimissions file.

For what consers the *output selectors,* we don't allow `path` and `type` definition for the reasons exposed above, and we enforce a `levelFrom=user` as already done by the `seapp_context` on 'classic' apps ran within untrusted_app domain.

5.6.2.4 MAC permissions

To handle the `mac_permission.xml` parsing *SELinuxMMAC* has been used. This java class is the one in charge to parse `system/etc/selinux/plat_mac_permissions.xml` and `vendor//etc/selinux/nonplat_mac_permissions.xml` all I did was reusing it to also parse and check the `mac_permissions.xml` file of the installing app, and gather the `seinfo` value associated to it.

After the validation and policy build steps have been done successfully a newly implemented method will add the `mac_permissions.xml` of the app to the list of `mac_permissions.xml` files loaded on the system.

5.6.3 Installd

The *install deamon*, *installd* is the process in charge to install applications and set up the application directory. To correctly work *installd* runs as root and has a lot of privileges across the entire system, so it was quite natural to use it to perform the policy compilation and load into the kernel.

installd is part of the Android native framework and is used by the Package Manager Service through the *Java Native Interface* (JNI).

The Java Native Interface (JNI) is a foreign function interface programming framework that enables Java code running in a *Java virtual machine* (JVM) to call and be called by native applications.

When the JVM invokes the function, it passes a `JNIEnv` pointer, a jobject pointer, and any Java arguments declared by the Java method.

On Android, one process cannot normally access the memory of another process. So to talk, they need to decompose their objects into primitives that the operating system can understand, and marshall the objects across that boundary for you. The code to do that marshalling is tedious to write, so Android handles it for you with AIDL.

The *Android Interface Definition Language* (AIDL) is similar to other IDLs, it allows to define the programming interface that both the client and service agree upon in order to communicate with each other using interprocess communication (IPC).

As we are defining a new method *reloadSELinuxPolicy* in the *installd Native Service* with the objective to make it available to the Package Manager Service we need to update the AIDL associated to *installd*.

After that, all we need to do is implementing the method signature we defined in the AIDL inside the *installd Native Service*. The code of *reloadSELinuxPolicy* its mostly what it is executed by the init to load the SELinux policy for the first time, the only difference is that now we need to also consider the policy files loaded by the apps.

```
binder::Status InstalldNativeService::reloadSELinuxPolicy
    () {
  ENFORCE_UID(AID_SYSTEM);
  std::lock_guard<std::recursive_mutex> lock(mMountsLock);
  LOG(INFO) << "Compiling SELinux policy";
  // Determine the highest policy language version
     supported by the kernel
  int max_policy_version = security_policyvers();
  if (max_policy_version == -1) {
    return error("Failed to determine highest policy
       version supported by kernel");
  }
  policy files context      char compiled_sepolicy[] = "/
     cache/selinux/sepolicy.XXXXXX";
  android::base::unique_fd compiled_sepolicy_fd(mkostemp(
     compiled_sepolicy, O_CLOEXEC));
  if (compiled_sepolicy_fd < 0) {
    return error(StringPrintf("Failed to create temporary
       file %s", compiled_sepolicy));
  }
  // Determine which mapping file to include
  std::string vend_plat_vers;
  if (!selinux_get_vendor_mapping_version(&vend_plat_vers)
```

```
    ) {
  return error("Failed to determine vendor mapping
      version");
}
std::string mapping_file("/system/etc/selinux/mapping/"
    + vend_plat_vers + ".cil");
const std::string version_as_string = std::to_string(
    max_policy_version);
// clang-format off
const char* def_args[] = {
  "/system/bin/secilc",
  "/system/etc/selinux/plat_sepolicy.cil",
  "-M", "true", "-G",
  // Target the highest policy language version
      supported by the kernel
  "-c", version_as_string.c_str(),
  mapping_file.c_str(),
  "/vendor/etc/selinux/nonplat_sepolicy.cil",
  "-o", compiled_sepolicy,
  // We don't care about file_contexts output by the
      compiler
  "-f", "/dev/null"};  // now it is available
size_t i, len = sizeof(def_args)/sizeof(def_args[0]);
char **compile_args, **pkg_policies = get_pkg_policy("
    sepolicy.cil");
if (pkg_policies) {
  size_t nof_pkgs = sizeof(pkg_policies)/sizeof(
      pkg_policies[0]);
  compile_args = (char **) calloc(len + nof_pkgs + 1,
      sizeof(char *));
  if (!compile_args) {
    free_pp(pkg_policies);
    return error(StringPrintf("%s:  Out of memory\n",
        __FUNCTION__));
  }
  for (i = 0; i < len; ++i)
    compile_args[i] = strdup(def_args[i]);
```

```
    for (i = 0; i < nof_pkgs; ++i)
      compile_args[len++] = pkg_policies[i];
  } else {
    compile_args = (char **) calloc(len + 1, sizeof(char
       *));
    if (!compile_args)
      return error(StringPrintf("%s:  Out of memory\n",
         __FUNCTION__));
    for (i = 0; i < len; ++i)
      compile_args[i] = strdup(def_args[i]);
  }
  compile_args[len] = nullptr;
  for (i = 0; i < len; ++i)
  LOG(INFO) << "compile_args[i] = " << compile_args[i];
  // clang-format on
  if (!fork_execv_and_wait_for_completion(compile_args[0],
      (char**)compile_args)) {
    unlink(compiled_sepolicy);
    free_pp(compile_args);
    return error("Failed to compile SELinux policy");
  }
  unlink(compiled_sepolicy);
  free_pp(compile_args);
  LOG(INFO) << "Loading compiled SELinux policy";
  if (selinux_android_load_policy_from_fd(
     compiled_sepolicy_fd, compiled_sepolicy) < 0) {
    return error(StringPrintf("Failed to load SELinux
       policy from %s", compiled_sepolicy));
  }
  return ok();
}
```

reloadSELinuxPolicy forks a child to execute the SELinux CIL compiler and waits for its completion. The compiler build a monolithic binary policy from:

- *plat_sepolicy.cil*: the platform policy

- *nonplat_sepolicy.cil*: the vendor policy
- *mapping_file*: the file that maps the vendor policy to the current platform mapping and allows the platform uprev
- *pkg_policies*: this array contains all the path of the third party app `sepolicy.cil` files that can be found at `/data/selinux/<package_name>/sepolicy.cil`

with options:

- *-M true*: build an mls policy.
- *-G*: expand and remove auto-generated attributes, done to avoid policy size explosion [9, 3]
- *-c*: build a binary policy with a given <version>, in our case the maximum policy version supported by the kernel
- *-o*: target file where the policy will be written, in our case `/cache/selinux/sepolicy.XXXXXX`
- *-f*: target file where the file_context will be written, in our case, since we don't have any interest in this, we throw it away in `/dev/null`

Finally loads the SELinux policy with `selinux_android_load_policy_from_fd` a function of libselinux that we modified in order to allow updating the policy at runtime.

5.7 Uninstall app

App uninstall is taken care of by the same system services that handle app installation.

Of couse in this situation we don't need to do any kind of checking, but only clean all the directories and services that store some app policy related information.

1. Remove the `/data/selinux/<package_name>` directory that contains all the policy files: file_contexts, seap_contexts, mac_permissions.xml and sepolicy.cil.

2. Remove /data/selinux/<package_name>/mac_permissions.xmlreferences from *SELinuxMMAC* data structures.

3. Build the policy without the uninstalled app and load it back into the kernel.

4. Restore /data/data/<package_name>/ default security context app_data_file to allow installd to correctly get rid of the directory

5.8 seapp and file contexts handle

As explained in Section 5.6.2 all policy related files that come with an APK are stored in /data/selinux/<package_name>.

We already talked about the required changes to correctly use the sepolicy.cil and the mac_permissions.xml files in the system, but we didn't perform any 'active' action to load file_contexts and seapp_contexts files. This is due to how these files are loaded by libselinux. *libselinux* loads file_contexts and seapp_contexts the first time a process indirectly calls respectively selinux_android_restorecon_common and seapp_context_lookup functions.

seapp_context_lookup is called in selinux_android_setcontext, the function used by the Zygote process to set the domain of newly spawned processes.

selinux_android_restorecon_common is called anytime a *restorecon* command is issued.

All we needed to do for the newly installed app to correclty load its contexts files is to extend the seapp_contexts and file_contexts files considered to the ones presents in /data/selinux/*/seapp_contexts and /data/selinux/*/file_contexts.

Here follows the implementation for the file_contexts handle.

```
static struct selabel_handle*
   selinux_android_file_context(const struct
   selinux_opt *opts,

   unsigned nopts) {
```

```
struct selabel_handle *sehandle;
struct selinux_opt *fc_opts;
size_t i, len;
// Get file_contexts files of 3rd-party apps
char **pkg_file_contexts = get_pkg_contexts("
   file_contexts");
if (pkg_file_contexts) {
  size_t nof_pkgs = sizeof(pkg_file_contexts)/
     sizeof(pkg_file_contexts[0]);
  fc_opts = malloc((nopts + nof_pkgs + 1) * sizeof(
     struct selinux_opt));
  if (!fc_opts) {
    selinux_log(SELINUX_ERROR, "%s:  Out of memory\
       n", __FUNCTION__);
    free_pp(pkg_file_contexts);
    return NULL;
  }
  for (i = 0; i < nopts; ++i) {
    fc_opts[i].type = opts[i].type;
    fc_opts[i].value = strdup(opts[i].value);
  }
  len = nopts;
  for (i = 0; i < nof_pkgs; ++i) {
    fc_opts[len].type = SELABEL_OPT_PATH;
    fc_opts[len++].value = pkg_file_contexts[i];
  }
} else {
  fc_opts = malloc((nopts + 1) * sizeof(struct
     selinux_opt));
  if (!fc_opts) {
    selinux_log(SELINUX_ERROR, "%s:  Out of memory\
       n", __FUNCTION__);
    return NULL;
  }
```

```
    for (i = 0; i < nopts; ++i) {
      fc_opts[i].type = opts[i].type;
      fc_opts[i].value = strdup(opts[i].value);
    }
    len = nopts;
  }
  fc_opts[len].type = SELABEL_OPT_BASEONLY;
  fc_opts[len++].value = (char *)1;
  for (i = 0; i < len - 1; ++i)
    selinux_log(SELINUX_INFO, "fc_opts[i] = %s",
       fc_opts[i].value);
  sehandle = selabel_open(SELABEL_CTX_FILE, fc_opts,
     len);
  free_opt(fc_opts);
  if (!sehandle) {
    selinux_log(SELINUX_ERROR, "%s: Error getting
       file context handle (%s)\n",
       __FUNCTION__, strerror(errno));
    return NULL;
  }
  // Compute fc_digest only on the system
     file_contexts
  if (!compute_file_contexts_hash(fc_digest, opts,
     nopts)) {
    selabel_close(sehandle);
    return NULL;
  }
  selinux_log(SELINUX_INFO, "SELinux: Loaded
     file_contexts\n");
  return sehandle;
}
```

At the beginning of the function we look for the apps `file_contexts` files, if not found the platform and non-platform `file_contexts` are loaded, otherwise on top of platform and non-platform `file_contexts` also the apps

`file_contexts` files are loaded.
As we said in Section 5.6.2.3 we decided to prefer the use of `file_contexts` over `seapp_context` to assign security contexts to files of the security-enhanced third-party apps. In order to implement this in the *libselinux* library, the function `pkgdir_selabel_lookup` has been changed, to perform the following steps:

1. Detect with what kind of app the function is dealing with.
2. Depending on the outcome of the previous point:
    - *'classic' app*: use seapp contexts handle to lookup the security context needed to label the file
    - *security-enhanced app*: use file contexts handle to lookup the security context needed to label the file

5.9 Addition to the Android API

A solution without the need to extend Android API might be feasible, but during my thesis work files created from the test app wasn't getting the right security contexts straight up after creation, instead they were always given `app_data_file`. As a consequence to assign the correct security contexts to files an explicit *restorecon* command needs to be performed after file creation. In order to allow apps to perform this kind of operation an extension to the API is required and since we were already modifying it for the *restorecon* I also added a *setcon* function to allow app to explicitly change SELinux domain at runtime.

When SELinux was first introduced within the AOSP a SELinux java class was created to implement methods equivalent to the one offered by *libselinux*.

The SELinux class and its methods are not available in the Android SDK, however if developing SELinux enabled apps within AOSP then reflection would be used (as an example the package `packages/apps/SEAdmin` was provided).

With the recent releases however this class has been sitting in the AOSP and not used given that all usage examples of it have been removed.

It turns out this is exactly what we needed to allow an API extension that does a JNI call to *libselinux* functions.

The *restorecon* command was already implemented, but in order to work for our purpose it needed to be visible (remove the `{@hide}` from the *javadoc*) and `SELINUX_ANDROID_RESTORECON_DATADATA` had to be passed to allow *restorecon* to go through the app directories.

We wasn't so lucky for the *setcon* command, in fact, this was not part of the SELinux java class methods. Anyway, since *setcon* is required by *Zygote* to spawn processes within the correct domain as specified by the `seapp_contexts` the underlying SELinux library implements it.

To add *setcon* to the SELinux java class methods

1. Add the native method in SELinux.java

```
/**
 * Changes the security context of the current
    process.
 * @param context new security context given as a
     String.
 * @return a security context given as a String.
*/
public static native boolean setContext(String
   context);
```

2. Add the implementation of the native method in the related `C++` file `android_os_SELinux.cpp`

```
/*
 * Function: setCon
 * Purpose:  set the security context of the
    current process
 * Parameters: context: the new security context
    of the current process
 * Returns: true on success, false on error
 * Exceptions: none
*/
```

```
static jboolean setCon(JNIEnv *env, jobject,
    jstring contextStr) {
  if (isSELinuxDisabled) {
    return true;
  }
  ScopedUtfChars context(env, contextStr);
  if (context.c_str() == NULL) {
    ALOGV("%s(%p): NULL pointer exception",
        __FUNCTION__ , contextStr);
    return false;
  }
  // GetStringUTFChars returns const char * yet
      setfilecon needs char *
  char *orig_ctx_str = NULL, *ctx_str =
     const_cast<char *>(context.c_str());
  int rc = -1;
  rc = getcon(&orig_ctx_str);
  if (rc)
    goto err;
  ALOGD("Current context: %s New context: %s.",
     orig_ctx_str, ctx_str);
  ALOGV("%s(%s):  Executing
     security_check_context ctx=%s", __FUNCTION__,
      ctx_str, ctx_str);
  rc = security_check_context(ctx_str);
  if (rc < 0)
    goto err;
  ALOGD("Security check for context %s passed.",
     ctx_str);
  ALOGV("%s(%s):  Executing strcmp ctx=%s
     orig_ctx=%s", __FUNCTION__, ctx_str, ctx_str,
      orig_ctx_str);
  if (strcmp(ctx_str, orig_ctx_str)) {
    ALOGD("Going to execute
```

```
            selinux_android_setcon.");
        rc = selinux_android_setcon(ctx_str);
        if (rc < 0)
          goto err;
    }
    ALOGD("Context has been set to %s.", ctx_str);
    rc = 0;
out:
    freecon(orig_ctx_str);
    ALOGV("%s(%s) => %s", __FUNCTION__, ctx_str,
        context.c_str());
    return (rc == 0);
err:
    ALOGV("%s(%s): Error setting context %s",
        __FUNCTION__, ctx_str, strerror(errno));
    rc = -1;
    goto out;
}
```

This native method checks the validity of the security context passed, compares it with the current domain and if they are different triggers *selinux_android_setcon* the *libselinux* function that takes care of setting the domain of the current process to the one specified as input of the function.

5.10 SEPolicy changes

All the changes done to implement this SELinux fine-grained protection of Android apps change also how some of the processes interact with already defined security contexts. To allow our solution to work a lot of changes have been done on the platform policy, here the most important ones will be described.

5.10.1 Installd

`installd.te` had some major changes:

- allow to create and delete a temporary file in `/cache/selinux`
- transition `installd` to `secilc` on execution of `secilc_exec` to grant policy access only to secilc execution
- allow *`installd`* to load the policy into the kernel

```
# Allow installd to create and delete the built
   policy
allow installd cache_file:dir { search };
allow installd cache_selinux_file:dir { search
   add_name write remove_name };
allow installd cache_selinux_file:file { create
   getattr open read write unlink };

# Transition to secilc domain on execution of
   secilc_exec
domain_auto_trans(installd, secilc_exec, secilc)
# Allow secilc to access sepolicy files
allow secilc sepolicy_file:file { open getattr read
   };
# Allow installd to load_policy
allow installd kernel:security { load_policy };
```

`domain.te` has been changed to remove the impact of a neverallow rule on installd.

5.10.2 Ueventd

`ueventd.te` had some minor changes to allow access to access to */data/selinux/*/file_contexts*.

```
allow ueventd file_contexts_file:dir { open read };
allow ueventd file_contexts_file:file { open };
```

Before

```
# Once the policy has been loaded there shall be none
    to modify the policy.
# It is sealed.
neverallow * kernel:security load_policy;
```

After

```
# Once the policy has been loaded only installd can
   modify the policy.
neverallow { domain -installd } kernel:security
   load_policy;
```

Table 5.3: domain.te - Before and after

5.10.3 Untrusted app

`untrusted_app.te` requires some more privilegies to be able to perform *restorecon* and *setcon* functions. On a production release this should be done with a system service to avoid granting all this permission to third party apps.

```
# Allow to read file_contexts_file
allow untrusted_app file_contexts_file:file { getattr
    open read };
# Allow to read seapp_contexts_file
allow untrusted_app seapp_contexts_file:file {
   getattr open read };
# Allow to relabelfrom ... and relabelto ...
allow untrusted_app app_data_file_type:file { getattr
    open read relabelfrom relabelto };
# Allow to change file/process context
allow untrusted_app selinuxfs:file { open read write
   };
allow untrusted_app kernel:security { check_context
   };
```

```
# Switch SELinux context to newly defined domains
   allow untrusted_app self:process { setcurrent };
   allow untrusted_app untrusted_app_all:process {
   dyntransition };
allow untrusted_app untrusted_app_all:binder {
   impersonate };
```

`app.te` and `app_neverallows.te` are changed to allow apps to use my API extension.

5.11 Test app

To develop and test third-party apps protection with SELinux functionalities a demo app has been realized.

5.11.1 How it works

Note all the security contexts described here have `com_example_sepolicytestapp` as prefix.

The app works as follows at app lunch, it runs inside the `init` domain, creates two files (`top-secret` and `unclassified`), calls the *restorecon* function to set the correct context to the newly created files (`top_secret_`file and `unclassified_`file) and goes into `top_secret` domain. At this point we can see that both file are accessible, but clicking the 'Switch to unclassified' button the domain is changed with the *setcon* function to `unclassified` and the top-secret file is no more accessible.

Originally the app was meant to be able to switch from `top_secret` and `unclassified` domain anytime the button was pressed, but the *setcon* function only allows to transition into a domain that is bounded to the corrent one and this don't allow to go back to `unclassified`.

This limitation however should be overcome by an app that, instead of personally transition into a domain, spawns new processes with stricter domains.

5.11.2 Policy files

This section shows the policy files associated to our basic app, `sepolicy.cil` might look quite complicated for such an easy app, this is due to the basic privileges that we need to grant *domains* in order to correctly function. Anyway its very likely that a more complex app won't require much more entries to work.

5.11.3 file contexts

```
.*  u:object_r:app_data_file:s0:c512,c768
files/unclassified  u:object_r:com_example_sepolicytestapp.
    unclassified_file:s0:c512,c768
files/top-secret  u:object_r:com_example_sepolicytestapp.
    top_secret_file:s0:c512,c768
```

5.11.4 seapp contexts

```
user=_app seinfo=sepolicy_test_app domain=
    com_example_sepolicytestapp.init name=com.example.
    sepolicytestapp levelFrom=user
```

5.11.5 MAC permissions

```
<?xml version="1.0" encoding="iso-8859-1"?>
<policy>
  <signer signature="3082...d9bc">
    <package name="com.example.sepolicytestapp">
      <seinfo value="sepolicy_test_app"/>
    </package>
  </signer>
</policy>
```

To extract the signature from the APK, `post_process_mac_perms` has been used.

5.11.6 SEPolicy

```
(block com_example_sepolicytestapp
    (type unclassified_file)
    (type top_secret_file)
```

```
(type init)
(type unclassified)
(type top_secret)

(typeattributeset app_data_file_type (unclassified_file
   top_secret_file))

(typebounds untrusted_app init)
(typebounds init top_secret)
(typebounds top_secret unclassified)

(typeattribute domains)
(typeattributeset domains (init unclassified top_secret))
(typeattributeset domain domains)
(typeattributeset appdomain domains)
(typeattributeset untrusted_app_all domains)

    ...

(allow domains app_api_service (service_manager (find)))
(allow domains surfaceflinger_service (service_manager (
   find)))
(allow init file_contexts_file (file (getattr open read)))

(allow init app_data_file (file (relabelfrom)))
(allow init top_secret_file (file (relabelto)))
(allow top_secret_file labeledfs (filesystem (associate)))
(allow init unclassified_file (file (relabelto)))
(allow unclassified_file labeledfs (filesystem (associate))
   )
(allow init selinuxfs (file (open read write)))
(allow init kernel (security (check_context)))
(allow init self (process (setcurrent)))
(allow init top_secret (process (dyntransition)))
(allow top_secret selinuxfs (file (open read write)))
(allow top_secret kernel (security (check_context)))
(allow top_secret self (process (setcurrent)))
(allow top_secret unclassified (process (dyntransition)))
(allow init unclassified (binder (impersonate)))
(allow top_secret unclassified (binder (impersonate)))
(allow init app_data_file (file (create)))
(allow init unclassified_file (file (getattr read)))
```

```
    (allow init top_secret_file (file (getattr read)))
    (allow unclassified unclassified_file (file (read)))
    (allow top_secret unclassified_file (file (read)))
    (allow top_secret top_secret_file (file (read)))
)
```

5.12 Add policy file to APK

To add the policy directory to the APK and make it accessible to the Package Manager Service the following steps are followed.

1. Unzip APK, copy the policy directory inside and zip it

   ```
   unzip app-debug.apk -d app-debug
   cp -a policy app-debug
   cd app-debug
   zip -r ../app-debug+policy.apk *
   cd ..
   ```

2. Align the APK

   ```
   zipalign -fvp 4 app-debug+policy.apk app-debug+
      policy.apk
   ```

3. Sign the APK

   ```
   apksigner sign --ks debug.keystore --out debug+
      policy.apk debug+policy.apk
   ```

4. Specify the keystore password

Chapter 6

Conclusions and future developements

Security is correctly perceived, both by technical experts and customers, as a crucial property of mobile operating systems. The integration of SELinux into Android is a significant step toward the realization of more robust and more flexible security services. The attention that has been dedicated in the *SEAndroid* initiative toward the protection of system components is understandable and consistent with the high priority associated with the protection of core privileged resources. Our implementation is the natural extension of that work, which demonstrated a successful deployment, toward a more detailed consideration of the presence of apps.

The thesis shows that the potential for the application of policy modules associated with each app is quite extensive, supporting scenarios where developers define their own app policy, and scenarios where policies are automatically generated to improve the enforcement of privileges and the isolation of apps. The extensive level of reuse of SELinux constructs that characterizes the language for the appPolicyModules demonstrates the flexibility of SELinux and facilitates the deployment of the proposed solution. An analysis of the evolution of the *SELinux* implementation in Android confirms that app policy modules identify a concrete need and that Android is evolving in this direction.

The introduction of SELinux policies for fine-grained protection of Android apps improves the definition and enforcement of the security requirements associated with each app. However, in the approach presented in the previous

section, we assumed that the extension to the MAC policy has to be defined by the developer, who knows the service provided by the app and its source code. Due to the size of the community, we can expect that many app developers will either be unfamiliar with the SELinux syntax and semantics, or know SELinux but not want to use it, avoiding the introduction of strict security boundaries to the app beyond those associated with untrusted app. There is also the risk generated by the presence in devices of a variety of versions of the system policy and the need to guarantee that the app policy is compatible with it.

However, we observe that the app developers that can be expected to be most interested in using the services of the MAC model are expert developers responsible for the construction of critical apps (e.g., apps for secure encrypted communication, or for key management, or for the access to financial and banking services). This community is possibly small, but its role is extremely critical. They can be expected to overcome the obstacles to the use of our methodology. In addition, the deployment of the policy modularity services opens the door to a number of other services. We consider here how it is possible to use them to enforce a stricter model on the management of *Android permissions*, relying on the automatic generation of app policies, solving all the issues identified above.

Looking at the workflow to build an app, developers are already familiar with the definition of security requirements in the `AndroidManifest.xml`, through the use of the tag `<uses-permission>`. In fact, in order to access system resources (e.g. access to the user's current location) the app has to explicitly request the associated Android permissions (e.g. *`android.permission-group.LOCATION`*), which correspond both to a set of concrete actions at the OS level and to a set of permissions granted at the SELinux level (e.g. open, read on files and directories). The system already offers both a high-level representation and a low-level representation of the privileges needed to access a resource, but they are not integrated and what happens, in the absence of policy modularity, is that the app is associated with the untrusted app domain, which is allowed to use all the actions that correspond to the access to all the resources that are invokable by apps, essentially using for protection only Android permissions. The integration of

security policies at the Android permission and MAC levels offers a more robust enforcement of the app policy.

This can be realized introducing a mechanism that bridges the gap between different levels, through the analysis of the high-level policy (i.e., the permissions asked by the app within the *Android Permission Framework*) and the automatic generation of an app policy module that maps those Android permissions to a corresponding collection of SELinux statements. The generator starts from the representation of the app security requirements expressed in the `AndroidManifest.xml`, builds a logical model of the structure of the app policy, and it finally produces the concrete implementation of the app policy module and verifies that all the security restrictions are satisfied.

A necessary step in the construction of the mechanism is the identification of a mapping between policies at the distinct levels. The *Android Permission Framework* contains more than 200 permissions and most of them present a mapping between the *Android permission* and a dedicated *SELinux domain*, already specified in the system policy. The current system policy does not cover all the permissions; e.g. the downloads, calendar, and media content resources are associated with the single platform app data file type. We expect this aspect to be manageable with a revision of the policy. However, there is a number of Android permissions that can only be partially supported by this mechanism due to current limitations in the security mechanisms provided by internal components (e.g., *SQLite*).

To summarize, it is already possible to capture most Android permissions in a precise way and some of them with some leeway, leading in all cases to a significant reduction in the size of the MAC domain compared to what would otherwise be associated with an app.

Bibliography

[1] Cil - common intermediate language. https://android.googlesource.com/platform/external/selinux/+/master/secilc/docs. Last accessed on April 1, 2018.

[2] Example: system and vendor on nexus 6p. https://source.android.com/devices/architecture/kernel/modular-kernels#example:system-and-vendor-on-n6p. Last accessed on March 31, 2018.

[3] Policy size explosion. https://source.android.com/security/selinux/device-policy#policy_size_explosion. Last accessed on March 31, 2018.

[4] Selinux project. bounds rules. http://www.selinuxproject.org/page/Bounds_Rules. Last accessed on March 27, 2019.

[5] split-appended-dtb github repository. https://github.com/dianlujitao/split-appended-dtb. Last accessed on March 31, 2018.

[6] Titanium backup - root needed. https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup&hl=en. Last accessed on April 1, 2018.

[7] David Elliott Bell. Looking back at the bell-la padula model. *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[8] E. Kirda C. Mulliner, W. Robertson. Virtualswindle: An automated attack against in-app billing on android. *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 459–470, 2014.

[9] Thomas Nyman Elena Reshetova, Filippo Bonazzi. Characterizing seandroid policies in the wild. 2015.

[10] Stefano Paraboschi Enrico Bacis, Simone Mutti. Apppolicymodules: Mandatory access control for third-party apps. 2014.

[11] A. Ludwig. Android - practical security from the ground up. http://goo.gl/z0RIwu, 2013.

[12] E. Magri M. Arrigoni Neri, M. Guarnieri. Conflict detection in security policies using semantic web technology. *Proceeding of the IEEE ESTEL*, 2012.

[13] O. Abendan M. Balanza, K. Alintanahin. Droiddreamlight lurks behind legitimate android apps. *Malicious and Unwanted Software (MALWARE)*, 2011.

[14] Stefano Paraboschi Simone Mutti, Enrico Bacis. Policy specialization to support domain isolation. *Proceedings of the 2015 Workshop on Automated Decision Making for Active Cyber Defense*, pages 33–38, 2015.

[15] Robert Craig Stephen Smalley. Security enhanced (se) android: Bringing flexible mac to android. *NDSS Symposium 2013*, 2015.

[16] W. Zhou Y. Zhou, Z. Wang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *NDSS*, 2012.