Extending Mandatory Access Control Policies in Android

Stefano Paraboschi, Enrico Bacis, and Simone Mutti

Università degli Studi di Bergamo, Italy
Department of Management, Information and Production Engineering
{parabosc,enrico.bacis,simone.mutti}@unibg.it

Abstract. Solutions like SELinux have recently regenerated interest toward Mandatory Access Control (MAC) models. The role of MAC models can be expected to increase in modern systems, which are exposed to significant threats and manage high-value resources, due to the stronger protection they are able to offer. Android is a significant representative of these novel systems and the integration of MAC models is an important recent development in its security architecture. Opportunities indeed exist to further enrich the support offered by MAC models, increasing their flexibility and integrating them with other components of the system. We discuss a number of proposals that have recently been made in this domain.
First, we illustrate the integration of SELinux and SQLite, named *SeSQLite*, which permits to apply MAC permissions at a fine granularity into relational databases, offering both a schema-level and row-level support. Then, *AppPolicyModules* are presented, which let app developers specify extensions to the system-level policy that protect the resources of each specific app. Finally, an integration between SELinux and the interprocess communication services is proposed, to further regulate the cooperation among separate apps and services. All these enhancements lead to a stronger and more detailed support of the complex security requirements that characterize modern environments.

1 Introduction

A clear long-term trend in computer security is the increasing complexity of the systems that have to be controlled, with a significant increase in terms of the attack surface, resource value, and complexity of user needs. In terms of the attack surface, larger software stacks and the presence of pervasive network connections increase the exposure of a system and the number of components whose failure may lead to a compromise of a system. The growth in resource value derives from the adoption of computer systems to support most activities. The wider impact of IT solutions also leads to the need to offer access to a larger number of users, each with the need to operate over a specific fraction of the system resources. In many domains the use of computers would indeed be even stronger in the absence of security worries. It can be argued that security management is a bottleneck to the development of modern information systems.

The construction of systems able to offer adequate protection has to consider a large variety of aspects [11, 1, 5]. At a high-level, it is necessary that components of IT systems are built using a security-by-design approach, giving to security the role of critical requirement considered in the first phases of the project. Each of the elements, at every level of the architecture, has then to provide robust behavior. An element that is perceived to increase in importance is the use of modern access control models and policy languages. These classical security tools require to be extended in order to meet the stringent requirements of novel applications.

The Android operating system is a representative example of modern computer systems, with novel challenges and the need to extend current security techniques and models. Android has become the most widely deployed operating system, with an impact on a variety of application domains (smartphones, embedded devices, domotics, automotive, etc.). In this paper we mostly refer to the use of Android in the smartphone domain, but the other domains can also benefit from the enhancements that we propose.

Android has to face an extensive collection of security challenges. One of the most visible threats in Android is represented by the installation of potentially malicious apps. Much of the utility of a smartphone derives form its ability to execute apps that suit specific user needs. Apps can be retrieved from monitored markets, like Google's Play Store, or from user-chosen repositories, which may host malicious apps. The risk associated with user-chosen repositories is far greater, but the design of Android has to assume that users will be able and will install potentially malicious apps. The goal is then to reduce the damage that a malicious app can do to the system. The problem can be considered a variant of classical multi-user operating systems, where users are assumed to potentially misbehave and access control aims at restricting the damage potential. Indeed, in Android each app is associated in the Linux kernel with a specific *uid* and *gid*, adapting to apps the Discretionary Access Control (DAC) services of Linux that were originally designed to support multiple users. At the file system level, each app is then able to label its files with *acl*s that will be consistent with the protection needs of each app.

The DAC model alone is insufficient to deal with the security challenges Android has to face. An important recent evolution in Android is the development of SEAndroid, which integrates SELinux into the design of the operating system [15]. SELinux implements a type-based Mandatory Access Control (MAC) model in Linux. The advantages of MAC models are particularly beneficial to Android. The presence of SELinux offers a strictly enforced central policy, able to limit the power (and abuses) of privileges. The availability of SELinux permits Android to better support the security principle of "isolation", containing the installed apps within a restricted space and limiting their ability to compromise system resources.

SEAndroid already provides significant benefits, but we see additional opportunities to extend its use in Android. The paper summarizes a few recent advances in this domain that could lead to an increase in the security of the system. A first aspect is the inability of the SQLite database to support fine granularity in the access to resources. An effort has been directed to the integration in SQLite of the support for MAC labels, allowing a table-level and row-level enforcement of access restrictions on the content of the database. Another research line considers the introduction of the ability of apps to be protected by SELinux. The current architecture mainly uses SELinux to protect system resources from the vulnerabilities and misbehaviors of system components and apps. Apps that are not considered part of the core system are all contained together in a single *untrusted_app* domain. An app interested in getting protection cannot receive support from the MAC system. Developers of apps that process sensitive data may instead be interested in getting this protection. To allow app developers to specify ad-hoc policies for their apps requires to satisfy several critical requirements, with a need for the correct management of policy modules that goes beyond what current systems provide. The proposal of AppPolicyModules wants to provide a mechanism that lets apps to be enriched with an ad-hoc MAC policy for their resources, with guarantees about the fact that this addition to the policy is not going to weaken the system MAC policy. The availability of this technology also opens the door to the realization of a more robust support of Android permissions.

2 Overall Android architecture

The Android architecture is composed by three layers (see Figure 1): (a) a Linux kernel (b) a middleware framework and (c) an application layer. The first layer (i.e., the Linux kernel) provides low-level services and device drivers to other layers and differs from a traditional Linux kernel. The Android team has taken the Linux kernel code and modified it to run in an embedded environment, thus it does not have all the features of a traditional Linux distribution.The second layer is composed by native Android libraries, runtime modules (e.g., Dalvik Virtual Machine) and an application framework.

The third layer is composed by applications. They can be divided into two categories (i) core applications (e.g., browser, dialer phone) installed by default and (ii) common applications, written in Java, installed by the user. Each application is composed by different *app components*. There are four different types of app components: (a) *Activities*, typically each screen shown to a user is represented by a single *Activity component* (b) *Services*, provide the background functionalities (c) *Content Providers* used to share data among applications and (d) *Broadcast Receivers* used to receive event notifications both from the system and from other applications.

2.1 Android Security Architecture

Android provides a *permission mechanism* that enforces restrictions on the specific operations that a particular process can perform. Basically, by default, an application has no permissions to perform any operation (e.g., reading or writing

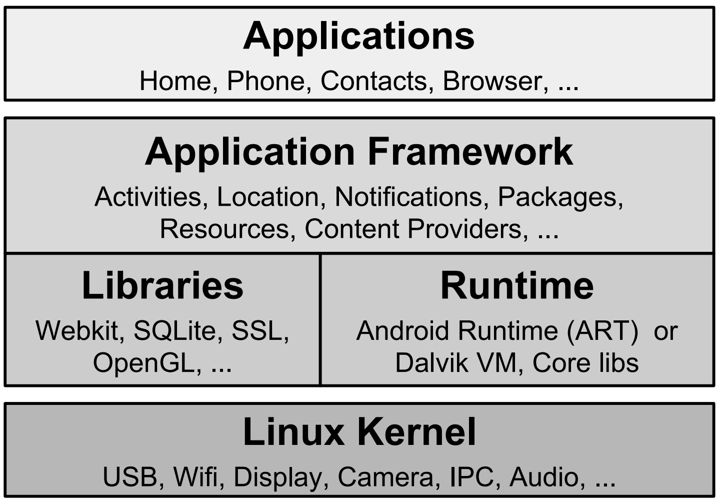

Applications
Home, Phone, Contacts, Browser, ...
Application Framework
Activities, Location, Notifications, Packages, Resources, Content Providers, ...
Libraries
Webkit, SQLite, SSL, OpenGL, ...
Runtime
Android Runtime (ART) or Dalvik VM, Core libs
Linux Kernel
USB, Wifi, Display, Camera, IPC, Audio, ...

Fig. 1. Android Architecture overview.

the user's private data, keep the device awake, performing network access). Furthermore, Android isolates applications from each other, with a sandbox. The sandbox mechanism relies on the use of *certificates*. All Android applications must be signed with a certificate whose private key is held by their developer and is a prerequisite for inclusion into the official Android Market. The purpose of certificates in Android is twofold: (a) to distinguish application authors; and (b) to grant or deny access to signature-level permissions.

Android assigns a unique user id (UID) and a group id (GID) to each app. Each installed application has a set of data structures and files that are associated with its UID and GID. Only the application itself and the superuser (i.e., root) have the permissions to access these structures and files.

2.2 SELinux

SELinux originally started as the Flux Advanced Security Kernel (FLASK) [7] developed by the Utah University Flux team and the US Department of Defense. The development was enhanced by the NSA and released as open source software. SELinux policies are expressed at the level of *security context* (also known as *security label* or just *label*). SELinux requires a security context to be associated with every process (or subject) and resource (or object), which is used to decide whether access is allowed or not as defined by the policy. Every request that a process generates to access a resource will be accepted only if it is authorized by both the classical DAC access control service and by the SELinux policy. The advantages of SELinux compared to the DAC model are its flexibility (the design of Linux assumes a *root* user that has full access to DAC-protected resources) and the fact that process and resource labels can be assigned and updated in a way that is specified at system level by the SELinux policy (in the DAC model, owners are able to fully control the resources). SELinux uses a closed world assumption, so the policy has to explicitly define rules to allow a *source* (the process) to perform a set of *actions* on a *target* (the resource). The rule also specifies the class of target on which the rule has to be applied (e.g., file, directory). An SELinux rule has the following syntax:

```
allow source_t target_t:class {actions};
```

3 SeSQLite

Android provides several ways to store apps data. For example, apps can store text files both in their own files directory and in the phone SD card. Sometimes, however, an app needs to be able to carry out complex operations on persistent data, or the volume of data requires a more efficient method of management than a flat text file. To this end Android provides a built-in *SQLite* database[1].

SQLite is the most widely deployed in-process library that implements a SQL database engine. It offers high storage efficiency and small memory footprint. A SQLite database is represented by a single disk file and anyone who has direct access to the file can read the whole database content. Usually, due to the fact that SQLite has no separate server, the entire database engine library is integrated into the application that needs to access a database. Furthermore, SQLite does not provide any kind of access control mechanism, it only provides a few proprietary extensions (e.g., database encryption).

Modern DBMSs provide their own authorization mechanisms[6, 4], with a corresponding set of SQL constructs, which permit access control at the level of tables, columns or views. DBMSs use a permission model that is similar to, but separate from, the underlying operating system permissions. Current information systems often make a limited use of these database access control facilities and tend to embed access control directly in the application program used to access the database. This choice derives from the perceived difficulty in keeping the database users aligned with the user population in the application, and from the flexibility obtained in the construction of the application thanks to the absence of access control restrictions to the database. (Our expectation is that this is going to change, but it will take a long time.)

The integration of the MAC model, defined at the level of the operating system, and the access control services, internal to the DBMS, promises to reduce the above obstacles and lead to an increase in the security of the system. Such integration would also provide *system-wide consistency*, because all the access control decisions would be guaranteed to be compliant with the system-level MAC policy. To realize such integration in Android it is necessary to extend SQLite with the support for SELinux controls.

[1] https://www.sqlite.org/

3.1 SQLite and Content Provider

In the Android platform, SQLite is used to store several types of information, like contacts, SMS messages, and web browser bookmarks. In order to let an app consume these types of information, Android provides specific components, called *Content Providers*. The Content Providers are daemons that provide an interface to the SQLite library, for sharing information with other applications.

Due to the fact that SQLite does not provide any security mechanism, access control is embedded directly into the Content Provider used to access the database. The Content Provider code that handles a query can explicitly call the system's permission validation mechanism, using the *Android Permission Framework*, to require certain permissions.

Besides the Content Provider, at the Linux kernel level Android provides both *DAC* and *MAC* access control [15]. Both DAC and MAC are designed to provide protection against an attempt to directly access the database by a process that is not the Content Provider.

3.2 SeSQLite Architecture

The file-level granularity provided by DAC is not sufficient if we want to provide a fine grained access control over SQLite databases. To this end, we introduced *Security-Enhanced SQLite* [10] (SeSQLite), which extends SQLite and integrates it with SELinux in order to provide fine-grained mandatory access control. In this way we can ensure that access control policies are consistently applied to every user and every application.

The requirements leading the implementation of SeSQLite are the following:

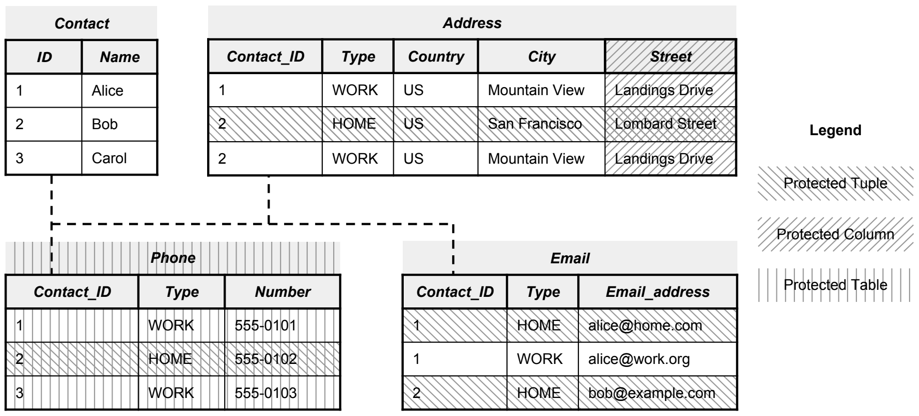
R.1 - Backward Compatibility SeSQLite is designed to maintain backward-compatibility with common SQLite databases (i.e., no modification to the SQL syntax);

R.2 - Flexibility SeSQLite is designed to provide everything needed to successfully implement a Mandatory Access Control module, while imposing the fewest possible changes to SQLite. Moreover, it is designed to be easily adapted to different implementation of MAC (e.g., *SELinux* or *SMACK[14]*);

R.3 - Performance SeSQLite must keep negligible the overhead on computational time and database size.

3.3 Access Control Granularity

In a SQLite database there are different types of SQL object, which can be grouped in two granularity levels: the *Schema Level* and the *Tuple Level*. To supply per-SQL object protection, SELinux requires that SQLite provides support for security labeling. However, a distinction among the approaches used to



Contact

ID	Name
1	Alice
2	Bob
3	Carol

Address

Contact_ID	Type	Country	City	Street
1	WORK	US	Mountain View	Landings Drive
2	HOME	US	San Francisco	Lombard Street
2	WORK	US	Mountain View	Landings Drive

Phone

Contact_ID	Type	Number
1	WORK	555-0101
2	HOME	555-0102
3	WORK	555-0103

Email

Contact_ID	Type	Email_address
1	HOME	alice@home.com
1	WORK	alice@work.org
2	HOME	bob@example.com

Fig. 2. Example of an access policy defined on a database of contacts leveraging both SeSQlite schema checks and tuple checks. An application that wants to issue a SELECT query over the shadowed parts needs an additional privilege.

manage schema and tuple object is needed, since schema and tuple objects address different needs and belong to different logical models, as it will be presented in the following.

Schema Level All SQL statements must be compiled before their execution. In SQLite, the compilation process follows the sequence of steps below:

1. Syntax check;
2. Semantic check;
3. Expansion;
4. Code generation.

The output of the compilation process is a piece of program containing the information about all the tables and attributes that have to be accessed.

Example 1. Consider the *Address* table in Figure 2 and the following query:

```
SELECT Country, City, Street FROM Address WHERE Contact_ID=1;
```

The statement accesses four columns within the *Address* table. The *Country*, *City* and *Street* attributes appear in the target list directly as a part of the query. The *Contact_ID* is used in the *WHERE* clause.

Using this information, SeSQLite introduces a *schema check* to control if the query complies with the SELinux policy, i.e., the user can access all the tables and columns specified in the query. An error is immediately raised if the user does not have all the privileges needed to perform the query.

According to the example policy in Figure 2, the check will raise an error if the issuer is not granted the access privilege, since the user does not have the privileges to select the attribute *Street*. The same approach can be applied to `INSERT`, `DELETE` and `UPDATE` statements.

Row Level At tuple level, the query is made to be always compliant with the policy. In fact, tuple level access control operates as a *filter* that automatically excludes any unaccessible tuple from the loop that scans the tables. This allows SeSQLite to process only the tuples that can be accessed by the user, according to the action requested.

Query rewriting is the most common mechanism used to provide fine-grained access control at tuple level because the modifications are internal to the database and do not require any adaptation at application level. Essentially this can be compared to automatically appending conditions to a SQL query's *WHERE* clause as it executes, and dynamically changing the result returned by the query.

Example 2. Consider the table *Email_address* in Figure 2 and the query:

```
SELECT Type, Email_address FROM Email WHERE Contact_ID=1;
```

Type	**Email_address**
HOME	alice@example.com
WORK	alice@example.org

Table 1. Result of the query in Example 2 without *check_tuple()*.

The output of the query is shown in Table 1. However, if we consider the same query and we want to enforce that a user can select only non protected tuples, with query rewriting the query becomes:

```
SELECT Type, Email_address FROM Email
WHERE Contact_ID=1 AND check_tuple();
```

The *check tuple()* function appends to the *WHERE* clause the predicate responsible to perform the access control filter. The query output is presented in Table 2.

Type	**Email_address**
WORK	alice@example.org

Table 2. Result of the query in Example 2 with *check_tuple()*.

SeSQLite uses a modified version of traditional query-rewriting, with the following traits:

	time	**overhead**
SQLite	5.846s	–
SeSQLite (2 contexts)	6.132s	+4.8%
SeSQLite (100 contexts)	6.741s	+15.3%
SeSQLite (1000 contexts)	6.749s	+15.4%

Table 3. Total CPU time and overhead for a suite of common operation in SeSQLite.

1. Due to the fact that SQLite does not provide a multi-user database, the decision to allow or deny the access to a SQL object depends on the policy provided by the *SELinux Security Server*. This design provides a characteristic feature called system-wide consistency in access control, which means that SELinux provides all the access control decisions based on a single declarative policy.
2. In order to maintain a negligible overhead, SeSQLite uses two different *query rewriting* approaches, based on the number of SELinux contexts used in the database (the experimental results can be found in Table 2):
   - The first approach is based on a custom SQL function, which checks for each row if its security context can be seen by the issuer of the query. This approach is selected when a substantial number of different SELinux contexts is used in the database.
   - The second approach is used when a limited number of SELinux contexts is present in the database (the most common case). Before starting the table scan, the visibility of all the contexts present in the database is computed based on the issuer of the query. The ones that can be accessed are enclosed in a SQL *IN* operator, so that only those tuples are accessed.

4 AppPolicyModules

SeSQLite assumes that different Android application use different SELinux contexts, so that it is possible to use this information in the access control checks.

Unfortunately, at the moment all the apps share a single SELinux context[2]: *untrusted_app*. We recently proposed the concept of *AppPolicyModules* (*APM*s) [2], which aim at providing the full benefits given by the use of Mandatory Access Control to third-party apps.

4.1 Providing Mandatory Access Control to Apps

APMs follow the principles of the Android security model, which aims at strengthening the boundaries among apps, introducing an additional mechanism to guarantee that apps are isolated and cannot manipulate the behavior of other apps. The additional mechanism is obtained with an adaptation of the services of the MAC model introduced by SELinux.

[2] There is also the context *google_app* used only for the apps signed by Google.

The introduction of APMs improves the definition and enforcement of the security requirements associated with each app. However, apps become known to the system only when the user asks for their installation. For this reason, the MAC policy has to be dynamic, with the ability to adapt to the installation and deletion of apps. This requires modularity and the capability to incrementally update the security policy, letting an app be able to specify the policy module it is associated with. In this way app developers, who know the service provided by the app and its source code, can benefit from the presence of a MAC model, letting them define security policies that increase the protection the app can get against attacks coming from other apps, which may try to manipulate the app and exploit its vulnerabilities.

It is to note that many app developers will either be unfamiliar with the SELinux syntax and semantics, or will not want to introduce strict security boundaries to the app beyond those associated with *untrusted_app*. 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 access to financial and banking services).

We consider here how it is possible to use APMs to enforce a stricter model on the management of Android permissions, relying on the automatic generation of APMs. If we consider the structure of the *AndroidManifest.xml* file provided by each app, it already contains the definition of security requirements through the use of the tag `<use-permission>`. For example, if an app developer wants to write on the SD card, she has to explicitly request the associated Android permissions (e.g., *android.permission.WRITE_EXTERNAL_STORAGE*), which corresponds both to a set of concrete actions at the OS level and to a set of SELinux rules.

The system already offers both a high-level and a low-level representation of permissions, but they are not integrated. Furthermore, in the absence of policy modularity, the apps are associated only with the *untrusted_app* domain, which is allowed to use the actions that correspond to the access to all the resources that are invokable by apps, essentially using for protection only the functions of Android permissions. The integration of security policies at different 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 APM that maps those Android permissions to a corresponding collection of SELinux rules.

In general, with the availability of *appPolicyModules*, the system could evolve from a scenario where each app is given at installation time access to the whole *untrusted_app* domain at the SELinux layer, to a scenario where each app is associated with the portion of *untrusted_app* domain that is really needed for its execution, with a better support of the classical "least-privilege" security principle. It is to note that Android M provides the feature to drop Android permissions at runtime. This confirms that APMs and more generally domain specialization identify a concrete need and that Android is evolving in this direction.

Example 3. Most mobile browsers (e.g., Chrome, Firefox) store confidential information such as *usernames* and *passwords* in a SQLite database. Following Google's best practices for developing secure apps, the password database is saved in the app data folder, which should be accessible only to the app itself. However, this is not enough to protect the password database by other apps with root privileges.

The use of MAC support offers 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 the domains introduced by the APM.

Figure 3 shows an example where the *untrusted_app* domain does not hold any permission on the file labeled as *password_file*, which is accessible only by the *browser1* domain.

It is to note that both *browser1* and *password_file* are typebounded (see [9]), thus *browser1* is not violating any restriction defined on the parent domain (i.e., *untrusted_app*). 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.

The management of policy modules provided by apps and integrated within the system-level SELinux policy has to satisfy four crucial requirements.

Req1, No impact on the system policy : the app must not change the system policy and only have an impact on processes and resources associated with the app itself;

Req2, No escalation : the app cannot specify a policy that gives more privileges than those given to *untrusted_app*;

Req3, Flexible internal structure : the app can define and activate separate domains, to limit potential vulnerabilities deriving from internal flaws, adopting the principle of "least privilege" and fragmenting its structure in a way that each component is only given the minimum amount of privileges it needs to execute properly;

Req4, Protection from external threats : the app can protect its resources from other apps, specifying its own types and then defining SELinux restrictions on them.

We refer to [2] for an extensive discussion of these principles and their impact on the realization of APMs.

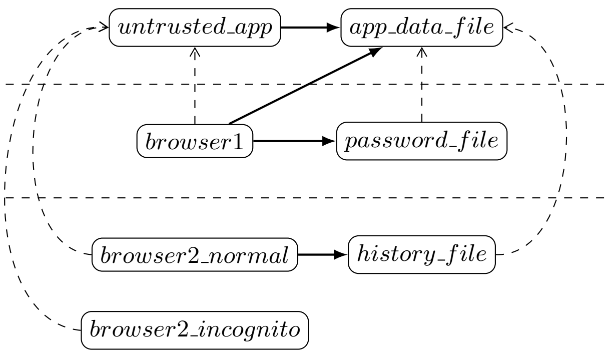


Fig. 3. Isolation of two applications using typebounds.

5 SEIntentFirewall

In order to cross the process boundaries (i.e., inter-process communication) an app can use a messaging object to request an action from another app or system component. The messaging object is called *Intent*.

5.1 Android Intents

Formally, Intents are asynchronous messages that allow application components to request functionality from other Android components. This mechanism has been denoted as *Inter-Component Communication* (ICC). Intents represent the high-level *Inter-process Communication* (IPC) technique of Android, while the underlying transport mechanism is called *Binder*.

Binder is a customized implementation of *OpenBinder* for Android. It has the facility to provide bindings to functions and data from one execution environment to another. The *OpenBinder* implementation runs under Linux and extends the existing IPC mechanisms. Android provides two types of intent:

Implicit intent : specifies the action that should be performed and optionally the data for the action. If an implicit intent is used, Android searches for all components that are registered for the specific action and the provided data type;

Explicit intent : explicitly defines the component that should be called by the Android system (i.e., using the Java class as identifier).

Intents can be used to: start *Activities*; start, stop, and bind *Services*; and, broadcast information to *Broadcast Receivers*. All of these forms of communication can be used with either explicit or implicit Intents.

5.2 SEIntentFirewall Architecture

Unfortunately, Intent messages can be intercepted by a malicious app, as shown in previous research [3, 12, 13], due to the fact that there is no guarantee that

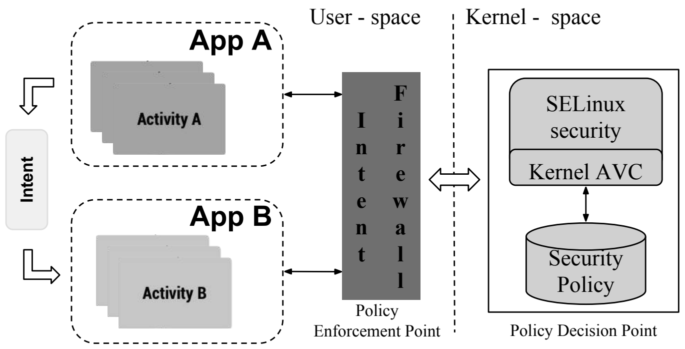


Fig. 4. Overview of the architecture used by SEIntentFirewall.

the Intent will be received by the intended recipient, and launch a malicious Activity in place of the intended Activity. To address this problem, Google has introduced the *Intent Firewall* component since Android 4.3. As it is explicit in the name, the *Intent Firewall* is a security mechanism that regulates the exchange of *Intents* among apps.

Although this approach provides several advantages in the protection against Intent-based attacks, it introduces two major drawbacks. Firstly, the *Intent Firewall* policy can be modified only by the root user (i.e., uid 0). Secondly, due to the fact that the system has to manage a new policy language, it introduces *policy fragmentation.*

SEIntentFirewall [8] is a built-in enhancement of *IntentFirewall*, providing fine-grained Mandatory Access Control (MAC) for Intent objects (see Figure 4). This approach leads to a more powerful control on the communication among apps. This aims at strengthening the barriers among apps, introducing an additional mechanism to guarantee that apps are isolated and cannot manipulate the behavior of other apps.

The SELinux decision engine will then operate as the Policy Decision Point. This choice offers a well-defined policy language and engine, leads to a simpler and better structured code base, and minimizes the implementation effort. It is to note that this design does not require to adapt apps source code. The *SEIntentFirewall* will be obtained with an adaptation of the services provided by APMs, discussed before [2]. This way, the security-demanding developers will be able to embed their SEIntentFirewall rules jointly with the AppPolicyModule.

6 Conclusions

The paper reported on a number of interconnected investigations, all characterized by the objective of extending the support of Mandatory Access Control in Android. The integration of SELinux and SQLite extends the use of labels to the database content, supporting the application of the system-level policy to the structured data contained in the many existing databases. The support

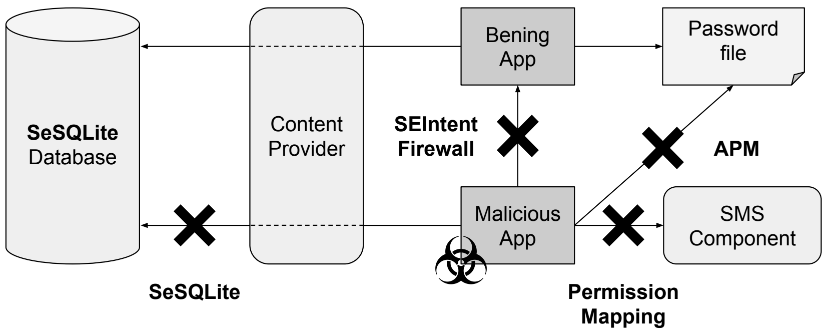


Fig. 5. A malign app is blocked in manifold ways thanks to the wide adoption of SELinux. The SeSQLite database blocks the retrieval of sensitive data; SEIntentFirewall intercepts malicious Intents; the APM specifies which app has access to the password file, while the permission mapping strengthens the permission control.

for AppPolicyModules offers to app developers the opportunity to protect with stronger guarantees their apps and data from vulnerabilities in other apps or system components. The introduction of the SEIntentFirewall finally lets the MAC model cover also the invocation of services.

As it is shown in Figure 5, all these elements support each other and permit the construction of a more robust operating environment, with the possibility to control the behavior of the system in different phases, all following the specifications of a modular policy applied at every access request.

Acknowledgments

The authors would like to thank Ivo Carminati, Paolo Carrara, Stefano Cattaneo, Valentina Centurelli, Andrea Durelli, Kevin Gotti, Niccoló Mangiagalli, Giada Mariani, Francesco Marrazzo, Simone Opreni, Jacopo Ratti, Marco Rosa, Gabriele Scotti, Paolo Vavassori, and Davide Vimercati for support in the implementation of the systems and in the experimental evaluation. This work was partially supported by a Google Research Award (winter 2014), by the Italian Ministry of Research within the PRIN project "GenData 2020" and by the EC within the 7FP and H2020 program, respectively, under projects PoSecCo (257129) and EscudoCloud (644579).

References

1. Arrigoni Neri, M., Guarnieri, M., Magri, E., Mutti, S., Paraboschi, S.: Conflict Detection in Security Policies using Semantic Web Technology. In: Proc. of IEEE ESTEL - Security Track (2012)

2. Bacis, E., Mutti, S., Paraboschi, S.: AppPolicyModules: Mandatory Access Control for Third-Party Apps. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security. pp. 309–320. ACM (2015)
3. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. ACM (2011)
4. Denning, D.E., Akl, S.G., Morgenstern, M., Neumann, P.G., Schell, R.R., Heckman, M.: Views for multilevel database security. In: Security and Privacy, 1986 IEEE Symposium on. pp. 156–156. IEEE (1986)
5. Guarnieri, M., Arrigoni Neri, M., Magri, E., Mutti, S.: On the notion of redundancy in access control policies. In: Proceedings of the 18th ACM symposium on Access control models and technologies. pp. 161–172. ACM (2013)
6. Knox, D.: Effective Oracle database 10g security by design. McGraw-Hill, Inc. (2004)
7. Lepreau, J., Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D.: The flask security architecture: system support for diverse security policies. Secure Computing Corp Saint Paul MN (2006)
8. Mutti, S., Bacis, E., Paraboschi, S.: An SELinux-based Intent manager for Android. In: IEEE Conference On Communications and Network Security. Florence, Italy (September 2015)
9. Mutti, S., Bacis, E., Paraboschi, S.: Policy Specialization to Support Domain Isolation. In: SafeConfig 2015: Automated Decision Making for Active Cyber Defense. Denver, Colorado, USA (October 2015)
10. Mutti, S., Bacis, E., Paraboschi, S.: SeSQLite: Security Enhanced SQLite. In: Annual Computer Security Applications Conference 2015 (ACSAC 2015). Los Angeles, California, USA (December 2015)
11. Mutti, S., Neri, M.A., Paraboschi, S.: An eclipse plug-in for specifying security policies in modern information systems. Proc. of the Eclipse-IT (2011)
12. Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P.: Semantically rich application-centric security in Android. Security and Communication Networks 5(6), 658–673 (2012)
13. Sbîrlea, D., Burke, M.G., Guarnieri, S., Pistoia, M., Sarkar, V.: Automatic detection of inter-application permission leaks in Android applications. IBM Journal of Research and Development 57(6), 10–1 (2013)
14. Schaufler, C.: Smack in embedded computing. In: Proc. Ottawa Linux Symposium (2008)
15. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In: Network and Distributed System Security Symposium (NDSS 13) (2013)