An SELinux-based Intent manager for Android

Simone Mutti, Enrico Bacis, Stefano Paraboschi
DIGIP — Università degli Studi di Bergamo, Italy
{simone.mutti, enrico.bacis, parabosch} @ unibg.it

Abstract—The support for Mandatory Access Control offered by SELinux has become a significant component of the security design of the Android operating system, offering robust protection and the ability to support system-level policies enforced by all the elements of the system. A well-known security-sensitive aspect of Android that currently SELinux does not cover is the abuse of intents, which represent the Android approach to inter-process communication. We propose SEIntentFirewall, an SELinux intent manager that provides fine-grained access control over Intent objects, permitting to cover within MAC policies the use of intents.

I. INTRODUCTION

The rapid success and wide deployment of mobile operating systems has introduced a number of novel and challenging security requirements, with a clear need for an improvement of security technology. 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. Nowadays, one significant attack vector is represented by the (ab)use of Intent objects [1], [2], [3]. Intents are the communication mechanism that can be used to exchange information between Android components of the same application or among distinct ones. The integration of SELinux into Android (briefly, SEAndroid [4]) is a significant step toward the realization of more robust and more flexible security services. However, SEAndroid does not take into consideration the use of Intent objects in the communication among apps. The paper introduces SEIntentFirewall, a Security Enhanced version of the IntentFirewall component. SEIntentFirewall uses the features provided by MAC models to provide fine-grained access control over Intent objects. The proposal adapts in the design the concept of appPolicymodule [5], an extension of SEAndroid that gives to each app the possibility to define its own SELinux policy under a set of well specified constraints.

II. PROBLEM

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 component. These messaging objects are called Intents. Formally, Intents are asynchronous messages that allow application components to request functionalities from other Android components (see Figure 1). This mechanism has been denoted as Inter-Component Communication (ICC). Intents represent the higher-level Android Inter-process Communication (IPC) technique, and the underlying transport mechanism used is called Binder. Android provides two types of Intent:

- Implicit intent: it specifies the action that should be performed and optionally data that is provided for the action. If an implicit intent is used, Android searches for all components that are registered for the specific action and are compatible with the data type;
- Explicit intent: it 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. Unfortunately, the exchange of intents represents an application attack surface, as shown in several papers. The main issue is that, when an application sends an implicit Intent, there is no guarantee that the Intent will be received by the intended recipient. A malicious application can then intercept an Intent and launch a malicious Activity in place of the intended Activity. Interception can also lead to control-flow attacks, like denial of service [1], [2], [3]. Several solutions have been proposed, using different approaches (e.g., static/dynamic analysis, control-flow mechanisms). However, none of them has been included in the Android Open Source Project (AOSP). To address this problem, Google has introduced the Intent Firewall component, since Android 4.3.

![Fig. 1. Abstract representation of Intent mechanism.](image)

III. INTENT FIREWALL

As explicit in its name, the Intent Firewall is a security mechanism that regulates the exchange of Intents among apps, by analyzing the type of data exchanged. A specific syntax was developed in order to build an Intent Firewall policy, represented by an XML file. Listing 1 shows a snippet of an Intent Firewall policy.

```xml
<rules>
  <activity block="true" log="false">
    <component-filter name="com.android.dialer.DialtactsActivity"/>
  </activity>
</rules>
```

Listing 1. Snippet of an Intent Firewall policy.
The example in Listing 1 defines a rule that blocks all intents to the Android phone dialer component (i.e., an Activity). Following the common architecture of access control services, the Intent Firewall realizes a classical Reference Monitor and it covers both the roles of a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP). Although, this approach provides several advantages in the protection against intent-based attacks, it introduces two major drawbacks. Firstly, the modification of the Intent Firewall policy can be done only by the root user (i.e., uid 0). This means that a common app developer cannot use this mechanism to protect its own app from malicious requests by other apps. Secondly, it introduces policy fragmentation, as the system will have to manage an additional policy language. From a policy management point of view, a system with several policy languages and PDPs is difficult to maintain, with no clear solution to the maintenance of the consistency among all the distinct policies.

IV. PROPOSAL

Our contribution is a built-in enhancement of IntentFirewall, providing fine-grained Mandatory Access Control (MAC) for Intent objects. SEIntentFirewall takes access control decisions based on a SELinux security policy (see Figure 2), in the same way as user access to file system objects is enriched by SELinux in the kernel. 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 AppPolicyModules [5].

V. IMPLEMENTATION AND EVALUATION

The work in [5] represents the basis for our work. The use of appPolicyModules allows each app developer to specify an ad-hoc SELinux policy for its app, offering guarantee about the integrity of the system policy and the ability of each app to benefit from the stronger protection offered by the MAC model. The implementation of appPolicyModules has been extended in order to manage Intent objects. The modifications can be structured into the following activities: (i) retrieve the security context associated with an app, (ii) allow the SELinux access control engine to handle permissions defined for Intent objects.

The first set of challenges concerns the modification of the JNI bridge in order to retrieve the security context associated with an app. At the Application Framework level, the SELinux class provides access to the centralized Java Native Interface (JNI) bindings for SELinux interaction. The android_os_SELinux.cpp file represents the JNI bridge. In the current AOSP version, the JNI bridge is able to retrieve the security context only of running apps, but an Intent could also be used to start an app. To address this limitation and retrieve the security context that will be associated with the app, we modified the android_os_SELinux.cpp file introducing a set of functions able to retrieve the needed information.

The second set of challenges concerns the introduction of a new security class, and the respective permissions, to let the SELinux engine handle Intent objects. Listing 2 shows an example of an SELinux rule using the new security class.

\begin{verbatim}
Listing 2. “SELinux rule using intent security class.”
allow appdomain appdomain:intent { send; }
\end{verbatim}

As clearly expressed in the design of Android, any security feature must introduce a minimal overhead in terms of performance. We executed a series of experiments for the evaluation of the performance impact of the techniques presented in this paper. Experiments have been run on a Nexus 6 with Android 5.1. The overhead introduced by the use of SEIntentFirewall is negligible, less than 3%.

VI. CONCLUSIONS

Security is correctly perceived 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. Our approach is a natural application of this design. The potential of an SELinux-based solution like SEIntentFirewall is extensive and leads to a significant improvement in access control enforcement and app isolation.

REFERENCES