Mandatory Access Control for the Android Dalvik Virtual Machine

Aline Bousquet[1], Jérémy Briffaut[1], Laurent Clevy[2], Christian Toinard[1], Benjamin Venelle[2]
[1] *LIFO - ENSI de Bourges - first.last@ensi-bourges.fr*
[2] *Alcatel-Lucent Bell Labs - first.last@alcatel-lucent.com*

Abstract

With the growing use of smartphones and other mobile devices, it becomes essential to be able to assure the user that his system and applications are doing exactly what they are supposed to do. Over the years and despite its configuration complexity, Mandatory Access Control has proven its efficiency in protecting systems. This paper proposes a solution providing a generic protection that doesn't need to modify the applications. Moreover, in order to face the complexity of defining an efficient MAC policy, a tool automatizes the generation of the policies required for the various applications.

However, to efficiently guarantee the security of a system, each layer that composes it must be secured. Therefore, MAC implementations should not be limited to the operating system, but should also protect the inside of the applications.

This paper presents Security Enhanced Dalvik (SEDalvik), a MAC approach for the Dalvik Virtual Machine in order to control the flows inside the Java applications running in Android.

SEDalvik proposes a new mandatory protection to block the attacks that exploit the weakness of the Dalvik VM. By controlling the information flows between the Java objects, SEDalvik could prevent the new vectors of attack coming from the threat of the Java virtual machine as explained by Kaspersky Labs[1]. In contrast with other approaches, our solution corresponds to a self-organizing system since it transparently protects existing Java applications without any modifications. An experiment on an Android phone shows the efficiency of the protection.

Keywords

Security, Java, Mandatory Access Control, Android, Dalvik

1 Introduction

Android is the most widely used system for smartphones and its security is therefore an essential challenge. Indeed, due to its considerable popularity, Android is more and more frequently the target of attacks and the users grow more concerned about the security of their devices. Indeed, 96 new threats on Android were detected in Q4 2012 by F-Secure [2] and 238 in the whole year, that is to say 79% of the threats detected on mobiles.

For instance, [5] describes a conceptual weakness in Android that allows privilege escalations attacks. Thus, it is possible for an unprivileged application to access a protected resource through a privileged application. This can happen because of the way the applications can interact: when an application accesses another one's components, Android does not ensure that the callee's permissions form a subset of the caller's permissions. Hence, the calling application can indirectly obtain the callee's permissions.

A similar weakness is also described in [17] which explains how a privileged application can store sensitive data on the SDCard. Since the SDCard is world readable (for Android versions 4.0 and earlier), the sensitive data becomes accessible to any other application, even an unprivileged one.

One way to handle this risk is to use Mandatory Access Control (MAC) to block malicious information flows inside a Dalvik Java application. SEDalvik offers such a solution to control the permissions between the caller and the callee objects.

SEDalvik is a new protection for Android, derived from a previous work, SEJava [16], that aims to protect the Java Virtual Machine (JVM). However, the Dalvik Virtual Machine differs from the JVM. Therefore, Dalvik requires a dedicated solution to enforce MAC policy.

Since requesting modifications of applications does not fit with self-organizing systems, SEDalvik can reuse an application without any change. Therefore, the pro-

tection works with all the applications.

Furthermore, regarding the complexity of the definition of the policy, a learning tool can generate the policy automatically when an application is installed.

Section 2 introduces some key concepts of Android as well as related works concerning the security of Android. Then, section 3 describes SEDalvik's concepts and implementation. Finally, section 4 shows the results obtained with SEDalvik, concerning both its efficiency and its performances.

2 Background

2.1 Android

Android is a system for mobile devices that includes an operating system based on the Linux kernel, a Java middleware and Java applications available from the Google store.

Android also provides some tools and APIs easing the development of third-party applications with the Java programming language.

2.1.1 Dalvik VM

Android uses its own virtual machine, named Dalvik [2] and acquired by Google. Dalvik is quite different from off-the-shelf implementations of the JVM. Dalvik was designed with optimization in mind, in order to run Java applications on devices with little memory, limited computational power and short battery life.

Dalvik is a register based virtual machine. Its instruction set contains 246 opcodes (i.e. bytecodes) which are essentially different from the 144 opcodes defined by the JVM specifications [15].

A standard Java compiler stores the program bytecodes into `.class` files, one `.class` file per defined Java class. The Android's Java compiler uses the `dx` tool to merge all `.class` files into one single `.dex` file (Dalvik Executable - Dex).

The `.dex` file format aims to minimize the VM memory usage by sharing data. In contrast with the JVM, Dalvik uses several memory pools shared among all classes to store data according to their nature.

2.1.2 Access Control

Android provides several mechanisms limiting the interactions between the system and the applications and between the applications themselves. This subsection describes these access control mechanisms.

To handle applications privileges, Android uses a specific model of permissions [7]. Each application requests a set of permissions, allowing it to perform specific actions. For instance, an application that needs to send SMS has to request the `SEND_SMS` permission. This is a security model based on capabilities.

Permissions are explicitly granted by the user during the installation of the application. Nevertheless, since Android does not allow a partial selection, the user must either accept all the permissions or cancel the installation. Moreover, the user cannot change the permissions afterwards: the only way is to uninstall the application. A solution, described in [10], has been implemented to allow the user to specify exactly what resources an application can use.

Many applications request too many permissions. There is two reasons 1) the developer usually asks for unnecessary permissions, because of the difficulty to define a minimal set of permissions and 2) the application is a malware that asks for illegal accesses. Since it goes against the least privilege principle, these applications present a damageable risk: too many privileges implies that the application may access resources for illegitimate purpose.

Android also implements an application sandboxing mechanism to isolate applications from one another. Each application runs in its own instance of the Dalvik VM and under a unique user identifier (`uid`). Thus Android enforces a Discretionary Access Control to restrict accesses to the application's resources.

However, a given `uid` can be used by several applications if they are signed by the same developer's certificate. Consequently, a misuse of the developer's certificate may disable the offered isolation.

Both Android's permissions model and the application's sandboxing are mechanisms derived from Discretionary Access Control (DAC). This means that the data's security is under the responsibility of its owner (i.e. the application) and that a super user such as root can access all data. Besides, a DAC system fails to guarantee security properties [8].

Therefore, the security of Android needs to be improved, as shown by the numerous studies that propose to address these problems.

2.2 Related Works and Motivation

TaintDroid [6] is an extension of Android that enables the tracking of information flows on Android smartphones. TaintDroid uses data tainting to track sensible data. It assumes that the applications installed by the user cannot be trusted. It monitors the user's data and aims to detect when some data leaves the system.

YAASE [12] is a security extension for Android that uses TaintDroid to provide a fine-grained access control mechanism. The user defines a set of policies to con-

trol the propagation of data. The policies are enforced thanks to hooks positioned in Android framework's components.

AppFence [9] makes privacy controls on Android applications by retrofitting the runtime environment. AppFence implements two systems: data shadowing, i.e. giving an application fake data (empty contact list...) instead of sensitive data, and ex-filtration blocking, i.e. preventing sensitive data (tainted by TaintDroid) from leaving the device.

However, these three tools produce an important number of false positives. Moreover, they cannot protect from escalations of privileges, that are the more common attacks on Android.

Saint [11] is a framework used to define policies for the applications. With Saint, it is for instance possible to restrict the access to a permission. Indeed, using Saint, an application declaring a new permission can specify under which conditions this permission will be granted to another application. Moreover, Saint controls the inter-applications communications at runtime.

Aurasium [18] is a protection solution that does not alter the Android OS. Indeed, Aurasium hardens Android applications by repackaging them in order to add its policy enforcement code. Thus, Aurasium can control access to sensitive information, such as IMEI number, location...

CRePE [4] presents a policy enforcement solution based on the contextual environment (geographical location, time of the day...). These environments are automatically detected by CRePE, and no action from the user is requested. Thus, users can disable some functions depending on the current situation.

SEAndroid [13, 14] is a port of SELinux on Android, but it extends SElinux controls to Android specific features, such as intents. SEAndroid comes with some predefined security policies for system processes and system applications. However, it has the same limitations as SELinux: for instance, it cannot prevent advanced attacks using 1) indirect information flows between applications or 2) flows inside an Android Java application.

As shown by this state of the art, a solution that monitors the interactions between the Java objects is really missing for the Dalvik VM. In contrast with the other approaches, a MAC protection using a fine-grained policy with type enforcement such as proposed by SEDalvik has several advantages. Firstly, SEDalvik does not require any modification of the applications, nor any bytecode injection. Secondly, it limits the false positive decisions associated with the over-approximation available in the tainting approaches. Thirdly, it precisely controls all kinds of flows between all the Java objects, thus supporting a large range of security properties dealing with confidentiality, integrity and escalation of privileges.

3 SEDalvik

SEDalvik extends SEJava to protect the Dalvik virtual machine, including the Android applications. Indeed due to major differences with the JVM, SEJava cannot work on Dalvik. Therefore, SEDalvik proposes a novel model of Mandatory Access Control to prevent malicious flows between the Dalvik objects.

3.1 Mandatory Access Control for Dalvik

Inside the Dalvik VM, SEDalvik monitors the interactions between a source and a target Java object i.e. an instance of a Java class. SEDalvik associates each object with a unique security identifier. A security identifier includes the Java type, which is unique, and a unique instance id (for example, the instance's address). Different instances associated with the same Java type share the same security context. Thus, a security context corresponds to a Java type.

SEDalvik controls two kinds of interactions:

1. a method M_1 of a source security context O_1 calling the method M_2 of a target security context O_2 (i.e. an `invoke` permission)
2. a method M_1 of a source object O_1 accessing an field F_2 of a target object O_2 (i.e. an `access` permission.)

Therefore, SEDalvik controls an interaction between two objects associated with the security contexts O_1 and O_2 that can be described as follows:

$$O_1 - \{ \text{ Permission } \} \rightarrow O_2$$
where
Permission : (M_1 `invoke` M_2) or (M_1 `access` F_2)

3.1.1 Policy files

SEDalvik defines a MAC policy using two kinds of files.

1. the `.vmc` files (for Virtual Machine Contexts) define the security context for each class signature.

   Here is a example of two security contexts associated with the *Object* and *Thread* classes:

   ```
   Ljava/lang/Object;    object_j
   Ljava/lang/Thread;    java_lang_thread_j
   ```

2. the `.vmr` files (for Virtual Machine Rules) list all the allowed interactions. By default, all the interactions not explicitly described in a `.vmr` file are forbidden since it is a mandatory protection.

   The format used to define rules is presented thereafter, as well as a sample rule allowing a thread to create an object:

```
allow <source_context> <target_context>
   from <source_method>
   invoke <target_method>
allow java_lang_thread_j object_j
   from (init)
   invoke (init)
```

To ease the definition of the policy, SEDalvik helps to learn the required policy through a dynamic computation of 1) the security contexts and 2) the interactions between these contexts.

3.2 Architecture

SEDalvik has two main components: a reference monitor including the interception of the methods and a decision engine i.e. a C library receiving an interaction as a request.

The interactions between the interception of the methods and the decision engine are described in figure 1.

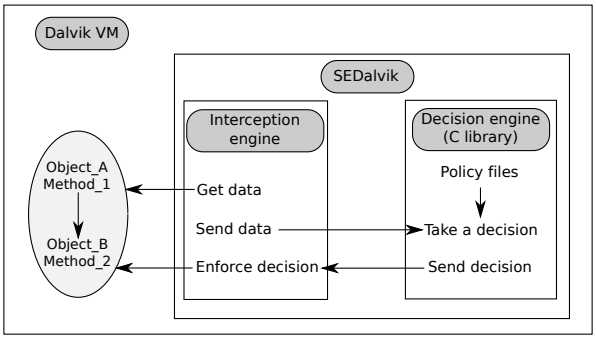

Figure 1: SEDalvik architecture.

When an interaction occurs, the interception engine retrieves some necessary data, computes a request corresponding to the interaction and sends the request to the decision engine. The decision engine checks the received interaction against the policy and sends back a decision to the interception engine, so that it continues (allows) or aborts (denies) the method.

SEDalvik's process can be summed up as follows:

1. The application is created (happens only once)
   (a) The C library is loaded
   (b) The security policy is loaded
2. The application runs
   (a) Each interaction is intercepted
   (b) A decision is taken for each interaction

The next subsections describe each of these steps.

3.3 Application Creation

When Android creates an application, SEDalvik is initialized. Some operations are performed during this initialization: the C library for the decision engine is loaded, as well as the policy files (the `.vmc` and `.vmr` files). Operations unrelated to SEDalvik are also executed, like the loading of classes.

The initialization of SEDalvik can be time-consuming, especially when a large policy is loaded. However, this process is done only once, when the application runs for the first time. Indeed, when an Android application exits, the system does not kill it, but only stop it. Thus, when the application is launched again, it does not need to be created again, and the initialization is not performed again.

3.4 Interception of methods

To be able to intercept all the methods executed by an application, a satisfying solution is requested for Dalvik.

Our reference monitor uses the same mechanism as the JDWP [3] agent that comes with Dalvik and provides debug functions for an external debugger.

The JDWP agent uses the Dalvik's internal debugger. They are deeply linked and the debugger is active only when a JDWP client is connected. This default behavior is modified so that the debugger is active when SEDalvik is running. Thus, SEDalvik intercepts the method calls without having the extra-cost of a running JDWP agent.

The interactions between the Dalvik's debugger, the JDWP agent and SEDalvik are described in figure 2.

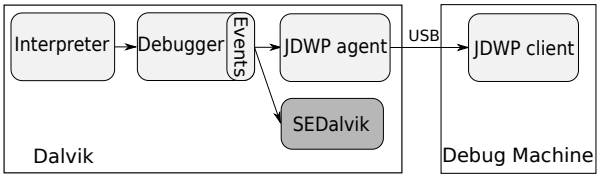

Figure 2: Interactions between the debugger and SEDalvik.

SEDalvik is directly connected to the Dalvik's internal debugger. Therefore, when an event occurs in the virtual machine, the interpreter warns the debugger. If the event is one of those monitored by SEDalvik - for instance, a METHOD_ENTRY event - the debugger will inform SEDalvik. The event will then be handled as needed.

Consequently, SEDalvik is able to intercept all the method calls occurring in the virtual machine and to handle them. Moreover, the methods are intercepted at an early stage, which is an advantage concerning the performances.

3.5 Handling Events

Once a method is intercepted, SEDalvik decides whether the operation should be allowed or blocked.

Each time a `METHOD_ENTRY` event is emitted, Dalvik's debugger is warned about it. The debugger then calls SEDalvik's events handler functions.

SEDalvik will retrieve the callee's and the caller's data. The corresponding interaction is then sent to the decision engine that has to find a satisfying rule in the MAC policy. When SEDalvik gets the answer back, it can continue or abort the callee method.

3.5.1 Interaction request

Each time a method is called, SEDalvik needs to get some data about it in order to compute the corresponding interaction request. The fields to retrieve are those needed to designate the objects, the methods and the attributes involved in the interaction:

- the names of 1) the two involved methods $M1$ and $M2$, their IDs and accessors or 2) the method $M1$ and attribute $A2$, their IDs and accessors
- the security context of the two objects $O1$ and $O2$ and their IDs

These fields enable to compute the interaction request. SEDalvik fetches these fields by interacting with Dalvik's internal debugger:

Indeed, Dalvik's debugger provides an interface to get the content of a stackframe (where the stackframes are the elements composing the application's call stack).

However, the debugger has to know which stackframe is concerned. Therefore, SEDalvik uses stack inspection to have access to the current frame (i.e. the target Java object) as well as the previous frame (i.e. the caller).

The debugger can then query the stackframe and get every needed field.

3.5.2 Labeling Process

In order to identify the source and target objects that are part of an interaction, SEDalvik assigns them unique contexts through a dedicated labeling tool. That tool provides automation to compute the label requested in the SEDalvik policy. Indeed, our labeling tool computes the security contexts dynamically at run time. Thus, the security officer simply reuses the computed labels. The labeling algorithm runs as follows:

1. Check if the object has a primitive type (in this case, the context is defined in the default policy)
2. Check if the object's type is declared in the policy
3. If no context is found, walk through the class hierarchy to find a parent class with a context and use this context. If there is no such class, the default context, `object_j`, will be applied

For performance issues, a context is not generated multiple times. Indeed, once a context has been created, it is stored in Dalvik's internal structures and will be directly fetched the next time it is needed.

3.5.3 Policy computation

The labeling tool computes the security contexts at runtime. In order to protect an application, a security policy is required. The SEDalvik policy consists of both a list of available contexts and a set of rules.

Our policy tool allows SEDalvik to learn the required contexts and rules. This learning process runs as follows:

1. The learning tool runs a first time
   (a) For each encountered class, generate a context based on the class signature (if the context has not yet been defined).
   For example, when a thread tries to create an object, the following contexts are generated:

   ```
   Ljava/lang/Object;  object_j
   Ljava/lang/Thread;  java_lang_thread_j
   ```

2. The learning tool runs a second time
   (a) For each interaction, get the previously defined contexts that are involved
   (b) Generate an `allow` rule authorizing the interaction.
   For example, the rule for the previous interaction would be:

   ```
   allow java_lang_thread_j object_j
      from (init)
      invoke (init)
   ```

3.5.4 Decision engine

Once the requested interaction is received, the decision engine can determine if the operation should be allowed or forbidden.

To take this decision, the engine takes as an input the security policy in which are declared all the allowed interactions.

Then, it looks for a rule matching the request generated by the application. If a matching rule is found, the decision engine allows the interaction. If no matching rule is found, the engine denies the interaction.

3.5.5 Mobile device improvements

Since SEDalvik is targeting mobile devices, the performance of the solution is an essential consideration.

A major improvement that is made to obtain good performances is to use the METHOD_EXIT events generated by Dalvik's interpreter. This allows the use of a stack of the method calls.

Indeed, when an METHOD_ENTRY event is reached, the caller method and object are pushed to the call stack. On the other hand, when it is METHOD_EXIT event, the last item in the stack is removed. Thus, SEDalvik's reference monitor knows which object/method couple is the caller and avoid fetching the same data several times.

Another improvement consists in a cache that stores the data for the interaction. Once an object or a method occurs again, the cache directly provides the requested information. Thus, the cache avoids multiple calls to the Dalvik's debugger in order to the compute the request and makes SEDalvik faster.

3.5.6 Algorithm

Figure 3 presents the final algorithm used by SEDalvik to control the interactions between Java objects.

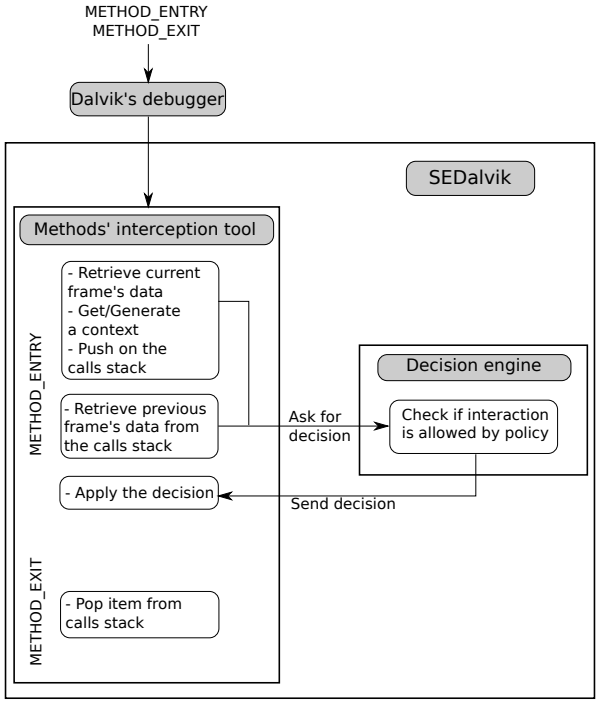

Figure 3: Global algorithm.

When a METHOD_ENTRY or a METHOD_EXIT event occurs, Dalvik's interpreter warns SEDalvik.

If the event is a METHOD_ENTRY, SEDalvik will query the debugger to get the needed data. SEDalvik will also generate a context for the called object if this was not done in a previous call; otherwise, it will only retrieve the existing context into the cache.

Then, SEDalvik gets the previously stacked data to get the information about the caller, and pushes the callee's data on the stack for future uses.

The computed request is then sent to the engine so that a decision can be taken. The decision engine sends back the result so that it can be enforced.

On the other hand, if the event is a METHOD_EXIT, SEDalvik only needs to pop the last item of the stack.

4 Experiments

4.1 Usecase

This usecase is based on a well-known Android security flaw. Indeed, the Android's permission model can lead to privilege escalation attacks where a malicious application can access a component or data belonging to another application without the corresponding permissions.

Figure 4 presents how this security flaw can be implemented with two Android applications.

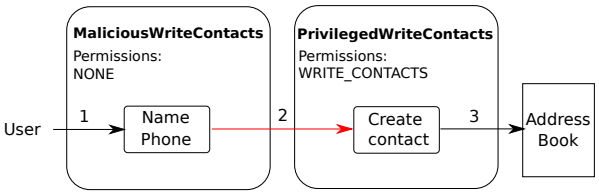

Figure 4: Privilege Escalation Scenario.

The applications involved in the usecase are:

1. A malicious application, *MaliciousWriteContacts*, is forbidden to access the address book.

   This application waits for the user to enter a name and a phone number (interaction #1).

   When both fields are known, the application sends a *intent* containing the data to the second application (interaction #2)

2. A privileged application, *PrivilegedWriteContacts*, has the `WRITE_CONTACTS` permission and can thus access the address book to perform write operations.

   This application can receive an *intent* containing a name and a phone number (interaction #2).

   It will then create the corresponding contact in the address book (interaction #3).

   However, this privileged application is vulnerable. Indeed, since Android's default setting allows any

application to send an intent, *MaliciousWriteContacts* can use the privileged application to create a contact.

These two applications show an example of a privilege escalation attack. Indeed, the *MaliciousWriteContacts* application is able to create new contacts in Android's address book, even without having the right permission.

In order to prevent this attack using SEDalvik, a security policy needs to be defined.

An excerpt from the contexts defined for the usecase can be seen in listing 1.

```
Lpkg/privileged/PrivilegedWriteContactActivity;
     privilegedwritecontactactivity_j
Lpkg/malicious/MaliciousWriteContactActivity;
    maliciouswritecontactactivity_j
```

Listing 1: Contexts Sample

Once the labels have been declared, the policy learning enables to compute a security policy, such as the one in listing 2.

```
allow android_widget_button_j
 android_content_intent_j
 from onClick
 invoke (init)
allow android_widget_button_j
 maliciouswritecontactactivity_j
 from onClick
 invoke startService
allow object_j
 android_content_intent_j
 from createFromParcel
 invoke (init)
```

Listing 2: Policy Extract

The first rule allows a button's event handler to create an intent. The second rule allows this event handler to send the intent (method *startService()*). The third rule allows an object to create an intent from an Android Parcel, i.e. Android's message container for the intents.

The obtained policy is controlling all the interactions between Java objects. Therefore, it is quite large. This usecase needs around 450 security contexts and 10200 security rules. These numbers include the contexts and rules needed by Android to control all the flows of these applications (for instance, the rules to launch an application or to create its graphical interface).

With this policy, the following result is obtained:

```
[traceid=62472;stamp=27241435;pid=592]
 type=allow
 scontext=object_j
 tcontext=android_content_intent_j
 { onClick invoke (init) }
 sinstance=5328 tinstance=7472

[traceid=62507;stamp=27251525;pid=592]
 type=allow
 scontext=object_j
 tcontext=maliciouswritecontactactivity_j
 { onClick invoke startService }
 sinstance=5328 tinstance=1736

[traceid=2044;stamp=31147460;pid=609]
 type=allow
 scontext=object_j
 tcontext=android_content_intent_j
 { createFromParcel invoke (init) }
 sinstance=0944 tinstance=2120
```

Listing 3: Output Extract

The first two traces correspond to the sending of the intent from the malicious application (`pid=592`). The first trace is the intent creation, while the second one is the function sending the intent.

The last trace is from the privileged application (`pid=609`): it corresponds to the intent reception.

SEDalvik can block this attack by changing the policy presented in listing 2. By commenting out the second rule, the *MaliciousWriteContacts* application won't be able to send any intent. However, a better approach is to have an additional control to be able to block illegal intents while allowing legal ones. Indeed intents, passing through an intermediate object that is a native library, need an additional control as described below.

Additional control of native components transmitting the intents To reach the *PrivilegedWriteContacts*, the intent pass through a third element, Android's *Binder* that is not controlled easily since the Binder uses a native library. Therefore, an additional control is required. As a consequence, SEDalvik needs to analyze the Binder activity through the analysis of the traces.

Android captures the traces of the intents as displayed in listing 4.

```
START {act=android.intent.action.MAIN
 cat=[android.intent.category.LAUNCHER]
 cmp=pkg.privileged.privilegedwritecontacts/
 .PrivilegedWriteContactActivity
 (has extras) u=0}
 from pid 592
```

Listing 4: Android's Log Extract

SEDalvik sees that the intent for the *PrivilegedWriteContacts* application is sent by the application with `pid=592`, that is to say *MaliciousWriteContacts*.

For this purpose, Android's Binder is modified in order to handle the intents between two applications.

The simplified sequence of the interactions for our use-case is described in figure 5.

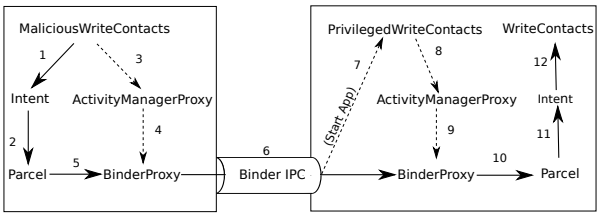

Figure 5: Sequence of Interactions Leading to a Privilege Escalation.

The plain arrows interactions (#1, 2, 5, 6, 10, 11 and 12) are the ones that need to be analyzed in order to block the privilege escalation attack.

Therefore, by generating a trace containing the PID/ObjectID pair for both the caller and the callee during interaction #6, SEDalvik will be able to link the sending and the receiving of an intent. Once the complete sequence is known, SEDalvik has all the elements to block the interaction #6.

4.2 Benchmark

In order to test SEDalvik's usability, the solution was tested on a physical device. The tests were done on a Galaxy Nexus phone, using Android 4.1.2 (Jelly Bean).

Two aspects have to be considered in order to determine the overhead generated by SEDalvik: the time needed to launch the application for the first time, and the time needed for the other runs. The first launch of an application is when Android loads the classes it needs and when SEJava loads its policy (contexts and rules). Therefore, it is always much slower than forthcoming runs.

As a consequence, the launching times will be presented for a "first run" and for "other runs", that is to say for the application's creation (when the classes and SEJava are loaded) and for the next runs. Both cases are compared with and without SEDalvik.

Four situations are considered:

1. The time Android takes to display the *MaliciousWriteContacts* application
2. The time *MaliciousWriteContacts* takes to send an intent to the second application
3. The time *PrivilegedWriteContacts* takes to handle the received intent and to create a contact
4. The time Android takes to display the *PrivilegedWriteContacts* application

The results are presented in figure 6.

They were obtained with *logcat*, the monitoring tool provided by Android. Indeed, *logcat* informs the user of the amount of time an application needs to be displayed. It also indicates the time each intent takes to be sent.

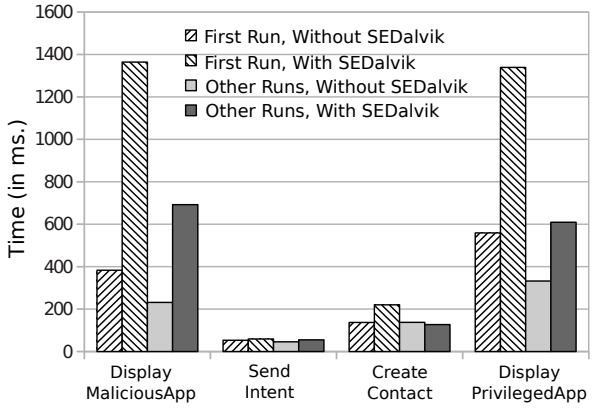

Figure 6: Usecase's performances on a Galaxy Nexus.

As observed, the overhead induced by SEDalvik is more important for the first runs.

Moreover, the overhead for the actual application's functions (sending an intent, creating a contact) is quite small. Indeed, the overhead is mostly due to Android's internal functions and the handling of the graphical interface. Therefore, an interesting improvement would be to only control the core of an application, without controlling its graphical part.

By doing this, SEDalvik's overhead would be cut down without really degrading the offered security.

5 Conclusion

The growing number of Android threats shows the importance of addressing the protection of both the Linux and Java parts of Android. Currently the majority of the threats on Android do not address the Dalvik Virtual Machine since the Linux and native applications of Android are really poorly protected. Thus, hackers currently do not need to attack Dalvik for compromising Android. However, the recent vulnerabilities of the Java Virtual Machines such as the one affecting Facebook and Twitter [4] show that the Java parts will be one of the main concerns for improving the security of Android. Indeed, when the Linux part will be better protected, the attacks will be targeting the Dalvik Virtual Machine.

This paper shows that the Java applications running into the Dalvik Virtual Machine are as vulnerable as the Java applications running into a classical Java Virtual Machine. This paper presents SEDalvik, a Mandatory Access Control implementation for the Dalvik virtual machine, that prevents advanced threats such as privileged escalations between Java objects.

SEDalvik is able to observe all the interactions between the Java objects. Thus, SEDalvik provides a reference monitor guaranteeing that the interactions between the Java objects satisfy the required MAC policy.

SEDalvik eases the management of the MAC policy between the Java objects. SEDalvik does not require the modification of the Java applications nor of the Dalvik bytecode. It is an extensible approach that runs for any application coming from the Google Play. The performances show that improvements can minimize the overhead by auditing only a limited subset of the Java components.

Future works deal first with the connection of SEDalvik to an external tool in order to block efficiently the sequences of interactions such as the one observed with the native Binder. PIGA [3, 1] is such a tool and could be connected to SEDalvik in order to prevent complex malicious activities. Secondly, SEDalvik can be coupled with a protection at the operating system level, such as the one offered by SEAndroid. These two MAC protections are needed to provide an in-depth protection going from the Java to the Linux levels.

References

[1] BLANC, M., BRIFFAUT, J., TOINARD, C., AND GROS, D. PIGA-HIPS: Protection of a shared HPC cluster. *International journal on advances in security 4*, 1 (Sept. 2011), 44–53.

[2] BORNSTEIN, D. Dalvik VM internals. In *Google I/O Developer Conference* (2008), vol. 23, pp. 17–30.

[3] BRIFFAUT, J., LALANDE, J.-F., AND TOINARD, C. Formalization of security properties: enforcement for MAC operating systems and verification of dynamic MAC policies. *International journal on advances in security 2*, 4 (Dec. 2009), 325–343. ISSN: 1942-2636.

[4] CONTI, M., NGUYEN, V., AND CRISPO, B. Crepe: Context-related policy enforcement for android. *Information Security* (2011), 331–345.

[5] DAVI, L., DMITRIENKO, A., SADEGHI, A., AND WINANDY, M. Privilege escalation attacks on android. *Information Security* (2011), 346–360.

[6] ENCK, W., GILBERT, P., CHUN, B. G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), USENIX Association, pp. 1–6.

[7] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 627–638.

[8] HARRISON, M. A., RUZZO, W. L., AND ULLMAN, J. D. Protection in operating systems. *Communications of the ACM 19*, 8 (1976), 461–471.

[9] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 639–652.

[10] NAUMAN, M., AND KHAN, S. Design and implementation of a fine-grained resource usage model for the android platform. *The International Arab Journal of Information Technology 8*, 4 (2011).

[11] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically rich application-centric security in android. *Security and Communication Networks 5*, 6 (2011), 658–673.

[12] RUSSELLO, G., CRISPO, B., FERNANDES, E., AND ZHAUNIAROVICH, Y. Yaase: Yet another android security extension. In *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)* (2011), IEEE, pp. 1033–1040.

[13] SMALLEY, S. Security Enhanced (SE) Android. *LinuxCon North America 2012* (2012).

[14] SMALLEY, S., AND R2X, T. The case for se android. *Linux Security Summit 2011* (2011).

[15] TIM LINDHOLM, FRANK YELLIN, G. B., AND BUCKLEY, A. *The Java Virtual Machine Specification, Java SE 7 Edition*. Oracle America Inc, july 2011.

[16] VENELLE, B., BRIFFAUT, J., CLEVY, L., AND TOINARD, C. Mandatory access control for the java virtual machine. In *16th IEEE Computer Society Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2013)* (2013).

[17] WEIDMAN, G. Bypassing the android permission model. Hack In Paris, 2012.

[18] XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security* (2012).

Notes

[1] `http://www.kaspersky.com/about/news/virus/2012/Oracle_Java_surpasses_Adobe_Reader_as_the_most_frequently_exploited_software`

[2] `http://www.f-secure.com/static/doc/labs\_global/Research/Mobile\%20Threat\%20Report\%20Q4\%202012.pdf`

[3] `http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html`

[4] http://www.forbes.com/sites/andygreenberg/2013/02/15/facebook-hacked-via-java-vulnerability-claims-no-user-data-compromised