# An extension to the Android access control framework

Master's Thesis

**Qing Huang**

Supervisor: Ludwig Seitz, SICS

Examiner: Nahid Shahmehri, IDA


October 2011

# Abstract

Several nice hardware functionalities located at the low level of operating system on mobile phones could be utilized in a better way if they are available to application developers. With their help, developers are able to bring overall user experience to a new level in terms of developing novel applications. For instance, one of those hardware functionalities, SIM-card authentication is able to offer stronger and more convenient way of authentication when compared to the traditional approach. Replacing the username-password combination with the SIM-card authentication, users are freed from memorizing passwords. However, since normally those kinds of functionalities are locked up at the low level, they are only accessible by a few users who have been given privileged access rights. To let the normal applications be benefiting as well, they need to be made accessible at the application level. On the one hand, as we see the benefit it will bring to us, there is a clear intention to open it up, however, on the other hand, there is also a limitation resulting from their security-critical nature that needs to be placed when accessing which is restricting the access to trusted third parties.

Our investigation is based on the Android platform. The problem that we have discovered is the existing security mechanism in Android is not able to satisfy every regards of requirements we mentioned above when exposing SIM-card authentication functionality. Hence, our requirement on enhancing the access control model of Android comes naturally. In order to better suit the needs, we proposed a solution White lists & Domains (WITDOM) to improve its current situation in the thesis. The proposed solution is an extension to the existing access control model in Android that allows alternative ways to specify access controls therefore complementing the existing Android security mechanisms. We have both designed and implemented the solution and the result shows that with the service that we provided, critical functionalities, such as APIs for the low-level hardware functionality can retain the same level of protection however in the meanwhile, with more flexible protection mechanism.

# Acknowledgments

I would like to say many thanks to my supervisors Ludwig and Christian at SICS, for everything, this great thesis opportunity, their full support and the endless patience on me.

I would also like to thank my supervisor and examiner Nahid at Linköpings universitet for her valuable inputs and feedbacks.

In the end, I want to give the deepest gratitude to my family and friends. Thank them all for always staying with me.

# Contents

# Acronyms

| | |
|---|---|
| **ABAC** | Attribute base access control |
| **AC** | Access control |
| **ADB** | Android Debug Bridge |
| **AIDL** | Android Interface Definition Language |
| **API** | Application Programming Interface |
| **CTS** | Compatibility test suite |
| **DAC** | Discretionary access control |
| **DDMS** | Dalvik Debug Monitor Server |
| **DVM** | Dalvik virtual machine |
| **ID** | Identifier |
| **IPC** | Inter Procedure Communication |
| **JVM** | Java virtual machine |
| **LBAC** | Lattice based access control |
| **MAC** | Mandatory access control |
| **MIDP** | Mobile Information Device Profile |
| **ODM** | Original Device Manufacturers |
| **OS** | Operating System |
| **PC** | Personal Computer |
| **PKI** | Public Key Infrastructure |
| **RBAC** | Role based access control |
| **SATSA** | Security and Trust Services API for J2ME |
| **SDK** | Software development kit |
| **SIM-card** | Subscriber identification module card |
| **SICS** | Swedish Institute of Computer Science |
| **SWiN** | Social Networking Wireless Secure Identification |
| **URI** | Uniform Resource Identifier |
| **UTPD** | Untrusted Third Party Domain |
| **WITDOM** | White lists & Domains |
| **XML** | Extensible Markup Language |

# List of Figures

# List of Tables

# Chapter 1

# Introductions

The security issues of mobile networks gain increasing concerns recently. This is partly caused by mobile malwares which are more written targeting our smart phones, and partly due to the access requests to our personal data by many social networking applications which leads to the privacy deficiency. In order to address these security problems, a project called Social Networking Wireless Secure Identification (SWiN) [1] has been carried out at Swedish Institute of Computer Science (SICS), in cooperation with Sony Ericsson and Ericsson. The project focuses on three aspects of security issues in mobile networks: authentication protocols, access controls in Android and privacy considerations.

This thesis is one part of the SWiN project, concentrating on the access models of the Android platform.

## 1.1 Motivations

Many hardware functionalities provide nice features, however, they are at this time only accessible by hardware functionalities providers: manufacturers and a very few privileged users, for example, the phone operators. Our manufacturers have the intention to open them up at the application level since they have seen the benefit it could bring to their third parties. For instance, the SIM-card authentication as one of this kind of hardware functionalities could be used in the novel mobile application to replace the username-password combination. As a return, it offers us a much stronger and more convenient ways of authentication. Specifically, when using this approach the users no longer need to remember passwords, and therefore relief the burden on users. In the context of SWiN project, this SIM-card authentication is to be integrated into a novel secure identification application which requires a stronger user authentication method.

Apart from the advantages, the functions that the SIM-card API exposes are security critical, therefore the access should be limited to a number of trusted third parties. In conclusion, the SIM-card authentication should be made available at

[1] https://www.sics.se/projects/swin

the application level. In the meanwhile it requires to be exposed in a controlled way so that only certain trusted third parties are allowed to use.

Since Android [2] is the platform that the SIM-card authentication functionality will be exported to, we have been studying on its security mechanism to see how this Application Programming Interface (API) could be exposed in accord with our requirements. What we have discovered is the Android platform is not able to offer a good way to solve this problem so far. More detailed information on why Android is not good for exposing the SIM-card authentication has been discussed in Chapter 4.

Another reason for us to look into the Android security is that Android phones have once suffered from malwares attacks. More than 50 malicious applications have been found in the Android market and the most popular one has been downloaded for approximately 200,000 times. Although Google solved this problem by adding security patches to its latest release, we saw the necessity of improving the security of the Android platform.

As we have identified the problem of exposing sensitive functionalities in the Android platform, it seems natural for us to investigate more on the Android security and modify its security mechanism in a way that better suits our requirements.

## 1.2 Methods

The method that we have adopted regarding to understanding the Android security mechanism is primarily based on the literature review and the code browsing.

We started off from the literature reviews. First, we have learned some fundamental concepts of access controls. Then we looked into several classic access control models. Last, we have studied on the related works.

With the help of the related work, we get a general idea of where the security mechanism is hard coded in the Android source code. From there, we take the package manager service as the starting point, which handles all applications' lifetime activities. We pay a special attention to the application installation logics since Android uses the static permissions granting approach and all permissions are granted at the install-time.

Based on the understanding of the existing Android security, we give a solution which allows alternative ways to deal with the APIs that are protected by a specific type of permissions in Android. This is further explained in Chapter 2. We get the input of our idea from a variety of sources, including Mobile Information Device Profile (MIDP) protection domain, as well as related works. We have introduced protection domains and white lists to Android. To find the detailed information of our solution, please read the Chapter 5.

As to the implementation, the main resources we have used are the online materials. Android is an open source project, there are many public tutorials and open technical documentations available. The places we visit the most are Android

[2] Android is the mobile platform developed by Google, see more in 3.1

homepage [3] and Google security discussion groups [4]. We have found that there are also many other technical blogs very helpful.

For the testing, the approach is to verify each part of functionality works properly once being implemented, and the system-level testing has been performed after the integration of all components.

## 1.3 Project goals

In our thesis, we have set up the following goals based on our requirements. Our general goal is to enhance the access control model in Android, and specifically, we are going to fulfill it from the following several aspects.

- Modify the existing access controls in Android in order to adapt the requirements of exposing security-critical hardware functionalities.
- Increase the flexibility of the Android access control model.
- Ensure the access is limited to a number of trusted third parties.

## 1.4 The thesis outline

**Chapter 2** gives the theoretical background of access controls and presents the findings of our study on related works.

**Chapter 3** provides an insight of the Android security.

**Chapter 4** presents the security requirements on the Android access control models that are both from the manufacturers and identified by ourselves.

**Chapter 5** shows the architecture of our design. There are descriptions on what kind of role each component is playing in the system, as well as the explanation on how they are integrated into the Android platform and how they interact with each other.

**Chapter 6** presents the implementation details from the code point of view. The results from the testing are later showed and discussed.

**Chapter 7** is an overall discussion based on the results and findings from the thesis. We have especially explored the opportunities for future works.

[3] http://www.android.com/
[4] http://groups.google.com/group/android-security-discuss?pli=1

# Chapter 2

# State-of-the-art analysis

In this chapter, the background information to the thesis is discussed. We begin with presenting Access control (AC) concepts including a few well-known access control models, and then we move to the Android security. The security mechanism of Android is discussed from the Linux kernel to the application framework level.

## 2.1 Access control basics

Access controls [1, 18] are security policy enforcements at the authorization point of system administrations. They are implemented as part of the system security to ensure the access rights of legal users, in the meanwhile protect system resources from unauthorized access. Security policies are the core of an access control system, which are set by a security manager as a means to accomplish system security goals. The policy is used to describe who can access what kind of resources through which type of operations. It assists the system manager to make decisions when receiving an access request from a user. To define a security policy, there are three parts we need to specify.

#### Subjects

Entities that request to access system resources. Subjects are often human beings and sometimes could be system processes running on behalf of computer users.

#### Objects

They are passively accessed by subjects. Typical objects are files, directories and hardware resources.

#### Operations

They are actions performed on objects by subjects. Take accessing a file for an example, operations are usually read, write and execute.

The access controls can be implemented at different levels of the computer system. At each level, there are specific system resources need to be protected.

#### Hardware level

When talking about the AC at the hardware level, it usually relates to the memory address access. The access is governed by the hardware functionalities.

#### Operating System level

AC at the operating system level addresses the protection to files and pipelines as they abstract the computer hardware and wrap them up as accessible interfaces at this level.

#### Middleware level

The AC at middleware level controls the access from application to application, as well as the access from application to system resources. For example, an application should obtain a permission from the system before using its APIs.

#### Application level

They are internal AC mechanisms of individual applications defined by developers themselves. In this case, the access control is enforced within an application and can be designed to enforce either fine-grained or coarse-grained access controls, and which is totally up to application authors.

There are several well-known access control models, which have been evolved over times. They are recognized as industry practise that developers could follow in terms of implementing their own access control system in their applications. Here we are going to look at four of those.

### 2.1.1 Discretionary access control

One of the most important characteristics of this Discretionary access control (DAC) [10] model is that subjects who is given a permission, at the same time automatically become the permission owner. Because of this, anyone who has been granted certain permissions is able to grant them to other subjects. The limitation is obvious: permission distributions are not centrally controlled and it makes permission revocation almost impossible.

### 2.1.2 Mandatory access control

In contrast to DAC, in the Mandatory access control (MAC) [2, 16] system, there is a security administrator who sets the access control policies, the user however cannot override or change them. All the access requests are checked compulsorily according to those policies, therefore, they are mandatory.

Regarding the policy, Subjects and objects are given labels to describe their levels of security. Subjects are labeled with the *security clearance* whereas objects

are labeled with the *security classification*. Because of the use of security labels, MAC is also known as Lattice based access control (LBAC).

### 2.1.3 Role based access control

Roles in the Role based access control (RBAC)[5, 23, 19], are users with a group of permissions. The associated permissions define what the user is capable of. Permissions can be seen as tuples of objects and operations, where objects specify which part of system resource that can be accessed and operations describe what kind of actions are allowed to perform.

To distinguish from Linux groups which are collections of users, roles in RBAC are collections of permissions. The advantage of RBAC is we can create pre-packaged sets of permissions based on the requirements of a specific job function, independently of the specific subject holding that function. It can thus simplify the management of access rights.

The subject is assigned roles however is not identified by a particular role. It is able to activate a specific role which allows it to use the associated permissions of this role. Moreover, a subject is not restricted to have a single role; instead it can have multiple roles, each of which entitles the user with a distinct job function.

### 2.1.4 Attribute base access control

Attribute base access control (ABAC) [6, 22] uses attributes to specify conditions to be fulfilled when the access is requested. ABAC is finer-grained AC model because various and detailed access senarios are able to be described by specifying attributes fields for them, for instance names and job titles of subjects, types of resources, the environment within which the access is taking place and so on so forth.

# Chapter 3

# Background of the Android security

## 3.1 Android platform

Android is an open source platform, designed for handset devices. They are widely spread in the market today and found mostly in mobile phones and tablets. The development of Android platform called Android Open Source Project is led by Google since 2007. Due to Android's open nature and rich features, manufacturers are able to tailor this platform to their needs in a quick and easy way. The same reason also makes Android the most popular mobile Operating System (OS). A survey of mobile platforms market share early in 2011 [17] indicates that the largest market share is contributed by Android phones.

There are many other reasons for Android becoming the NO.1 mobile platform. From the manufacturers' point of view, they are able to produce mobile devices with spending little efforts on customizations by utilizing rich features offered in Android. As a return, Android phones have been marketing as mainstream phones by many manufacturers which boost the development of Android applications. From the developers' point of view, Google provides well-documented APIs and online tutorials, which makes Android very developer-friendly and easy to start with. Besides, the freedom and equality philosophy of Android encourages developers to create and publish more applications in the Android market [7].

### 3.1.1 Android Architecture

Android is not just a mobile OS, it is a software stack including the middleware and a number of key applications. Figure 3.1 gives an overview of the Android architecture. As we can see from the figure, Android is base on the Linux kernel. It uses several native libraries and employs the Dalvik virtual machine (DVM) for its applications' runtime environment. The middleware are written in Java providing development APIs and the system service. Key applications allow user to use basic phone functionalities.

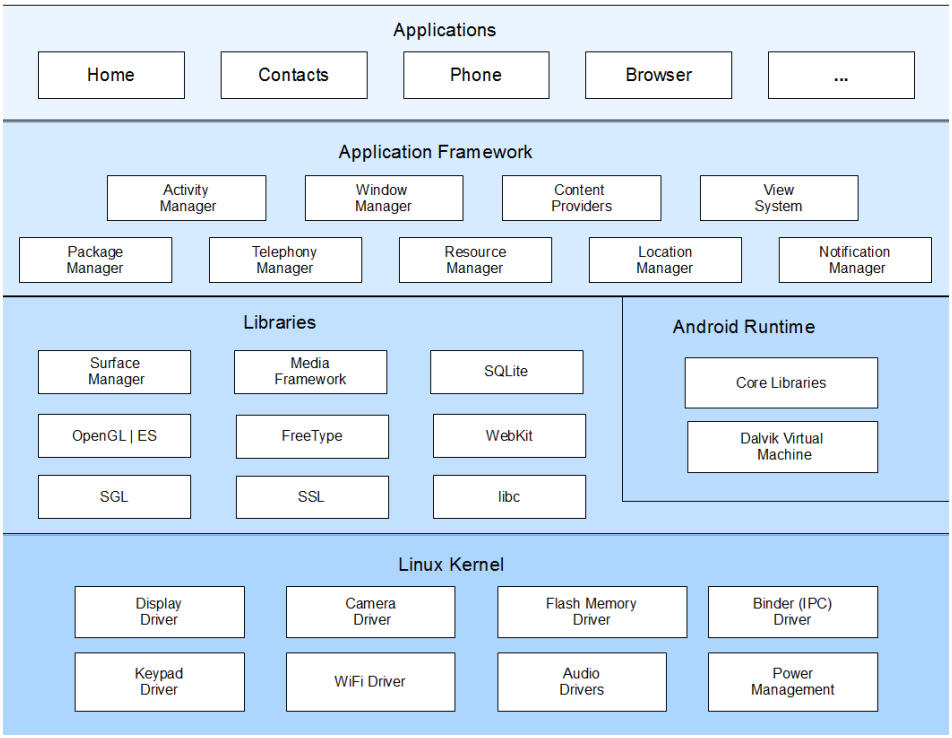


**Figure 3.1.** Android System Architecture [8]

The underlying Android OS is base on Linux kernel 2.6. It is located at the bottom of the system delegating calls to hardware resources.

On top of the Linux kernel are native libraries in use. A variety of libraries are included, from the *Surface Manager* to *libc*, written in multiple languages. Among them, some of them have been tailored to better suit resource-restricted devices.

Android runtime uses Dalvik virtual machine (DVM) instead of Java virtual machine (JVM) to handle the process management. JVM requires heavy computation will have serious impact on system performance. To solve this low memory constraits, DVM has been specially designed to replace it for the Android platform. Process isolations that a vitual machine offers are viewed as one of security enforcements points in the Android platform. Further discussion on this part can be found in the section 3.2.1.

The application framework defines the system service and developer APIs. The system service mediates the access to a variety of low-level functionalities while the framework APIs wrap up the system functionalities in components and make them reusable. Android defines its own static permissions at this level which is used to govern the access between application components. More information of Android permissions are provided in the section 3.2.3.

All installed applications are placed in the Android application layer. A wide range of applications can be written by using developer APIs or reusable components provided by other applications. Originally, Android has already included several applications with basic functionalities such as making phone calls and managing contacts. However, more space is left for developers to fill in. Developers are encouraged to take full advantage of Android's features when writing their own applications.

### 3.1.2 Components of the Android application

An Android application is made up of several types of components. We list some keys components and focus on the security related parts. More information can be found in the online Android developer guide [9].

#### Activities

Activities provide visible screens that mobile users can interact with. An activity is also responsible for monitoring and reacting to the operations that a user have performed on the screen.

The life cycle of an activity includes several states. It begins from `onCreate()` and ends at the time when `onDestroy()` is called. After an activity has been created, `onStart()` is the point that the activity becomes visible to users. `onResume()` also shows a state the activity is visible, however different from `onStart()`, it restores a previous state. `onPause()` represents a state that the current activity is placed in the background, it is active and ready to be brought back into focus at any time. Though the activity at the state of `onStop()` is still alive, it is disattached from the window manager and can no longer be restored.

The activity which is started at the application launch time is called the main activity. An application can have a series of activities and one activity is capable of creating another one. When a new activity is started, the old one won't be killed; instead, its state is pushed into the stack. The old activity will be restored by retrieving its state and regain the focus if the user navigates back.

#### Services

Services work quite similar to activities, the only difference is that the service usually runs in the background and performs a long term task; As a result, it doesn't provide any graphic interfaces.

Services can be started in two different ways. Calling the method, `startService()`, allows us to run an independent task, the service quits automatically when the task is finished. The other way to start a service is through application bindings. A bound service is subjected to an application, thus the application has to decide when to active it and when to kill it.

#### Content providers

Content providers work as the database for the application. The data in content providers can be shared across applications but only when the access is allowed. The application is also able to use the public content providers managed by Google. When storing data to the content provider, the user needs to specify the name of the data by following the Uniform Resource Identifier (URI) scheme so that the data can be identified and retrieved by name.

#### Broadcast receivers

Registering a broadcast receiver lets our application listen to a particular state of either the system or other applications. They are especially useful when we want to activate some service at a specific point. Supposing we want our application to get started as soon as the phone is finished with initialization. If we register for receiving the broadcast of the phone boots completed, we will be notified at that specific point and we can then ask the system to launch the application for us.

The notification message sent between is called *intent* in Android. It is serialized when it is sent. The message consists of the data together with the operation that will be performed. Intent filters are used to filter out unwanted intents so that users are informed by interested ones only.

### 3.1.3 Configurations of Android applications

The AndroidManifest.xml file is the configuration file of the Android application. It specifies the components that the application owns and external libraries it uses. As to the Android permissions, it declares permissions it requests as well as permissions that are defined to protect its own components. The structure of AndroidManifest.xml is shown in figure 3.2.

```
<manifest xmlns:android= http://schemas.android.com/apk/res/android
    package="com.example.android.wiktionary"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/app_icon" android:label="@string/app_name"
        android:description="@string/app_descrip">

        <!-- Browser-like Activity to navigate dictionary definitions  -->
        <activity
            android:name=".LookupActivity"
            android:theme="@style/LookupTheme"
            android:launchMode="singleTop"
            android:configChanges="orientation|keyboardHidden">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:scheme="wiktionary" android:host="lookup" />
            </intent-filter>

            <intent-filter>
                <action android:name="android.intent.action.SEARCH" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>

            <meta-data android:name="android.app.searchable" android:resource="@xml/searchable" />
        </activity>

        <!-- Broadcast Receiver that will process AppWidget updates -->
        <receiver android:name=".WordWidget" android:label="@string/widget_name">
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
            <meta-data android:name="android.appwidget.provider"
                android:resource="@xml/widget_word" />
        </receiver>

        <!-- Service to perform web API queries -->
        <service android:name=".WordWidget$UpdateService" />

    </application>

    <meta-data android:name="android.app.default_searchable" android:value=".LookupActivity" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-sdk android:minSdkVersion="3" android:targetSdkVersion="4" />

</manifest>
```

**Figure 3.2.** AndroidManifest.xml [4]

## 3.2 The Android security

The Android security lies both in the Linux kernel and in the application framework. The security inherited from Linux is the user ID. Along with this, at the framework level, a mandatory access control mechanism is enforced by Android permissions to control access between components.

### 3.2.1 Linux kernel

Different from Linux where users are people who login the system [11, 20, 21], users however correspond to individual applications in Android. Each Android application is assigned a distinct Linux user Identifier (ID) when it is installed. Based on its identity, the system allocates a unique process for it to run within. More importantly, the user ID of an application is used to distinguish itself from others and stays valid throughout the lifetime of this application.

For the runtime environment, Android employs Dalvik virtual machine (DVM) instead of JVM for the reason that DVM is more lightweight and has low memory requirements. The Dalvik virtual machine (DVM) is developed for resource-restricted devices such as mobile phones.

DVM provides most of functionalities that JVM does. It offers process isolation and allows multiple instances running at the same time. The difference is that DVM runs executable file in dex format. However the Android application is written in Java and compiled by the Java compiler, to run in an instance of DVM, the system needs an additional step to translate the compiled Android program to dex executable file. A tool called dx in Android Software development kit (SDK) is dedicated for this.

The DVM also has contributions to the Android security. The code isolation it provides can minimize the damage that is potentially caused by a compromised application. Even if a malware is mistakenly installed in the system, its capability is limited within the process, therefore, preventing it from taking over control the whole system.

For a large scale application, it might not be practical to pack everything in one package, Android offers a flexibility to allow different packages running in the same process, but the exception is only given to two packages, and the condition is that two packages should be signed by the same key. After being verified, a `sharedUserId` is given to both of them, which determines a shared process for them to run. The `sharedUserId` is related to one type of permissions in Android and we will revisit them in section 3.2.4.

### 3.2.2 Android applications signings

Android requires every application to be signed. The main purpose of application signing is to distinguish applications from one to another. For individual developers, they always do the signning with their own private keys. The private keys are supposed to stay secret and known only to their owners. After a signed applicaiton

is installed on the phone, the system is able to use its signature information to distinguish it from other application.

### 3.2.3 Android permissions

Permissions are the core concepts in the Android security. One thing we need to notice is that Android permissions are completely different from Linux file permissions. Existing in the system in forms of strings, they have been used widely to control the access from one application component to another. Permissions are involved in quite a few places.

All permissions are granted at install-time. In order to be grant a permission, it should be requested in the Android manifest file when specifying properties for an application. the system then evaluates it and makes a final decision on whether to grant or deny.

After the application has been lauched, permission checks are enforced before the actual access take place. For instance, an online game can never really be connected to the internet if it is found missing a internet connection permission.

The system has provided developers more than 60 built-in permissions. They are defined in the form of `android.Manifest.permission.X`, where `X` is the name of a particular permission. In addition to the built-in permissions, developers are also allowed to create their own permissions (called dynamic permissions in Android) through permission declaration in *AndroidManifest.xml*.

<*permission*> is the place where developers are able to define their permissions for protecting their application-specific APIs or components. The name of permission needs to be globally unique and descriptive so that other components are able to know and request it by name. The picture 3.3 shows parameters associated with a permission.

```
<permission android:description="string resource"
            android:icon="drawable resource"
            android:label="string resource"
            android:name="string"
            android:permissionGroup="string"
            android:protectionLevel=["normal" | "dangerous" |
                                     "signature" | "signatureOrSystem"] />
```

**Figure 3.3.** permission declaration [8]

<*use-permission*> lets the developer to request a permission so they get access to certain functionalities in the system. It could either be a built-in permission or a dynamic permission. By default, an application has no permission associated with it, thus all permissions should be requested explicitly. The figure 3.4 shows an example of how we request a permission in the manifest file.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

**Figure 3.4.** permission requesting [8]

### 3.2.4 Android protection levels

Android has four protection levels. The protection level is a parameter of a permission and needs to be specified when defining our own permissions. Each level of protection enforces a different security policy. From weak to strong, we have normal, dangerous, signature, and signatureOrsystem protection levels.

#### Normal

Normal permissions are the default setting, providing the weakest protections. They are often used to protect less security-critical functionalities. If the protection level is not specified, the permission is assumed to be normal. When a normal permission is requested, the system grants it without asking users. However, the users are able to check some of the granted permissions in its application properties if he/she wishes.

#### Dangerous

Dangerous permissions are the ones decided by phone users. Permissions at this level might ask for accessing the user privacy or certain hardware service. An example for dangerous permissions is asking for accessing some functionalities cost money. When an application requests a dangerous permission, the system shows the permission information in a screen to users and users need to accept all permissions if they want to install the application on their phones.

#### Signature

This protection is evaluated and decided by the system without the users' involvement. To be granted a signature permission, the requesting application to be signed by the same key as the application that the permission protects. To explain it in another way, the condition for being granted the access between two packages which used signature permissions is those two packages need to be signed by the same key. The *sharedUserId* is discussed in the section 3.2.1.

#### SignatureOrSystem

SignatureOrSystem permissions have two conditions,satisfy either one of them, the access is granted. One of the conditions requires the package resides in the system

image while the other one requires the accessing package to be signed by the same key as a package resides in the system image.

Google uses a static permission approach for its security. In Android, all permissions are requested and granted at install-time. Once being granted, they cannot be changed and will be valid throughout the lifetime of this application. The granted permissions can be checked by the system before the actual access takes place or can be explicitly called by developers.

## 3.3 Related work

We have found related works in three scientific papers. All of them have modified and extended the Android access control model, however focusing on different factors and taking different approaches.

### 3.3.1 Saint

Machigar Ongtang et al. [14] have observed that smart phones become more common nowadays, the mobile platforms are however lack of security in many ways. After giving considerations on a few typical mobile phone scenarios, they have gathered a list of security requirements that are mobile phone specific. Consequently, they propose a framework called Saint, based on the Android mobile platform. What Saint provides can be summarized as two parts: first is the refinement of the install-time permission checks and second is the dynamic access control enforcement that governs the runtime behaviors of applications.

Saint defines a set of policies, from different abstraction levels of the Android OS, as supplimentary security policies to the original ones. See Saint policy tree in figure 3.5.

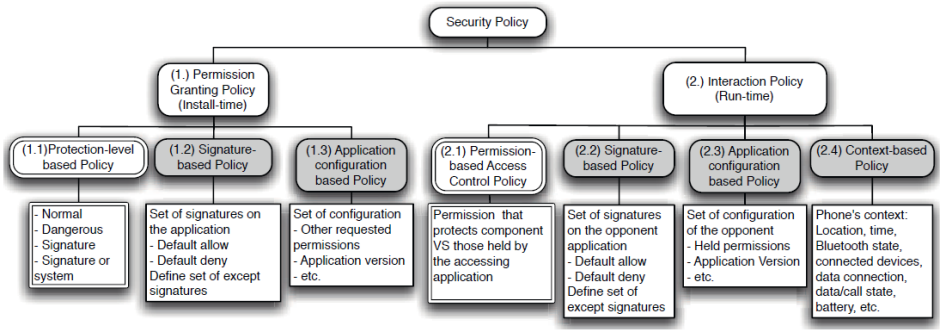


**Figure 3.5.** Saint Policy [14]

For the install-time policy, the modifications and integrations occurred mainly in the following two parts.

1. The signature based policy is modified to allow the client to define a set of except keys. The except keys may have their attributes been set to either *default allow* or *default deny*. At permission granting point, the signature of an application is checked against the key lists, if its key can be found in the default allow list, all permissions are granted. In the contrary, the permissions are always denied if the signing key is in the default deny list. When the key is found belong to none of the lists, the application then needs to go through the standard permission checks.

2. Configuration policy which is the newly added enforcement imposes the security checks on other properties of an application, such as version numbers or user-defined permissions.

As to the run-time policy, it consists of four parts.

1. Modified Permission based policies, used to control the access via Inter Procedure Communication (IPC)s (Inter-procedure communication).

2. Signature based policies at runtime.

3. The application configuration based policy.

4. The context-based policy.

In Saint, two types of policies are defined for the IPC security. The access policy controls over whether the callee can receive or initiate an IPC. The expose policy verifies whether or not the callee is legal to receive the IPC. To establish a successful IPC, access conditions on both sides should be satisfied.

Context-based policy describes the access control decision is made dynamically when the context of the mobile phone changes. The system is able to detect the switches of phone states, such as the location, the current time, Bluetooth etc. Permissions are granted or revoked based on that information. For example, the camera should be disabled when an employee's phone has been detected in the company building.

### 3.3.2 CRePE

CrePE [3] presents a concept of a context sensitive security for Android. The key idea is to use the current state of the phone, including its location, time, and temperature and so on as the supporting information to make decisions. Different from other electronic devices, the high mobility characteristic of our phones makes the context of mobile phones vary from time to time. The author argues that the context needs to be considered as an important attributes when defining security policies.

Unlike Saint, CRePE is user-centric, it is up to the users to define what kind of security policies should be applied to their phones when the context switches. The context are described by several attributes like the time and location as we mentioned before. Besides the security policy definition, the user is able to activate and deactivate policies. At the time of enforcing the CRePE security, the state of

phone is discovered at runtime so that the system can enable a specific policy for it.

The CRePE security is integrated into the Android platform by hooking the service to permission checks. Some existing code has been modified to include the CRePE components. See its architecture in figure3.6 for an overview.

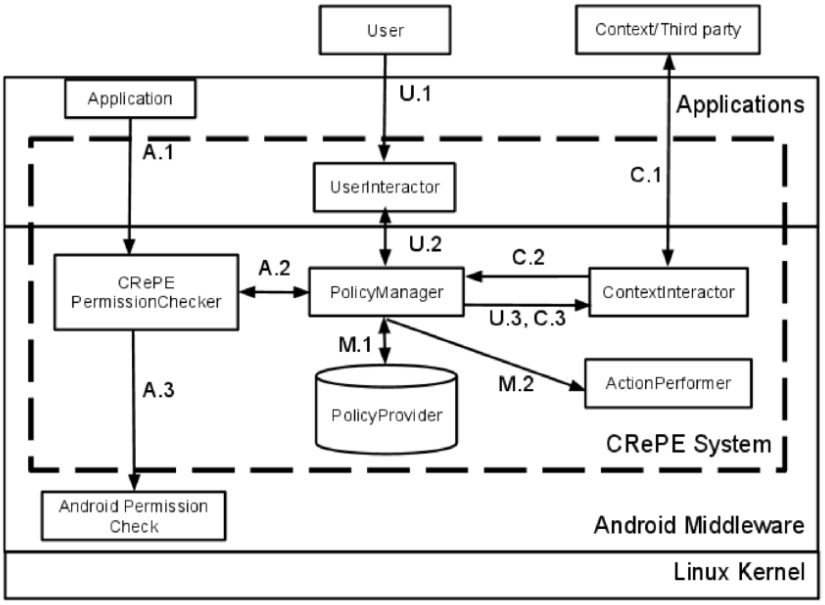


**Figure 3.6.** CRePE Architecture [3]

### 3.3.3 Apex

The authors have seen the limitations in the Android security that the users have no choice but to accept all requested permissions if they wish to use an application. They think using dangerous permissions are risky because many applications tend to ask for more permissions than it needs, and this could be even more dangerous if being made used of by a malicious application. In order to have a finer-grained permission granting mechanism to applications, they insert their Apex [12] security service to Android.

Apex has two core features. The first one is user-friendliness which is evident in the user interface they have implemented. It lets the user decide which permissions are allowed and which one should be denied. The other is runtime constraints, which enables dynamical granting and revoking permissions.

Although Apex framework has modifications and extensions in a number of places of the original Android code, another point worthy to mention is Apex is backwards compatible with the existing Android security.

Apart from the modification at the middleware level, Apex has extended the application installer to let phone users specify security rules. Each rule states the condition of granting a permission and is stored in the policy repository. The repository is consulted when permissions are requested.

# Chapter 4

# Requirements of Android Access controls

This chapter describes the requirements gathered both from the Original Device Manufacturers (ODM) side as well as the ones identified by ourselves. The discussion starts with a problem that our ODM is currently facing. Then, we analyze interests of different stakeholders. In the end of this chapter, we state goals that we are aiming for.

## 4.1 Problems and security requirements

### 4.1.1 Malicious Applications

Android has previously been affected by malicious applications. The news from BBC [13] shows that more than 50 apps have been found malicious in the Android market and they might have been downloaded for up to 200,000 times. Google eventually solved this problem by providing tools to remove the malware and adding security patches to their latest OS version. The incident shows that Android is potentially vulnerable in the complex mobile environment.

### 4.1.2 Issues with security-sensitive APIs

In addition to the malware threats, ODMs discovered the inflexible security design of Android when they intend to make security critical APIs available to developers at the application level. One example of them is the SIM-card authentication API. This API exposes the hardware functionalities and needs to be handled more carefully. It is initially made only available to handset manufacturers and the mobile phone operators. Operators are considered as privileged users and are given access to low-level functionalities.

Considering novel mobile applications such as mobile social networks could benefit from this SIM-card authentication API as well, ODMs intend to open it

up but only to a limited number of third party users. However, there are following security concerns that need to be solved beforehand.

- Android treats every application equally and they can be published freely on the market, which also leaves a hole for malwares.

- Four levels of protections, *normal*, *dangerous*, *signature* and *signature or system* in Android are not very well designed and make them inflexible to use in some occasions. There are gaps between need to be filled .

  Permissions at *dangerous* level require users to confirm on every requested permissions, causing a problem by putting too much security responsibilities on users. Since there might be some users who simply want the application works on their phone, and would consequently accept all permission requests without knowing what these permissions really can do. An even worse situation is that some malwares might take advantage of it to exploit the system. For the SIM-card authentication, any kind of deficiencies are not tolerant. Hence the dangerous permission is inadequate to protect security-critical APIs.

  The *signature* permission offers stronger protection than the *dangerous* permission due to more security constraints it has. The system makes decisions for mobile users instead of leting the user take the control. However, using the signature permission in our case turns out to be very inflexible because the condition of being granted a signature permission requires two packages signed by the same key. This is the problem identified by our ODM.

  We use a simple scenario to demonstrate how the SIM-card authentication is used when it is protecte by a signature permission. The SIM-card authentication API is developed and provided by our manufacturer. There is a third party application which implements this API. In order to be installed successfully, this third party application needs to be signed by the ODM instead of the developer since this API checks the application has the same signature. This creates a problem for ODMs to approve and sign every third party applications if they implement this API. Signing for a third party application is obvious cumbersome and risky. Therefore, using signature permissions are also unable to full satisfy requirements from ODM.

  As our ODM have seen many problems when applying Android permissions to sensitive functionalities. They think it is still not a good time to open them up at the application level and a solution to enhance the existing Android security need to be made.

## 4.2 Stakeholder of interests

Based on the problems we have observed, we sketched a use case diagram to help identify the requirements. See the diagram in figure 4.1. The scenario is described below from different stakeholders' point of view.

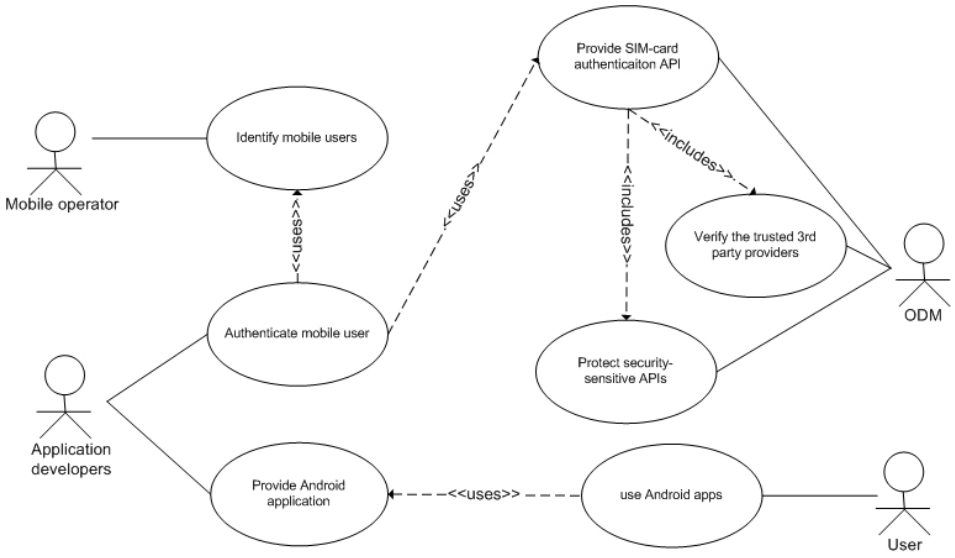


**Figure 4.1.** Use case diagram for the SIM-card authentication

### 4.2.1 Original Device Manufacturers

The ODM is the developer and provider of security sensitive API, such as the SIM-card authentication API. They have the intention to make these API available at application level while at the same time, limit the access to certain trusted third parties, to prevent them from being misused.

The ODM is supposed to decide who is allowed to access the security-critical API and who is not. Their responsibility also includes verifying and authorizing a third party user.

### 4.2.2 Mobile users

As the owner of the mobile device, users should be able to know what is going on in his/her phone. For the granted permissions, regardless of the protection level of permissions, they should be kept in a central pool where users can keep track of the information of the granted permissions whenever they desire to. Though now in Android, users can go to the setting and view the properties of the installed applications, the traceable permissions just include the *normal* and *dangerous* permissions. Permissions that are decided by the system like the *signature* and *signatuerOrsystem* permissions, however are never shown.

### 4.2.3 Application developers

Application developers are direct users of security-sensitive APIs. The SIM-card authentication is one of those functionalities that developers can choose to imple-

ment in their novel mobile applications. Replacing the traditional authentication approach, i.e. the username-password combination with the SIM-card authentication can give us a stronger way of authentication. Through the user point of view, they would also benefit from this since they do not need to remember passwords any more.

### 4.2.4 Mobile network operators

The fourth category of stakeholders is the mobile network operators. Originally, they are the only users who have been given the privileged access to the SIM-card authentication API from the low level. We can say they are are ultimate users who have been given the maximum access rights.

Other than being a user of some sensitive hardware functionalities, they are the potential providers of APIs which expose network functionalities, and in this case, they are in the equivalent position of ODMs.

Those are not the only reasons to make them important. As the phone operator, they are administrator of mobile networks where the authentication actually takes place.

## 4.3 Initial ideas

Before we present our ideas, we want to give some background information on MIDP [15] since that is where our idea initiates. MIDP is the development tool kit of Java ME [1]. With its help, developers are able to implement mobile applications running on Java platform.

Being shipped with MIDP, there is an optional API called Security and Trust Services API for J2ME (SATSA) being offered to developers as an interface to use the SIM-card authentication hardware functionality. It is up to evelopers to choose whether to implement or not depending on their needs. As an good example,SATSA makes the SIM-card authentication API available at application level, we see a possibility to do this for the Android platform as well.

The mechanism of granting permissions in MIDP works as follows: Each application is bound to a protection domain at install-time. The application is granted with the associated permissions of its domain. Any permissions beyond the domain permissions are denied. The advantage of doing this is the system can restrict the behavior of the application in a predicable way, preventing the application from gaining some privileged permissions, therefore protecting the system from being potentially exploited by malwares.

Unfortunately, we cannot directly apply the security domain of MIDP to Android. The main reason is that MIDP is dedicated for developing mobile applications on the Java mobile platform whereas Android is totally different from it. Although one may argue they have many in common since both of them use Java, we should also notice that the differences are larger. Android is not fully written in Java, only the framework layer is implemented in Java. Besides, it includes

[1] Java Platform, Micro Edition

several native libraries written in other languages. Android provides a whole set of APIs to developers, independent of Java platform. Most important, an Android application is zipped in an apk file other than a jar file for Java applications. All of these differences prevent us from reusing the security domain component from MIDP. However, since MIDP describes the API for the SIM-card based authentication, we gained insights about the ODM requirements from this profile and could bring some of their ideas to our design.

By studying the mechanism of MIDP and brainstorming, we have some initial ideas of the protection domains for our design, and they are listed and explained below.

### 4.3.1 Security domains

**What are protection domains?**

- Security domains are associated with a collection of permissions that allow us to carry out some application behaviors.
- The behavior can be described as a group of operations.
- The application should be assigned to a specific protection domain at the install-time.
- The application can be granted with the requested permissions if they belong to its domain permissions.
- If the application asks for a permission that do not belong to its domain permissions, the request is denied.

In conclusion, in order to be granted with the access right, the requested permissions of an application should always be a subset of its domain permissions.

**How the protection domain works?**

- The protection domain assignments happen at the application install-time. Every application should be appointed to a specific protection domain.
- Once an application is bound to a specific domain, it won't be able to change unless the application is uninstalled.
- If an application found belongs to several different domains, we assign it to a merged protection domain which is associated with permissions from all found domains.

### 4.3.2 The list of third parties

The list of third parties is referred when we need to determine which protection domain the installing application goes to.

- The white list is used to keep track of our trusted third parties. It describes who are allowed to access the sensitive device APIs.
- The target users of the third parties list are device manufacturers and phone operators. In order to let them manage the list, a configuration interface needs to be provided to them.
- The white list needs to use a distinct attribute of the application which allows the system to distinguish applications. Our initial thought is to use the certificate information of the application, since the application signing mechanism has already been used in the system for identifying the application, and it could be reused here for the same purpose.

### 4.3.3 Usability

- An explicit requirement from our ODM is the modified Android platform should stay backwards compatible. The backwards compatibility ensures that an application can be installed successful on an Android phone regardless of its version. As for our case, we want our solution works not only on a specific version of Android but also on other versions. Hence, the backwards compatibility is an important goal for us to achieve.
- As we have already discussed before, a configuration interface should be provided to our ODMs and operators so that they are able to maintain the third parties who have been given the access rights to their sensitive APIs.

## 4.4 Goals

The ideal solution shall fulfill the following goals.

- We provide an alternative way of dealing with the signature permission in Android. The new approach should be flexible and allow the ODM keep using signature permissions to protect their sensitive hardware functionalities.
- Third party developers who write applications that use the device API should be able to sign their applications with the own developer keys. When their applications are installed on the Android phone, the system should be able to identify they are trusted by our ODMs or operators, therefore, being granted the privileged access to signature protected APIs.
- Provide a configuration interface to sensitive functionalities providers, in our case, they are ODMs and phone operators. The configuration interface should allow them to include new trusted third parties and disable the existing ones. Our ODMs could have several sensitive APIs exposed, but they might want only one of them to be accessible by a particular third party. In this case, we let ODMs to define the specific APIs that each third party could access.

In summary, the general goal for us is to enhance the existing Android security model and keep the modified Android OS staying backwards compatible. The given solution complements the existing security mechanism and offers alternative and flexible ways of dealing with signature protected functionalities.

# Chapter 5

# The Extension design of Android access controls

This chapter presents the design points of security extensions based on the latest release of the Android OS: Android 3.2. Our extended access control solution has introduced protection domains and white lists to the Android OS. In our design, we keep the original static permissions granting mechanism. Evaluating and granting all permissions at the applications' install-time.

We begin with an overview of the high level architecture of WITDOM. In order to make it easier to understand how our WITDOM service is integrated into Android, we give some descriptions to the relevant packages of Android. After that, we explain what modifications that have been made in them and why. Our WITDOM components are described in the end.

## 5.1 Access control extension architecture

Our raw idea has already been presented in the chapter 4. We want to introduce the white lists and protection domains to the Android so we name our solution WITDOM which stands for WIhIte lists and protection DOMains for Android.

The architecture of WITDOM is based on the current access control mechanism of the Android platform. With respect to the requirements, it is designed in a way open for extending and backwards compatible. The WITDOM service consists of three parts, and they are `DomainManager`, `Domain`, `Whitelist` respectively.

WITDOM complements the access control mode of Android. We use the protection domain to restrict the behavior of applications and consult the white list to see which protection domain one application should be assigned to.

The domain binding is hooked into the package installation routine. Specifically, it evaluates the requested permissions from the installing application before the regular Android permission checks. In the Android source code, the application installation is handled by a method called `installNewPackageLi` which is located in the class called `PackageManagerService`.

We use the service hook to direct the installing application to our WITDOM service after it has been parsed by the package manager. Once we bound the application to a specific domain, it is returned to the package manager. The package manager continues installing the application till the point the Android permissions checks are called. We trap the application to our WITDOM service again, using the domain permissions to evaluate the requested permissions then making decisions based on the evaluation results.

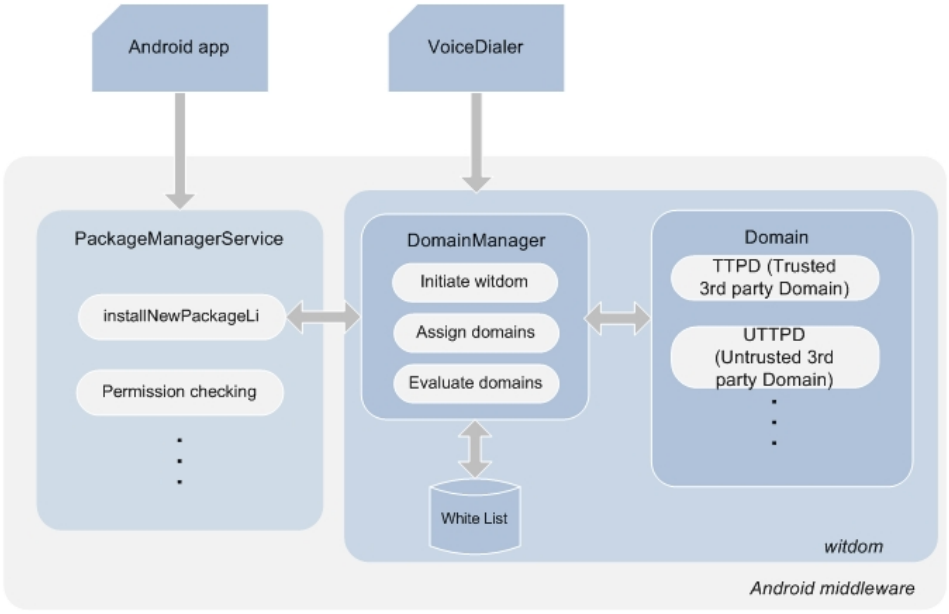


**Figure 5.1.** WITDOM extension architecture

WITDOM structure is illustrated in figure 5.1, showing how different components interact with each other and how WITDOM is integrated into the Android OS. The packages labeled with WITDOM are the extension we have designed for Android. The rest parts belong to the original Android OS where some modifications are made to accommodate the WITDOM service. The WITDOM service resides in the framework layer of Android which is written in Java, therefore, we naturally use Java in our implementation.

Since the design is not independent from other Android packages and methods, we will first go through the existing classes that are interested to our WITDOM service.

For each Android application, there is an `AndroidManifest.xml` used to describe package properties, so that when it is installed, the system could know how to allocate system resources, assign process, link components as well as other handling for this application. More importantly, this is also the place where permissions are declared, the part we are most interested in.

### 5.1.1 PackageParser

The information located in `AndroidManifest` file is parsed at the application install-time and the system service `PackageParse` is dedicated to extracting this information. The package parsing happens before the real package installation and the parsing steps are described in the section 5.1.3.

### 5.1.2 PackageManagerService

`PackageManagerService` manages the lifecycle of an application, from being installed to being removed. One thing we need to pay attention is that permission granting only happens at the application install-time, permission checks however can called at several places during the application runtime. More discussion is provided in the next section.

### 5.1.3 Applications installation steps

The figure 5.2 shows the method calling flow when installing a new package.

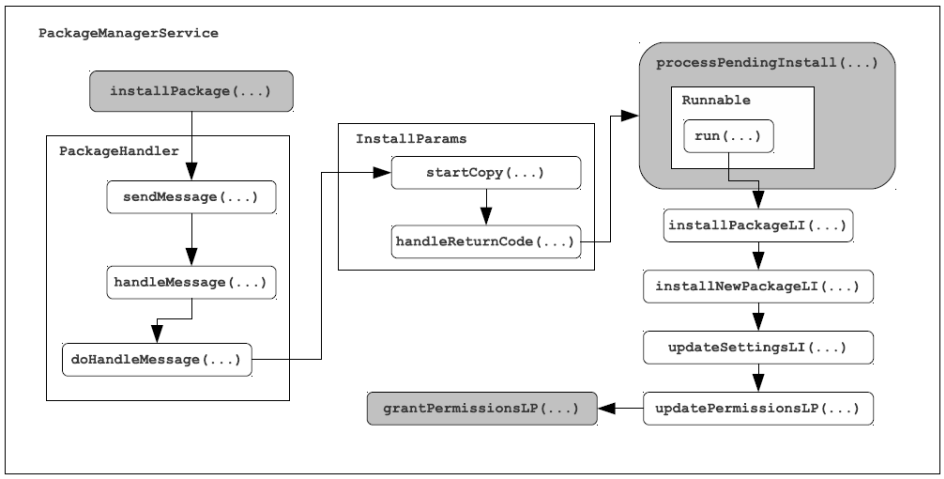


**Figure 5.2.** Methods calling flow of the application installation

1. The `PackageManagerService` gets a request of installing an application from the user.

2. The `PackageManagerService` creates an instance of the `Package` to store all data read from the new application. The package parsing is taken care of by `PackageParser` including extracting the source code of the application as well as parsing the configuration information from the `AndroidManifest.xml`

3. When the `PackageParser` has been called to parse the `Package` object, it extracts information of the `AndroidManifest` file and stores them to the corresponding parameters.

4. When the `PackageManagerService` gains back the populated package instance, the actual installation begins. The installation starts with checking the disk space which ensures there is enough storage for the application, and then the package is examined to see whether it is a new one or an update.

5. The `PackageManagerService` also needs to assess the requested permissions, and decisions are made by calling the Android permission checks which evaluates the permissions according to Android security policies.

6. After being granted with the requested permissions, the executable file is copied to the system. An instance of DVM is created to host the running process of this application. Till this point, the installation is finalized.

## 5.2 WITDOM design

The WITDOM service comprises three main parts: the white list, protection domains and the domain manager. Each component is detailed described below.

### 5.2.1 Target users

The target users of the WITDOM service are ODMs and phone operators. The WITDOM service is designed to control the access to security critical functionalities at the application level that are provided by ODMs and phone operators, thus they are our target users.

### 5.2.2 The white List

The white list is used by the system when it needs to decide a specific protection domain for the installing application. It associates the unique application information to protection domains so when we look up the list, we are able to know which trusted third parties are allowed to be given the access to certain sensitive functionalities. The unique information of the application we use here is the certificate. Certificates are chosen for two reasons. Firstly, the certificate contains a public key that can be used to verify the identity of application authors. Secondly, it contains the information of the certificate issuer which can be used to verify those non-self-signed applications.

The white lists are assumed to be managed statically by ODMs, mobile operators or other security critical functionalities providers. The configuration is done through the Extensible Markup Language (XML) files that have been placed in a built-in Android application. The structure of the white list is shown in the figure 5.3.

At the install-time of an application, its protection domain assignment is made base on the result of the white list lookup. The WITDOM service will first find

```
<whitelists>
    <whitelist name="ODM_WHITELIST">
        <owner domain="Original Device Manufacturers Domain">
            <certificate>MIIDqjCCApKgAwIBAgIES+BgtzANBgkqhkiG9w0BAQUFADCBljE.....</certificate>
    </owner>
    </whitelist>

    <whitelist name="MNO_WHITELIST">
        <owner name="Mobile Network Operators Domain">
        </owner>
    </whitelist>
</whitelists>
```

**Figure 5.3.** The configuration file of white lists

its certificate and compare it to the white list. When the corresponding domain has been found, we assign the application to it. If none protection domain can be found, our service then extracts its issuer certificate and does the same white lists lookup. If we still cannot find a protection domain for this application, we assign it to the untrusted domain which is associated with no permissions.

Please notice that we might end up in domain conflicts since the certificate of an application can be allowed in several different protection domains. To deal with this conflict, we combine all found domains together. The merged domain is associated with permissions from all found protection domains.

### 5.2.3 The protection domain

A protection domain is associated with a set of permissions which draws the boundary of operations that the application is allowed to perform.

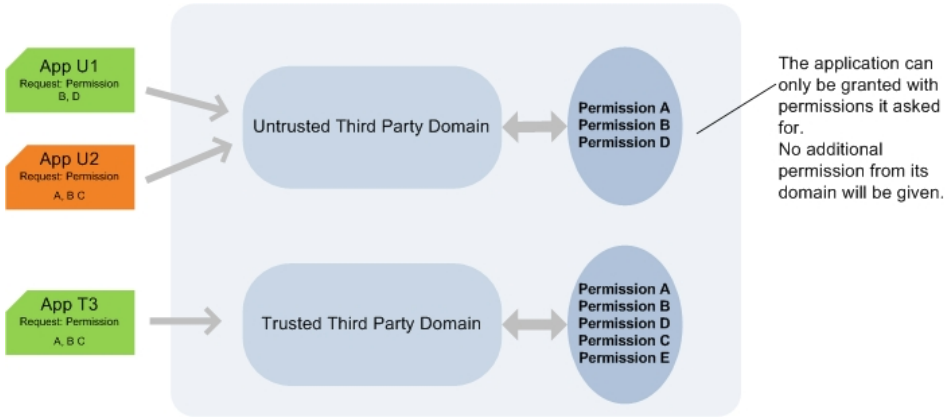


**Figure 5.4.** Protection domain

One thing we should notice here is the protection domain that we defined for our WITDOM service is different from the security domains in MIDP. The security

domain in MIDP defines that when an application is assigned to a protection domain, it automatically gets all permissions associated with the domain. However, our protection domain does not grant additional permissions than the applications request for. The associated permissions are only used to evaluate if the application is requesting any permission beyond its domain.

The WITDOM service works as an alternative way to the *signature* permission checks. We made this design choice since we want to change the inflexible mechanism while at the same time keep the system as original as possible.

We also choose to keep the static permission approach in our design since this approach is enough to fulfill our requirements. However, our design is open for extending, the dynamical permission granting will not be difficult to implement if we find it is needed.

The protection domain is also managed through a XML configuration file, and an example of its structure is shown in the figure 5.5.

```
<domains>
    <domain name = "Untrusted Third Party Domain">
    </domain>
    <domain name = "Trusted Third Party Domain">
        <permission>Manifest.permission.INTERNET</permission>
        <permission>Manifest.permission.CALL_PHONE</permission>
    </domain>
    <domain name = "Original Device Manufacturers Domain">
        <permission>Manifest.permission.SEND_SMS</permission>
        <permission>Manifest.permission.CALL_PHONE</permission>
        <permission>Manifest.permission.RECEIVE_SMS</permission>
    </domain>
    <domain name = "Mobile Network Operators Domain">
        <permission></permission>
    </domain>
</domains>
```

**Figure 5.5.** The configuration file of protection domains

### 5.2.4 The domain manager

The domain manager handles logics of protection domain binding and the permission evaluation. Instead of talking to domain or white list directly, `PackageManagerService` delegates the application to the domain manager. The `Domain Manager` mediates the collaborations between the `Whitelist` and `Domains`. In addtion, the initialization of the white list and protection domain is also handled by the `Domain Manager`.

At the time that the phone boots completed, the configuration data is sent to the `Domain Manager`. The `Domain Manager` is woken up at this point. It starts to initialize white lists and protection domain and then gets into the standby mode when the initialization is finished.

At the applications' install-time, the `Domain Manager` finds out and assigns the installing application to a specific protection domain by looking up its certificate in the white list.

At the time for evaluating the requested permissions, the `Domain Manager` checks the requested permissions of the installing application against its protection domain permissions. The access rights are granted if the requested permissions is a subset of domain permissions, otherwise, the access is denied.

### 5.2.5 The service hook in Voice Dialer

The two configuration files for the white list and protection domain are placed in the resource folder of a built-in Android application: *Voice Dialer*.The parsing of data is handled by a internal XML parser called `Xmlpullparser` provided by Android.

*Voice Dialer* is the application that is activated at phone startup time. It has registered to receive the boots complete broadcast so that it knows when to lauch. In order to make sure that on application can bypass our WITDOM security checks, we need to wake up our service at the same time so we place a service hook in this application. The modified *Voice Dialer* will not only start its own service but also wake up the `Domain Manager` for us when it is notified the phone's booting process is finished.

# Chapter 6

# Implementations and testings

In this chapter, we look at the WITDOM from the code point of view. The classes design is explained with the help of the class diagram. The work flow is then illustrated by using a sequence diagram. We will see the implementation details of each WITDOM components and how they are functioning as a system. The second part of this chapter is the discussion on the results from the tests that have been performed on the WITDOM service.

## 6.1 WITDOM implementation

We have provided our service interface via the Android Interface Definition Language (AIDL) to enable the communication between the class: *SendConfigData* and the *DomainManager*. This is done by registering our service into the service manager.

In the description of the WITDOM design, we have already introduced the three key components. The `DomainManager` resides at server side, handling the domain assignment for all packages and evaluating the requested permissions at installation time. The `Protection Domian` and `White list` store security policies for the `DomainManager` to consult and they are configurable through XML files. The service hook: `SendWitdomService` is placed in a built-in Android application: *Voice Dialer*, taking care of parsing and sending the configuration data to the `DomainManager`.

### 6.1.1 The development environment

The implementation is made on the open master of Android source, and the version number is Android 3.2. Our WITDOM solution extends the framework of Android, which is written in Java. Hence we naturally choose Java for our implementation.

### 6.1.2 The Class diagram

The figure6.1 illustrates how those classes interact and collaborate with each other.

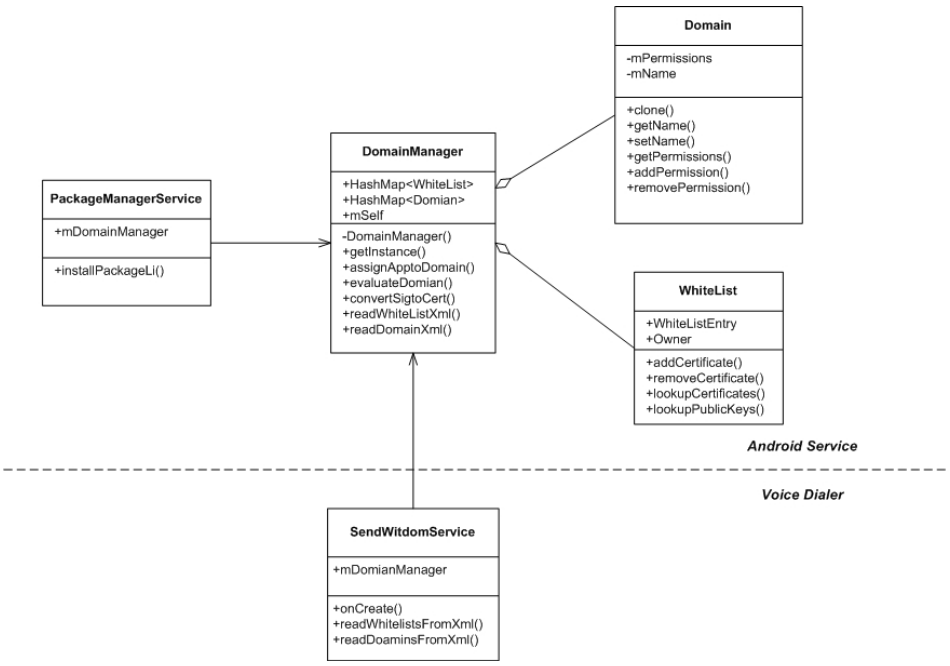


**Figure 6.1.** WITDOM class diagram

### 6.1.3 The sequence diagram

The sequence diagram below depicts the domain assignment mechanism; it shows how control flows from module to module when a new package is binding to a specific domain.

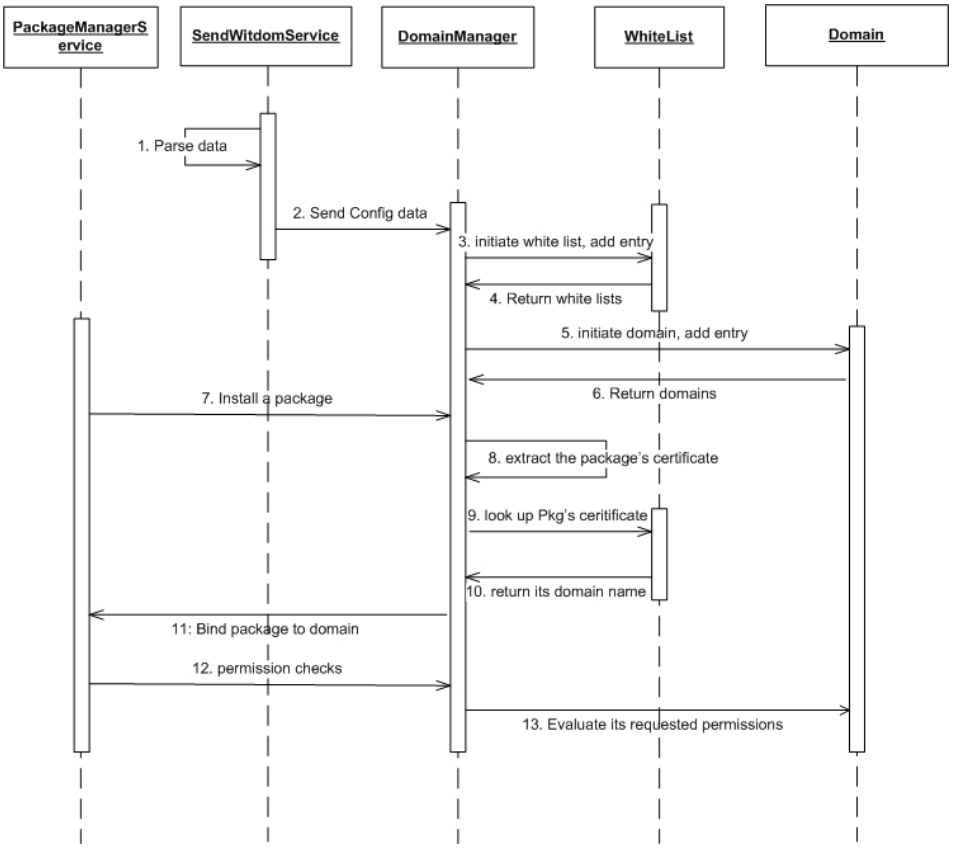


**Figure 6.2.** WITDOM sequence diagram

**At the phone startup time**

1. `SendWitdomService` parses the configuration data of white lists and domains from a XML interface.
2. `SendWitdomService` gets the instance of Domain Manager, and send the parsed data to it.
3. Upon receiving the data of the white list, we store them in form of `Whitelist` object which we have defined beforehand.
4. `DomainManager` adds the new `Whitelist` to the white lists pool.
5. Upon receiving the data of domains, we store them in form of `Domain` object which we have defined beforehand.
6. `DomainManager` adds the new `Domain` to the domains pool.

**At the application install-time**

1. We get an application installation request from the user.
2. The certification information is extracted from the package.
3. `DomainManager` searches through the white list pool and determine which domain the package belongs to.
4. Get the name of the corresponding domain.
5. `DomainManager` binds the package to the domain we have found.
6. `PackageManager` continues with other installation steps as well as checks the requested permission.

### 6.1.4 Implementations of classes

#### Domain Manager

`DomainManager` handles the logics of domain assignment and evaluation.

**Domain Assignment** When a package asks for installation, `DomainManager` determines a specific domain for the package by checking up its certificate. The certificate is compared to the white list. Despite the case that the certificate is found belong to a particular domain, there are two other possibilities we need to take into consider.

- When the certificate is not found in the white list, we bind the package to Untrusted Third Party Domain (UTPD).
- When the certificate is found belong to more than one domain, we create a new domain which is the union of all found domains.

**Domain evaluations** Domain evaluation takes places at the permission checks and it applies only to *Signature* permissions. In another word, other types of permissions won't be affected and are taken care by the regular Android permission checks. At the permission check point, if a signature permission fails the standard checks, we evaluate it against its domain permissions. Regrant the permission to the package when it passed the doamin checks.

| Domain Manager | |
|---|---|
| Variables | Description |
| `mWhitelists` | A white list pool which stores all white lists configured in the XML file at client app. |
| `mDomains` | A domain pool which stores all domains defined in the XML file at client app. |
| Methods | Description |
| `assignApptoDomain` | It binds a given package to a specific domain by comparing the certificate it contains to the white list. |
| `evaluateDomain` | It checks if a package belongs to a domain, and returns a boolean value. |
| `convertSigtoCert` | It retrieves the certificate from the signature of package. |
| `initWhitelist` | It receives white list data sent from the client side and stores it to mWhitelists. |
| `initDomain` | It receives the domain information sent from client side and stores it to mDomains. |

**Table 6.1.** Domain Manager

### Domains

`Domain` is the object class that defines the elements it consists of as well as the configuration methods.

| Domain | |
|---|---|
| Variables | Description |
| `mName` | It is the name of domain. |
| `mPermissions` | They are permissions that the domain is associated with. |
| Methods | Description |
| `addPermission` | It adds a permission to the domain. |
| `removePermission` | It removes a permission from the domain. |
| `getPermissions` | It returns all the permissions that are bound to this domain. |
| `comparePermission` | It checks if the provided permission can be found in the domain permission list. |

**Table 6.2.** Domain

### White lists

`White list` is also an object class. Similar to `Domain`, it includes several attributes which are specified by member parameters and a few configuration methods.

| White list | |
|---|---|
| Variables | Description |
| `mName` | It is the name of white list. |
| `mCertificates` | They are certificates that the domain has. |
| Methods | Description |
| `addCertificate` | It adds a certificate to the white list. |
| `removeCertificate` | It removes a certificate from the white list. |
| `lookupCertificate` | It searches through the certificate list of the white list, if the given certificate can be found, return true, else otherwise. |

**Table 6.3.** White list

### SendWitdomService

`SendWitdomService.java`, as one of the core parts of our WITDOM service, is placed at the client side. It takes care of for parsing and sending the domain and white lists data. It has been added to one of the start-up applications, *Voice Dailer*. This choice is made due to the following considerations.

- We need to instantiate the domain and white list at the booting time. With registering for receiving the broadcast of boots complete, our service is automatically activated when the phone is finished with booting.
- All the operations should be done in the background, and it is not necessary to interact with the user, therefore, we think to extend the service class is more appropriate than subclassing the Android activity, though choosing either of them can fulfill our needs.

| SendWitdomService | |
|---|---|
| Methods | Description |
| `onCreate` | It overrides unimplemented methods since this class extends the Android activity. It defines the behavior of this class when it is initialized. In our case, we call two methods: readWhitelistFromXml and readDomainFromXml to extract the data from configuration file. |
| `readWhitelistFromXml` | It parses whitelist.xml and send it to `DomainManager`. |
| `readDomainFromXml` | It parses domain.xml and send it to `DomainManager`. |

**Table 6.4.** Send Configuration data

## 6.2 The WITDOM testing

In order to ensure that our implementation can fulfill the requirements have been identified, we performed a set of tests on our implementation. The primary test approaches we have adopted for our implementation are the unit test and the system testing.

To assist developers testing their applications, Android provides a couple of useful development tools. They have been also used in our implementation and testing. For more information on the Android development tools, please refer to the appendix A.

### 6.2.1 The unit testing

The unit test is often used to verify each building block of code functions as it is supposed to be. We have performed unit tests on all WITDOM components, and the result shows that our implementation functions correctly.

### 6.2.2 The system testing

In addition to unit tests, we run the system test to verify that our implementation can be well integrated into the original Android platform and work correctly. The system test helps us verify the collaborations between components. To test the WITDOM service, we choose two sample applications and set the pre-condition for them. We assign them to different white lists and see how their permissions are finally granted.

We have run tests on signature permissions of the sample application and the result shows that WITDOM service can provide an effective way to deal with the inflexibility of signature permissions, which is in line with our goals. Screenshots from our testing results are shown below. Figure 6.3 shows the initiliazation of domains and white lists at the boot-time. Figure 6.4 shows requested permissions of an exmaple application are granted or denied accordingly by checking them against the WITDOM policy.

### 6.2.3 The compatibility test suite

The Compatibility test suite (CTS) is provided by Google and used to test the backwards compatibility of the modified Android platform. The backward compatibility is mentioned as a requirement when we had a discussion with ODMs. But our WITDOM service cannot pass the test at this point. This is the only requirement that we cannot fulfill so far. To enable the communication between the system service and the Android application, we have added our own interfaces and modified the system service which violate the backwards compatibility. However, we will keep investigating on this and find a way to solve the problem.
In general, we are satisfied with the testing results. We have realized all functional requirements and we also see a few things we could work on and they are described in the chapter 7.

```
10-16 09:55:39.432: DEBUG/SendWitdomService(296): service being started...
10-16 09:55:39.432: DEBUG/SendWitdomService(296): sendConfigData(): get R.xml resource
10-16 09:55:39.462: DEBUG/SendWitdomService(296): sendConfigData(): get parser
10-16 09:55:39.472: DEBUG/SendWitdomService(296): name: LPD_WHITELIST
10-16 09:55:39.472: DEBUG/SendWitdomService(296): owner: Least Privilege Domain
10-16 09:55:39.472: DEBUG/DomainManager(68): initWhiteList: LPD_WHITELIST
10-16 09:55:39.502: DEBUG/SendWitdomService(296): name: ODM_WHITELIST
10-16 09:55:39.502: DEBUG/SendWitdomService(296): owner: Original Device Manufacturers Domain
10-16 09:55:39.512: DEBUG/DomainManager(68): initWhiteList: ODM_WHITELIST
10-16 09:55:39.532: DEBUG/SendWitdomService(296): name: MNO_WHITELIST
10-16 09:55:39.532: DEBUG/SendWitdomService(296): owner: Mobile Network Operators Domain
10-16 09:55:39.542: DEBUG/DomainManager(68): initWhiteList: MNO_WHITELIST
10-16 09:55:39.552: DEBUG/SendWitdomService(296): name: TTP_WHITELIST
10-16 09:55:39.552: DEBUG/SendWitdomService(296): owner: Trusted Third Party Domain
10-16 09:55:39.552: DEBUG/DomainManager(68): initWhiteList: TTP_WHITELIST
10-16 09:55:39.582: DEBUG/SendWitdomService(296): finished with sending whitelist data
10-16 09:55:39.582: DEBUG/SendWitdomService(296): sendConfigData(): get R.xml resource
10-16 09:55:39.582: DEBUG/SendWitdomService(296): sendConfigData(): get parser
10-16 09:55:39.592: DEBUG/DomainManager(68): initDomain: Untrusted Third Party Domain
10-16 09:55:39.592: DEBUG/DomainManager(68): initDomain: Least Privilege Domain
10-16 09:55:39.602: DEBUG/SendWitdomService(296): name: Trusted Third Party Domain
10-16 09:55:39.642: DEBUG/DomainManager(68): initDomain: Original Device Manufacturers Domain
10-16 09:55:39.662: DEBUG/DomainManager(68): initDomain: Mobile Network Operators Domain
10-16 09:55:39.662: DEBUG/SendWitdomService(296): finished sending domains xml data
```

**Figure 6.3.** Initialization of WITDOM

```
10-16 10:36:38.973: INFO/WhiteList: Certificate(68): The certificate is found in ODM_WHITELIST
10-16 10:36:38.973: INFO/WhiteList: Certificate(68): The certificate is found in TTP_WHITELIST
10-16 10:36:38.983: DEBUG/DomainManager(68): Merged domain: 0:MergedDomain:net.jimblackler.newswidget has been
successfully added to mDomains
10-16 10:36:38.983: INFO/PackageManager(68): net.jimblackler.newswidget is assigned to:
0:MergedDomain:net.jimblackler.newswidget
10-16 10:36:40.023: DEBUG/PackageManager(68): New package installed in /data/app/net.jimblackler.newswidget-1.apk
10-16 10:36:40.033: DEBUG/PackageManager(68): WITDOM is activated. Perm = android.permission.INTERNET;Domain =
0:MergedDomain:net.jimblackler.newswidget from package: net.jimblackler.newswidget
10-16 10:36:40.033: DEBUG/DomainManager(68): checkDomainPermission, domain =
0:MergedDomain:net.jimblackler.newswidget
10-16 10:36:40.043: DEBUG/PackageManager(68): android.permission.INTERNET is denied by its domain
10-16 10:36:40.043: WARN/PackageManager(68): Not granting permission android.permission.INTERNET to package
net.jimblackler.newswidget (protectionLevel=1 flags=0xbe44)
10-16 10:36:40.043: DEBUG/PackageManager(68): WITDOM is activated. Perm =
android.permission.WRITE_EXTERNAL_STORAGE; Domain = 0:MergedDomain:net.jimblackler.newswidget from package:
net.jimblackler.newswidget
10-16 10:36:40.043: DEBUG/DomainManager(68): checkDomainPermission, domain =
0:MergedDomain:net.jimblackler.newswidget
10-16 10:36:40.053: DEBUG/PackageManager(68): android.permission.WRITE_EXTERNAL_STORAGE is denied by its domain
10-16 10:36:40.053: WARN/PackageManager(68): Not granting permission android.permission.WRITE_EXTERNAL_STORAGE to
package net.jimblackler.newswidget (protectionLevel=1 flags=0xbe44)
10-16 10:36:40.053: DEBUG/PackageManager(68): WITDOM is activated. Perm = android.permission.READ_PHONE_STATE;
Domain = 0:MergedDomain:net.jimblackler.newswidget from package: net.jimblackler.newswidget
10-16 10:36:40.053: DEBUG/DomainManager(68): checkDomainPermission, domain =
0:MergedDomain:net.jimblackler.newswidget
10-16 10:36:40.053: DEBUG/PackageManager(68): android.permission.READ_PHONE_STATE is denied by its domain
10-16 10:36:40.053: WARN/PackageManager(68): Not granting permission android.permission.READ_PHONE_STATE to
package net.jimblackler.newswidget (protectionLevel=1 flags=0xbe44)
```

**Figure 6.4.** Testing WITDOM on the example application

# Chapter 7

# Conclusions

This chapter states the experience we learned throughout the processes of the thesis work, both positive and negative. They are shown at a phase basis. Other than that, we perform an evaluation on the result to see to what extend our solution can fulfill the requirements. In the end, we go through our solution again and explore the possibility for our future work.

## 7.1 Discussions

We have discovered the Android platform developers mixed up the concepts of signatures and certificates when studying the source code. For more information, please see the appendix B.

The general impressions that we get from Android are:

- The Android OS has a huge code base. It includes hundreds of classes, many having up to 10,000 lines of code. To find the right point to start with is not an easy task as most of the internal code is not well documented. Thus, a lot of efforts has been put in order to understand the code.

- The security in Android is ill-patched work. The code for the access control is spread out into the Android source code. It is really difficult to get an overview of the Android security.

We have encountered some problems when we need to give an appropriate solution to the requirements. We are facing very specific requirements and there is almost no previous experience that we can relate to. By looking into the access control model in other mobile platforms, the protection domain in MIDP has come into our sight. Getting an insight of protection domains, we think this could be a suitable solution for our problems.

At the stage of the design, there is less problems came out. We think this is due to we have well defined requirements and a approved solution. Our understanding of the existing Android access controls allows us to easily find a point to start

with. Some minor changes are made to the initial design when we come into the implementation details. But for the main ideas, they stay the same.

The result of our thesis shows that our WITDOM service complements the existing security mechanism in Android and is able to provide a more flexible way when dealing with the functionalities protected by signature permissions. We see our WITDOM service can fullfil our requirements except for the requirement on backwards compatibility. In general, we are satisfied with the result. However, we also see some improvements we could make in the future. We start with the limitations.

## 7.2 Limitations

In our WITDOM service, we keep using the static permission approach from Android. Applications are assigned to protection domains when they are installed. Decisions for whether or not granting certain requested permissions are also made at the install-time. We also see the solution to enable the dynamic access control in the beginning, but we still stick to the static approach for two reasons. One is the static permissions keeps the access control mechanism simple. As the Android platform is used in resource-restricted devices, we don't want to introduce complicated mechanism to the system. This is also consistent with Google's philosophy for the Android development. The second reason is the changes that we can make on the static permission granting approach are enough to suit our requirements. Therefore, we don't need a dynamic permission approach for our case.

At this point, our WITDOM service handles all permission in the background. To trace whether the requested permissions are granted or denied are only available in the system log and they are not visible in the user interface.

The configuration interfaces we have exposed to ODMs and phone manufacturers are placed in a built-in native Android application. They need to share the application with other parts of functionalities. Concerning on the important information those two configuration files contain, better protections for them are expected. Hence, it is necessary to improve this part.

As we have mentioned in the testing section 6.2.3, the WITDOM service a problem to pass CTS [1], which is used to validate the modified system stay backwards compatible. However, that is the only requirement we cannot fulfill by now and we need more time on investigating the problem.

## 7.3 Future works

From the limitations that we have identified, we think the following parts worthy working on in the future.

- In terms of providing better user experiences, a user interface which lets the phone users to keep track of the granted permissions could be provided.

[1] Compatibility Test Suite

- To protect the configuration data from being potentially hacked, an dedicated Android application for handling the configuration data needs to be developed.
- The backwards compatibility is another part we want to work on. More investigations on this are required and a solution should be given.

# Bibliography


[1] Ross Anderson. *Security Engineering - A Guide to Building Dependable Distributed Systems*. Wiley, 2 edition, January 2001.

[2] D. Elliott Bell and Leonard J. LaPaluda. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-278 vol. 1, The Mitre Corp., Bedford, MA, 1973.

[3] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] epfl sync. Scala toolchain for eclipse. http://developer.android.com/guide/index.html, June 2010.

[5] David Ferraiolo and Richard Kuhn. Role Based Access Control. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pages 554–563, Baltimore, Maryland, USA, October 1992. NIST.

[6] Simon Godik and Tim Moses. eXtensible Access Control Markup Language (XACML). Standard, Organization for the Advancement of Structured Information Standards (OASIS), February 2003. http://www.oasis-open.org/committees/xacml.

[7] Google. Android market. http://market.android.com, July 2011.

[8] Google. Android security and permissions. http://developer.android.com/guide/topics/security/security.html, July 2011.

[9] Google. The developer's guide. http://www.assembla.com/code/scala-eclipse-toolchain/git/nodes/docs/android-examples/android-sdk/Wiktionary/AndroidManifest.xml?rev=f2fdb3144d0225487cafc7d628adf64889772db4, July 2011.

[10] Butler Lampson. Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems, Princeton, 1971. Reprinted in ACM Operating Systems Rev.*, volume 8, 1, pages 18–24, 1974.

[11] Matt Welsh Matthias Kalle Dalheimer. *Running Linux, Fifth Edition*. O'Reilly Media, 2 edition, December 2005. Print ISBN:978-0-596-00760-7 ISBN 10:0-596-00760-4.

[12] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[13] BBC News. Android hit by rogue app malware. http://www.bbc.co.uk/news/technology-12633923, March 2011.

[14] Machigar Ongtang, Stephen Mclaughlin, William Enck, and Patrick Mcdaniel. Semantically rich application-centric security in android. In *Annual Computer Security Applications Conference*, ASIACCS '10, 2009.

[15] Oracle. Mobile information device profile (midp); jsr 118. http://www.oracle.com/technetwork/java/index-jsp-138820.html, 2005.

[16] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur.*, 3:85–106, May 2000.

[17] Don Reisinger. Gartner: Android leads, windows phone lags in q1. http://news.cnet.com/8301-13506$_3$ − 20064223 − 17.*html*, $May$2011.

[18] Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOUNDATIONS OF SECURITY ANALYSIS AND DESIGN (TUTORIAL LECTURES*, pages 137–196. Springer Verlag, 2001.

[19] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38 –47, feb 1996.

[20] Ian Shields. Learn linux, 101: Manage file permissions and ownership, November 2010.

[21] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: general security support for the linux kernel. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 213 – 226, 2003.

[22] E. Yuan and J. Tong. Attributed based access control (abac) for web services. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, pages 2 vol. (xxxiii+856), july 2005.

[23] Gaoshou Zhai and Yaodong Li. Analysis and study of security mechanisms inside linux kernel. In *Security Technology, 2008. SECTECH '08. International Conference on*, pages 58 –61, December 2008.

# Appendix A

# Android tools

There are several Android development tools which are useful and they have been used intensively during our implementation.

## A.1 Android Debug Bridge

Android Debug Bridge (ADB) is a command line tool. It can connect and switch between multiple emulators. We use this tool to install and remove applications.

## A.2 Emulator

Emulator simulates Android on Personal Computer (PC) so that developers can view how their applications behave on real phones.

## A.3 Dalvik Debug Monitor Server

Dalvik Debug Monitor Server (DDMS) generates the system and applications logs when Android is running, as well as profiles the resource occupation during a given time slot. It also contains detailed information on each running process and threads.

## A.4 Android Interface Definition Language

AIDLs are IDLs in Android enables the IPC, which could bridge the communication between the client and the service.

# Appendix B

# A Confusion on Signatures and Certificates in Android

When studying the code of the Android OS, we have found there is a confusion between signatures and certificates. Although both of them are concepts from Public Key Infrastructure (PKI), however, they mean totally different.

The signature is a piece of encrypted information. It is used to validate whether or not the received content is tampered during the message transmission while the certificate gives a proof to the user's identity.

In Android, we have found that certificates are stored in a variable called signature, which apparently, does not conform to its definition. Signatures are actually only used when the certificates are loaded. This might lead to misunderstandings of the code.

Since we need to handle certificates in WITDOM, a function, `convertSigtoCert` is created to convert the so-called signature in Android.