Privacy Preserving Controls for Android Applications

by

Narasimha Aditya Gollapudi

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved November 2014 by the
Graduate Supervisory Committee:

Partha Dasgupta, Chair
Guoliang Xue
Adam Doupé

ARIZONA STATE UNIVERSITY

December 2014

ABSTRACT

Android is currently the most widely used mobile operating system. The permission model in Android governs the resource access privileges of applications. The permission model however is amenable to various attacks, including re-delegation attacks, background snooping attacks and disclosure of private information. This thesis is aimed at understanding, analyzing and performing forensics on application behavior. This research sheds light on several security aspects, including the use of inter-process communications (IPC) to perform permission re-delegation attacks.

Android permission system is more of app-driven rather than user controlled, which means it is the applications that specify their permission requirement and the only thing which the user can do is choose not to install a particular application based on the requirements. Given the all or nothing choice, users succumb to pressures and needs to accept permissions requested. This thesis proposes a couple of ways for providing the users finer grained control of application privileges. The same methods can be used to evade the Permission Re-delegation attack.

This thesis also proposes and implements a novel methodology in Android that can be used to control the access privileges of an Android application, taking into consideration the context of the running application. This application-context based permission usage is further used to analyze a set of sample applications. We found the evidence of applications spoofing or divulging user sensitive information such as location information, contact information, phone id and numbers, in the background. Such activities can be used to track users for a variety of privacy-intrusive purposes. We have developed implementations that minimize several forms of privacy leaks that are routinely done by stock applications.

To all who supported me

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Partha Dasgupta for his support and guidance throughout my thesis. Through his encouragement, this thesis was evolved from a tiny idea in a course project to what is being presented in this document. Also, I would like to thank my parents for their love and encouragement.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Android Operating System is one of the most prominent platforms for mobile devices. A recent survey claims that Android dominates the world smart phone market with more than 80 percent share. One of the main reasons can be attributed to its transparent nature and open-source approach. Despite these characteristics, Android does pose few threats when concerned to user privacy. Android uses permissions mechanism to grant access to certain privileges and components to an app. This information of what permissions the app requires is specified beforehand for the user to decide upon whether to install the app or not. Though this sounds like a good approach the access control is app-driven rather than user-driven. In other words, it is the app which decides what it can access but not the user, the only thing which the user can control at this point is whether he/she wants to install the app or not, given its permission requirements. Moreover, consider a scenario where an app has access to sensitive information which the user does not allow to be shared with other entities both internal, which can include other apps in the same device, and external entities. Currently, there is no way for the user to know if the app is not misusing its granted permissions, which makes the user to blindly trust the app. Given this current architecture, the central idea of this thesis work is to design a framework in Android which would rather give control to the user on what components a particular app can access. Also, the thesis is aimed at designing a way to let the user know if an app is trying to misuse its permissions.

Date	Applications available
May, 2011	200, 000
Jan, 2012	400, 000
May, 2012	500, 000
July, 2013	1,000,000
July, 2014	1,300,000

Table 1.1: Google Play Applications Timeline

1.1 Motivation

1.1.1 Need for User Driven Permission Management

With the current trend in technology, a mobile phone is capable of taking pictures, tracking location and even do processing almost equivalent to that of a desktop computers. Users have also adapted to these advances in mobile device technology and started using mobile phones for a variety of purposes thanks to the rich app library available across various platforms. Specifically this thesis concentrates on the Android framework, which provides more than 1.3 million apps available to be downloaded by users, as of July 2014. The number of apps in Google Play seems to be increasing in a rapid way with a growth of over 600% in the span of 3 years. The stats in Table 1.1 show the rate at which Android applications increase in Google Play over the past 3 years[42].

1.1.1.1 Increasing Malware

The increasing number of applications in Google Play store shows how prominent Android has become among users, but on the downside this also gives more scope to introduce malware and threats to a broader audience. There are many studies/surveys

conducted over the years which also suggest the increasing malware apps in the Google Play market. A study on increase of malware in Android conducted by Sophos Labs [29], claims that the malware tend to increase 81 times since 2010 to 2011. Later the malware rose even 41 times more in the time span on 2011 - 2012. A similar study by Network World [43] claims the count of malware specimens available in Google Play in a time span from June to October 2012 increased by 600%. This rise of malware in Android market makes the system quite vulnerable and a recent study on Mobile Threat Report by Webroot [40] claims more than 40% of Android apps analyzed were labeled as either malicious, suspicious or unwanted. Moreover, about 6 out of 10 Android apps are a potential security concern, these apps have a high chance of jeopardizing users' privacy.

These malwares are responsible for leaking user's personal data on many occasions. For instance, the popular Brightest Flashlight app which has more than a million downloads was reported to have misused ID and location data collected from its users [8] . This data collected was later shared with ad networks which has powerful tools to classify a user based on his personal and location data. On another instance in 2012, the widely used business-oriented social networking service LinkedIn has been reported to transmit user's data without their knowledge [10]. Similarly another social networking giant application Path was also known to upload entire address book of users to their servers [11]. These are only a few instances quoted among the various user data leaks from user's Android mobile devices due to either malware or bad-intention/ignorance of the developers of the apps.

1.1.1.2 Increasing Over-privileged apps

Another important factor which plays a key role in protecting a user's privacy is the number of applications which tend to require far more extra permissions than that are required for the app behavior. Though few extra permissions which seem to be contex-

tually inapplicable for the purpose of an app might make the users stop and think about installing the app onto the user's device. But there might be very few users who tend not to install the application due to this, an example of this can be taken for Facebook Android app. Though there weren't any reports of data leak or threats originating from the Facebook application there was a small user group which was alarmed at the update for Facebook earlier this year which gives the privilege for the app to read SMS. Though later a clarification from an engineer from Facebook sorted the debate by explaining that they require the permission to read SMS only to intercept login approval SMS messages as part of 2-factor authentication for their accounts. It doesn't stop the application to read SMS of users for another malicious purposes thus risking the privacy of the user [33].

In this aspect a previous study made by Felt *et al.* [21], which intends to find out how many apps are over privileged. Among their sample of 940 applications, about one-third of the applications seem to be over privileged. More than 50% of these over-privileged applications seem to have one extra permission and about 6% request more than four extra permissions. On a related study by Mario Frank *et al.* [26], the permission request activities of over 188,000 various Android applications were tracked. This study enlists the top 15 Android permissions which are being requested by these apps in various scenarios. Out of the top 15 Android permissions more than 50% belong to the set of dangerous permissions which involve reading personal information, read contents of the SD card, user's location activity, network connectivity and also permissions related to services that cost money for the user like placing phone call, sending SMS etc. Also out of all the permission request patterns the percentage of permissions related to services involved in user's privacy accounted to more than 70%. This study is a conclusive evidence that majority of the apps have access to sensitive user data which may have a potential chance of landing in the wrong hands.

Correlating the above information of how apps are being used in the current scenario right from trivial pastime activities to critical applications which involve interacting with sensitive user information related to IDs, Credit cards, Bank accounts, Location history, Phone book and other personal information. It makes more sense for the user to control what an app can access rather than blindly relying on the app itself. The above factors are one of the main motivating factors for necessity of User driven permission management.

1.1.2 Motivating Example

The research for this thesis is to investigate the possibilities and limitations of an app trying to compromise privacy of a user in Android. The idea originated while developing an app as part of one of the course projects. The Android app developed intends to help an instructor to keep track of the attendance of a class with the help of barcode scanning using the camera application of a user's smart phone. The app was built successfully which leverages on the Barcode Scanner application to be pre-installed on user's mobile device. The following steps describe the use case which involves reading the barcode:

- Invoke the Barcode Scanner app from Attendance Tracker app
- Scan the barcode using the already installed Barcode Scanner app
- Process the barcode from the image scanned
- Barcode Scanner app returns the processed barcode to the requested app
- Attendance Tracker app logic flows ...

In the above steps it should be noted that the actual barcode is scanned from the Barcode Scanner app but not the Attendance Tracker app. In this scenario since scanning the barcode requires the android.permissions.CAMERA, it seems intuitive that the Attendance Scanner app also possesses the same permission. But with the current way Android

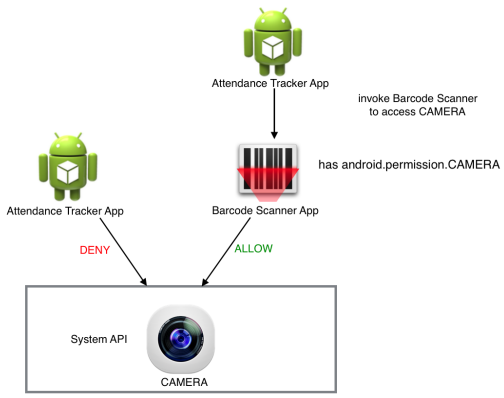


Figure 1.1: Pictorial Representation of Permission Re-delegation

permissions are processed for apps the Attendance Tracker app does not need to require the android.permissions.CAMERA permission. The argument being, since the scanning of barcode is done in the Barcode Scanner app, it is the Barcode Scanner app which requires the android.permissions.CAMERA permission. Attendance Tracker app was able to indirectly access the information processed by Barcode Scanner by accessing Camera, even when the Attendance Tracker app cannot access Camera by itself. The attack in this scenario where an under-privileged app delegates a task to a better-privileged app in order to get the information from resources to which the under-privileged app does not have access to is known as Permission Re-delegation. This is visually illustrated in Figure 1.1.

This scenario can be replicated between any two apps using Intents in Android, which are explained in detail in the coming sections. Although this seems trivial in this scenario we could imagine the same situation happening between a Malicious app with no access to user-sensitive data and a Sensitive app with access to user-sensitive data. Even though Malicious app cannot access the user-sensitive data on its own, it can take advantage of Sensitive app which might allow other apps to read the data. This scenario was the motivating factors which made me look into and understand various actions performed in Android related to security.

1.1.3 Need for Monitoring the Activities of Android Apps

Giving complete control to end-user in choosing which apps can have access to which permissions, though seem to be a reasonable solution to control user's privacy. Felt *et al.*[23], conducted a behavioral survey of Android users, the results of which indicated only 17% of the users paid checked keenly about the permissions the app is requesting during installation. On the negative side more than 42% of the people were not even aware of the permissions being requested by the apps during installation. This survey proves that rather than completely leveraging on the end-user to choose proper permissions for apps, there should also be a system which continuously tracks the permission request patterns of an application and notify the user/centralized-system [1] about the possible threats to user's privacy. Thus the user/centralized-system can take effective counter measures in vetting the app there by ensuring the threat can be avoided further.

The above consolidate the main motivating factors for formulating the main idea of this thesis work.

1.2 Organization

1.2.1 Methodology

The work proposed and presented in this thesis is done following an Incremental model of Software Development Life Cycle (SDLC). The software proposed is done in an evolutionary fashion. So rather than considering the entire work as a single project, it is divided into multiple projects which in turn follow the usual phases: requirement gathering, design, implementation and evaluation, as identified in a typical SDLC. Subsequent projects take the results of previous projects as inputs and the software continues to evolve until the final product.

[1]The centralized-system in this context refers to the Google Play entity in Android.

1.2.2 Document Outline

With the methodology explained, the rest of the document is organized in the following way. Chapter 2 gives an overview of the Background information about Android architecture along with its permission enforcement scheme. Also it explains the process of installation and running of an Android app.

Some more information is provided with respect to the related work in Chapter 3. These works come under the domain of security, permissions and user privacy in Android.

Chapter 4 deals with the project involving static analysis of Android applications to track potential intents that can make a permission re-delegation attack.

Chapter 5 deals with the project allowing users to choose the permissions of Android applications during install time on their devices.

Chapter 6 deals with the project allowing users to dynamically enforce constraints during runtime with respect to permissions being applicable to applications.

Chapter 7 encompasses the project which tracks the permission usage of Android applications especially taking into consideration the running context of the application.

Chapter 8 gives a formal conclusion of this thesis along with a brief outline on the future work that can be done to make Android a much better system in ensuring the privacy of users.

Chapter 2

BACKGROUND

Android, Inc. has its roots in 2003, during which it was originally intended to be developed as an advanced operating system for digital cameras. However after realizing the lack of market for such an operating system built for cameras, the goal was shifted to make Android as a smart phone operating system. After its acquisition into Google in 2005, the latter wanted to make Android as a platform for smart phones which succeeded immensely. Android kept improving its performance since then and released updates periodically with Android L (Lollipop) being the latest one release in 2014.

2.1 Android Architecture

Android Architecture is majorly renown for its efficient customization of Linux kernel in building a platform base to run applications. The architecture based on Linux kernel down the stack, exercises each application to run independently in a sandboxed environment created by Dalvik Runtime or Android Runtime which can be considered analogous to the renown Java's Virtual Machine environment. Each application running in Android even though independent of one another are still tied together by the Binder mechanism for inter-process communication (IPC). This IPC mechanism is developed with the ideal characteristics and Security, Scalability, Stability, Memory optimization are a few to name. Android utilizes this Binder along with adding many more useful customized features like AIDL, Process Identification from UID/PID etc. In fact the most frequent Activity callbacks for app which include onResume(), onDestroy() in the high level services, like `ActivityManagerService`, use binder in the low-level for its implementation. The coming sections briefly describe the basic blocks of an Android applica-

tion before going into the details with respect to Android Permission Model and process model which are topics of significant importance to this thesis.

2.2 Android Application Components

Typical Android applications include the following components which serve as building blocks of an application.

2.2.1 Activity

An activity represents the visual view of an app. Usually an app consists of a list of Activities, instances of which are loaded every time user is trying to interact with the app in the foreground. Activities together can be termed as the face of an app. Activities in turn use Fragments, Views to render UI related entities. As mentioned in the previous section an Activity Lifecycle consists of various states[13].

- An activity is present in *active* or *running* states if it is in the foreground.
- An activity is in *paused* state if it has lost focus but is still visible for the end-user. Though the activity is paused it retains a copy of its state and other information from it is active state.
- An activity is in *stopped* state if it is no longer visible to end-user. The user can still retrieve the state of the app if the app is still running. Under low memory conditions Android system will often kill these no longer used apps to free memory to be used by other active applications.

Storing the state information of an Activity can be facilitated using various methods available as part of the Activity API in Android.

2.2.2 Intent

An intent in Android is an abstract way of performing operations. Intents are used to invoke few entities which usually include from launching an Activity to starting a service. Usually intents are also provide a way to invoke Activities from other applications thus facilitating late runtime binding between different applications[15]. This is a way for inter-app communication in Android where apps run in a sand-boxed environment. Further sections in this thesis also deals with ways of dealing with intents which might be a potential way to leak data among multiple applications.

2.2.3 Service

A Service in an Android app represents a background running operation without the interaction with end-user. Applications usually use Services to gather data in the background and make it ready for Activities in the app when it is in the foreground.

2.2.4 Broadcast Receiver

A Broadcast Receiver can be made to listen for any incoming messages from system and other applications via intents. The Broadcast Receivers are used to notify other related components when specific events occur.

Linking of all these components is responsible for the smooth running of an application in Android.

2.3 Internals of an APK file

The previous sections briefly outlined various components of Android system which are necessary for an app to run in Android. This section gives an overview of various components of an APK file. An APK (application package) file is the file format used to

distribute applications in Android. Architectures of Android runtimes Dalvik and ART have clearly explained the key component of an Android application which is the dex file created after compiling the byte codes of all the java source files in Android. The following lists various directories and files of an APK file[41].

AndroidManifest.xml

Every application in Android is required to have this file in its root directory. The manifest file usually contains app specific information like name, icon, sdk-version, app compatible settings etc. Along with the above it also lists all the components that are present in this applications. The components include Activities, Services, Intents, intent-filters etc. The next key feature of the manifest file which is probably the point of interest for this thesis work is Permissions. Since an application in Android is run its sand-boxed environment, it should let the system know which external resources the app in intended to utilize. These are usually represented in the form of Permissions in Android. The `<uses-permission />` tag in the manifest file represents the permissions to use system resources. This manifest file acts as an integrating entity which puts together all the components of the application.

lib/

This directory usually contains compiled code of various supporting libraries which are referred usually in the application components. This directory in turn has different directories representing the processor base for which the libraries are compiled for e.g., arm, x86, mips etc.

res/

This directory contains the resources which are used in the application. An example of resources can be various images required in the UI layout of an Activity.

assets/

This directory contains application assets. These can be accessed programmatically in Android using AssetManager.

classes.dex

The dex file generated after compiling the bytecodes of all the java source files in the applications.

resources.arsc

This is a precompiled binary of the contents of resource directory mentioned above.

META-INF/

In order for the system to maintain a unique identity of authors of applications, apps are required to be signed before installation in Android. The contents of this directory contain the manifests and certificates of its digital signature.

2.4 Android Permission Model

Android operating system which is built on top of a Linux Kernel, runs each application as a Linux process in low-level. During the install time every package is assigned a unique Linux user ID. The UID remains constant as long as the package remains on the device. Although in the low-level the application is represented by a Linux process, the application as a whole is run in specific runtime either Dalvik Virtual Machine or Android Runtime as seen in the previous sections. Assigning an unique UID to each application encourages the sand-boxed environment created for each application in Android. Even though multiple applications are installed on a device having a unique id for each application wouldn't let an app to access contents of any other app by default.

In order to facilitate Inter Process communication (IPC), Intents are utilized which would allow the applications to pass data among them. Intents can be filtered using intent-

filters which control the access level of intents from apps. If an app specifies a filter to receive implicit Intents, then it is public and any intents to this are allowed. On the contrary they would not be allowed.

Coming to access control semantics, Android enforces access controlling features using Permissions. There a total of 146 permissions[16] introduced as of API level 20. Each of these permissions can be categorized into the several ways, below listed are few important categories of permissions:

Services that cost you money

These permissions usually refer to services capable of calling and sending text.

Network communication

These mainly refer to the INTERNET connectivity permission. Others can include WiFi, Bluetooth, NFC etc.

Your Location

These have the capability to access fine (GPS), accurate (WiFi), coarse (network-based) location.

Microphone

These have the capability to record audio.

Camera

These have the capability to control the Camera of the device.

Storage

These have access to read, modify or delete contents of internal storage or SD card.

The above categories are only a few of the lot and are listed to emphasize their importance with respect to user's privacy. The permissions of an app should be listed using the

`<uses-permission />` tag in AndroidManifest.xml file. The Figure 2.1 shows sample listing of permissions in manifest file[17]:

```
<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 2.1: Sample of Android Manifest File with App Using READ_SMS Permission

Other than these 146 predefined permissions, Android also allows developers to create their own permissions and these custom permissions each be defined with a protection level[18], listed below, based on the type permission.

Normal

Normal permissions refer to the type of permissions which have their impact on end-user experience but they are not harmful to the user with respect to type of content they have access to.

Dangerous

Dangerous permissions refer to the above list of categories which have access to services that cost the user, dealing with personal data of user etc. These are regarded as highly sensitive with respect to user privacy.

System/Signature

These are probably the most powerful permissions, which have the ability to control/kill other apps running in the background. Hence these permissions are not generally granted to app developers. Rather these are granted only to the applications which are installed during the manufacturing process of the device.

The figure 2.2 shows a sample definition of a dangerous level permission in the context of com.me.app.myapp application[17].

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >
    <permission android:name="com.me.app.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />
    ...
</manifest>
```

Figure 2.2: Sample of Android Manifest File Defining a Custom Permission `DEADLY_ACTIVITY`

With the above context on Permissions in Android, figure 2.3 depicts the basic security model for the system[34].

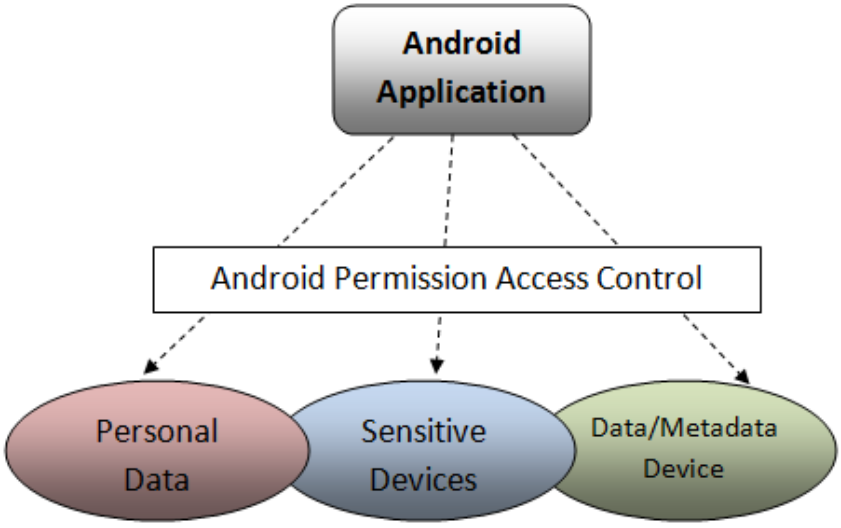


Figure 2.3: Permission Access Control Mechanism in Android

The next subsection gives an idea about how these permissions are enforced for an app in Android.

2.4.1 Permission Enforcement in Android apps

Most of the permissions in Android are enforced at the Android framework level present in the Android software stack. Typically from the framework level, there is only one publicly available API method in `ActivityManagerService` to check for the permissions for a given app. The coming sections illustrate how `ActivityManagerService` plays a key role in launching an application, given such high privilege it only fits that the checkPermission API method belongs to the same class.

Listing 2.1: checkPermission() method in ActivityManagerService

```
/**
 * As the only public entry point for permissions checking, this method
 * can enforce the semantic that requesting a check on a null global
 * permission is automatically denied. (Internally a null permission
 * string is used when calling {@link #checkComponentPermission} in
 * cases when only uid-based security is needed.)
 *
 * This can be called with or without the global lock held.
 */
public int checkPermission(String permission, int pid, int uid) {
    if (permission == null) {
        return PackageManager.PERMISSION_DENIED;
    }
    return checkComponentPermission(permission, pid, uid, -1);
}
```

The listing 2.1 illustrates the implementation of the only publicly available API for checking permissions in `ActivityManagerService`. Since each app in Android is entitled with a distinct unique UID, majorly the outcome of checkPermission is dependent

on the UID along with the listed permission. The later calls in the stack starting from `checkComponentPermission` method eventually refer to the `PackageManagerService` which stores the repository of permissions for the app by parsing its `AndroidManifest.xml` file.

Not all permissions in Android are enforced at the framework level, there are few permissions which are enforced at the kernel level using the concept of supplementary group-id[39]. Among these permissions which are enforced at group id level are the `INTERNET`, `CAMERA`, `WRITE_EXTERNAL_STORAGE`, `BLUETOOTH` etc. In Unix with 4.2 BSD, the concept of supplementary group IDs was introduced. This allowed the users if part of a specific group to have all the privileges of that group. The user could also belong to 16 such groups along with tailored access specified at user id level. All the above mentioned permissions were enforced using the supplementary group ids. The supplementary group ids were assigned based on the permissions the app has in its manifest file. This mapping of permissions with group id's are fetched from the platform.xml file. Below is an example of snippet from the `data/etc/platform.xml` file:

Listing 2.2: Permission and Group-ID Mapping from data/etc/platform.xml

```
<permission name="android.permission.INTERNET" >
        <group gid="inet" />
</permission>
```

If an app has any of these kind of permissions, then at the install time the group ids are set to the corresponding uid of the app accordingly. The below methods in listing 2.3 are utilized to set the enforce the group id.

Listing 2.3: API Methods Available for set/get/init Group-IDs in Linux

```
#include <grp.h>   /* on Linux */
#include <unistd.h> /* on FreeBSD, Mac OS X, and Solaris */
```

```
int getgroups(int gidsetsize, gid_t grouplist[]);
/* Returns: number of supplementary group IDs if OK, 1 on error*/

int setgroups(int ngroups, const gid_t grouplist[]);
int initgroups(const char *username, gid_t basegid);
/* Both return: 0 if OK, 1 on error */
```

Also the permissions can be used to secure components of an app. For instance if an Activity is declared with a permission, only apps with the corresponding permissions have the ability to start the Activity using intents. The below declaration in the manifest file depicts the same.

Listing 2.4: Securing Activity Using Permissions in Android Applications

```
<activity android:name="SecureActivity"
android.permission="com.mypermission.SECURE_ACTIVITY_PERMISSION">
<intent-filter>
...
</intent-filter>
</activity>
```

In the above example, only applications bearing the custom permission can invoke the SecureActivity using intents, the rest of intents are blocked. In the same way permissions can be used to secure other components like Services, Content Providers etc, in Android.

After illustrating the permission model in Android, the coming section describes how each of these come into picture during the installation of an application package (apk) file in Android system.

2.5 Android Application Installation

In Android an application is installed using the APK file. The default system application `PackageInstaller` is responsible for initiating the installation process of the app. The previous section shows how each app has a distinct user id thus enabling the app to be installed in its own sand-box environment. The following are the steps involved in installing the package[31].

1. Add package to installation process queue
2. Identify location for installing package
3. Determining type of operation install or update
4. Copy apk file to the concerned directory (usually /data/app/)
5. Assign a distinct UID for the package
6. Create directories for application and set privileges to the UID assigned in the previous step
7. Extract the classes.dex file from the APK file into the directory
8. Update `packages.xml` file with the new package
9. Notify user with the installation complete status

From the above steps, the major part involved with parsing/extracting the apk files, is dealt with `PackageManagerService`. Creating the directories and setting the appropriate rights for the UID is accomplished by the deamon process called `installd`. Once the installation is complete, the user is notified appropriately with the status by the activities of `PackageInstaller` application. Also a repository of package related data is stored

for faster access in other operations, and `/data/system/packages.xml` is one file that is updated along with all the permissions of the package. The following listing shows the structure of `packages.xml`.

Listing 2.5: Structure of /data/system/packages.xml

```
<packages>
<last-platform-version external="15" internal="15">
<permission-trees>
<permissions>
<item name="android.permission.CHANGE_WIFI_MULTICAST_STATE"
    package="android" protection="1">
<item name="android.permission.ASEC_ACCESS" package="android"
    protection="2">
...
</item></permissions>

<package codepath="/path/to/package.apk" name="com.app.mypackage"... >
<perms>
<item name="android.permission.mypackage.permission">
</perms>
</package>
...
</packages>
```

`PackageManagerService` uses repositories like these to list information of package when queried by other applications. Once the package is installed, the next section describes how the process management works for Android applications.

2.6 Android Application Process Management

Previous sections describe about how an application in Android is a separate Linux process that runs in its own runtime either Dalvik Virtual Machine or Android Runtime, but this doesn't mean that every process in Android runs in a similar. For instance there are many low level Linux processes which get spawned at the start up. These low level Linux processes are generally responsible for handling hardware interfaces. Apart from these low level handler processes, there exists another special process called Zygote.

2.6.1 Zygote

Zygote, as the name suggests, is the origin of all the processes related to Android applications. Typically when and Android application is launched, Zygote plays a key role in the lower level. Zygote is the process which spawns a new Linux process with the specific arguments for each application based on the configuration of the app. Zygote is also a termed as a "Warmed up" process since it is already linked and equipped with all the core libraries of Android which were described in the previous section. The main advantage of spawning new applications from Zygote is that, all the processes related to spawned applications do not need to have a copy of the core libraries. The memory/libraries are copied to the newly spawned process only in the case where the process intends to change them. Thus having a shared library with well-equipped Zygote process is majorly responsible for better performance with less memory footprint. Once Zygote is initialized, it opens a socket and continuously loops for any incoming socket connections. The next subsection illustrates how an application is launched from Zygote in Android[12].

2.6.2 Android Application Launch

After the start up, an application is typically launched by the end user by interacting with the Launcher. There are 3 major phases involved in the Application launch[28].

2.6.2.1 Creating the Process

When the application launch is requested, system first checks for any existing instance of the process in the memory. If it exists the system will load the process and return the same process id back to `ActivityManagerService`. Otherwise the package is initially parsed to gather its permissions which are transformed into a specific group ids which take play a key role in the access control mechanism of Android. This process of transformation is illustrated in the previous sections which describe about Android Permission Model. After parsing this package and gathering the required uid, gids and process specific details a Binder call is made to `ActivityManagerService` which is responsible for instantiating the Process record[19] for the Application. Through `ActivityManagerService`, a ZygoteSocket connection is established. After establishing a socket connection with Zygote, it is responsible for forking process and thus returning the process id back to `ActivityManagerService`.

2.6.2.2 Binding the Application

Once the process has been created from Zygote, each process spawns a new thread called the Activity thread which is responsible for forwarding appropriate messages to handler. This is typically done by the thread by sending a BIND_APPLICAITION message to the handler. This phase is responsible for invoking the methods which loads the app specific libraries/classes into the memory.

2.6.2.3 Launching the Application

Once the process is created and the corresponding classes for the application are loaded into the process's memory, the next step is to launch the activity which was initially sent to the `ActivityManagerService` from the Launcher. This process involves sending LAUNCH_ACTIVITY message to the handler, which then propagates down the stack and the appropriate activity is launched in the foreground to the end user.

Figure 2.4 illustrates the typical use-case of launching an application described above.

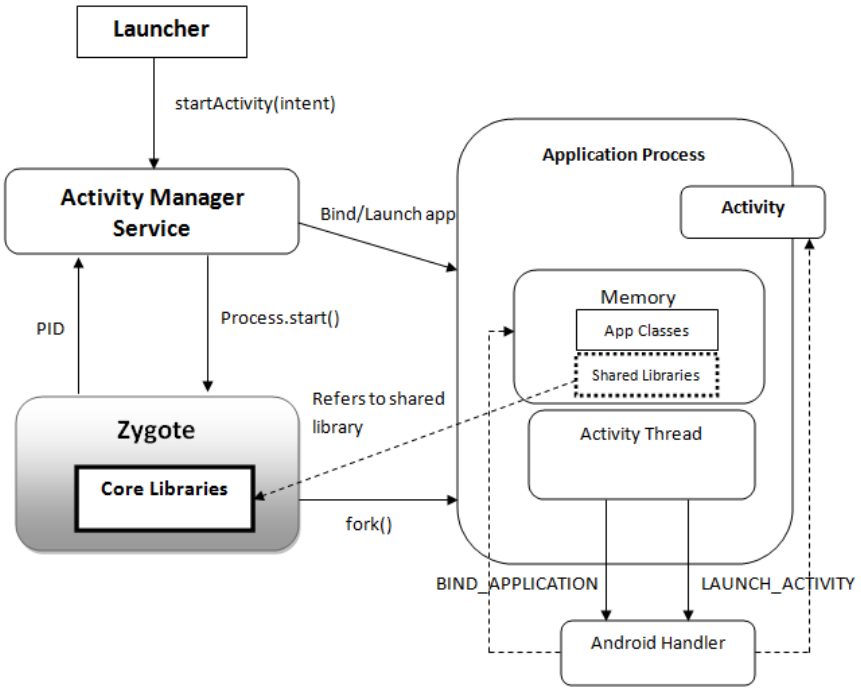


Figure 2.4: Android Application Launch Phases: 1. Creating the Process 2. Binding the Application 3. Launching the Activity

Chapter 3

LITERATURE REVIEW AND RELATED WORKS

Previous chapters evidently show that permissions play a key role in determining what an app can access and what not. So it is for the best to ensure that permissions are not utilized maliciously, breaking the trust and breaching the privacy of the user. Felt *et al.* devised a tool called Stowaway[21] to find out how many apps are over privileged. Their study shows that about one-third of the applications (among their sample of 940 applications) are using more number of permissions than required for the app.

3.1 Permission Re-Delegation

One of the other possible attack is Permission Re-delegation or Permission Escalation, which was already illustrated on Chapter 1. With the help of Android's inter-process communication entity, Intents, permissions re-delegation might be a potential threat to user's privacy. Felt *et al.* have implemented a IPC Inspection mechanism[22] for browser and Android and were able to prevent the Permission Re-delegation. In their implementation, they have come up with simple rules to block the attack where malicious applications tend to take leverage from other applications to access unauthorized content. Felt *et al.* designed the inspection system largely based on, restricted invocation of the Deputy app as they presented in their work. Part of this thesis work is aimed at blocking the Permission Re-delegation attack and the defense mechanism for this attack also took roots from the similar approach where the deputy app is launched with set of permissions which are common to both the requester and deputy app. Thus ensuring no chance of leakage of unauthorized content to the requester app from the deputy app.

3.2 Static Analysis to Find Malicious Intents

Before moving to the dynamic approach of restricting attacks and blocking applications, preliminary part of the thesis also tried to perform static analysis of Android applications to detect any possibility of intents being misused to perform permission re-delegation attack. Enck *et al.* had done a study of Android Application Security[20], where they have proposed a way to thoroughly perform static analysis of Android applications. Through this work they essentially have shown how traditional Static Code Analysis techniques can be applied in Android Applications. The entire analysis used four approaches: control flow, data flow, structural, and semantic analysis. This study using existing static code analysis tools, was able to show how Phone Identifiers, Location details etc. were exposed. Apart from findings where applications transmit user sensitive data across the internet, the control flow analysis part also included analysis of IPC calls in Android. Though their study tried to showcase a wide-range of findings with respect to their analysis of Android applications, this thesis tries to perform the static analysis of Android applications specifically to identify intents that might be potential causes for Permission Re-delegation attack.

3.3 Enforcing Dynamic Constraints on Permission Usage in Android

Although static analysis says what can possibly happen, the dynamic monitoring is the only way to control what exactly happens. With the same idea, this thesis also aims at providing a runtime permission model which would be able to restrict applications on how they take leverage of permission-model in Android. Apex[30], proposed by Nauman *et al.*, extends the Android Permission model by providing the users with the ability to enforce permissions as well as constraints related to the same for their applications. App Ops[14] in Android also allows users to control the permissions granted to their ap-

plications. App Ops which was accidentally released to the AOSP code base in Android 4.3 version, provides the user with fine-grained privacy controls for the end users. Figure 3.1 depicts the interface provided by Android's App Ops. This App Ops feature was removed in the next installment from Android, mentioning this isn't a tool for end users to have[27].

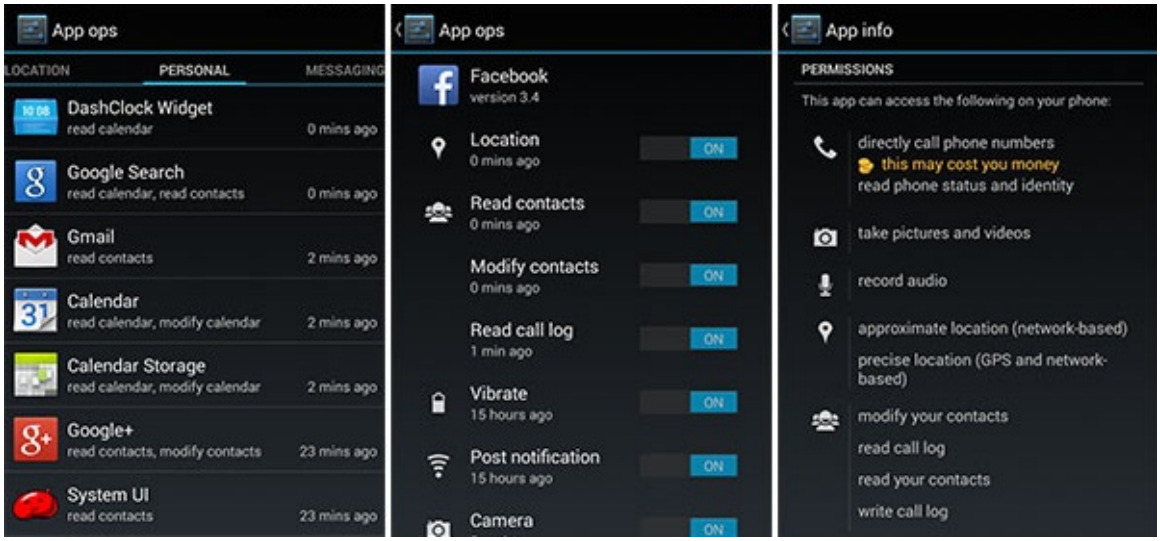


Figure 3.1: App Ops Interface in Android 4.3 (a) Categorizing Apps by permissions (b) Fine-grained Permission Control (c) AppInfo Listing Permission Details of an Application

Apart from providing the user with fine grained permission control, there is also a need to analyze how these permissions are being used in Android. Xu *et al.* proposed Permlyzer[44], a tool to analyze the permission usage in Android Applications. They were able to identify majorly for what purpose a set of permissions are invoked and how they are invoked. Along with this, they also identified potential scenarios on how a malicious application will target to gather user's personal information. They have provided a detailed study of how permissions are being used, and proposed a sustainable design to handle large set of applications. This thesis also analyzes the permission usage of applications in runtime and also takes into consideration the context of the application running in determining further action, i.e., allow or deny. The coming chapters explain the de-

sign and implementation details of various security related frameworks proposed in this thesis.

Chapter 4

STATIC ANALYSIS OF ANDROID APPLICATION FOR POTENTIAL MALICIOUS INTENTS

This chapter gives a complete overview of the problem analysis and specific requirements, design, implementation and results for the Static Analysis phase of Android application.

4.1 Problem Analysis and Requirements

A thorough static analysis of an Android Application can be done which would reveal potential places where leakage of user-sensitive information can be tracked from it. As described in the previous chapter there are already works which are capable of performing a complete static analysis of the Android application. Instead of reinventing the wheel, this thesis only concentrates on listing potential malicious intents that are capable of performing a permission re-delegation attack. The functional requirements related to this phase are listed below.

1. List various possible outgoing intents in the application.
2. Filter them to remove intents whose destination is in the scope of the current application.
3. Filter to keep only intents which return the control to the current application from the deputy application.
4. List out permissions of deputy involved in satisfying the intent to see any possible information leakage from the deputy application.

4.2 Design

This section gives a complete overview of the design and various design related decisions taken in static analysis phase of the project. As described in the previous chapter, thorough static analysis of an Android application can give an idea regarding what can happen when the application is run. The previous work from Enck *et al.* and various others included performing static analysis of Android applications to find out any possible malicious code in different possible scenarios. The following figures 4.1, 4.2 show a couple of scenarios where an application is tracking the IMEI number of the device[20].

```
public void run()
{
    ...
    r24 = (TelephonyManager) r21.getSystemService("phone");
     url = (new StringBuilder(String.valueOf(url))).append
("&vid=60001001&pid=10010&cid=C1000&uid=").append(r24.getDeviceId()).append
("&gid=").append(QConfiguration.mGid).append("&msg=").append(QConfiguration.getInstance
().mPCStat.toMsgString()).toString();
     ...
}
```

Figure 4.1: Sample from Static Analysis Depicting the App Tracking IMEI Number of the Device

```
public void onCreate(Bundle  r1)
{
    ...
    IMEI = ((TelephonyManager) this.getSystemService("phone")).getDeviceId()
     ...
}
```

Figure 4.2: Static Analysis Depicting Another Instance of IMEI Tracking

Although the static analysis phase of this thesis is not to completely search for various potentially dangerous candidates but rather the scope is restricted to find the possible intents which could be used as a potential threat to perform permission re-delegation attack. The following scenario demonstrates the use of such intents in an application.

4.2.1 Scenario: Permission Re-delegation by Intents

Permission Re-delegation attack as described in the previous chapter can be accomplished with the help of Intents, which are used for inter-process communication in An-

droid. Intents can be used to invoke Activities of other applications in Android and thus can be taken advantage of to read unauthorized content by apps. The following are the assumptions considered for this phase of the project.

1. Intents referring to activities in the same application are considered to be good in this context. Since there is no danger of permission re-delegation.
    - So generally any intent start with method like `startActivity()` can be ignored since, the control would be shifted to the application to which the Activity belongs.
2. An intent is only considered harmful if the control returns from the deputy application back to the original application.
    - So any intent started with method like `startActivityForResult()` can be considered as a potential candidate if the intent refers to an Activity out side the scope of the current application.
3. It is also assumed that application files of deputy applications identified are available for further static analysis on them.

With the above assumptions and required functional requirements presented in the previous section, the coming section gives more details on the design.

4.2.2 Design Details

Now that the functional requirements are set for the static analysis, this section gives details on the actual design of this phase of the project. As we have already seen in the previous chapters that APK file is a compressed version of various files related to the application. From a developer's point of view the main code files where the actual functional

and business logic goes in are the Java files related to the application. For the sake of simplicity, excluding the native code in this scenario since the context refers to permission re-delegation using Intents, a high-level API call in Android. All the java code was bundled together into one .dex file and is packed inside the APK file. Although we could perform bytecode analysis from .dex, keeping in mind the existing tools which can perform static analysis in Java it is for the best if the .dex file can be decompiled back to .java files. Moreover, decompiling the .dex files back to .java files gives a better idea from developer perspective to correlate what a particular developer intended to perform with this code.

Assuming the .dex files can be decompiled back to the actual .java sources, the next step is to identify for the intents along with the listing target Activity/Action for each of the intent. This process boils down to a pattern matching problem and which can be solved with the help of a right parser. Once the intents are identified, the next step is to identify intents which match the specific cases presented in the steps above, i.e., when the control returns back from deputy application to that of original application. The eventual outcome of this system are a list of Intents in the application that can be potential cause for Permission Re-delegation attack. The Figure 4.3 shows a pictorial representation of detailed design of the system.

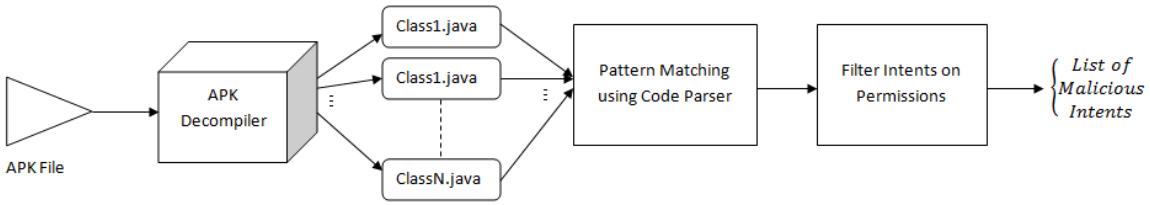


Figure 4.3: Design for Static Analysis of Android Applications to Identify Potential Malicious Intents Causing Permissions Re-delegation Attack

The next section talks about the implementation details and the related results.

4.3 Implementation and Results

4.3.1 APK Decompiler

The process of decompiling an Android application to its Java source files involves a 2 step process. First, the `class.dex` file in the apk is converted to jar and then the jar file is decompiled using the existing methods for java bytecode.

As discussed previously in the Background chapter, `classes.dex` in an APK file is an optimized format of all the class files in the Android application. The basic high level idea of compressing `.class` files to a `dex` file involves, integrating the data and constant pools of all the class files in one place which would reduce redundancy. The next step involves gathering all the class definitions together in the `.dex` file. The Figure 4.4 gives a typical idea of how `.java` sources are converted to `.dex` file in Android application development process[20].

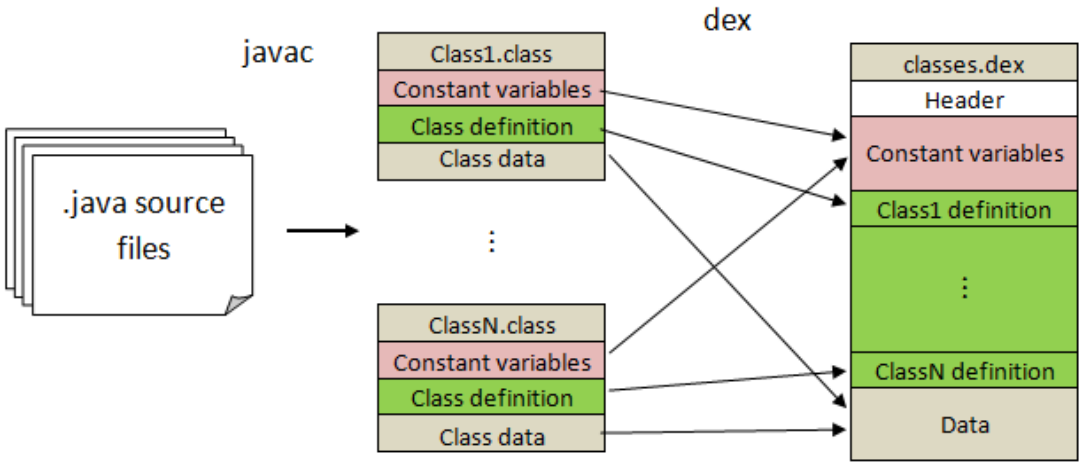


Figure 4.4: Conversion of Java Source Files to `classes.dex` File in Android Applications

The same logic is applied to reverse engineer `classes.dex` file to jar file which is nothing but an archive of `.class` files. There are existing tools like *dex2jar*[5] which

do the reverse engineering process of converting apk file into jar file. Once the jar file is obtained from the apk file, the existing java decompiler tool jad is used to decompile the .class files to the corresponding .java source files. The anatomy of decompiling an APK to its java sources is depicted in the figure 4.5. Although this process doesn't decompile the given apk file back to 100% original source files, but this does work in most cases.

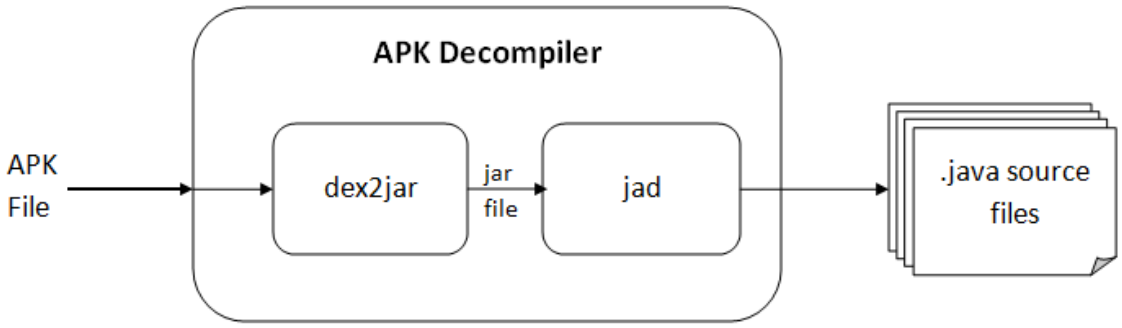


Figure 4.5: Pictorial Representation of APK Decompiler

4.3.2 Parser to Identify Intents

Once the `.java` source files are decompiled from the given Android application file, traditional java static code analysis techniques were used to identify the list of possible intents by parsing the entire source files. The total number of Activities in the current application which can be retrieved in the AndroidManifest.xml file, as described in the Background chapter. The intents identified are cross referenced from the activities of the current application and filtered as per the requirements. The Android Asset Packaging Tool `aapt` is used to parse the AndroidManifest.xml file from the apk file[1]. Once these intents are filtered out the remaining list of intents may correspond to Activities outside the scope of the current application, among these the only the intents which are called using `startActivityForResult()` method are identified. These are the possible list of potential intents which might cause the permission re-delegation attack.

4.3.3 *Track Permission from API Calls*

Once the external intents are identified from the previous step, the apk files for the corresponding files are analyzed. The analysis of deputy applications refers to the Activities to which the original application had made API calls to using intents. From a predefined list of permissions being correlated to API calls in Android, the possible permissions are listed out for the concerned Activities. APKInspector[32] tool already utilizes the same concept in identifying the permissions used with the help of a predefined API calls list. The figure 4.7 shows a sample snippet on how the API calls are mapped to a permission.

```
"BLUETOOTH_ADMIN" : {
        "android.bluetooth.BluetoothAdapter" : [
                ["F","cancelDiscovery()","public boolean"],
                ["F","disable()","public boolean"],
                ["F","enable()","public boolean"],
                ["F","setName(java.lang.String)","public boolean"],
                ["F","startDiscovery()","public boolean"],
        ],
},
```

Figure 4.6: Snippet from APKinspector with Mapping of Bluetooth API calls to Permission `BLUETOOTH_ADMIN`

Using these, the APKinspector lists the possible permissions the methods in the concerned Activity would need. The figure below shows the screenshot of how APKinspector lists the permissions that are mapped to the API calls originated from the Activity.

Using all the existing tools and techniques the static analysis of Android applications is implemented to identify the list of potential candidates for Intents that may be responsible for permission re-delegation attack.

4.4 Results

The static analysis is tested on a small sample set of Android applications, which delivered mixed results. There are cases where the list of possible malicious intents are

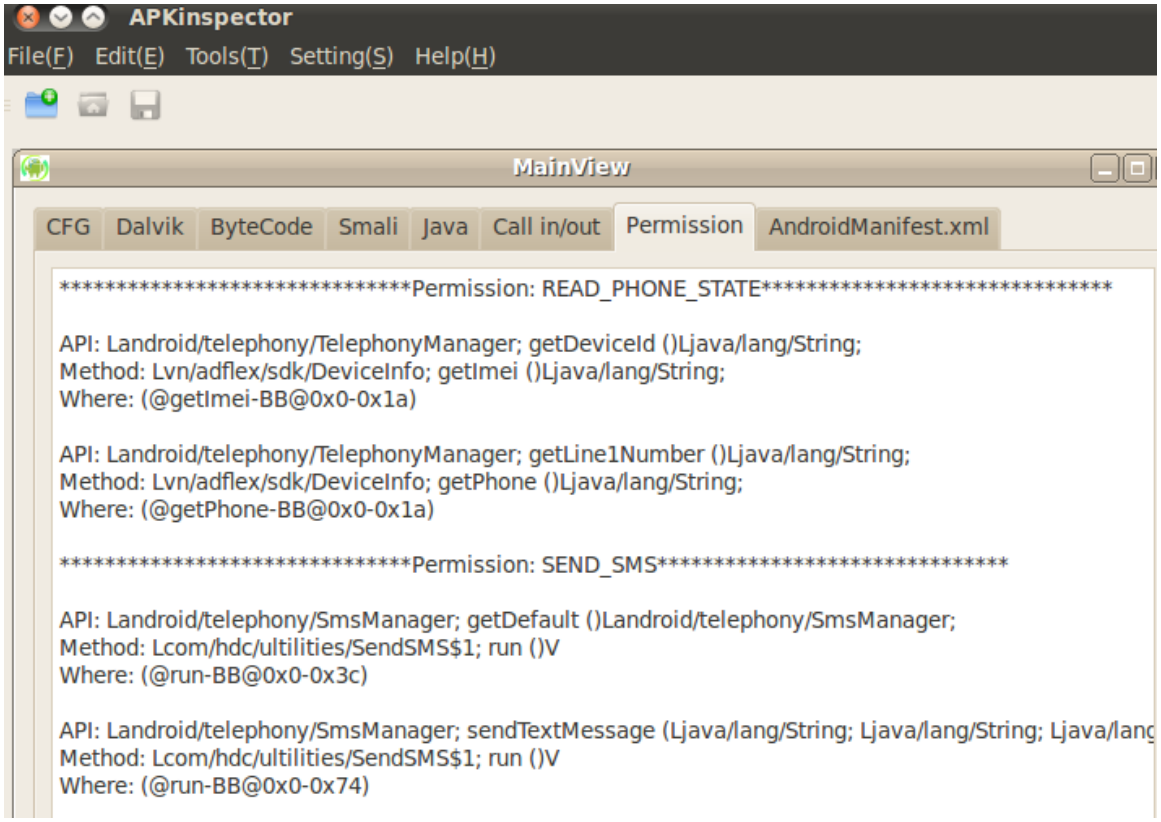


Figure 4.7: Screenshot from APKinspector Identifying Permissions for API calls from Activities in Android Applications

identified and there are cases where the analysis failed miserably. Listed below are the shortcomings of the process.

4.4.1 Shortcomings

Lack of context

Although there have been a list of Intents identified, this set doesn't exactly mean that the app is necessarily bad.

Failed parsing and permission mapping

The decompilation techniques are not 100% fool proof, there are ways like code obfuscation which would deliver improper results in the decompilation. Thus the dependent static code parsing steps would become ineffective.

In the similar way the mapping of permissions cannot be completely identified due to the large number of API methods available. More over the possibility of performing the same action in multiple ways make the mapping of permissions with API methods make it cumbersome.

Availability of Deputy Applications

This process is completely based on the assumption that the application files for all the deputy applications detected will be available which is not the case unless this technique is implemented along with access to a full repository of available Android applications.

Certainty of attack

Although static analysis may be able to identify how an app behaves, but through static analysis the possibility of attack may be uncertain.

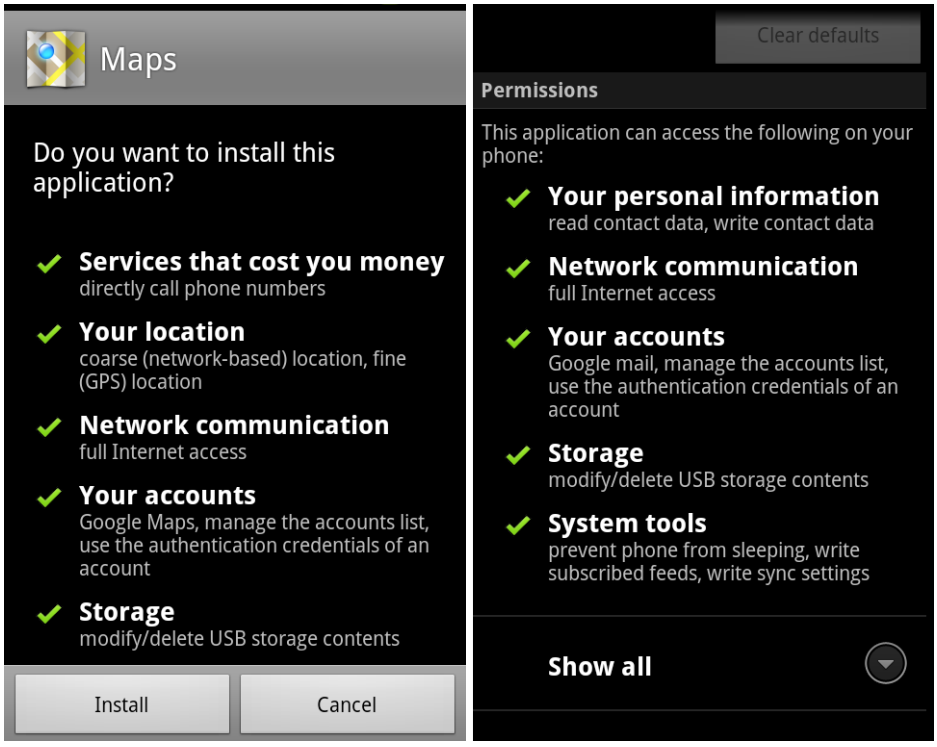
Though this approach of static analysis can be valuable for the specific attack under consideration, with all the above list of shortcomings for further research of this thesis is concentrated on dynamic/runtime behavior of applications.

Chapter 5

ENFORCE PERMISSIONS ON APPLICATIONS DURING INSTALL TIME

The shortcomings for static analysis of Android application illustrate the importance of permission system in Android. Since access control for every application in Android depends on the permissions the app possesses. The current way Android is built users doesn't have a way to change the permissions of an application. Instead the user is only notified about the permissions the app is requesting during the install time and the user has only the capability of not to install the application, if he/she is not satisfied with the permissions the app needs. The Figure 5.1 shows how an typical install prompt for an application being installed from Google Play Store. The figure also shows list of permissions of an already installed application through the Settings in Android.

As depicted, the user only has the option to either choose to install the application or not. Typically an app is required to be installed with the required set of permissions in order to avoid any unwanted behavior. But removing a permission from this set doesn't necessarily mean the app wouldn't be able to serve its purpose in the first place. For instance, the example of Facebook which requires many permissions to during installation, suppose a user feels that Facebook application doesn't require to have the `android.permission.READ_SMS` permission. Even though majority of operations concerned to Facebook would still work without having a problem, only the use case where the application tries to read the SMS from the device would tend to fail due to lack of permission. Analyzing the Facebook application further reveals that it only tries to read SMS from the device during the case where Facebook sends a security code as an SMS in its process of two-step authentication. In this way given the opportunity to choose the



(a) (b)

Figure 5.1: (a) An App Listing the Required Permissions at Install-time (b) Permissions Listed for an App Post-installation.

permissions to the end-user, they will be able to still use the application and feel secure about the app trying to access user sensitive data.

So this project is aimed at providing a new way for the user to choose the permissions of the application during the install time. The coming sections describe briefly the functional requirements, design and implementation of this project.

5.1 Problem Analysis and Requirements

As discussed before, this project aimed at providing the user with a new way to choose permissions during the install time of the applications. This might result in undesirable/inadvertent crashing of the app due to lack of required permissions. So this project works on the following assumptions:

- Developers take the responsibility of making sure the app has access to certain permission before actually performing required operation. For instance Figure 5.2, shows the notification user would get in iOS when an app tries to access a functionality for which the access is blocked by the user.

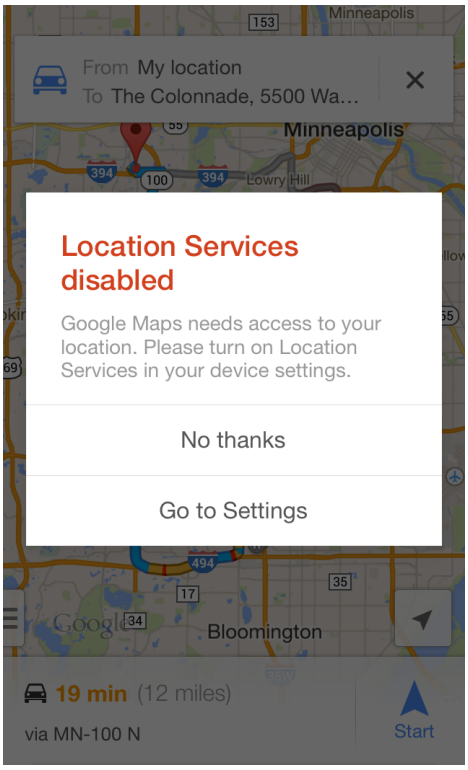


Figure 5.2: Notification in iOS to User Depicting the Requirement of Location Services by the App for the Required Functionality

This project assumes that the Android applications probe for permissions before accessing the concerned services. This can be achieved using the API calls `checkPermission`, `checkCallingPermission` and `checkCallingOrSelfPermission` in Android. Listing 5.1 shows how the application checks for `READ_PHONE_STATE` permission before making an instance to TelephonyManager service to read the phone number of the device.

Listing 5.1: Application Probing for Related Permission, `READ_PHONE_STATE`, Before Accessing the `TelephonyManager` Service

```
...
String permission = "android.permission.READ_PHONE_STATE";
int result = mAppContext.checkCallingPermission(permission);
if (result == PackageManager.PERMISSION_GRANTED) {
  TelephonyManager tMgr = (TelephonyManager)mAppContext.
                      getSystemService(Context.TELEPHONY_SERVICE);
  String mPhoneNumber = tMgr.getLine1Number();
} else {
   displayErrorMessage();
}
...
}
```

- The above approach will let the user to re-install the app, if required, this time by choosing the proper set of permissions to avoid the undesirable behavior.

5.1.1 Functional Requirements

With the above assumptions in place, the below are the functional requirements that are listed for providing the facility for users to choose permissions of app during install time.

1. Parse the APK file to list the permissions of the app.
2. Enable users to choose from the list of permissions for the app.
3. Reconstruct the APK file by only taking into account the permissions user has chosen.
4. Deliver the reconstructed APK file to the user to be installed in the device.

5.2 Design

With the above requirements specified, the permissions of the app had to be made configurable by the end user at install time. Since the permissions are listed altogether in `AndroidManifest.xml` file, the basic approach for this project would be to identify the permissions listed in the file. Based on the configuration of user, the permissions in `AndroidManifest.xml` file are updated accordingly and the application is rebuilt with the chosen permissions. With this premise, the steps to design the application are as below:

1. Decompile the required Android app.
2. Parse the `AndroidManifest.xml` file, to list the permissions the app is requesting.
3. Allow user to choose from the above list of permissions.
4. Update the `AndroidManifest.xml` file with only the permissions the user had chosen.
5. Rebuild the whole application from the extracted content. Allow user to install the updated Android app.

Figure 5.3 shows the pictorial representation of the design mentioned in the above steps.

5.3 Implementation and Results

As seen in the previous chapter, static analysis of an app can be done with the help of Android application decompilers. Even though as per the design decompiling the Android application is required, it should be noted that high level of granularity in decompiling the application is not required. Instead of decompiling to the core java sources the app can be decompiled to an intermediate `smali`[4] source files. `smali`/textttbaskmali

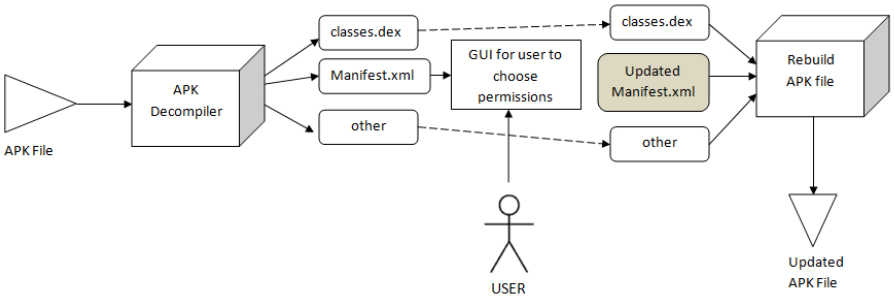


Figure 5.3: Design for Enabling the User to Choose Permissions of an Android Application During Install Time

is an assembler/disassembler for dex format used by Dalvik. The syntax is loosely based on Jasmin/dedexer's syntax, and supports full functionality of dex format. Similarly, the app can also be recompiled directly from the `smali` codes to dex and build the APK file from them.

5.3.1 apktool & aapt

Apktool[2] is an open sourced utility which provides the functionality of decompiling, the APK file to the `smali` codes. This process converts the `classes.dex` to the equivalent `smali` codes. The remainder of the application is extracted as it is, which contains `AndroidManifest.xml`, `assets`, `lib`, `build`, `res` folders typically. As seen from the Bacground chapter, these are the contents of the APK file other than `classess.dex` file.

Once these files are extracted using the apktool, the permissions for the app can be parsed from the `AndroidManifest.xml` and presented for the user to be chosen. Once the permissions are chosen, the `AndroidManifest.xml` is updated accordingly to only contain the permissions chosen by the user. After updating the manifest file, the contents could be rebuilt to a APK file using the Application Asset Packaging Tool (`aapt`). As

discussed in Background chapter the `aapt` tool is responsible for building the contents into a APK file in the first place.

5.3.2 Prototype Model

As a proof of concept, the above model is implemented using a simple web server which acts as an Android market, giving the user to install Android applications from. The user can use this web service to configure the application present in the market and then install the applications with the chosen set of permissions. The figure 5.4 lists the screen-shots of the prototype application.

5.3.3 Results

This approach decompiles the Android application and only tries to parse the `AndroidManifest.xml`, rather than changing any other sources in the application file. Hence, this approach less susceptible to failure due to previously seen problems in static analysis of APK. This was evident from the results for this approach, which has 100% successful results with a test sample of more than 50 android application files. Through this project, this thesis proposes an Android application to be configurable one rather than a specific APK file. The configuration can be then mapped to multiple APK files that can be generated with various combinations of permissions chosen by the user at install time. This approach also makes the user feel secure by controlling of the type of content the app can access. Thus making the Android applications to be having user-driven access control mechanism rather than the traditional app-driven access control mechanism.

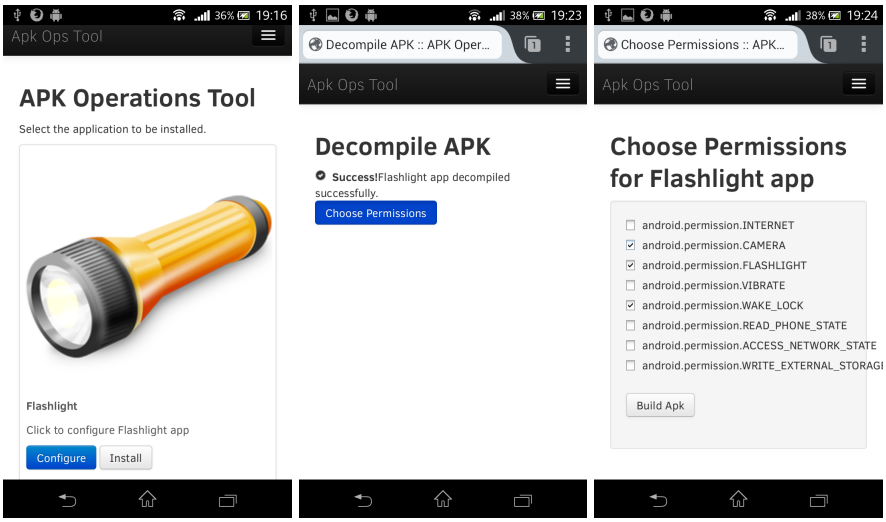


Figure 5.4: (a) APK Operations Tool Listing Flashlight App as Configurable App (b) Decompiling the Flashlight App (c) User Choosing the `CAMERA`, `WAKELOCK`, `FLASHLIGHT` Permissions Only (d) APK File Rebuilt with the Chosen Permission Configuration (e) Configured Flashlight Showing Only the Selected Permissions

Chapter 6

A DYNAMIC FINE-GRAINED PERMISSION ENFORCEMENT SYSTEM

The previous chapter proposed how users can be given control of the permissions of an Android app during install time. Ideally users are less likely to know what permissions to choose before actually installing and having prior knowledge of the app. This phase of the project aims at providing the users with a way to enforce permissions for their apps dynamically during runtime. Such a facility provided at runtime gives more flexibility for the users to dynamically adjust the access privileges of an application. Compared to the permission enforcement at install time proposed in the previous chapter, this facility lets the user to still change the access privileges of the application without the need to re-install the application. This way the user can still save his current settings/session in the application and if the user wants to enable/disable a certain permission for the app he/she could do so right away. This makes the runtime access control for apps as a more robust and flexible feature compared to the previous approach.

Also to be noted that, during the course of this work, a similar feature was rolled out as an accident in Android called AppOps in the initial update of Android 4.3, but later this feature was removed from Android Source Repository. A further comparison is made between the proposed `AppAccessControl` and AppOps in section 6.5. The privacy control feature released in iOS 6 also enables the user to control the usage of sensitive resources from an application in a similar manner. But unlike the privacy controls in iOS, this proposed system is aimed at providing the users with fine-grained permissions enforcement

at runtime. Before going into the design & implementation details, the coming section gives a little background on the way permissions are enforced currently in Android.

6.0 Background: Current Way of Permission Enforcement in Android

Compared to the previous approach where permissions are listed in the manifest file of the APK file, this approach deals with once the application has been installed on the device. So in order to achieve this a further understanding is required on how the permissions listed in the `AndroidManifest.xml` file are transformed to control the application access rights on the device. As seen from the Background chapter, each Android application installed runs as a Linux process in the lower level with a specific access control mechanism controlled by the permissions of the app during install time. In Android applications a permission can be enforced in two ways:

Android framework level permissions

These are the kind of permission which are enforced during runtime. In particular these permissions provide a higher level abstraction and are enforced at a higher level than the Linux process. Majority of the permissions in Android are of this category.

Linux process level permissions

These are the kind of permissions which have a direct effect on the low level Linux process concerned to the Android application. Each of these permissions are transformed into a unique supplementary group-id which is associated with the Linux process. In turn each group-id has a specific set of access privileges that are predefined. Many permissions which involve direct interaction hardware components on the device. These include `INTERNET`, `CAMERA`, `WRITE_EXTERNAL_STORAGE`, `BLUETOOTH` etc.

With the above types of permissions, clearly the associating/disassociate an existing application with any of the permissions depends on the type of permission. An Android framework level permission can be controlled directly at the framework level where it is enforced in the first place. Coming to the Linux process level permissions, the app's Linux process has to be associated/disassociated with the corresponding group-id(s), in order for the enforcement to take effect. Though for the latter to be successful, the Linux process has to be restarted. Now with this contextual knowledge, the coming sections describe about the design & implementation details of this phase of project.

6.1 Problem Analysis and Requirements

This project which involves manipulating the access rights of an Android application, which traditionally could be taken for granted for a particular Android application once installed. This project assumes that developers take the responsibility of making sure the application first probes for the availability of certain access before trying to perform the required operation. This ensures that app would notify the problem to the user rather than crashing. Figure 6.1 is a screen shot of a famous application Viber which notifies the user that the concerned operation requires the user to enable location services. Although the application is in iOS, the underlying message is that this project assumes the permission-related failures are handled accordingly by the developers of the app.

6.1.1 Functional Requirements

The below listed are the functional requirements for providing the facility to enforce permissions during runtime to users.

1. Provide user with a listing of apps on the device along with fine grained permissions concerned to each app.

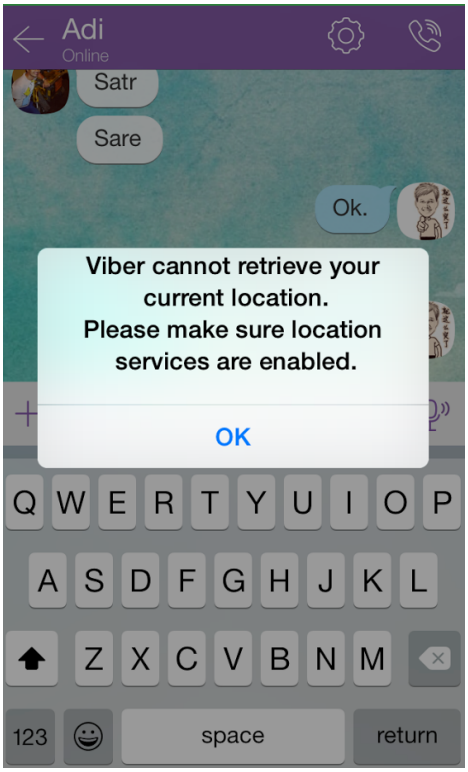


Figure 6.1: Notification in iOS to User Depicting the Requirement of Location Services by Viber App for the Required Functionality

2. Maintain a centralized repository to keep track of user's changes to the permissions of apps.

3. Ability to enforce the permission at the right (Android Framework/ Linux Process) level dynamically.

6.2 Design

With the above functional requirements, the figure 6.2 shows the design diagram of the dynamic permissions enforcement mechanism proposed called `AppAccessService`. The `AppAccessService` is a system level service which is responsible for enforcing permissions at runtime for an application. The service takes care of whether the permission is Process level or Framework level enforced. Based on the application's operations in runtime, the dynamic permission check are regulated using the `AppAccessService`.

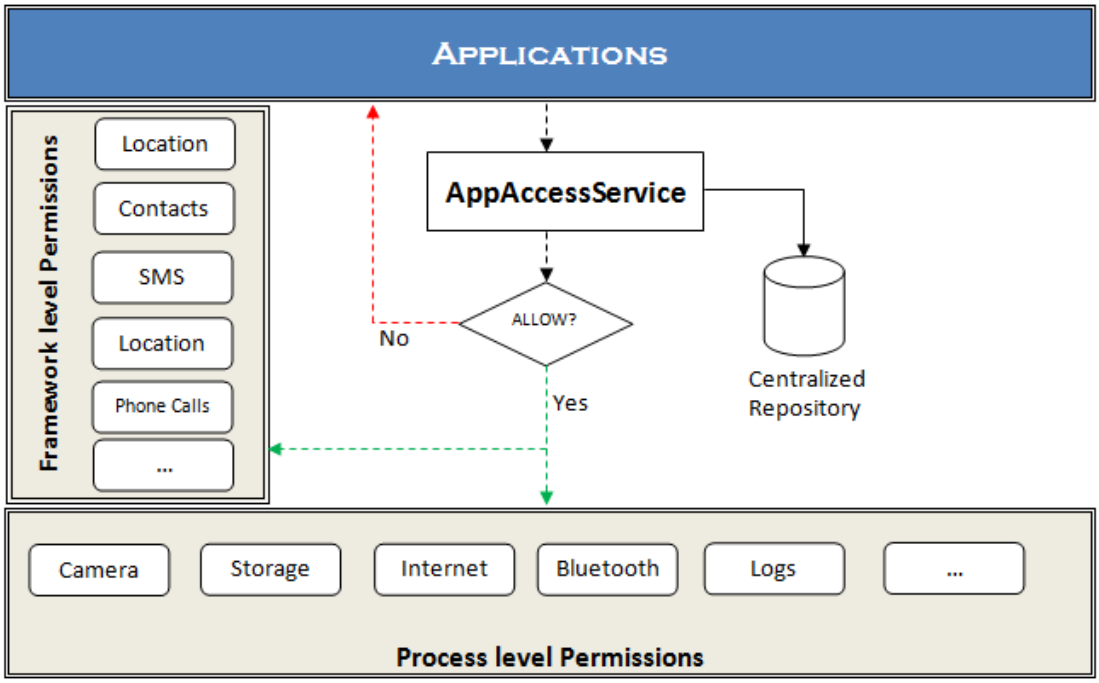


Figure 6.2: High-level Design Depicting the Control of Access for Apps Using AppAccessService

6.3 Implementation and Results

With the above higher level design, the main objective of `AppAccessService` is to enforce an additional check to control the access privileges of an app at runtime. With the two types of permissions described in the previous section, the following are the approaches to block the concerned permission.

6.3.1 Approach

6.3.1.1 Enforce Permission at Framework Level

As described earlier in the Background chapter, the `checkPermission()` method from `ActivityManagerService` is claimed[3] to be the only public entry point for per-

missions checking. Figure 6.3 shows the method definition from the latest AOSP [1] repository by Google.

```
/**
 * As the only public entry point for permissions checking, this method
 * can enforce the semantic that requesting a check on a null global
 * permission is automatically denied.  (Internally a null permission
 * string is used when calling {@link #checkComponentPermission} in cases
 * when only uid-based security is needed.)
 *
 * This can be called with or without the global lock held.
 */
@Override
public int checkPermission(String permission, int pid, int uid) {
    if (permission == null) {
        return PackageManager.PERMISSION_DENIED;
    }
    return checkComponentPermission(permission, pid, UserHandle.getAppId(uid), -1, true);
}
```

Figure 6.3: `checkPermission` in `ActivityManagerService` Mentioned as the Only Public Entry Point for Permissions Checking

Clearly modifying the AOSP to add additional check in the `checkPermission()` method is the simpler approach to enforce a permission at Android Framework level.

6.3.1.2 Enforce Permission at Linux Process Level

Unlike the framework approach, in order to enforce a permission at Linux process level for an app in Android involves set/remove the corresponding supplementary group-id for that application. In order to achieve this, there should be a clear understanding first on how an application is launched in Android. The section 2.6.2 from the Background chapter, clearly elucidates how an application is launched in Android. This involves three steps: 1) Creating the process, 2) Binding the application and 3) Launching the application. The steps for creating the process are analyzed carefully and the function calls are back tracked from the stack to identify where the group ids are set for an app's Linux

[1] Android Open-Source Project

process. The following are the key functions which are involved in setting the group-id for a package.

Get gids of a Package

The Background chapter clearly shows how `PackageManagerService` is responsible for parsing the Android application files and caching the key information related to a package. In this regard, the method `getPackageGids()` from `PackageManagerService` is responsible for returning the gids which are assigned to the Linux process being created for the application from `ActivityManagerService` as described in section 2.6.2. Figure 6.4 shows the definition of the method from `PackageManagerService`.

```
@Override
public int[] getPackageGids(String packageName) {
    // reader
    synchronized (mPackages) {
        PackageParser.Package p = mPackages.get(packageName);
        if (DEBUG_PACKAGE_INFO)
            Log.v(TAG, "getPackageGids" + packageName + ": " + p);
        if (p != null) {
            final PackageSetting ps = (PackageSetting)p.mExtras;
            return ps.getGids();
        }
    }
    // stupid thing to indicate an error.
    return new int[0];
}
```

Figure 6.4: `getPackageGids()` in `PackageManagerService`

The above code excerpt, doesn't exactly specify how the permissions are mapped to the gids. A careful analysis of the call sequence shows that after a bunch of calls, the permission are then mapped to the group ids from the `data/etc/platform.xml` and figure 6.5 depicts an excerpt from the same.

The above file is used as a base by Android framework to translate Android permissions to group IDs but the actual translation to the numeric group id is done somewhere at

```
<!-- This file is used to define the mappings between lower-level system
     user and group IDs and the higher-level permission names managed
     by the platform.

     Be VERY careful when editing this file!  Mistakes made here can open
     big security holes.
-->
<permissions>

    <!-- ================================================================= -->
    <!-- ================================================================= -->
    <!-- ================================================================= -->

    <!-- The following tags are associating low-level group IDs with
         permission names.  By specifying such a mapping, you are saying
         that any application process granted the given permission will
         also be running with the given group ID attached to its process,
         so it can perform any filesystem (read, write, execute) operations
         allowed for that group. -->

    <permission name="android.permission.BLUETOOTH_ADMIN" >
        <group gid="net_bt_admin" />
    </permission>

    <permission name="android.permission.BLUETOOTH" >
        <group gid="net_bt" />
    </permission>
```

Figure 6.5: Excerpt from `data/etc/platform.xml` Depicting the Mapping of Low-level Group-IDs with Permissions in Android

a deeper level `android_filesystem_config.h`. This is the header definition which the native code utilizes to map the gid like `net_bt` to the actual numeric equivalent gid 3002. The figure 6.6 clearly show how in the lower-level reserved user IDs, group IDs and range of app user IDs that are mapped to an Android application once during its installation.

With the above knowledge, instead of adding additional checks to the lower level entities in the system to interact with `AppAccessService`, the additional checks could be placed in the `getPackageGids()` method. Since this method is involved in fetching the group IDs related to the Linux process of the Android application, the check in this place gives a simpler solution that would enforce the permission at Linux process level.

```
36. /* This is the master Users and Groups config for the platform.
37.  * DO NOT EVER RENUMBER
38.  */
39.
40. #define AID_ROOT             0  /* traditional unix root user */
41.
42. #define AID_SYSTEM        1000  /* system server */
43.
44. #define AID_RADIO         1001  /* telephony subsystem, RIL */
45. #define AID_BLUETOOTH     1002  /* bluetooth subsystem */
46. #define AID_GRAPHICS      1003  /* graphics devices */
47. #define AID_INPUT         1004  /* input devices */
48. #define AID_AUDIO         1005  /* audio devices */
49. #define AID_CAMERA        1006  /* camera devices */
```

(a)

```
86. /* The 3000 series are intended for use as supplemental group id's only.
87.  * They indicate special Android capabilities that the kernel is aware of. */
88. #define AID_NET_BT_ADMIN  3001  /* bluetooth: create any socket */
89. #define AID_NET_BT        3002  /* bluetooth: create sco, rfcomm or l2cap sockets */
90. #define AID_INET          3003  /* can create AF_INET and AF_INET6 sockets */
91. #define AID_NET_RAW       3004  /* can create raw INET sockets */
```

(b)

```
101. #define AID_APP         10000  /* first app user */
102.
103. #define AID_ISOLATED_START 99000 /* start of uids for fully isolated sandboxed processes */
104. #define AID_ISOLATED_END   99999 /* end of uids for fully isolated sandboxed processes */
105.
106. #define AID_USER        100000  /* offset for uid ranges for each user */
107.
108. #define AID_SHARED_GID_START 50000 /* start of gids for apps in each user to share */
109. #define AID_SHARED_GID_END   59999 /* start of gids for apps in each user to share */
```

(c)

Figure 6.6: (a) Lists the Masters Users and Groups That Are Predefined in the System (b) Supplementary Group-IDs of the Hardware Components Interacting with the Kernel (c) Range of User IDs Assigned to the Installed Apps on the Device

6.3.2 Prototype Model

With the approach defined clearly from the previous section, a prototype model of the proposed `AppAccessService` is implemented on a Android 4.0 ice-cream sandwich repository. The following are the main components of the implementation.

SystemServer

Traditional SystemServer responsible for starting various services during the start up of an Android Device. This is responsible for the start up of `AppAccessService` after bringing up the required services.

IAppAccessService

The Android Interface Definition Language file that is linked with the actual `AppAccessService`. This enables the Binder IPC communication between the actual service and other interacting services/components.

AppAccessService

The actual service where the operations are defined, the primary operations are listed below:

1. *getBlockedPermissions()*: Returns the list of blocked permissions for the given package.
2. *updateBlockedPermissions()*: Update the blocked permission list for an application with the new list passed to it.
3. *getBlockedGids()*: Returns the list of blocked group IDs for the given package.
4. *isPermissionBlocked()*: Returns whether the given permission is blocked by the user for the given app.

AppAccessControl app

The UI app which is presented to the end user to control the fine-grained permissions of applications installed on the device.

PackageManagerService

The traditional PackageManagerService with an updated `getPackageGids()` method which removes the blocked gids returned from `AppAccessService` for the app.

ActivityManagerService

The traditional ActivityManagerService with an updated `checkPermission()` method which now runs an additional check to see if the permission is blocked as per `AppAccessService`. If so, this method behaves as if the checkPermission returns `DENY` response.

With the above components updated in the AOSP, figure 6.7 depicts the UML class diagram representation of only the modified components of AOSP.

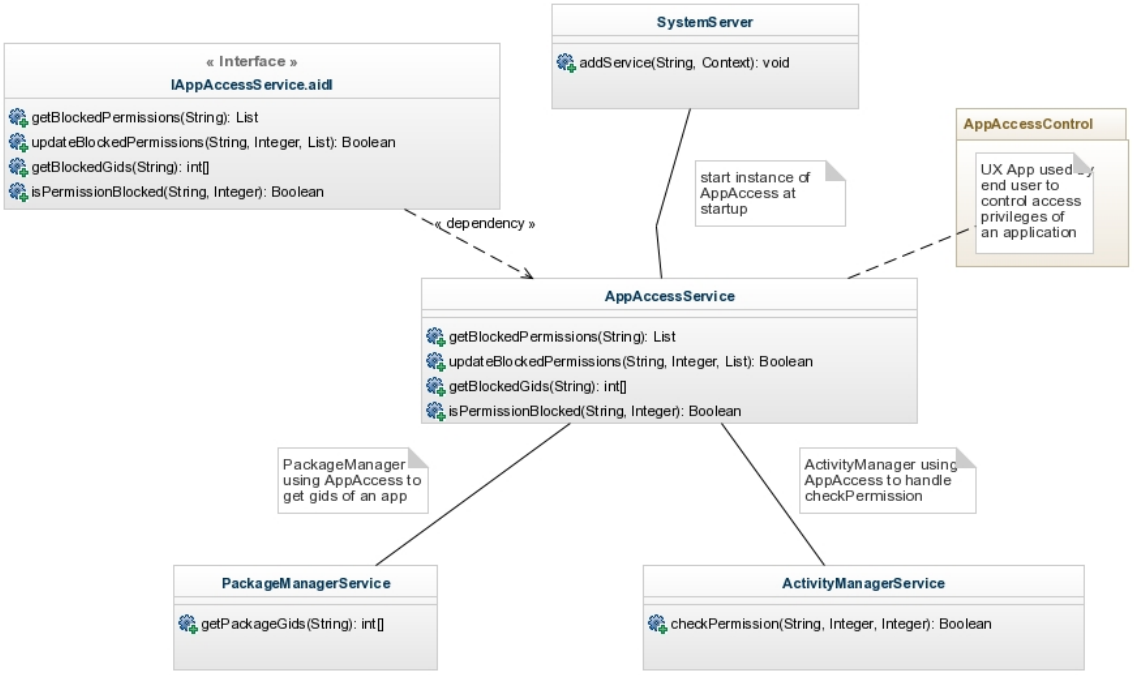


Figure 6.7: Class Diagram of the Implemented Dynamic Permission Control Prototype `AppAccessService`

6.3.3 Repository to Store Blocked Permissions

The list of blocked permissions for each application need to be stored in persistent memory for it to be accessed by `AppAccessService`. An xml format is chosen as a repository for simplicity and consistency. Below is the structure of the xml element which stores the list of blocked permissions. This file named as `acl.xml` file is created, if not already present, at the `/data/` directory in Android device.

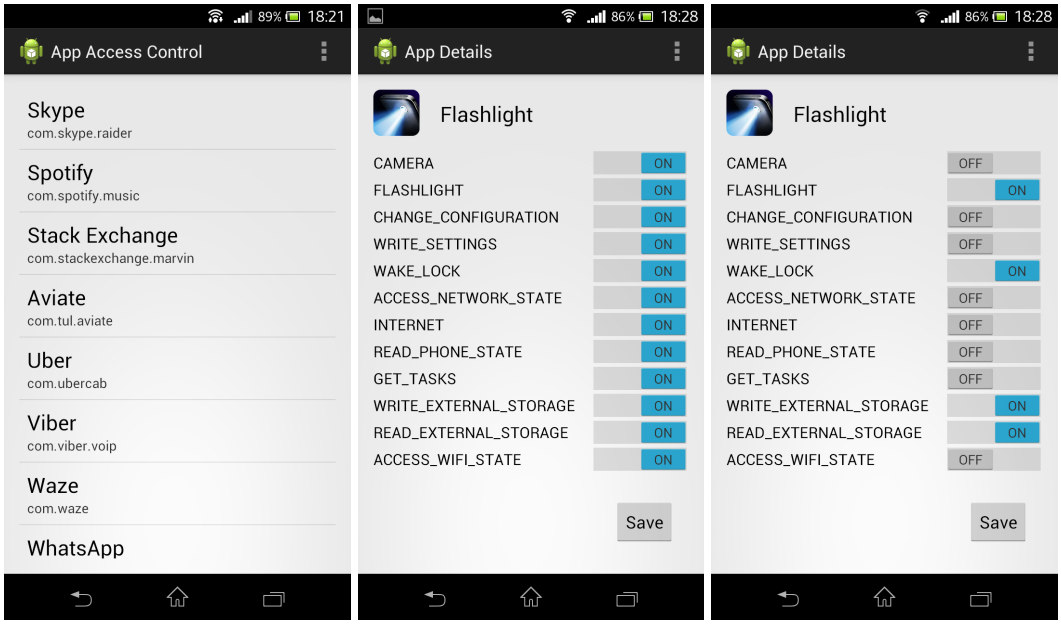
Listing 6.1: Structure of XML File Used as Repository to Store Blocked Permissions

```
<packages>
  <package name="pkg.name" uid="pkguid">
      <blocked-permission>PERMISSION_1</blocked-permission>
      <blocked-permission>PERMISSION_2</blocked-permission>
      ...
  </package>
  ...
</packages>
```

The API methods listed in `AppAccessService`, read from the above xml file and return the results accordingly. Also, it should be noted that the above xml file doesn't store group IDs related to the permission. `AppAccessService` is designed to map the permissions to their corresponding group IDs and the result of getBlockedGids() is returned after the conversion of the blocked permissions from xml file to their equivalent group IDs. This is done using a custom mapping data structure combining the data shown in figures 6.5 and 6.6.With the service well designed the coming sections discuss about the results and other effects of `AppAccessService`.

6.3.4 Results

The above prototype implemented on ice-cream sandwich AOSP repository. The resultant services are then tested on emulator at first in Android. Tests show that `AppAccessService` was capable of enforcing all type of permissions on apps. Figure 6.8 shows the screenshots of App Access Control app which serves as a interface between end user and the `AppAccessService`. This shows how a user controls the permissions for an app.



(a) (b) (c)

Figure 6.8: (a) App Access Control Listing All the Installed Applications on the Device (b) Listing of Permissions Requested for a Particular App, Flashlight along with the Current Status of Permissions (c) Enabling the User to Configure the Permissions for Flashlight Application

6.3.5 Restricting Permissions for Facebook Application

As part of testing the `AppAccessService`, the renown Facebook app is used for blocking both types of permissions. Facebook application was chosen for testing the permission control framework since the app was developed with exceptional handling in most of the cases which arise due to disabling permissions for apps using `AppAccessService`.

6.3.5.1 Linux Process-level Permissions

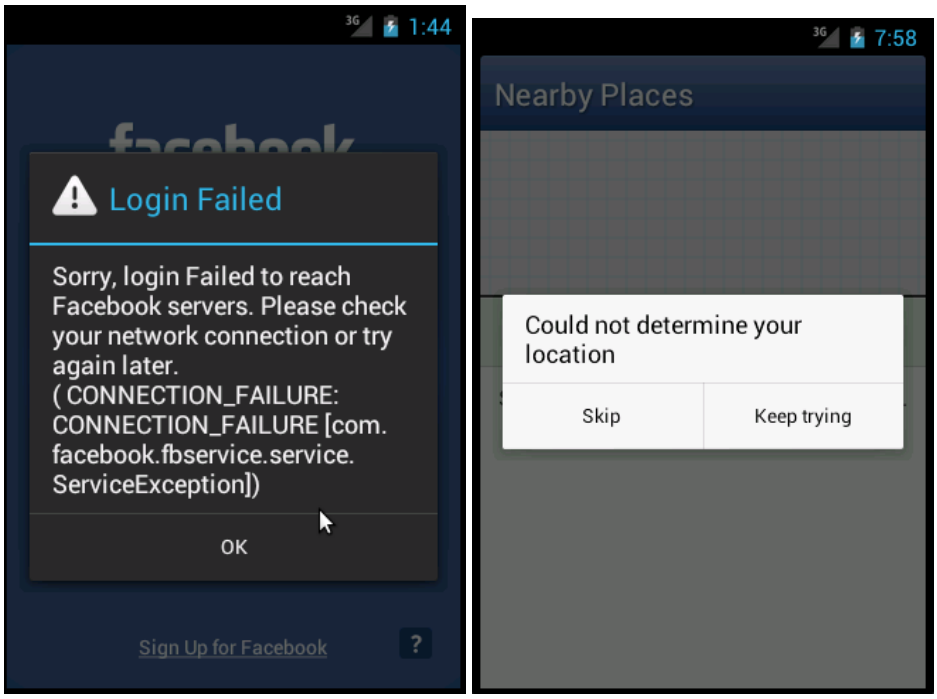
App Access Control app was used to disable `INTERNET`, `CAMERA & LOCATION` permissions for Facebook app. After disabling these permissions, launching the Facebook application would result in restricted access for the app. The log statements tracked show

the group ids with which the Facebook app's Linux process was created. This is depicted in the figure 6.9.

```
I/WindowManager(   77): createSurface Window{41337528 Starting com.facebook.katana paused=false}: DRAW NOW PENDING
V/PackageManager(   77): blockedGids ***** [3003, 1006]
I/ActivityManager(   77): Start proc com.facebook.katana:nodex for activity com.facebook.katana/.LoginActivity: pid=610 uid=10036 gids={1015}
W/NetworkManagementSocketTagger(   77): setKernelCountSet(10036, 1) failed with errno -2
D/dalvikvm(   34): GC_EXPLICIT freed 37K, 4% free 6825K/7107K, paused 5ms+31ms
```

Figure 6.9: Log Depicting Facebook App Launched with Just Single Group-ID 1015 Related to `WRITE_EXTERNAL_STORAGE`. The Blocked Group-IDs 3003, 1006 Refer to `INTERNET` and `CAMERA` Respectively

With `INTERNET` permission blocked, the figure 6.10a shows how the app is unable to login. Correspondingly the log also shows the details error with respect to how the app is unable to resolve the domain name `api.facebook.com` as depicted in figure 6.11.



(a) (b)

Figure 6.10: (a) Facebook App Unable to Login with the Error Prompt Showing `CONNECTION_FAILURE` Message (b) Facebook App Unable to Gather Location Due to Lack of `LOCATION` Permission

```
E/fb4a(:<default>):FeedDataLoader(  626): newer story fetch failed.
E/fb4a(:<default>):FeedDataLoader(  626): com.facebook.fbservice.service.ServiceException: CONNECTION_FAILURE: CONNECTION_FAILURE
E/fb4a(:<default>):FeedDataLoader(  626):       at com.facebook.fbservice.ops.DefaultBlueServiceOperationFactory$DefaultOperation.b(DefaultBlueServiceOper
E/fb4a(:<default>):FeedDataLoader(  626):       at com.facebook.fbservice.ops.DefaultBlueServiceOperationFactory$DefaultOperation.c(DefaultBlueServiceOper
E/fb4a(:<default>):FeedDataLoader(  626):       at com.facebook.fbservice.ops.DefaultBlueServiceOperationFactory$DefaultOperation$4.run(DefaultBlueService
E/fb4a(:<default>):FeedDataLoader(  626):       at android.os.Handler.handleCallback(Handler.java:605)
E/fb4a(:<default>):FeedDataLoader(  626):       at android.os.Handler.dispatchMessage(Handler.java:92)
E/fb4a(:<default>):FeedDataLoader(  626):       at android.os.Looper.loop(Looper.java:137)
E/fb4a(:<default>):FeedDataLoader(  626):       at android.app.ActivityThread.main(ActivityThread.java:4340)
E/fb4a(:<default>):FeedDataLoader(  626):       at java.lang.reflect.Method.invokeNative(Native Method)
E/fb4a(:<default>):FeedDataLoader(  626):       at java.lang.reflect.Method.invoke(Method.java:511)
E/fb4a(:<default>):FeedDataLoader(  626):       at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:784)
E/fb4a(:<default>):FeedDataLoader(  626):       at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
E/fb4a(:<default>):FeedDataLoader(  626):       at dalvik.system.NativeStart.main(Native Method)
W/fb4a(:<default>):BlueServiceQueue(  626): Exception during service
W/fb4a(:<default>):BlueServiceQueue(  626): java.net.UnknownHostException: Unable to resolve host "api.facebook.com": No address associated with hostname
W/fb4a(:<default>):BlueServiceQueue(  626):     at java.net.InetAddress.lookupHostByName(InetAddress.java:400)
W/fb4a(:<default>):BlueServiceQueue(  626):     at java.net.InetAddress.getAllByNameImpl(InetAddress.java:242)
W/fb4a(:<default>):BlueServiceQueue(  626):     at java.net.InetAddress.getAllByName(InetAddress.java:220)
```

Figure 6.11: Log Depicting Facebook App Unable to Resolve the Domain Name `api.facebook.com` Due to Lack of `INTERNET` Permission

6.3.5.2 Framework-level Permissions

Just the same way as the Linux process level permissions, the permissions enforced at the framework level also are tested with Facebook application. The most notable `LOCATION` permission is disabled for Facebook application which results in the failure prompt as shown in figure 6.10b.

6.4 `AppAccessService` to Block Permission Re-delegation

6.4.1 Problem due to intents

As discussed above the same Facebook application launched by disabling the `CAMERA` permission. But the figure 6.12 shows the camera functionality still working in Facebook application. This failure in letting the Facebook application use camera even after blocking by `AppAccessService` can be attributed to Permission Re-delegation. Blocking the permissions for an application only restricts the API calls that originate from that application in Android. After a careful analysis of the Facebook application it can be understood that CameraActivity in Facebook, with package name `com.facebook.katana`, launches the camera activity in the device's camera denoted by the package name `com.android.camera`. In this way even after blocking the permission for an application, the

Figure 6.12: Camera Being Used by Facebook Even after Launching the App after Disabling the Camera Permission Using `AppAccessService`

app can still get to use some of the blocked resources with the help of Intents. To avoid this `AppAccessService` was used to block Permissions Re-delegation attack.

6.4.2 Problem Analysis and Requirements

The previous chapters already demonstrated in detail about permission re-delegation attack that is feasible using the IPC mechanisms in Android. Although the above projects in Chapter 4 show static analysis of Android applications to list the Intents (Android's IPC mechanism) that can be a potential cause for permission re-delegation attack. The static analysis also has a lot of shortcomings due to which the attacks cannot be guaranteed. This phase of the project tries to evade the permission re-delegation attack at runtime using the `AppAccessService`.

Consider app A is trying to invoke app B using intents, let $P(X)$ denote the original permissions of app X. Assume $P^{A}(B)$ indicates permissions of app B when launched by

app A expecting data to be exchanged from app B to app A. This thesis proposes the following as the requirement in order to block permission re-delegation,

$$P^{A}(B) = P(A) \cap P(B)$$

The key point in the above scenario is identifying if the control is coming back to the calling application, which in this example is app A. The previous chapters had already showed that an intent can be launched from Activities majorly using the variants API methods `startActivity()`, `startActivityForResult()`. Among these Chapter 4, already describes `startActivityForResult()` as the method which is susceptible for launching other Activities using intents and returning the control back to the caller Activity. The below subsections demonstrate the design and implementation aspects of using `AppAccessService` to block permission re-delegation attack.

6.4.3 Design

The previous section identifies the API methods to be fixed as `startActivityForResult()` method of Activity class. On a higher level, the solution proposed is to identify the intents passed to this API method, the intent is then mapped to the concerned application. Consider the application from which the API call is being made for `startActivityForResult()` as the *caller app* and the application to which the intent is mapped as the *callee app*. At each invocation `startActivityForResult()`, both the *caller* and *callee* apps are identified. If the *caller app* and *callee app* are not the same, then ideally there is a chance of permission re-delegation attack. Even though the above condition does eliminate permission re-delegation completely, but this would also block applications not to load properly in the first place. The reason being application launch it self uses the same API call from Launcher application in Android. In order to avoid such application loading problems, if the *caller app* is any of the preloaded Android applica-

tions on the device then these API calls are exempted from this check. With the above assumption, the figure 6.13 shows the updated class diagram. As highlighted the `startActivityForResult()` methods from Activity interacts with the `AppAccessService` to `blockPermissionRedelegation()`.

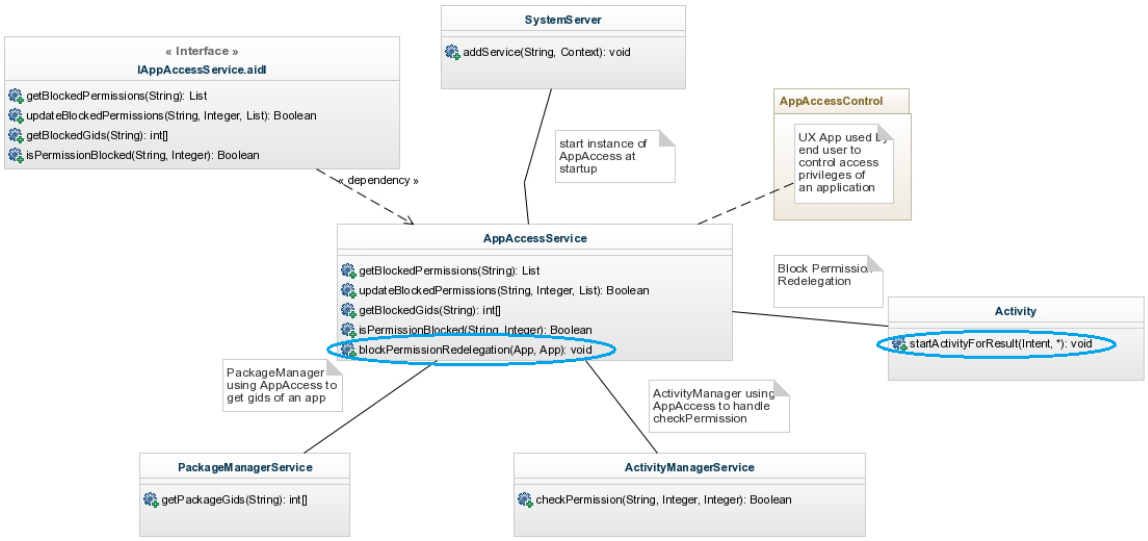


Figure 6.13: Updated Class Diagram of `AppAccessService` Depicting the New `blockPermissionRedelegation()` Method API

6.4.4 Implementation

The design subsection elucidates what exactly should be in place to block permission re-delegation attack. Since `AppAccessService` already has the capability to block permissions for the apps, the same can be used to block permission re-delegation attack. The idea is to identify the permissions of *caller* and *callee apps*, as defined in the requirements section identify the permissions of *callee app* which doesn't satisfy the condition $P^{A}(B) = P(A) \cap P(B)$. The permissions identified above constitute the set of blocked permissions for *callee app*, these can be blocked using `updateBlockedPermissions()` method of `AppAccessService`. Once the control is given back to the *caller app*, the previous block operation can be undone to restore *callee app* to its usual state with respect

to its access control permissions. Listing 6.2 shows the high level pseudocode for this operation.

Listing 6.2: Pseudocode for blockPermissionRedelegation() Method

```
void blockPermissionRedelegation(String callerApp, String calleeApp) {
    Set callerPermissions = getCurrentPermissions(callerApp);
    Set calleePermissions = getCurrentPermissions(calleeApp);
    Set blockedCalleePermissions = {};
    foreach (permission : callerPermissions) {
           if ( !calleePermissions.contains(permission) ) {
           blockedCalleePermissions.add(permission);
       }
    }
    // API call to AppAccessService to update permissions for callee app
    updateBlockedPermissions(calleeApp, blockedCalleePermissions);
}
```

6.4.5 Effectiveness

Blocking permissions for apps temporarily to avoid permission re-delegation although seems to effect other applications considerably. But it actually doesn't have a problem since, the way Android is designed as a single app running in the foreground. Although Android supports multiple processes to run in the background, at a given time only one application seem to be running at the foreground for the user. This feature in Android helps to support the proposed design to block permission re-delegation in this thesis. Moreover, the background applications are not guaranteed to run since Android system kills the least recently used background applications to free memory to accommodate

for other active applications. In the same way the *callee app* can be restarted in order to enforce the new permissions during the even of permission re-delegation.

6.4.6 Results

The proposed design has been implemented as a prototype on the ice-cream sandwich Android repo base. The scenario presented in the motivating example in Figure 1.1 is recreated. To recapitulate the scenario involves a dangerous app without `CAMERA` permission trying to access the resource using deputy application in this case Barcode Scanner app represented by package `com.google.zxing.client.android`.

With the scenario elucidated, the figure 6.14 shows the logcat statements which clearly show how the permissions of *callee app* are blocked to make sure the *caller app* doesn't get any extra information from resources which are not accessible for *caller app*. The log files have been split for better understanding, it should be noted that the below log constitutes of a single action when loading the Barcode Application from the dangerous app to use camera. The resultant response presented to the user in the foreground is depicted in figure 6.15. Thus the permission re-delegation/ permission escalation attack is shown to be evaded with the help of `AppAccessService`.

6.5 Comparing `AppAccessService` with AppOps

As discussed previously, AppOps was released by Android for a brief timespan. The AppOps also provides the user to controlled fine grained permissions for applications on the device. On the foreground, both AppOps and AppAccesscontrol app look much similar and the same is depicted in figure 6.16. Even though the interfaces look similar, the internal design concerned with how an Android application is being restricted differ a lot. For instance, `AppAccessService` restricts the permissions of applications by loading the app with a restricted set of permissions, and inserted code to check if permission is

blocked by `AppAccessService` is present majorly at two places: 1) `PackageManager-Service` for gids 2) `ActivityManagerService` for permission checks.

On the contrary for AppOps, restricting the Applications to various services is done by intercepting about a dozen services. The following code analysis in figure 6.17 shows that AppOps had inserted a bunch of permission checking functions in a variety of services[35]. In this aspect, the approach proposed in this thesis for restricting the permissions of applications at runtime is much simpler than that of AppOps.

```
I/startActivityForResult(  496): Caller Package: com.example.dangerous.app
I/startActivityForResult(  496): Callee Package: com.google.zxing.client.android
I/startActivityForResult(  496): Caller permissions: [READ_CALENDAR, WRITE_CALENDAR, INTERNET]
I/startActivityForResult(  496): Callee permissions: [CAMERA, INTERNET, VIBRATE, FLASHLIGHT,
READ_CONTACTS, READ_HISTORY_BOOKMARKS, WRITE_EXTERNAL_STORAGE, CHANGE_WIFI_STATE,
ACCESS_WIFI_STATE]
I/startActivityForResult(  496): AppAccess Working
I/AppAccessService(   77): Caller gids: [3003]
I/AppAccessService(   77): Callee gids: [1006, 3003, 1015]
```

(a)

```
I/AppAccessService(   77): Blocked callee gids: [1006, 1015]
I/AppAccessService(   77): Blocked callee permissions: [CAMERA, VIBRATE, FLASHLIGHT,
READ_CONTACTS, com.android.browser.permission.READ_HISTORY_BOOKMARKS, WRITE_EXTERNAL_STORAGE,
CHANGE_WIFI_STATE, ACCESS_WIFI_STATE]
I/AppAccessService(   77): Xml file updated
I/AppAccessService(   77): <?xml version="1.0" encoding="UTF-8"?><app-access-list><package
name="com.google.zxing.client.android"
uid="10036"><blocked-permission>CAMERA</blocked-permission><blocked-permission>VIBRATE</blocked-p
ermission><blocked-permission>FLASHLIGHT</blocked-permission><blocked-permission>READ_CONTACTS</b
locked-permission><blocked-permission>com.android.browser.permission.READ_HISTORY_BOOKMARKS</bloc
ked-permission><blocked-permission>WRITE_EXTERNAL_STORAGE</blocked-permission><blocked-permission
>CHANGE_WIFI_STATE</blocked-permission><blocked-permission>ACCESS_WIFI_STATE</blocked-permission>
</package></app-access-list>
```

(b)

```
I/ActivityManager(   77): START {act=com.google.zxing.client.android.SCAN
cmp=com.google.zxing.client.android/.CaptureActivity (has extras)} from pid 496
V/BLOCKEDGIDS??!!(   77): Proc: com.google.zxing.client.android blockedGids: [1006, 1015]
I/ActivityManager(   77): Start proc com.google.zxing.client.android for activity
com.google.zxing.client.android/.CaptureActivity: pid=521 uid=10036 gids={3003}
I/a       (  521): Using implementation class
com.google.zxing.client.android.common.executor.HoneycombAsyncTaskExecInterface of interface
com.google.zxing.client.android.common.executor.AsyncTaskExecInterface for SDK 11
I/WindowManager(   77): createSurface Window{411f6eb8
com.google.zxing.client.android/com.google.zxing.client.android.CaptureActivity paused=false}:
DRAW NOW PENDING
I/WindowManager(   77): createSurface Window{412f0160 SurfaceView paused=false}: DRAW NOW PENDING
I/a       (  521): Using implementation class
com.google.zxing.client.android.camera.open.GingerbreadOpenCameraInterface of interface
com.google.zxing.client.android.camera.open.OpenCameraInterface for SDK 9
V/EmulatedCamera_Factory(   36): getCameraInfo: id = 0
V/EmulatedCamera_Camera(   36): getCameraInfo
I/GingerbreadOpenCamera(  521): Opening camera #0
I/AppAccessService(   77): *** PERMISSION BLOCKED *** Permission: CAMERA, Process:
com.google.zxing.client.android
E/BLOCKEDPERMISSION(   77): User blocked android.permission.CAMERA for process with uid 10036
W/ServiceManager(   36): Permission failure: android.permission.CAMERA from uid=10036 pid=521
E/CameraService(   36): Permission Denial: can't use the camera pid=521, uid=10036
W/CaptureActivity(  521): Unexpected error initializing camera
W/CaptureActivity(  521): java.lang.RuntimeException: Fail to connect to camera service
```

(c)

Figure 6.14: (a) `startActivityForResult()` Method Listing Permissions and Group-IDs of Both *Caller & Callee Apps* (b) Shows `AppAccessService` Blocking All the Extra Permissions of *Callee App* (c) Shows the Launch of *Callee App* with Restricted Permissions, Mainly `CAMERA`, The Activity in Barcode Scanner Unable to Load Camera Due to Lack of Privilege

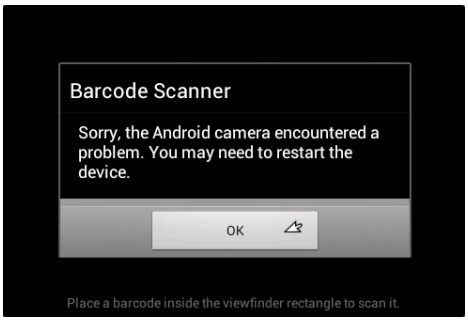


Figure 6.15: Error Prompt on Barcode Scanner Application Generated by Blocking Permission Re-delegation.

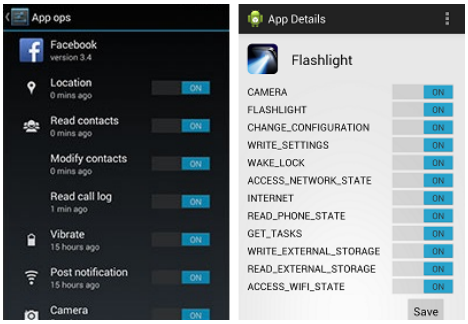


Figure 6.16: Similarity Between the Interfaces of AppOps Vs AppAccessControl User Facing Apps

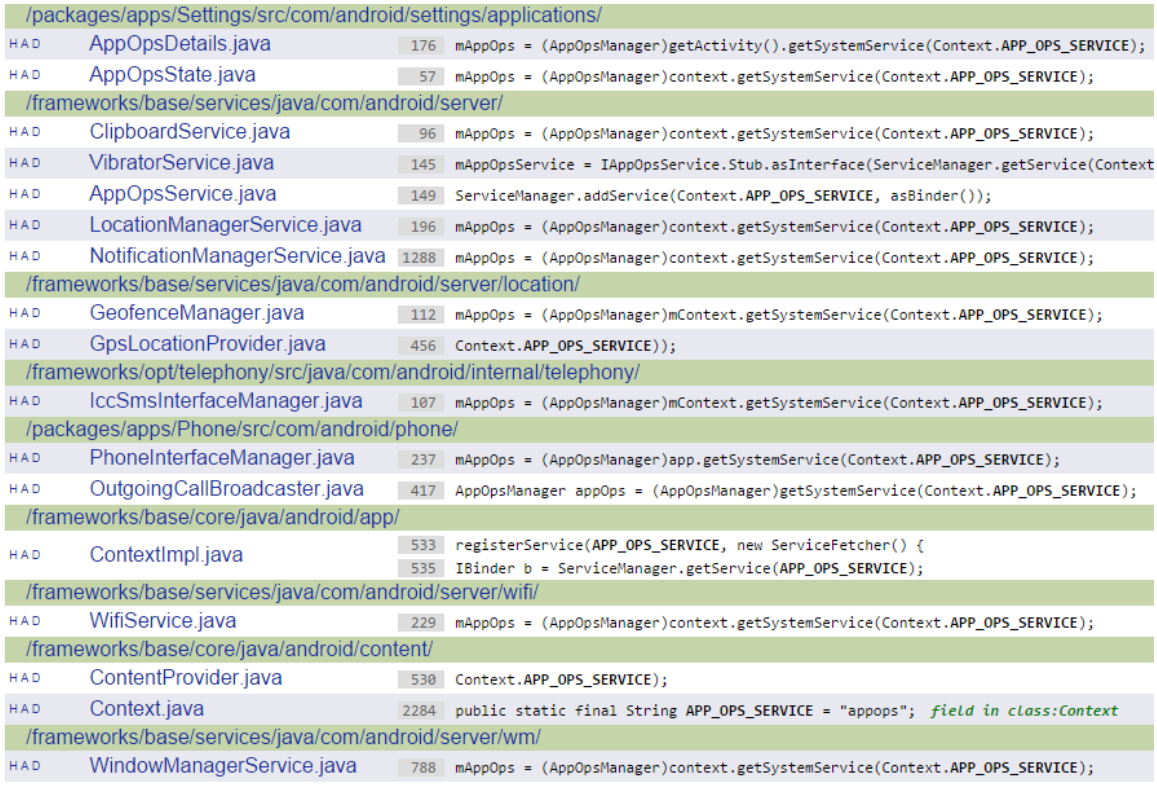


Figure 6.17: Traces of AppOps Being Present Across All the Services in Android Source

Chapter 7

BACKGROUND PERMISSIONS USAGE OF ANDROID APPLICATIONS

`AppAccessService` proposed in the previous chapter is capable of blocking permission re-delegation attack and more importantly providing the user with the ability to control the access privileges of Android applications installed on the device. One of the major problem with the permission model in Android is that if an *app* is granted a particular *permission* then the *app* can make use the concerned resources as long as the app is installed on the device. As described previously, these permissions are enforced silently at execution time by the *app*. Hence there seem to be no restrictions what so ever on the amount of usage of the resources governed through the *permission* for the *app*.

Of the related works in this aspect with providing a user centered approach, Apex[30] proposed by Nauman *et al.* suggested an extension to the Android permission model by enforcing user-defined constraints. Bai *et al.* proposed a context-aware usage control for Android[7], which takes into consideration the context data of user to enhance the protection of user's privacy. Further work in this thesis aims to provide a user centered approach to enhance privacy by taking into consideration by taking the context of the application running. Also, during the course of this work involving restricting applications based on running context, a similar feature was released as part of iOS 8 privacy controls which enables the user to control Location services with regards to the running status of the app. A further description of this feature is presented in section 7.5.

7.1 Problem Analysis and Requirements

As described before the permissions are silently enforced at runtime for the Android applications, this doesn't stop the application from accessing resources the app has been

privileged to use even if it is not strictly required to serve end user. Especially if the resources effect the privacy of the user, the risk is even higher of the application unnecessarily accessing the user-sensitive data from the device. A motivating example which explains this scenario, relates to those kind of applications which utilize the user's current location to identify nearby related services for the user. There are many such renowned applications and Foursquare, Yelp, Google Places are a few among this list. Although the end user needs these kind of applications to access the current location of the user, it doesn't stop the application from constantly tapping the user's location even if the user is not actively using these applications.

As seen in related work by Felt *et al.*[23] suggests that very less percentage of users pay attention to check and understand the permissions an application is requesting during installation. This definitely proves that completely leveraging of the end-user to take care of his/her privacy is also not a viable option. With the above scenario in mind, this thesis proposes to restrict applications based on the context of the applications running on the device. When this chapter mentions about context of an Android application running, it refers to whether the application is running in the background or it is running in foreground. Thus this project aims at providing the user to control applications' resource usage based on the running context of the applications. In addition to the `AppAccessControl` which already is proposed to provide the users with a fine-grained permission management system in Android, allowing the users to enforce permissions on the app based on the application's running context also helps the users to ensure their privacy from background spoofing applications.

On the other hand, the permission usage statistics for applications along with their running context definitely helps in identifying malicious applications when implemented on large scale. In general there is a conception that an application is made to undergo vetting process before the applications are made available in Google Play store to be down-

loaded by end user. Even if there exists one vetting process the surmounting statistics on the percentage of malware applications reported[24], makes the vetting process quite ineffective. Many instances where the malware application was taken down from the Google Play store after reporting suggests even more about the ineffectiveness of the vetting process in Google Play store[9]. This thesis also aims at providing a viable solution to identify the background spoofing applications from permission usage analysis. This solution if utilized by the centralized app governing systems like Google Play store, Amazon appstore etc, might be able to identify malicious applications even more effectively and can eventually ensure the security of the end user.

This phase assumes that the permission usage related to the applications which are running in the foreground are not considered a threat to the user's privacy. With the above basic assumption and a clear problem statement, the following are the functional requirements for tracking the background permission usage of Android applications.

7.1.1 Functional Requirements

1. Identify the running context of an Android application.
2. Provide a way for the end user to enforce constraints on permission usage of application based on the running context.
3. Block the application from accessing a particular resource based on the constraint specified by the user on the governing permission.
4. Provide a way for a centralized entity like Play store, to notify background spoofing activity of applications running on user's device.

7.2 Design

The major challenge from the above proposed requirement lies in identifying the proper running status of the application. Refreshing the basics of how an application is run in Android, sure does help in order to correctly identify the running status of the Android application. As described in detail in the previous chapters, an application in Android in the low level is a Linux process(es) which is launched with a unique user ID. Also traditionally Android is a variant of single foreground tasking operating system, which essentially means there is one active application at a given time running in the foreground. Even though there can be multiple applications running in parallel in Android and each of which can be made active by selecting them from the list of running apps. The Figure 7.1 shows the latest rendition, from Android L, of running apps stacked and an application from the stack can be made active.

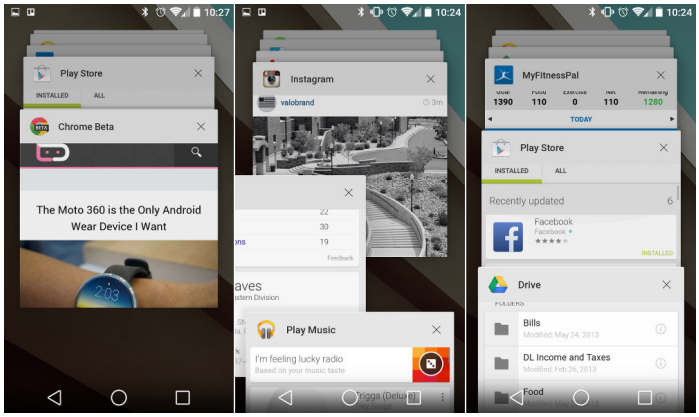


Figure 7.1: Multi-tasking in Android

So even though Android supports multitasking there could only be one foreground application running at a given time. This feature can be taken advantage of in trying to determine the running context of an application. During permission checking, an application can be termed as foreground if an Activity from the application is currently in foreground and vice versa. Once the status of the application is known, based on the

user preference read from `AppAccessService` the permission check can be `ALLOWED` or `DENIED`.

Irrespective of the choice above, the permission usage can be logged for background applications to be further provided as input for the centralized app governing body, which then can decide whether the application is malicious based on the volume of permission usage and the risk factor of the requested permission.

With the above design characteristics, now the higher level design would modify the design proposed for `AppAccessService` earlier to the Figure 7.2.

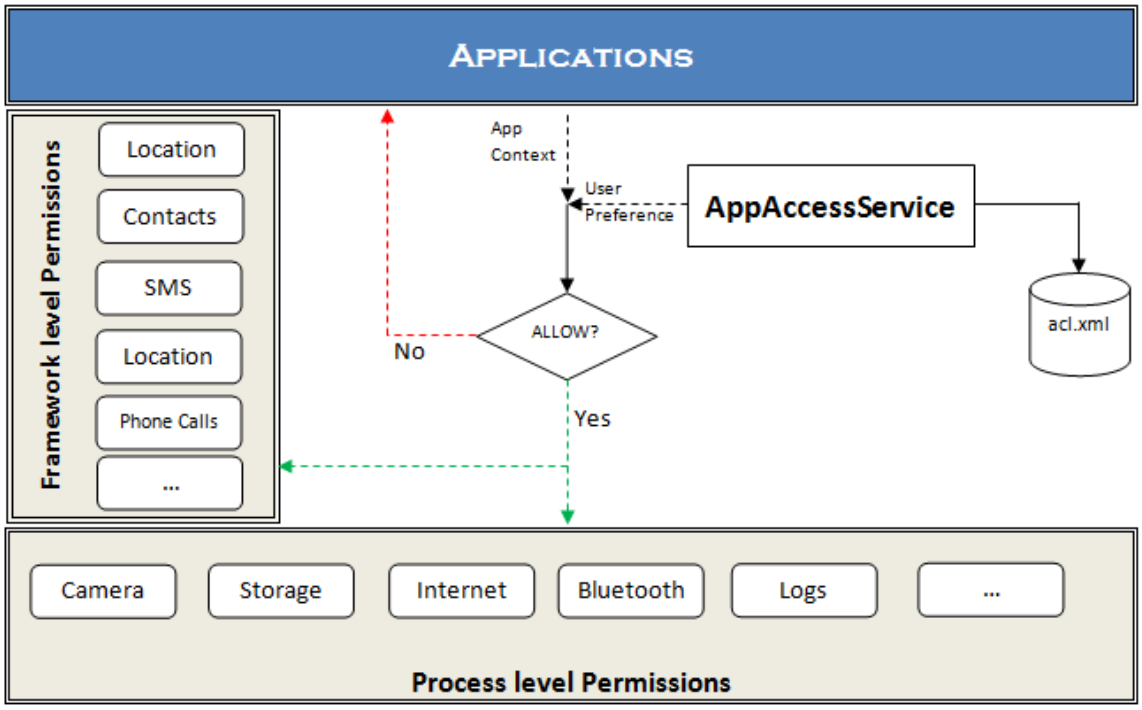


Figure 7.2: Design of Permission Check Being on App's Running Context

7.2.1 Limitations

Although this model aims to protect background spoofing for applications, this works only with the type of permissions which are enforced at Android framework level rather than the ones enforced at a low level Linux group IDs. Although `AppAccessService` can

be made to enforce the Linux group ID level permissions by restarting the application whenever the application running context changes from background to foreground and vice versa, but this approach wouldn't preserve the current state of the application since it needs to be restarted. The major group ID permissions which can be used by third party apps and are considered to govern access to resources sensitive to user's privacy are `EXTERNAL_STORAGE`, `INTERNET`, `CAMERA` and `BLUETOOTH`. Apart from the above permissions, the proposed solution only prevents background spoofing of Android applications with respect to all permissions enforced at Android Framework level.

7.3 Implementation and Results

As described previously, the only API call which is publicly accessible to check for permissions and is called internally by various system level services to check for permissions is `checkPermission()` API call in `ActivityManagerService`. `ActivityManagerService` maintains a stack of Activities that the user is interacting with, labeled as `mMainStack`. Ideally the top of the stack in `mMainStack` represents the foreground Activity that is shown to the end user. Whenever an API call to `checkPermission()` is made, the package name is identified for the corresponding application and then it is compared to that of the application the top Activity in the stack belongs to. If they both match the application making the API call is labeled as a foreground application at the given instant otherwise it is labeled as a background application. Listing 7.1 shows the code snippet representing the simple way of labeling the running status of an Android application in `ActivityManagerService`.

Listing 7.1: Code Snippet Determining Application's Running Status

```
public int checkPermission(String permission, int pid, int uid) {
    ...
    boolean isForeground = false;
```

```
        String packageName = getPackageName(uid);
        ActivityRecord top = getTopOfStack(mMainStack);
        isForeground = top
                       .intent
                       .getComponent()
                       .getPackageName()
                       .equalsIgnoreCase(packageName);

        ...
    }
```

Once the application running context is determined as above, the rest is quite trivial as described in the previous chapter with respect to using `AppAccessService` to block the permission call made by the application. The status of the permission API call can also be logged for further parsing from Play store servers as proposed in the problem statement.

7.3.1 Prototype

A prototype was built to identify and block the permissions for applications based on running context. The figure 7.3 shows the change in the interface built for the user, to control framework level permissions according to the application running context, and shows how it is transformed from the original interface depicted in figure 6.8. The figure clearly shows the preliminary difference in how the permission control is represented to the user. All the depicted permissions for Facebook application, can now be set based on the running context of the application. Not all permissions can be chosen to be enforced based on the app's running context, figure 7.4 shows how the App Access Control interface differs for the renowned Flashlight application.

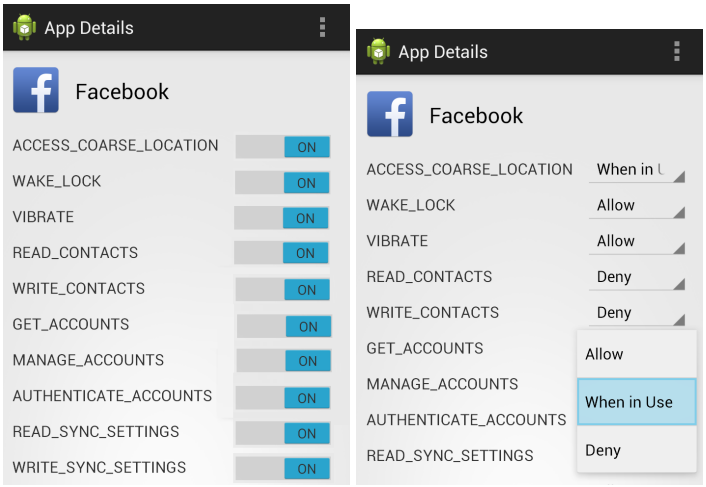


Figure 7.3: Old Vs New Interface of `AppAccessService` Enabling the Users to Configure Permissions Based on Running Context of the Application

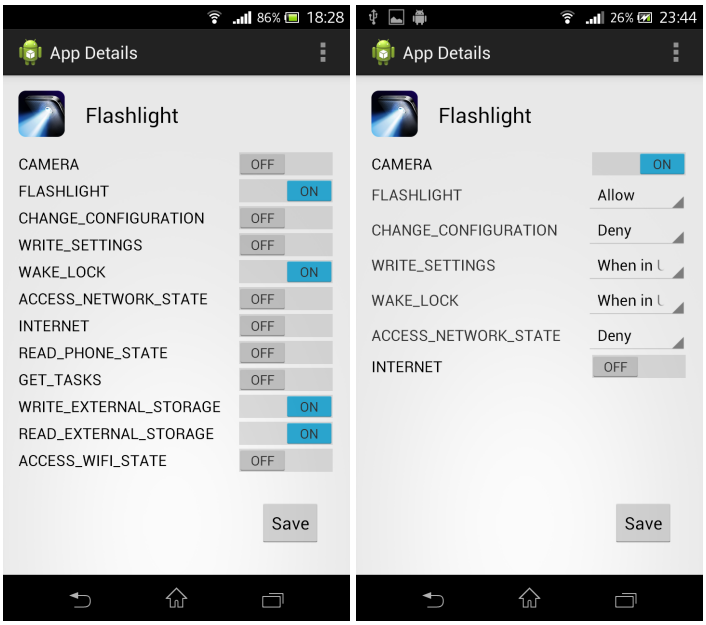


Figure 7.4: Old Vs New Interface of `AppAccessService` for Flashlight Application

As we can see a framework level permission can have three choices now, instead of two as seen before, *Always/Never/When In Use*. By choosing the *When In Use* option, the user would be able to selectively allow the permission only when the app is running in the foreground. All the background permission checks of the app with respect to that

	Total Count	Background Count	Percentage
All apps	50342	47172	93.70%
Most used apps sample	2876	1847	64.22%
Spyware apps sample	308	134	43.51%

Table 7.1: Permission Check Count Analysis for Sample Applications

permission are denied. Thus the user can feel safe of sensitive data not being tracked when the app is not active.

7.3.2 Permission Usage Analysis

A sample of the applications were tested to find out the permission usage statistics especially with respect to the running context of the application. The sample consisted of set of ten top downloaded applications currently in Android Play store along with a small sample of five known malware/spyware applications in Android. The sample of these 15 applications along with a set of more than 30 default applications that come with the device already are tested to track the permission usage of applications with respect to the application's running context. The permission usage analysis was determined by running through various possible use cases of the sample applications and all the calls to the public API `checkPermission()` are to tracked obtain a list of detailed statistics. The table 7.1 summarizes the total number of permission checks made by applications during the period of testing.

As depicted in table 7.1 more than 83% of the overall 50342 permission checks are done by the default system applications which is quite understandable since the sample is dominated by stock Android applications preloaded on the device. Even though the table shows more than 60% of permission checks done by the sample of most used applications

turns out to be background checks, this data cannot be used to classify any of the background checks as a threat to user's privacy. The same can be attributed to the spyware sample as well. Though table 7.1 cannot be used to analyze the list of permission checks that can be categorized as a threat to user's privacy, the data is actually presented to get an idea of the amount of processing which goes in the background with respect to simple actions made by end-user by using the applications in the foreground.

Further analysis involved tracking of permission checks of only the sample applications barring the stock Android applications, and also taking into consideration only the permissions which relate to user's privacy like `LOCATION`, `SMS`, `PHONE_STATE`, `CONTACTS`, `RECORD_AUDIO` etc.

7.3.2.1 Spyware Sample Analysis

The five malware/spyware applications chosen are listed along with their known attacks possible which let the application steal information considered as sensitive to the end user.

SMS RU - org.me.androidapplication1

Contains a known Trojan capable of sending SMS from the effected device to premium numbers like 3353, 3354[38].

Roidsec - cn.phoneSync

Contains a Trojan capable of sending sensitive user information like SMS, call log, contacts, location and other usage information to remote location. Once installed the application doesn't have a launcher[36].

Ear spy - com.microphone.earspy

Known spyware tool available in Play store. Capable of eavesdropping from the surroundings.

	LOCATION	SMS	IMEI	RECORD	CONTACTS
SMS RU		✓			
Roidsec	✓	✓	✓		✓
Ear spy			✓	✓	
Antsmasher	✓		✓		
Android SPY		✓			

Table 7.2: Critical Background Permission Checks for Malware Applications Sample

Antsmasher - com.bestcoolfungames.antsmasher

A spy-ware game application capable of stealing information in the background while the user is playing the game[37].

Android SPY - sms.monitoring.app

A spyware tool capable of sending the received SMS to a configured email.

With the background on the sample spyware applications, the table 7.2 shows the critical background permission checks detected with the proposed methodology in this thesis.

It is to be noted that the spyware applications tested didn't completely cover all possible usecases during testing and the results shown in the table 7.2 only correlate to the limited usage of the application. Even this limited usage proved to be significant in detecting the background permission checks made by these applications, especially with respect to the `SMS & CONTACTS`. This demonstrates the effectiveness of the approach proposed in this thesis.

7.3.2.2 Most Used Sample Analysis

The most used applications sampled for testing include renown applications in the following six categories in the Android application market.

Social	Travel	Music
• Facebook • Messenger	• Google Maps • Foursquare	• Pandora
Arcade	**Shopping**	**Utilities**
• Prize Claw	• REI – Shop Outdoor Gear	• Firefox • Clean Master • Flashlight

Figure 7.5: Most Used Sample Applications

The testing on the most used applications sample, identified the `LOCATION` permission being used in the background by almost all applications in the sample. Especially with respect to Facebook app (`com.facebook.katana`), the total number of permission checks for `ACCESS_FINE_LOCATION` in the tested usecases turned out to be 100% in the background (64 out of 64 checks). Table 7.3 depicts the traces of background permission checks from the applications identified with regards to the critical permissions.

The instance of permission check involving the `KILL_BACKGROUND_PROCESS` permission. Typically Android applications bearing this permission have the capability to kill few apps running in the background majorly to conserve memory and CPU time. In the same way Clean Master, one of the famous utility app, tries to kill few applications in the background and the same traces have been identified. This instance also demonstrates the effectiveness of the proposed approach in tracing the background activities of applications.

	LOCATION	IMEI	CONTACTS	RECORD	KILL_BG_PROC
Facebook	✓				
Messenger	✓		✓		
Foursquare	✓				
REI	✓				
Prize Claw	F	F		F	
Clean Master		✓			✓
Flashlight	F	F		F	
Google Maps	✓	✓			

Table 7.3: Critical Background Permission Checks for Most-used Applications Sample. F - Foreground Permission Checks.

7.3.2.3 Static Analysis Based on Permission Usage

Other than `LOCATION & READ_PHONE_STATE` permissions which being tracked in the background, the other notable finding from this sample are the permission checks originating from Prize Claw application to the permissions `RECORD_AUDIO & CAMERA`. Although the application do not bear these particular permissions, further static analysis of the application determined that these checks originated due to the probing for permission in the background. Figure 7.6 depicts code originated from Prize Claw application that is responsible for the permission checks for unauthorized permissions.

Once the permission check statistics for the application are known, static analysis of the application in order to identify the code snippet causing the permission check in the first place becomes easier. Since the details of the outcome are known and hence static analysis is more result-driven reverse engineering. Thus this approach can be used to identify vulnerable applications in Android after being released for usage of end user.

7.4 Crowd-sourcing Way of Finding Vulnerable Apps Post Release

The previous section demonstrates the effectiveness of permission usage of applications taking into consideration the running context of the application. The same approach can be utilized in identifying potential malicious apps that put the user's privacy at risk. The following steps demonstrate the design proposed for this scenario.

1. Traditionally Google Play acts as a centralized authority to load Android applications onto a device.

2. Each device keeps track of the permission usage of applications with respect to the running context.

3. The permission usage statistics are collected by another centralized authority for analysis.

4. The applications found misusing the permissions by the analysis of permission usage statistics are further made to undergo careful vetting process.

5. The vetting process aims at causal analysis of the abnormal permission usage statistics.

6. Based on the causal analysis the malicious applications which potentially cause risk to user's privacy can be identified to take further action.

The design proposed involves the methodologies of crowd sourcing where in each android device acts as an individual entity in crowd sourcing rather than making the end user boar the onus of securing privacy by aptly choosing application settings using `AppAccessService`. The proposed design is pictorially represented in figure 7.7.

7.5 iOS 8 Privacy Controls: Location Services

iOS 8 is the latest version of Apple's mobile operating system, which was released recently during September 2014. iOS is renown for its famous security update in iOS 6 which provides the user with a privacy controls enabling the users to control access to sensitive data for applications on the device. The figure 7.8 depicts the privacy settings provided in iOS 6 which remained unchanged in iOS 7 as well.

The advent of smart phones revolutionized the mobile marketing which is targeting the audience bearing smart phones to gather important user sensitive information especially user's location. Most of the applications tend to transmit user location along with preferences to the marketing agencies risking the privacy of many users[25]. In addition to providing the privacy settings, Apple also provided a way for the user to allow Location services *while using the app* as well instead of the previous seen *Allow* and *Deny*. The following is the excerpt from iOS 8 Security documentation by Apple.

"From Settings, access can be set to never allowed, allowed when in use, or always, depending on the apps requested location use. Also, if apps granted access to use location at any time make use of this permission while in background mode, users are reminded of their approval and may change an apps access. "[6]

The figure 7.9 shows the change in the privacy control feature in iOS 8 especially related to Location Service. This updated privacy setting in iOS 8 looks a lot similar to the interface in the proposed design as shown in figure 7.3. Even though the ideologies of the iOS 8 updated privacy settings and the `AppAccessService` permission enforcement based on running context of the application are similar. The fact that iOS 8 is developed as a commercial product environment unlike open sourced Android makes it completely coincidental for this similarity. Moreover the similar idea already put into action although

not in a fine-level granularity, also affirms credibility and the importance of the approach proposed in this thesis.

```
private boolean isCameraAvailable()
{
  Context localContext = (Context)this.contextRef.get();
  boolean bool = false;
  if (localContext != null)
  {
    int i = localContext.getPackageManager().checkPermission("android.permission.CAMERA",
    bool = false;
    if (i == 0)
    {
      Intent localIntent = new Intent("android.media.action.IMAGE_CAPTURE");
      int j = localContext.getPackageManager().queryIntentActivities(localIntent, 65536).s
      bool = false;
```

(a)

```
const-string v3, "android.permission.RECORD_AUDIO"

invoke-static {}, Landroid/os/Binder;->getCallingUid()I

move-result v4

invoke-virtual {v0, v4}, Landroid/content/pm/PackageManager;->getNameForUid(I)Ljava/lang/String;

move-result-object v4

invoke-virtual {v0, v3, v4}, Landroid/content/pm/PackageManager;->checkPermission(Ljava/lang/String;Ljava/lar

move-result v0
```

(b)

```
public final void a()
{
  if (!this.g)
  {
    this.g = true;
    if (this.d != null) {
      try
      {
        b.a(this, "Starting recorder");
        this.d.f();
        return;
      }
      catch (Throwable localThrowable)
      {
        b.a(this, "Error starting recorder", localThrowable);
      }
    }
  }
  for (;;)
  {
    bb localbb = this.c;
    localbb.a(4);
    return;
    b.c(this, "Recorder already started");
  }
}

public final void a(final cx paramcx)
{
  b.a(this, "Capturing audio from recorder");
  if (this.b == null)
```

(c)

Figure 7.6: (a) Code from Millennial Media Ad-packages Used in Prize Claw Application Checking for Availability of `CAMERA` Permission (b) Smali Code Snippet from Inmobi Ad-packages in Prize Claw Probing for `RECORD_AUDIO` Permission (c) Code Snippet from Prize Claw Application Capable of Recording the Audio from the Device If Granted the Permission `RECORD_AUDIO`

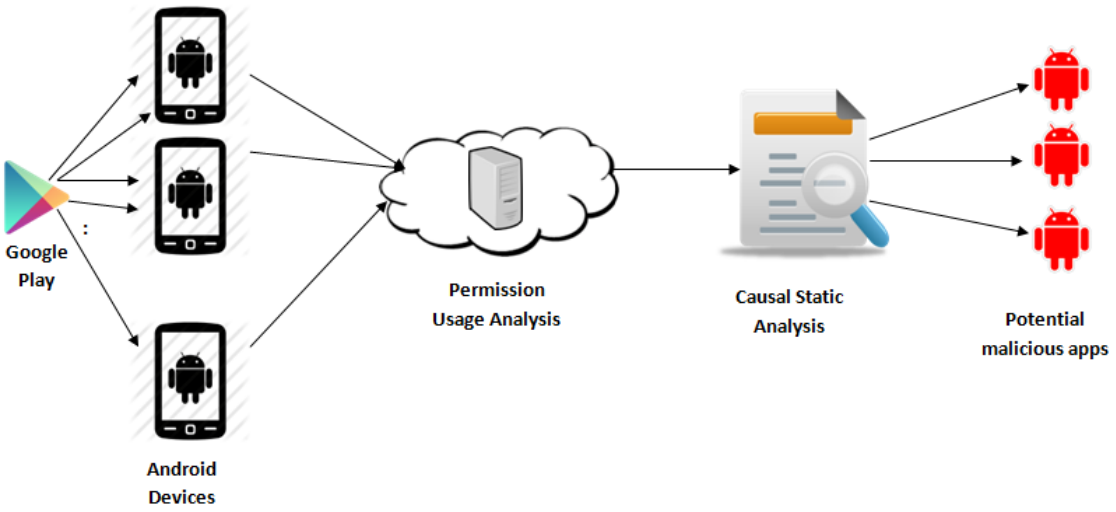
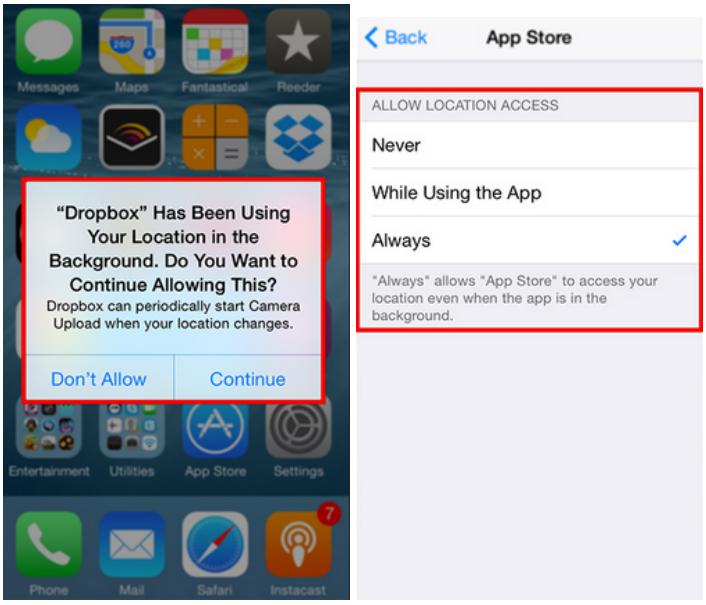


Figure 7.7: Proposed Design for Identifying Malicious Android Applications Post-Release Using the Background Permission Usage Analysis

Figure 7.8: The Privacy Settings in iOS 6 Listing the Apps Which Use Contacts and Location Services



(a) (b)

Figure 7.9: (a) Location Service's Privacy Control Prompting the User to Choose Background Location Setting of Application (b) Privacy Control Updated View of Location Services Having 3 Options Never, While Using the App, Always

Chapter 8

CONCLUSION & FUTURE WORK

Present day smart phones tend to become more powerful and helpful resource for many users. The increasing computing ability of the device is also bringing the users more closer to the smart phones and definitely they are no longer used just as a means of elementary communication like talk and text. The growing computing ability also demands a responsibility of the smart phone to ensure the privacy of the user. With Android having the majority of the smart phone market share, this thesis proposes a better mechanism in Android to enhance user's privacy protection. The current status of Android relies on app-driven permission control management, which makes the user having less control of what privileges to grant to an application in Android.

This thesis proposes an approach to make the permission management of Android applications more user-driven rather than app-driven. Taking into consideration of the current security architecture in Android, a simple and effective security framework is proposed which allows users to control the permissions of applications in Android. Apart from allowing users to choose permissions for the applications on the device, this thesis also showcases the robustness of this simple `AppAccessService`. The same service can be used to block a typical attack in Android called Permission Re-delegation or Permission escalation attack which is inherent due to the IPC design in Android. Also this thesis proposes a novel approach of restricting applications based on running context of themselves. The fact that the application running context permission usage proposed to be used in a crowd sourced manner in assessing the risks of the the app at runtime makes it unique.

Even though this thesis addresses runtime issues and accesses the vulnerability of the application after the malicious app attempts to do some damage, more work needs to be

done in trying to assess the dangers involved in apps accessing resources in the foreground. Thus the future work needs to focus more on extending similar approach proposed to classify the permission checks from the foreground which doesn't relate to the user's activity as a potential threat and should be blocked.

REFERENCES

[1] Building and running | android developers. URL https://developer.android.com/tools/building/index.html.

[2] Android apktool: A tool for reengineering Android apk files. URL https://code.google.com/p/android-apktool/.

[3] Activitymanagerservice.java - platform/frameworks/base - git at google. URL https://android.googlesource.com/platform/frameworks/base/+/android-4.4.4_r2.0.1/services/java/com/android/server/am/ActivityManagerService.java.

[4] smali - An assembler/disassembler for Android's dex format - Google Project Hosting. URL https://code.google.com/p/smali/.

[5] dex2jar: A tool for converting Android's .dex format to Java's .class format, 2013. URL https://code.google.com/p/dex2jar/.

[6] Apple. ios security september 2014. URL https://www.apple.com/privacy/docs/iOS_Security_Guide_Sept_2014.pdf.

[7] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010*. Springer.

[8] BBC. BBC News - Data haul by Android Flashlight app 'deceives' millions, 2013. URL http://www.bbc.com/news/technology-25258621.

[9] UTB Blogs. More evidence of google's play store vetting process. URL http://utbblogs.com/android/more-evidence-of-googles-play-store-vetting-process/.

[10] CNET. Linkedin's app transmits user data without their knowledge, 2012. URL http://www.cnet.com/news/linkedins-app-transmits-user-data-without-their-knowledge/.

[11] CNET. Path shares photos–oh, and uploads your contacts, too, 2012. URL http://www.cnet.com/news/path-shares-photos-oh-and-uploads-your-contacts-too/.

[12] Colt. Android os - processes and the zygote!, 2010. URL http://multi-core-dump.blogspot.com/2010/04/android-application-launch.html.

[13] Android Developers. Process, 2014. URL http://developer.android.com/reference/android/app/Activity.html.

[14] Android Developers. Appopsmanager, 2014. URL http://developer.android.com/reference/android/app/AppOpsManager.html.

[15] Android Developers. Intent, 2014. URL http://developer.android.com/reference/android/content/Intent.html.

[16] Android Developers. Manifest.permission, 2014. URL http://developer.android.com/reference/android/Manifest.permission.html.

[17] Android Developers. System permissions, 2014. URL http://developer.android.com/guide/topics/security/permissions.html.

[18] Android Developers. Android manifest permission protectionlevel, 2014. URL http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel.

[19] Android Developers. Intent, 2014. URL http://developer.android.com/reference/android/os/Process.html.

[20] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, 2011.

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM, Conference on Computer and Communications Security CCS*, 2011.

[22] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.

[23] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: user attention, comprehension, and behavior. In *Symposium On Usable Privacy and Security, SOUPS, '12, Washington, DC, USA - July 11 - 13, 2012*, page 3. ACM, 2012.

[24] Forbes. Report: 97 percentage of mobile malware is on android. this is the easy way you stay safe, . URL http://onforb.es/OOCrZD.

[25] Forbes. Is location based advertising the future of mobile marketing and mobile advertising?, . URL http://onforb.es/1evJLP5.

[26] Mario Frank, Ben Dong, Adrienne Porter Felt, and Dawn Song. Mining permission request patterns from android and facebook applications. 2012.

[27] Dianne Hackborn. Well, darn. It's confirmed Android 4.4 KitKat is missing the App Ops, 2013. URL https://plus.google.com/+DannyHolyoake/posts/FkfBxA5i3iG.

[28] Radhika Karandikar. Android application launch, 2010. URL http://coltf.blogspot.com/p/android-os-processes-and-zygote.html.

[29] Sophos Labs. Android vs. iOS Security | Why iOS is Safer than Android | Sophos Enterprise Mobile Control, 2013. URL http://www.sophos.com/en-us/security-news-trends/security-trends/malware-goes-mobile/why-ios-is-safer-than-android.aspx.

[30] 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. ACM.

[31] Ketan Parmar. In Depth : Android Package Manager and Package Installer, 2012. URL http://www.kpbird.com/2012/10/in-depth-android-package-manager-and.html.

[32] The Honeynet Project. honeynet/apkinspector. URL https://github.com/honeynet/apkinspector/.

[33] Reddit. Why facebook android app needs to read user sms and mms?, 2014. URL http://www.reddit.com/r/WTF/comments/1t5z45/facebook_why_the_hell_do_you_think_its_okay_to/ce4y6x2.

[34] Android Source. Android security overview, 2014. URL https://source.android.com/devices/tech/security/index.html.

[35] Mingshen Sun. Android 4.3 app ops hidden feature analysis | mingshen sun's pearls, 2013. URL http://blog.mssun.me/security/android-4-3-app-ops-analysis/.

[36] Symantec. Android.roidsec technical details | symantec. URL http://www.symantec.com/security_response/writeup.jsp?docid=2013-052022-1227-99&tabid=2.

[37] Virus Total. Antivirus scan for antsmasher6.22.apk at 2014-09-11 05:54:23 utc - virustotal. URL http://bit.ly/1vOO8z3.

[38] VRT. Vrt: Malware on android? big deal! URL http://vrt-blog.snort.org/2010/08/malware-on-android-big-deal.html.

[39] Stephen A. Rago W. Richard Stevens. Advanced programming in the unix environment: Second edition. Addison Wesley Professional, 2005.

[40] Webroot. Webroot Mobile Threat Report 2014, 2014. URL http://www.webroot.com/shared/pdf/WR_MobileThreatReport_v4_20140218101834_565288.pdf.

[41] Wikipedia. APK (file format) — Wikipedia, The Free Encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=APK_(file_format)&oldid=624778628.

[42] Wikipedia. Google Play - Wikipedia, The Free Encyclopedia, 2014. URL http://en.wikipedia.org/w/index.php?title=Google_Play&oldid=624732374.

[43] Network World. Android malware exploding, says Trend Micro, 2012. URL http://www.networkworld.com/article/2160862/network-securityndroid-malware-exploding--says-tre/network-security/android-malware-exploding--says-trend-micro.html.

[44] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing permission usage in android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*.