Journal of Mobile, Embedded and Distributed Systems, vol. VII, no. 3, 2015
ISSN 2067 – 4074

Ensuring Privacy in the Android OS by Hooking Methods in Its API

Andrei-Stefan CONSTANTINESCU
IT&C Security Master
Department of Economic Informatics and Cybernetics
The Bucharest University of Economic Studies
ROMANIA
andconstantinescu@gmail.com

Abstract: The paper presents a mean to ensure security and privacy on Android devices available at the moment by applying a need to know principle for information that the installed applications can access. The presented method requires root access and an Open Source framework that allows hooking functions in the Android system, called Xposed, to be installed.

Key-Words: Security, Privacy, Application, Android, Xposed, Hooking Functions

1. Introduction

The paper talks about ensuring *security* and *privacy* on *Android devices*, by using means to avoid the *unnecessary use* of *granted permissions* for a user-installed application.
In today's context, mobile devices are extremely important. The mass adoption started in 1999 and nowadays, they became extremely popular. For example, the "phone" part in the smartphone has become one of the many things it can actually do, such as Email, Text/Instant messaging, Web surfing, GPS, Camera and so on. People use all those functions daily, through various applications installed on their smartphones and this obviously raises a concern regarding security and privacy, because most of the time, users cannot control what happens to the gathered information, who is able to access it or what it is used for.
Android smartphones have the biggest market share at the moment – 53.2% in the U.S., followed by Apple's smartphones with 41.3%, according to a recent study[1].
The Android platform security architecture is composed of the following key security features: OS level through the Linux Kernel, mandatory application sandbox, secure inter-process communication, application signing and application-defined and user-granted permissions. All of them have been improved over time, with every update.
The part of Android's security that I would like to talk about is the "application-defined and user-granted permissions". This part is controlled by the installed applications, who declare the permissions in their AndroidManifest.xml file. The user has to grant them the requested permissions, in order to be able to install and use them. Most of the users never check the requested permissions of an application they install, because of various reasons and that may lead to a leak of private information that could have been easily avoided.

2. Controlling Applications Permissions

Controlling privacy and security in the Android system can be done through granted permissions. The user is not allowed to modify them directly, as a system setting (in the recent versions of the mobile OS) but he must always accept the declared permissions of an installed application in order to be able to install and use it. However, there are multiple ways to get control of the installed applications' permissions.

[1] The study can be found here http://www.comscore.com/Insights/Market-Rankings/comScore-Reports-January-2015-US-Smartphone-Subscriber-Market-Share, accessed in May 2015

2.1 Available solutions

One of them is installing and configuring a **Firewall** for the Android OS that is usually based on the Linux IP tables and requires root access to be enabled on the device. It will enable the user to block a specific applications' ability to connect to the Internet, such as, for example, a Unit Measurement Converter application that requires the ability to connect to the Internet without actually needing it. That way, the user makes sure that the application does not publish his private information to a server, to be gathered and used. But there is a problem, for example with more advanced scenarios as: the user wants to block the Facebook applications' possibility to use his Location without having to disable the Internet connection or the Locations service. This cannot be achieved by a firewall, it is limited, in terms of controlling permissions, just to Internet connection restrictions. But instead it is pretty easy to use and almost certainly, won't cause applications' crashes or a complete system crash that may require a complete reinstall and even loss of data, which is possible with the following presented methods, especially if the user does not own a backup.

Another possibility is installing an application called **App Ops**, developed by "Lars Team". Once installed, it may or may not need root access, depending on the OS version and it enables access to some hidden settings that allow the user to disable or renew permissions for every application. It also gives the possibility to restrict the System's applications permissions. The problem with this kind of control is that it can easily make applications crash when they will try to access the restricted item. If the user restricts some System application, it can even crash the entire Android system (soft brick).

A more elegant way to achieve this is using applications that are based on frameworks that enable developers to *hook functions* in the Android applications. This method has the advantage of being able to even send fake data on system function calls instead of just block the applications' permissions. For example, using this method, the Facebook application will receive an empty list of contacts every time it tries to access it or a random location, when it tries to access location information.

There are two alternatives of frameworks that achieve this, one of them is called *Xposed*, released in 2012, which is open-source and the other is called *Substrate*, released in 2013 which is not open-source, but instead has a better documented API. They are pretty similar, both with pros and cons.

For the installation of the Xposed or Substrate frameworks, the user must have root access enabled on the target Android device, he should install the framework installer application, that is available either on the Google Play Store or the developer's pages and run it, following the required steps.

I chose Xposed because it is more popular, it gets updates more frequently (already supports Android 5.0) and it already developed a pretty strong developer's community. Moreover, Substrate support appears to have been discontinued because the latest update was back in 2013, though it seems to still work on more recent versions of Android.

2.2 General context of an Xposed solution

The mechanism behind the hooking frameworks is similar, though the hooking is done differently, which allows users to use them in parallel on the same device. At installation, Xposed copies an extended *app_process* executable to the */system/bin* directory of the Android system. App_process is a C++ program that is run at the boot-up of the operating system and the resulting process is called *"Zygote"*. The Zygote is a very important part of the OS, because every application is started as a copy of it. The extended app_process enables Xposed to act in the Zygotes' context.

The method hooking or replacing facilities that it provides are extremely powerful because the user / developer does not need to decompile / recompile / sign anything for them to work as he should have done if he wanted to modify the behavior of an application. Instead, the code can be injected before or after the method or replace a method. It cannot, though, modify code in a method of an application. In XposedBridge, the jar file provided by the author to the developers there is a generic native function called *hookMethodNative* that is called every time a hooked method is called, instead of that particular method. Inside that method, the *handleHookedMethod* is called in order to pass arguments, the *this* reference, handle the callbacks etc.

3. Developing a Solution Using The Xposed Framework

Developing an Xposed solution generally involves a strong research on the Android API functions that are called when a specific action is made. For example, hooking the method that brings an application the current location requires checking how it is acquired and finding the class that holds the method that must be hooked or replaced.

3.1 Specific configurations

In order to register an application as an Xposed module, that can be recognized by the framework and that can be enabled, you have to add some configurations to it. First of all, the *AndroidManifest.xml* must be modified. This file is required by every application to be present in the root directory and provides essential information about the application to the Android system, needed before it can run it. Three *<meta-data>* nodes with properties must be added in the *<application>* node of the *AndroidManifest.xml*:

```
<meta-data
    android:name="xposedmodule"
    android:value="true" />
<meta-data
android:name="xposedminversion"
    android:value="54" />
<meta-data
android:name="xposeddescription"
    android:value="Short description of
the module" />
```

For Xposed to recognize the APK installed as a module, the developer has to include a file called *xposed_init* to the *assets* directory of the application. The file contains the package and the class name, such as, for example: *com.example.XposedHookLoader* of the Xposed entry points
Then, the application must include the *XposedBridgeApi.jar* provided by the framework’s developer. In the jar, the user can find the framework’s classes that can be used, such as:

- *XposedBridge* where the logging, hooking functions are defined;
- *XposedHelpers* where the helper methods such as those used to find methods, call methods, get fields in a class, get process PID and so on.

3.2 Using Xposed API for hooking / replacing functions

To define a Xposed entry point, the developer has to create a class that implements an interface from the XposedBridgeApi, such as IXposedHookLoadPackage and specify it in the *assets/xposed_init*. It would look like this:

```
public class XposedHookLoader
implements IXposedHookLoadPackage {

    @Override
    public void
handleLoadPackage(LoadPackageParam
loadPackageParam)

       throws Throwable {
       //implementation added here
    }
}
```

In this method, the developer checks whether the application that he wants to restrict has been loaded by using the *loadPackageParam* received parameter and hooks or replaces some methods for it.
In order to hook a method, the developer has to use the Java Reflection mechanism to retrieve the Class reference and pass it to an Xposed method. For example:

```
Class<?> clazz =
Class.forName("android.location.Locatio
nManager", false, classLoader);
XposedBridge.hookAllMethods(clazz,
"getLastKnownLocation", new
XC_MethodHook() {
    @Override
    protected void
afterHookedMethod(MethodHookParam
param) throws Throwable {

param.setResult(XposedHookUtil.getFakeL
ocation());
    }
});
```

This is a way to hook one of the functions that applications use to retrieve the location and set the returned value to a fake location of choice. For example, I made a local Taxi ordering application believe that I am currently in Paris, after enabling this module. An example is presented in the Figure 1.

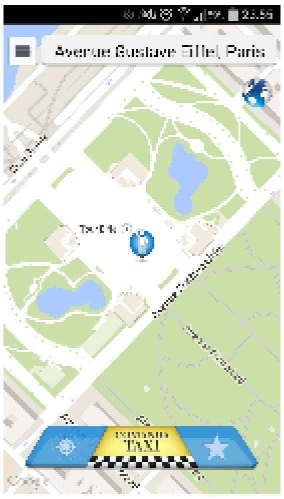

Figure 1. Location in the Taxi ordering app

After getting the module to be *recognized* by the Xposed Framework and *enabling* it, its hooks (registered callbacks) and method replacements become active.
Enabling it is done through a checkbox, as shown in the Figure 2, in the Xposed Installer application, as a security measure, allowing a user to decide which modules get to be enabled.

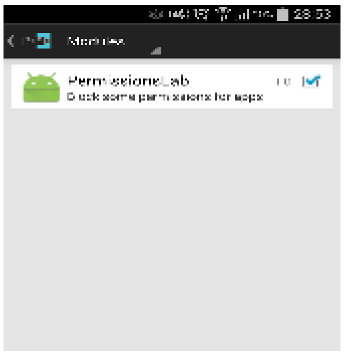

Figure 2. Module enabling in Xposed

3.3 Making the Xposed module change settings after being loaded

Even though the Xposed module and the application with the interface come into the same .APK file, they are two separated things, so it is not possible to communicate with the module directly, from an Activity of the application, for example. The module is a separated instance running in the Xposed framework context. The Activity is one of the main components of an Android application and it is used to display a screen that a user can interact with. An application usually has one or more activities that are bound to each other.
If we want to *pass something to the module* at run-time, a workaround to the inability to communicate with it directly would be to modify a file (an XML or other format) from the Activity and check the modifications at run-time, from the module. But that may lead to inconsistencies so I chose another available solution, proposed by a developer from the Xposed developer's community, using a Service registered in the Service Manager, by using Xposed at the initialization of the Zygote. That is achievable by implementing an interface called *IXposedHookZygoteInit* provided in the *XposedBridgeApi.jar*, in the Xposed class, that has only one method, *initZygote*, called by default. The method implementation looks like this

```
@Override
public void
initZygote(IXposedHookZygoteInit.St
artupParam startupParam) throws
```

```
Throwable {
    ModComService.inject();
}
```

The *ModComService* is the implementation class of the service I register through the ServiceManager, which is started before any other service and can be viewed as a registry for all the available services in the system. The inject method contains Xposed specific code used to call the *addService* method of the ServiceManager, that registers the service in the system, so it can be used from the Activity, for example.

4. Conclusion

The chosen solution, using Xposed, has some *advantages* such as being able to feed the "restricted" applications fake data that can actually be user-chosen. For an application that requires location information, that can be crucially important. This way, the user also makes sure that the application he wants to use (because that is why he installed it) will not crash because of a restricted permission, which actually may happen with other type of permission control applications, such as AppOps. This kind of solution, proves itself very elegant also because the application that uses a permissions does not have any mean to detect that it received some fake data or not.
It also has some *limitations*, for example in case of applications that are developed by the same company, they can easily communicate and one application can obtain sensitive data through another. So you may think you have restricted its access to that information, but it may still have access through another application. For example, nowadays, there is a method to obtain location through Play services, so restricting the Location for such an application will not work unless you restrict Google Play services ability to access the Locations service and this may be uncomfortable for users, because it would block an ability that may actually be needed in other scenarios.
A small *disadvantage* may also be the fact that a user with no technical knowledge may not understand the mechanism and what he must do in order to obtain the needed result, or maybe the installation process would be too hard for them. However, the targeted users for this kind of application would be those who know how to obtain root access at least and maybe flash a new image of the Android OS, to recover from a malfunction that causes a boot loop, for example.
Privacy and security on a mobile device that we carry everywhere and that can gather a lot of information about its user is becoming a real issue nowadays and I would like to say that I believe that it is an *elegant solution* to this very common problem, especially considering the fact that obtaining this kind of result (being able to pass fake data to applications) would probably not get any easier in the future

Acknowledgment

Parts of this paper were presented at The 8th International Conference on Security for Information Technology and Communications (SECITC 2015), Bucharest, Romania, 11-12 June 2015.

References

[1] James C. Sheusi, *Android Application Development for Java Programmers*, published by Cengage Learning, 2013;
[2] Onur Cinar, *Pro Android C++ with the NDK*, published by Apress, 2012;
Zigurd Mednieks, L. Dornin, G. B. Meike, M. Nakamura, *Programming Android*, published by O'Reilly Media, 2012;
[3] Jeff Six, *Application Security for the Android Platform: Processes, Permissions, and Other Safeguards*, pages 21-51, published by O'Reilly Media, 2011;
[4] Nikolay Elenkov, *Android Security Internals: An In-Depth Guide to Android's Security Architecture*, pages 13-37, published by No Starch Press, 2014;
[5] Anmol Misra, A. Dubey, *Android Security: Attacks and Defenses*, pages 3-29, published by CRC Press, 2013;
[6] J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, G. Wicherski, *Android Hacker's Handbook*, published by John Wiley & Sons, 2014;

[7] ***, *Security Overview*, https://source.android.com/devices/tech/security/overview/index.html;
[8] ***, *Xposed Framework* | *Cydia Substrate*, http://www.cydiasubstrate.com/id/34058d37-3198-414f-a696-73e97e0a80db/;
[9] ***, *Gain control over app permissions with Ap Ops*, http://www.techrepublic.com/article/gain-control-over-app-permissions-with-ap-ops/;
[10] ***, *Xposed Development Tutorial*, https://github.com/rovo89/XposedBridge/wiki/Development-tutorial;