Android Security Permissions – Can we trust them?

Clemens Orthacker, Peter Teufl, Stefan Kraxberger, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber

University of Technology Graz, Institute for Applied Information Processing and Communications, Graz, Austria
clemens.orthacker|peter.teufl|stefan.kraxberger|guenther.lackner@iaik.tugraz.at

Abstract. The popularity of the Android System in combination with the lax market approval process may attract the injection of malicious applications (apps) into the market. Android features a permission system allowing a user to review the permissions an app requests and grant or deny access to resources prior to installation. This system conveys a level of trust due to the fact that an app only has access to resources granted by the stated permissions. Thereby, not only the meaning of single permissions, but especially their combination plays an important role for understanding the possible implications. In this paper we present a method that circumvents the permission system by spreading permissions over two or more apps that communicate with each other via arbitrary communication channels. We discuss relevant details of the Android system, describe the permission spreading process, possible implications and countermeasures. Furthermore, we present three apps that demonstrate the problem and a possible detection method.

Key words: Android Market, Security Permissions, Android Malware, Android Services, Backdoors, Permission Context, Side Channels

1 Introduction

The opening of mobile device platforms to third party developers was probably the most significant move of the IT industry in the last years. The availability of a multitude of applications (apps) has boosted user acceptance and usefulness of mobile devices like smartphones and tablet computers. Regardless whether for business or private use, these devices and their apps have the potential to facilitate and enrich the user's everyday life. Mobile platform vendors recognized the importance of opening their systems to third party developers in order to attract a wide range of apps. However, the large number of third party developers make it very hard to provide uniform quality standards for the repository providers.

While, i.e., Apple enforces tight policies on software distributed via their AppStore for iOS, regarding security and content, Google emphasizes a more *open* philosophy, providing many liberties to Android developers, distributing their products via the Android Market. Apps submitted to the Android Market

are rudimentarily checked but the process is not as strict as it is for the AppStore. Google seems to pursuit a *delete afterwards* strategy if apps have been found, which are malicious or are of low quality. Android implements a kill switch to remotely remove apps installed on customer devices[1].

Google introduced a permission system for their Android platform, allowing developers to define the necessary resources and permissions for their products. The customer can decide during the installation whether she wants to grant or deny access to these requested resources such as the address book, the GPS subsystem or the phone functionalities. Although, this process is challenging to the standard user, at least an expert will have the ability to draw conclusions about the theoretical capabilities of an app based on its permissions.

Thereby, the security of the permission system is mainly based on two different aspects: the meaning of the permission itself and even more important, the meaning of combined permissions. For example, when the Internet permission is combined with the read contacts permission, a possible malicious app could transfer your private contact data to the Internet. This implication and the functionality is lost when both permissions are not used in the same application. Therefore, a large part of the security of the permission system and the trust in it is based on the assumption that an app only gains access to the resources that are declared via permissions.

In this work we give a detailed description of the possible communication paths between applications. We discuss the issue of what we call *spreading of permissions* which exploits interprocess communication to allow a transfer[2] of security permissions to apps which did not request them at installation time. We substantiate this threat by presenting three prototype apps that highlight the permission spreading problem and demonstrate the detection of a Service based communication path[3].

2 Related Work

In the article *Understanding Android Security* [1] Enck et al. took a look at the Android application framework and the associated features of the security system. One pitfall of Android, as the authors describe, is that it does not provide information flow guarantees. Also the possibilities of defining access policies in the source code introduces problems because it clouds the app security since the manifest file does not provide a comprehensive view of the application's security anymore.

SMobile has done some research on the Android Market and its permission system. They have documented specific types of malicious apps and threats.

[1] http://www.engadget.com/2010/06/25/google-flexes-biceps-flicks-android-remote-kill-switch-for-the/

[2] This is not an actual transfer of the permission, but has the same effect.

[3] These apps can be downloaded from: http://www.carbonblade.at/wordpress/research/android-market/

In their latest paper [6] they have analyzed about 50,000 apps in the Android Market. They looked for apps which could be considered malicious or suspicious based on the requested permissions and some other attributes.

Their key findings are that a big number of apps, available from the market, are requesting permissions that have the potential of being misused to locate mobile devices, obtain arbitrary user-related data and putting the carrier networks or mobile device at risk. Although the Android OS and Android Market prompt users for permissions before the installation, users are usually not ready to make decisions about the permissions they are granting. The most important statement they make is that fundamental security concerns as well as increase in malicious apps can be related to poor decisions of the user. Toninelli et al. came to the same conclusion in [5].

Nauman et al. [2] investigated the Android permission system with special focus on introducing more fine-grained permission assignment mechanisms. The authors argue that the current permission system is too static since it does not take into account runtime constraints and that the accept all or none strategy is not adequate. Thus, they propose Apex, an extension to the Android policy enforcement framework which allows users to grant permissions more selectively as well as to impose constraints on the usage of resources at runtime. Their extension provides some additional security features which allows a more fine-grained security policy for Android. A similar approach is outlined in the work of Ongtang et al. [3].

Similar to [1], Shabtai et al. [4] performed a security assessment of the Android framework in the light of emerging threats to smartphones. They made a qualitative risk analysis, identified and prioritized the threats to which and Android device might be exposed. In addition, they outlined the five most important threat categories which should be countered by employing proper security solutions. They provide a listing with adequate countermeasures and existing solutions for the specific threat categories. One of the main propositions is that the permission system should be hardened to protect the platform better from misuse of granted permissions.

3 Interprocess Communication

Android apps may activate components of any other app if the other app allows for it. The four different types of components providing entry points for other apps are Activities[4], Services, Broadcast Receivers and Content Providers.

If a certain component is requested, the Android system checks whether the corresponding app process is running and the component is instantiated. If either of both is not available, it is created by the system. Thus, if a requesting app is allowed to access a background Service provided by another application, it can access it at any time, without the Service having to be started by the user.

[4] For all subsequent Android specific terms we refer to the Android developer documentation located at http://developer.android.com/index.html

3.1 RPC Communication

The Android system provides means for interprocess communication (IPC) via remote procedure calls (RPC). Since different processes are not allowed to access each other's address space, methods on remote objects are called on proxy objects provided by the system. These proxy objects decompose (marshal) the passed arguments and hand them over to the remote process. The method call is then executed within the remote app component and the result marshaled and returned back to the calling process. The app programmer merely defines and implements the interface. The entire RPC functionality is generated by the system based on the defined interface and transparent to the application.

Interfaces for interprocess communication are defined using the Android interface definition language (AIDL). The resulting Java interface contains two inner classes, for the local and remote part, respectively.

Typically, the remote part is implemented within an app component called *Service*, allowing clients to *bind* to it in order to receive the proxy object for communication with the remote part. The Service returns the Stub class in its `onBind()` method called by the system upon a connection request from the client. The client, on the other hand provides a `ServiceConnection` callback object along with the bind request in order to receive the proxy object for interprocess communication with the Stub.

Specific messages, called *Intents*, identifying the targeted Service, represent bind requests. A client's Intent is passed to the Service's `onBind()`, so that the Service can decide whether or not to accept the connection. Upon a successful connection establishment, the system passes the proxy object corresponding to the Stub returned from `onBind()` to the client's `ServiceConnection` callback.

Services may declare required *permissions*[5] that are enforced during the binding. Applications binding to the Service have to declare the use of these permissions correspondingly.

3.2 Communication via Intents

Intents are logical descriptions of operations to perform and are used to activate Activities, Broadcast Receivers and Services. Since these components may be part of different applications, Intents are designed to cross process boundaries and may transmit information between applications. Note that Intents are mainly intended to identify a component, optionally adding a limited amount of additional information to more precisely specify the targeted operation.

Neither Activities, nor Broadcast Receivers or (unbound) Services provide persistent RPC connections for interprocess communication. Still, they may all be activated by Intents which will be passed to their respective activation methods `startActivity()`, `startService()`, `sendBroadcast()` and others. Via their `extras` attribute, Intents therefore provide a simple means for

[5] see Section 4

transmitting a limited amount of data to another process' component. Additionally, Activity components provide a way to return a result back to their caller. If launched via `startActivityForResult()` the calling process may receive a result Intent via its `onResult()` method, thus allowing for simple two-way communication.

3.3 Alternative Communication Paths

Content Providers are intended to share data between applications. They are uniquely identified by URIs and can be accessed via `ContentResolver` objects provided by the system. Note that Content Providers are not activated by Intents. Reading and writing to Content Providers allows for two-way interprocess communication if the participating processes have the required permissions.

Apart from the described methods, apps may also communicate by exposing data via the filesystem and setting global (world) read/write permissions on the files. Alternatively, apps may share resources if they request the same UID. In that case they are treated as being the same app with the same file system permissions. UID sharing is possible for apps signed by the same developer.

We consider these communication approaches as *side-channel* communication.

3.4 Information Flow Overview

Based on the IPC methods described above, there are four possible communication paths between two apps A and B (see Figure 1). The S block depicts any of the aforementioned interprocess interfaces that provides or receives information. The arrows indicate the information flow between an app and the remote component.

- **One-Way**: An app A can either transfer/receive information to/from an app B, by using a one-way communication method. This could also be described as pushing or pulling information to/from an application.
- **Real Two-Way**: App A exchanges information with app B, by using two-way communication. Thereby both apps can receive and transmit information from A to B and vice versa. This involves an RPC interface as provided by Services that a client can bind to.
- **Pseudo Two-Way**: In this variant, two one-way communication channels are combined to create a two-way communication channel. In this example, app A transmits information to app B via interface S2 provided by B. In addition, app B pushes information to A by calling its S1 interface. For this example two push channels were used, however an arbitrary combination of push and pull channels could be used. On an abstract information flow level, this method is equal to the real two-way communication method (regardless of the employed push/pull combination).

Especially the pseudo two-way method and one-way push method can be used to transmit information over side channels (e.g. communication by reading and

writing to logging facilities). The available communication paths influence how potential malicious apps can avoid detection and where they locate the actual malicious code.

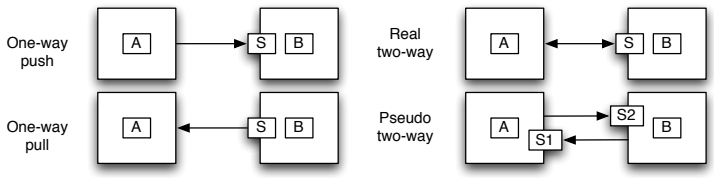

Fig. 1. Two apps A and B can exchange information via one-way or two-way communication.

4 Android Permission Mechanism

Application isolation, distinct UIDs for all apps and permissions are the three building blocks of Android's security architecture. Isolation of apps from each other as well as from the system is assured by executing every app within its own Linux process. Further, every app runs with a distinct user- and group ID (UID), assigned at app installation. This allows for protection of memory and file system resources. Communication and resource sharing are subject to access restrictions enforced via a fine-grained permission mechanism. Applications are allowed access to resources if they are granted the respective permissions by the user.

The isolation of applications, called *sandboxing*, is enforced by the kernel, not the Dalvik VM. Java as well as native apps run within a sandbox and are not allowed to access resources from other processes or execute operations that affect other apps.

Applications must declare required permissions for such resources within their manifest file. These permissions are granted or denied by the user during the installation of the application. The user does not deny or grant permissions during the runtime of the application[6]. Permissions are enforced during the execution of the program when a resource or function is accessed, possibly producing an error if the app was not granted the respective permission. The Android system defines a set of permissions to access system resources such as for reading the GPS location, or for inter-application/process communication. Additionally, apps may define their own permissions that may be used by other apps.

There is no central point for permission enforcement, it is scattered over many parts of the Android system. At the highest level, if permissions are declared in an application's manifest file for a component, they are enforced at access points to that component. These are calls to `startActivity()` or `bindService()` for

[6] There is no dynamic permission granting as with the Blackberry system.

activities or services, respectively, that would cause security exceptions to be thrown if the caller is not granted the required permission.

Permissions may control the delivery of broadcast messages by restricting who may send broadcasts to a receiver or which receivers may get the broadcast. In the first case a permission for the protected receiver is declared in the manifest file (or when registering the receiver programmatically, respectively). It gets enforced *after* a sender's call to `sendBroadcast()` and will not cause an exception to be thrown. Rather, the message will simply not be delivered to that receiver if the sender does not have the required permission. A sender, on the other hand, may declare a permission within the `sendBroadcast()` call, which will also be enforced without the sender noticing.

Permissions for granting read or write access to Content Providers are declared within the manifest file. Apart from that, content providers allow for a finer grained access control mechanism, via URI permissions. They control access to subsets of the content provider's data, allowing a content provider's direct clients to temporarily pass on specific data elements (identified by a URI) to other applications. A dedicated flag on an Intent[7] indicates that the recipient of the Intent is granted permission to the URI in the Intent's data field, identifying a specific resource, such as a single address book entry. The granted URI permission is finally enforced once the recipient of the Intent queries the content provider holding the URI by calling on a `ContentResolver` object[8]. Content Providers declare support for URI permissions in the manifest file. Enforcement of URI permissions results in security exceptions being thrown if the caller does not have the required permissions.

Applications may at any time query their context whether a calling PID or package (name) is granted a permission. This allows for custom-tailored permission enforcement for specific app requirements. Certain system permissions are mapped to Linux groups. On app installation, the application's UID is added to the respective group (GID). Permission enforcement involves GID checks on the underlying OS level. The permission to GID mapping is declared within the system's `platform.xml` file. Another specificity are *protected broadcasts*, which only the system may initiate.

5 Permission Spreading

As the user grants permissions on installation of an application, it is crucial to consider all permissions an app requests *in context*. The combination of specific permissions may indicate security flaws to the user. Within this work, we consider the user to be capable of critically analyzing an application's declared permissions and understand the implications of granting permissions.

Therefore, the permission system and the decision of the user is based on the assumption that an app can only use the functionality for which the appropriate permissions are available. We argue, that this security function can

[7] Intents are the entities used to activate app components, cf. Section 3.2

[8] Content Providers are not activated via Intents.

be circumvented by spreading security permissions over two or more apps that use interprocess communication. Thereby the apps are able to gain additional functionality for which they do not have the corresponding permissions.

5.1 Demonstration

In this section we describe two demo apps that hide the transmission of private location data to the Internet by employing permission spreading via an implemented Service (see Figure 2). The app *Backdoor* requires the *access fine location* (GPS) permission, which could be justified by posing as GPS app that displays the current position. However, not detectable by the user the app also implements an Android Service that provides the GPS position to other apps. The second app – *TwitterApp* – has the *Internet* permission and could pose as a simple app for accessing Twitter, which again would not raise any suspicion during its installation. However, the user is not able to see that *TwitterApp* has malicious code that determines the current GPS position by calling the Service of app *Backdoor*. This GPS position is then posted to a Twitter account[9], which requires the *Internet* permission. Therefore, the app *TwitterApp* gains additional capabilities by calling a Service of another app and uses its own *Internet* permission to publish this information. Although the permission system is not directly circumvented (the app is still not able to get the GPS position without the Service of the second app), there are serious implications when analyzing this method in the context with malicious applications.

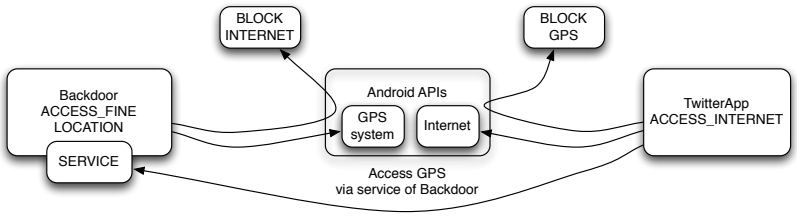

Fig. 2. The app *TwitterApp* uses the Service of *Backdoor* to determine the users's GPS position and submits it to Twitter via its own *Internet* permission.

5.2 Implications for malware

When taking a closer look at these two demo apps and the permission spreading method, we come to several conclusions:

Losing the permission context in the Android Market? The Android Market permission system is intended to support the user in her decision whether to trust an app before its installation.

Thereby, primarily the context in which multiple permissions are used and not only the permissions themselves, will alarm a user when inspecting a possible

[9] http://twitter.com/demolocator

malicious application. However, exactly this context is lost in our presented attack, since permissions can be distributed over different apps and do not occur within the same context.

Trust in the permission system? The Android permission system conveys a level of trust when an app is installed, since the system functionality can only be accessed when the appropriate permissions are available. However, when employing permission spreading, this trust leads to a wrong sense of security. In this case even an app without any permissions at all can gain additional functionality by calling functions within other apps that have these permissions.

Where is the malicious code? Assuming, we have a malicious app that transmits private information information to an attacker without using permission spreading, then, firstly this app must declare all the required permissions (e.g., read contact data and Internet access) and secondly it must contain the malicious code that reads the private data and transmits it to the Internet. Such an app might raise suspicions due to the employed permissions and the lack of an adequate description or app use case that would necessitate these permissions. When analyzed thoroughly, the malicious activity could be detected by decompiling the application, capturing network traffic or employing other methods for detecting malware.

However, when employing permission spreading it is not necessary that the malicious code is contained within the app that has the permissions required for the malicious activity. For instance, app *Backdoor* only contains a Service that provides the GPS position, but not the malicious code that transmits this data to an attacker. Therefore, an arbitrary malware detection/analysis method would never detect the malicious activity when inspecting *Backdoor* only. In fact, *TwitterApp* carries the malicious code and uses *Backdoors's* permissions and Services to gain the information required for the malicious activity.

Furthermore, the app *TwitterApp* could come without any permission at all and just acquire the functionality through calling Services on multiple other apps (e.g., providing contact data, GPS position, Internet access).

Backdoors? Regardless of the malicious code's location and how the various available communication paths are employed, permission spreading still requires the installation of multiple apps by the smartphone users. At a first glance this might make the likelihood of a successful attack smaller. However, permission spreading could also be viewed as a classic backdoor that could either be injected or integrated on purpose into a popular application:

– Such a backdoor Service could be injected into existing source code that is not protected adequately. Since, the malicious code is not present but only the code required for providing certain information or functionalities, it might be difficult to detect such a backdoor – especially when communication side channels are employed (e.g. by writing data to a system log).

- The backdoor could be injected on purpose into a popular app by the company developing the app itself, by a developer that is involved in the development of an popular app or by a government[10].
- The backdoors could be integrated into multiple apps created by the same developer who then convinces the user to install more than one of the apps (e.g. by promoting add-ons, splitting functionality, additional levels for a game, by using a common API for multiple apps, by providing demo/full versions of applications, etc.).

6 Countermeasures

During the analysis of the permission spreading problem, we have also investigated countermeasures and implemented another demonstration app that focuses on the detection of a communication path between permission spreading apps. Concerning detection, we need to deal with the following question: *How can we detect malware that employs permission spreading?* The answer strongly depends on the employed communication method. We will give a short overview about the possible detection methods in the following sections.

6.1 Service detection

Android Services are the simplest method for establishing a two-way communication path in Android. Services must be declared within the app manifest and get an identifier that is used when calling the Service. The detection therefore can be categorized into the detection of a service and the detection of a call to this service:

Detection of a Service

- **Android Market - by the user**: The Android market does not state which Services are provided within the application. Therefore, the user is not able to get information about the Services employed by an application.
- **Android Market - by Google**: Since the manifest of each app is readable, Google would be able to gain information about the employed Services within all market applications.
- **Android smartphone - by the user**: The user is able to get information about running Services from the Android system. However, non-running Services are not displayed by the standard apps bundled with the Android system.
- **Android smartphone - by an app (e.g., a virus scanner)**: An arbitrary app without any permissions can query the Android `PackageManager` for installed applications. For each of these installed apps it is possible to list the declared Services. In addition the `ActivityManager` can be used to list the running Services.

[10] The possible attempt to catch Facebook account data in Tunisia is a good example for such an attack: http://www.wired.com/threatlevel/2011/01/tunisia/

Detection of Service calls

Services are called by using Intents as parameters for the `startService()` or `bindService()` methods. The `bindService()` method enables the calling app to maintain a communication channel that is used for information exchange. The direct detection of such an Intent, the activation of a Service, or an established communication channel would require direct access to the Android system, which to our current knowledge is not possible without adding appropriate functions to the Android source code. At least for the `bindService()` method we have discovered a simple method, that allows us to determine when a Service is called and limit the possible apps that issued this call. We have also created a simple demo app (*ServiceBindDetection*) that can be installed as background Service and notifies the user whenever a Service is called by another application:

- The Android system Service allows an app to retrieve a list of all running Services (extracted via the `AcitivityManager`). In addition, for each Service the number of connected clients can be extracted. A client is connected when a *ServiceConnection* is maintained between a Service and the app that calls this Service.
- The detection app runs a loop in the background that in each iteration stores the running Services and their client count. Whenever the client count changes the user is notified within the Android notification bar.
- Whenever such a change occurs the detector could also get a list of currently running tasks (also via the `ActivityManager`) and thereby limit the possible callers[11]. By observing different calls to the same Service over time the possible perpetrators could be narrowed down.

We emphasize that this method does not work when `startService()` is used, since it does not maintain a communication channel and therefore does not list the calling app as connected client. Furthermore, we might miss certain calls when the duration of a `bindService() ServiceConnection` is smaller than the idle time of the Service checker loop.

6.2 Detection of Alternative Communication Paths

As described in Section 3, numerous ways for communication between applications exist. Acquiring information on interprocess communication other than via binding to Services turns out to be difficult. Information about Intents being sent would be valuable to detection of permission spreading. However, there does not seem to be a user-mode facility to obtain such information. Ongoing research will focus in this area.

Detection of communication via side channels like reading and writing to Content Providers seems to be even more difficult. Only in-depth analysis of the involved applications might yield satisfactory results.

[11] The most probable caller is the app that is currently running, however it cannot be ruled out that another app running in the background issues the call.

7 Conclusions and Outlook

The security of the Android permission system and the trust placed into the system is based on the assumptions that an app only has access to the functionality defined by the stated permissions and that all employed permission are displayed to the user within the same context (the app to be installed). As we show, these assumptions are not valid, since permissions can be spread over multiple apps that use arbitrary communication paths to gain functionality for which they do not have the appropriate permissions.

The intention of this work is to highlight the possible dangers and the wrong sense of security when trusting the permission system. Thereby, possible countermeasure range

- from making changes to the permission system including requiring permissions when using IPC between applications, or displaying communication interfaces prior to app installation,
- over implementing automatic detection systems within the Android Market, or performing an in-depth analysis of APK files,
- to shift the detection to the Android smartphone, by detecting communication events caused by permission spreading.

Addressing the detection on smartphones, we have presented a method to detect a covert communication channel involving Services. However, further investigations are necessary, since there is a large number of possible communication channels, ranging from documented IPC to not so obvious side channels.

References

1. William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7:50–57, 2009.
2. 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*, pages 328–332. ACM, 2010.
3. Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. *2009 Annual Computer Security Applications Conference*, pages 340–349, December 2009.
4. Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security & Privacy Magazine*, 8(2):35–44, March 2010.
5. Alessandra Toninelli, Rebecca Montanari, Ora Lassila, and Deepali Khushraj. What's on Users' Minds? Toward a Usable Smart Phone Security Model. *IEEE Pervasive Computing*, 8(2):32–39, April 2009.
6. Troy Vennon and David Stroop. Android Market: Threat Analysis of the Android Market, 2010.