A Concept of Dynamic Permission Mechanism on Android

Lukas Aron, Petr Hanacek

Department of Intelligent Systems
Faculty of Information Technology
Brno University of Technology
Bozetechova 2 612 66 Brno, Czech Republic

iaron@fit.vutbr.cz
hanacek@fit.vutbr.cz

Abstract. This paper discuss the main security topic in mobile security area and this topic is protect user against the leakage of data. This work primarily contains the proposal of concept of dynamic permission mechanism for Android operating system. This mechanism deals with assignment or enforcement permissions to the application according to files that the application works with. Application has set of permissions that can use, but if the user opens confident files the application permissions should change its permission set and there should not be possible leakage of this secret data. The permissions set should be stricter according to opened confidential file or more open if the file is not secret file. The concept proposes the solution for protecting this data leakage. Idea covers rule that user should be avoided of change this permissions himself, but this behavior should be dynamic, automatic and independent. This proposal is mainly aimed to Android operating system, but the concept can be applied to other mobile platforms with some implementation changes.

INTRODUCTION

Almost everything that users can access on their desktop can also be accessed on their mobile devices, in particular on smart phones. These mobile devices have become powerful with capabilities that desktops did not even have several years ago. Increased capabilities have come hand in hand with increased security threats.

In these days, all our private, business data or confidential data is accessible on our phones or tablets. Thus, these devices increasingly have become the targets of malicious attacks. Main reasons is the purpose of the data which are usually confidential, like corporate secret plans or accesses to bank.

The most successful and most widespread operating system of today's mobile devices is Android [1]. Needless to say, the operating system plays a key role for the security and for the privacy of these devices. However, we have different types of operating systems, the security principles are very similar to all platforms. There is another very important role in these principle and this role is the weakest point in the whole security chain. This weakest point is always user. This main aim of this paper is the proposal of protection to the user of data leakage. Although, the proposal is design to Android operating system, the main principle can be applied to other platforms, as well.

Permissions play a major role for applications on Android operating system. Giving permissions to applications can lead to data leaks because it may, for example, allow these applications to access contact data and to access the Internet. A malicious application may thus send all sensitive data to a server on the Internet. Fine-grained permission setting is not (yet) possible in Android. This article was inspired by introduction in [2].

Unlike other mobile operating systems, Android has a unique permission mechanism. At deployment time, an application developer needs to explicitly defined permissions, which the application needs, by including them in an application configuration file. This file is called Manifest (*AndroidManifest.xml)* and has specific structure

(explanation follows later in this paper). During installation, each user needs to review the permissions that the application requests and explicitly grant them for the duration the application is installed.

Currently, there are over 150 permissions which Android applications can request in API level 22. These permissions are defined by Android platform. Generally, an application can ask for permissions to use protected APIs for phone resources (e.g., storage, NFC, WiFi, etc.) or information available on the phone (e.g., contacts, location, call logs, etc.). For example, if an application wants to use APIs that control the camera, it needs to request the specific permission. (*android.permission.CAMERA*).

Although considered to be robust, the current Android permission mechanism has a number of deficiencies. The burden of deciding to grant permissions is placed on the user, but the permissions themselves provide little contextual information on how sensitive APIs are leveraged by the app. For example, it is unclear if an app with the permission to access the internet, as well as the phone's SIM card, exposes the private telephony data stored on the SIM card to the outside world. This article was inspired by introduction in [16].

In this paper, we will introduce a concept of dynamic permission mechanism for Android OS and its security features. There are solutions for tracking the flow of data [2] or flow of permissions [16]. Moreover there are modifications of Android operating system [15] where the user has absolute power over policy enforcement. However these solutions are secure and precise they are not so widely used. The main reason is the complexity of the systems or the settings of the application is not designed for the classical (non-technical) user. In this paper we are proposing another new approach in this area. The user will not have to decide what permission will be granted or denied. The system will automatically decide this according to files or documents that the application opens. This approach can be very useful in BYOD [4] system, where the employees can bring their own devices to company and can work on them.

In the first section of this paper is described the Android security architecture. Following section provides the related work in this area. Next section is the central section of the paper with concept of the dynamic permission mechanism. The last section is conclusion and future plans.

ANDROID SECURITY ARCHITECTURE

Android [3] is a privilege-separated operating system, in which each application runs with a distinct system identity (Linux user ID and group ID). Parts of the system are also separated into distinct identities. Linux thereby isolates applications from each other and from the system.

Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.

A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another application's files, performing network access, keeping the device awake, and so on.

Because each Android application operates in a process sandbox, applications must explicitly share resources and data. They do this by declaring the permissions they need for additional capabilities not provided by the basic sandbox. Applications statically declare the permissions they require, and the Android system prompts the user for consent at the time the application is installed.

The application sandbox does not depend on the technology used to build an application. In particular the Dalvik/Art VM [5, 6] is not a security boundary, and any app can run native code. All types of applications — Java, native, and hybrid are sandboxed in the same way and have the same degree of security from each other.

User IDs and File Access

At install time, Android gives each package a distinct Linux user ID. The identity remains constant for the duration of the package's life on that device. On a different device, the same package may have a different UID; what matters is that each package has a distinct UID on a given device.

Because security enforcement happens at the process level, the code of any two packages cannot normally run in the same process, since they need to run as different Linux users. You can use the *sharedUserId* attribute in the file *AndroidManifest.xml* - manifest tag of each package to have them assigned the same user ID. By doing this, for purposes of security the two packages are then treated as being the same application, with the same user ID and file permissions. Note that in order to retain security, only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID.

Any data stored by an application will be assigned that application's user ID, and not normally accessible to other packages. When creating a new file with *getSharedPreferences(String, int)*, *openFileOutput(String, int)*, or *openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory)*, you can use the *MODE_WORLD_READABLE* and/or *MODE_WORLD_WRITEABLE* flags to allow any other package to read/write the file. When setting these flags, the file is still owned by your application, but its global read and/or write permissions have been set appropriately so any other application can see it.

Android Permissions

A basic Android application has no permissions associated with it by default, meaning it cannot do anything that would adversely impact the user experience or any data on the device. To make use of protected features of the device, developer has to include in *AndroidManifest.xml* [7] file one or more *<uses-permission>* tags declaring the permissions that current application needs.

For example, an application that needs to monitor incoming SMS messages would specify:

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

At application install time, permissions requested by the application are granted to it by the package installer, based on checks against the signatures of the applications declaring those permissions and/or interaction with the user. No checks with the user are done while an application is running; the application is either granted a particular permission when installed, and can use that feature as desired, or the permission is not granted and any attempt to use the feature fails without prompting the user.

Often times a permission failure will result in a *SecurityException* being thrown back to the application. However, this is not guaranteed to occur everywhere. For example, the *sendBroadcast(Intent)* method checks permissions as data is being delivered to each receiver, after the method call has returned, so you will not receive an exception if there are permission failures. In almost all cases, however, a permission failure will be printed to the system log.

However, in a normal user situation (such as when the app is installed from Google Play Store [8]), an application cannot be installed if the user does not grant the app each of the requested permissions. So you generally don't need to worry about runtime failures caused by missing permissions because the mere fact that the application is installed at all means that your application has been granted its desired permissions.

The permissions provided by the Android system can be found at *Manifest.permission*. Any application may also define and enforce its own permissions, so this is not a comprehensive list of all possible permissions. A particular permission may be enforced at a number of places during your program's operation:

- At the time of a call into the system, to prevent an application from executing certain functions.
- When starting an activity, to prevent applications from launching activities of other applications.
- Both sending and receiving broadcasts, to control who can receive current broadcast or who can send a broadcast to current application.
- When accessing and operating on a content provider.
- Binding to or starting a service.

The Application has defined permission as was described earlier, there is the basic schema of the flow of communication between application and operating system, as it is implemented in Android. On top of the schema is the application which uses its permission defined during creating (programming). These permissions are mapped into

operating system calls. There are also user files (it is usually documents, pictures etc.) stored on the current mobile device. If the user of the device wants to work with the files through the application, there are system calls for input/output operations which are needed to communication with Application. We have a concept how to change this behavior to have automatically (dynamically) change the application permission enforcement according to files, which the application works with. The concept is described in the section Concept of Dynamic Permission Mechanism on Android.

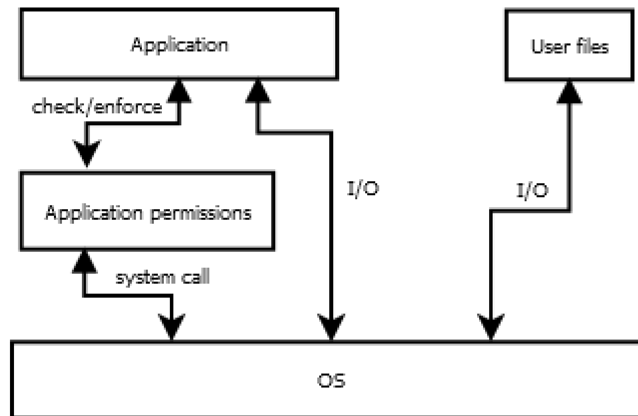

Figure 1 Application Communication Flow

RELATED WORK

This part discuss the similar work or projects. The first few projects are important from the point of view of this paper. The idea behind the concept of this paper become real from the connection of few existing projects and some additional value. The first paper which is related to this paper is the whole overview of the Android permission concept and its evolution during past few years: Permission evolution in the android ecosystem [18].

APEX is an extension to the Android permission framework. Apex allows users to specify detailed runtime constraints to restrict the use of sensitive resources by applications. The framework achieves this with a minimal trade-off between security and performance. The user can specify her constraints through a simple interface of the extended Android installer called Poly. The extensions are incorporated in the Android framework with a minimal change in the basic codebase and the user interface of existing security architecture [9].

Aurasium is a solution that bypasses the need to modify the Android OS while providing much of the security and privacy that users desire. Aurasium automatically repackages arbitrary applications to attach user-level sandboxing and policy enforcement code, which closely watches the application's behavior for security and privacy violations such as attempts to retrieve a user's sensitive information, send SMS covertly to premium numbers, or access malicious IP addresses. Aurasium can also detect and prevent cases of privilege escalation attacks [17].

MockDroid is a modified version of the Android operating system which allows a user to 'mock' an application's access to a resource. This resource is subsequently reported as empty or unavailable whenever the application requests access. This approach allows users to revoke access to particular resources at run-time, encouraging users to consider the trade-off between functionality and the disclosure of personal information whilst they use an application. Existing applications continue to work on MockDroid, possibly with reduced functionality, since existing applications are already written to tolerate resource failure, such as network unavailability or lack of a GPS signal [11].

FlowDroid is highly precise static taint analysis for Android applications. A precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and

object-sensitivity allows the analysis to reduce the number of false alarms. Novel on-demand algorithms help FlowDroid maintain high efficiency and precision at the same time [19].

Kynoid is a real-time monitoring and enforcement framework for Android. Kynoid is based on user-defined security policies which are defined for data-items. This allows users to define temporal, spatial, and destination constraints which have to hold for single items. In this way, Kynoid is the first extension for Android which enables the enforcement of security policies of data-items stored in shared resources [21].

PScout is a tool that extracts the permission specification from the Android OS source code using static analysis. PScout overcomes several challenges, such as scalability due to Android's 3.4 million line code base, accounting for permission enforcement across processes due to Android's use of IPC, and abstracting Android's diverse permission checking mechanisms into a single primitive for analysis [22].

CONCEPT OF DYNAMIC PERMISSION MECHANISM ON ANDROID

This part describes the dynamic permission mechanism on Android operating system. First, the schema of the concept is described and then the details of main parts will be described in more detail. The idea behind the concept is related to BYOD principle. The user of the mobile device can hold private/corporate data which is usually classified. The same user uses the device as his private device at home that means there are another data.

We have to deal with two approaches to access these data. User’s personal data are not classified even if they are not public. There exists a lot of solutions how to control the flow of the data or flow of permissions. Some of them are presented in the above section. The usual approach how to handle this division between corporate and personal data is to have two devices or switching between two modes on the mobile device. Implementation details could be different, but the switching is the same – the user has to do something more than just use the application of the device like usual.

The main idea in this concept is that the user can use his mobile device as it is the personal device. He can work with application as he used to it. However the application will distinguished what data it is working with. The application will dynamically enforce permission in the restricted way. That means if the application is working with personal files it is working with all permissions that the developer gives it. On the other hand when the application works with corporate/confidential files the permissions are stricter and some of them (which were allowed by developer), that may lead to data leakage, are forbidden.

The example will bring some light to the previously defined idea. Let’s have an application that works with photographs. This application can open pictures, edit them, send them through the internet or by email as attachment. When the user opens a personal picture he can do everything that the application tender. The same user can open corporate/confidential picture (in the same application) and the behavior is different. There is no way how to send the picture to the internet or by email.

The main schema of the concept (Figure 2 Concept of communication flow) describes the data flow and permission flow related to the application on Android operating system. Main difference between previously defined schema (Figure 1) is the Permission controller. This part is the arbiter of permissions and the whole communication point from/to application to/from operating system. All data are going through this point. Controller should contains some AI logic to distinguish difference between confidential and personal data. This concept do not cover the AI. The easiest proposal of AI may contain the decisions only by the path of the file. For example the confident files will be always stored in specific folder on the device or the path will always contain the confidential keyword.

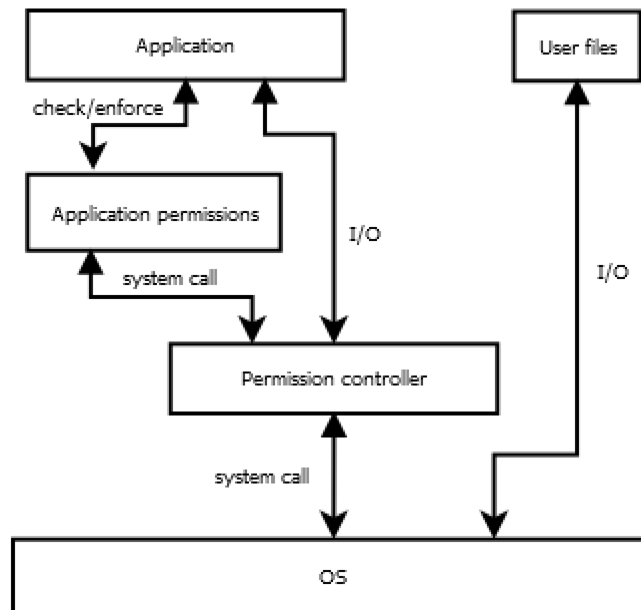

Figure 2 Concept of communication flow

The Permission controller is the most important part of the schema. The idea behind this approach comes from Flow permissions for Android [16]. Where the principle of permission division into two groups is explained. It basically means we have *source* permissions and *sink* permissions. *Source* permissions are those that can be in some way dangerous like sharing sensitive information from device (e.g. phone numbers, position etc.) and the *sink* permissions are these permissions where can be leakage of the data (e.g. access to SD card, internet, log etc.). More details and the whole description is in article of Flow permissions for Android [16].

The concept relies on the idea from Aurasium [17] research to catch or follow all necessary system calls from the application. The concept handle all important system calls (also input/output system calls) and related to these calls can dynamically change the permissions of the application. All system calls needed to catch are:

- *ioctl()* - This is the main API through which all IPCs (inter process communications) are sent. By interposing this and reconstructing the high-level IPC communication, we are able to monitor most Android system APIs. According to Linux manual pages: The ioctl() function manipulates the underlying device parameters of special files. In particular, many operating characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests [20].
- *getaddrinfo() and connect()* - These functions are responsible for DNS resolving and socket connection. Intercepting them allows us to control the application's Internet access [20].
- *dlopen() , fork() and execvp()* - The loading of native code from Java and execution of external programs are provided by these functions [17, 20].
- *open() , read(), write()* - These functions reflect access to the file system. Intercepting them allows us to control private and shared files accesses [17, 20].

There are two possible approaches for handling these system calls. The first is modification of Android operating system and catch all possible calls. This approach is for example possible by adjusting the MockDroid project or the classical edition of Android operating system.

The second is put the application into another sandbox and run the application inside. The advantage of the second principle is that we do not need to modify each version of Android operating system. There is existing solution for this approach and it can also unpack the whole application and run it with all needed catching of system calls. The solution is called Aurasium [17]. The Aurasium project can be used with our proposal concept. The schema will be

preserve and around it will be the cover Aurasium which will handle the communication between Permission controller and Android operating system.

The benefit with using Aurasium is we do not need to cover the low-level handling signals. We can focus directly to change the behavior of the permissions related to the files which the application works with.
Additionally we can define a list structure of permissions called *source permission* which are not allowed to use with confidential files. Related to this approach we should define a dummy data that will be provided to application when required permission from the *source* group of permission is called. Dummy data is needed for preserve the behavior of an Android applications. These applications will fail when the required permission is denied, so the resolution for this approach is to provide dummy data (e.g. if the access to phone list of contact is required the empty list will be provided or some non-existing number is returned to the application).

CONCLUSION AND FUTURE WORK

The security issues in Mobile world become more and more sophisticated and their detection is more complicated. Main responsibility is on the user even if he uses the mobile device to work. The user is the weakest point in the whole string of security attacks. The reason of this is that the operating system is very complex. Even the small part of this system like security is very complicated for non-technical user and the professional attacker can have a profit from that. There should be features and steps for protecting users a keep them in safe zone. This paper propose the basic schema of the protection confidential data on mobile devices. The concept dynamically change the permission model and protect user from data leakage. A Concept for dynamically permission enforcement was described and the main principles or overview was defined by schema.

This approach could save corporate data and should deny the leakage of secret data from the device. This concept protects the user to share confident data to the network, to files, to the log or outside of the device. The next step of this work will need more mathematical model and proofs that this concept works. Afterwards the implementation of this concept follows with proof of concept.

ACKNOWLEDGEMENT

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by the project CEZ MSM0021630528 Security-Oriented Research in Information Technology and by project FIT-S-11-1 Advanced Secured, Reliable and Adaptive IT

REFERENCES

1. Top 10 Mobile phones operating systems http://www.shoutmeloud.com/top-mobile-os-overview.html [retrieved: July., 2015]
2. Kern, M., & Sametinger, J. (2012). Permission Tracking in Android. In *The Sixth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies UBICOMM* (pp. 148-155).
3. Developers, A. (2011). What is android
4. Song, Y. (2014). "Bring Your Own Device (BYOD)" for seamless science inquiry in a primary school. *Computers & Education*, *74*, 50-60.M. P. Brown and K. Austin, Appl. Phys. Letters **85**, 2503–2504 (2004).
5. Oh, H. S., Kim, B. J., Choi, H. K., & Moon, S. M. (2012, October). Evaluation of Android Dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems* (pp. 115-124). ACM.
6. Cinar, O. (2015). Android Platform. In *Android Quick APIs Reference* (pp. 1-14). Apress.
7. Developers, A. (2012). Android Manifest Permissions.
8. MacLean, D., Komatineni, S., & Allen, G. (2015). Deploying Your Application: Google Play Store and Beyond. In *Pro Android 5* (pp. 677-696). Apress.
9. Nauman, M., & Khan, S. (2011). Design and implementation of a fine-grained resource usage model for the android platform. *Int. Arab J. Inf. Technol.*, *8*(4), 440-448.
10. Ongtang, M., McLaughlin, S., Enck, W., & McDaniel, P. (2012). Semantically rich application-centric security in Android. *Security and Communication Networks*, *5*(6), 658-673.
11. Beresford, A. R., Rice, A., Skehin, N., & Sohan, R. (2011, March). Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (pp. 49-54). ACM.
12. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., & Sadeghi, A. R. (2011). Xmandroid: A new android evolution to mitigate privilege escalation attacks.*Technische Universität Darmstadt, Technical Report TR-2011-04*.
13. Zhou, Y., Zhang, X., Jiang, X., & Freeh, V. W. (2011). Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing* (pp. 93-107). Springer Berlin Heidelberg.
14. Sam Lu. aSpotCat (app by permission). play.google.com/store/apps/details?id=com.a0soft.gphone.aSpotCat [retrieved: July, 2015]
15. Team, C. Cyanogenmod. www.cyanogenmod.org/ [retrieved: July, 2015]
16. Holavanalli, S., Manuel, D., Nanjundaswamy, V., Rosenberg, B., Shen, F., Ko, S. Y., & Ziarek, L. (2013, November). Flow permissions for android. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (pp. 652-657). IEEE.
17. Xu, R., Saïdi, H., & Anderson, R. (2012, August). Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium* (pp. 539-552).
18. Wei, X., Gomez, L., Neamtiu, I., & Faloutsos, M. (2012, December). Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference* (pp. 31-40). ACM.
19. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J. & McDaniel, P. (2014, June). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices* (Vol. 49, No. 6, pp. 259-269). ACM.
20. Pages, L. M. Linux Manual Pages. http://linux.die.net/man/ [retrieved: July, 2015]
21. Schreckling, D., Köstler, J., & Schaff, M. (2013). Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android.*Information Security Technical Report*, *17*(3), 71-80.
22. Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012, October). Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 217-228). ACM.