BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

Dynamic security policy enforcement on Android

Matúš Vančo

Abstract
This work deals with the concepts of Android security and proposes the system for dynamic enforcement of access rights. Each suspicious application can be repackaged by this system, so that the access to selected private data is restricted for the outer world. In the first phase, interprocess communication and existing frameworks, which are capable to intercept communication between application and the operating system on the level of system calls, are explored. After that, the system is designed and developed, utilizing the possibilities of one of the compared frameworks – Aurasium framework. The system adds an innovative approach of tracking the information flows from the privacy-sensitive sources using tainting mechanism without need of administrator rights. There has been designed file-level and data-level taint propagation and policy enforcement based on Android binder.

Keywords: private data — Aurasium framework — operating system — system call — binder driver — Android security — policy enforcement — security policy

Supplementary Material: Downloadable Code

*xvanco02@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Android's fast growth of popularity has a lot of causes and consequences. Growing number of applications, increasing a number of devices and growing level of integration have been interfered and influenced each other, implying increasing volume of the *private data*. People have put their trust in their devices and become more dependent on mobile technologies, using it for socialization, trading or entertainment. Hovewer, the private data is being used for the profit still more often during globalization, because it is the base for the knowledge-based business and targeted advertising. Even Google's free Android generates the significant part of its revenue just this way [1].

Since this asset is seized by many groups of people using illegal ways, Android has become the most assaulted mobile operating system facing the wave of malware, which is even more capable and stealthy, and can even establish a permanent presence on the device [2]. In order to address these challenges, Android includes permission model that protects access to sensitive resources. However, since permissions are overly broad and misunderstood, applications are provided with more access than they truly require. In particular, they are granted statically during install-time and so does not correspond to the actual use at the time. This implicates big vulnerability even if a certain application is not intended to misuse the private data because it can be exploited.

Based on this insufficient built-in Android security and his later refinements, plenty of third-party frameworks seek to supplement overall security. The current state of the art comprises several effective countermeasures to issues like coarse granularity of permissions,

over-claim of permissions and permission escalation attack. Hovewer, most of these solutions are rather too complex and less straightforward. They replace the whole Android permission model, or tries to track the information flow on the level of operating system which requires the rooted device. In contrast, there is Aurasium framework, which automatically repackage and harden chosen applications, interposing the sandboxing code in the applications themselves. Nevertheless, it is robust enough to interpose almost all types of interactions between the application and the operating system.

The aim of this work is to develop the system, which utilizes the Aurasium framework and restrict the access of private data outside the device through the selected applications. In contrast to original Aurasium framework, this work focuses more on the real asset, the private data, and especially on the high usability and easy deployment. The solution is focused on tracking the information flows from the privacy-sensitive sources to the system sink where they aim to leave the system.

2. Android Security

Android security has been built upon fundamental security concepts of the operating systems themselves. Android's Permission Label Model (PAM) has been built upon Linux security mechanisms including Mandatory Access Control (MAC) mechanism and Principle of Least Privilege (PLP) [3]. Security enforcement using MAC uses two types of permissions – granted permissions (used or requested permissions) and required permissions (access permissions). Granted permissions are manifested during installation and are inherited by all of the application's components. On the other hand, required permissions are usually created by the developer to protect their important components. Required permissions are always assigned to application components separately. When the application is started, the launcher component is invoked and the other components are called subsequently from the same or the other application or system. The mechanism of the passing to another component and access control is transparent in both situations. Communication between components is based on Linux Inter-component Communication (ICC), because each application runs in the separate process. It uses message passing, where messages contain data with the required action and are called intents (ACTION_SEND, ACTION_VIEW, etc.) [4]. Android maintains this communication using Reference Monitor (RM), which is part of the Android OS Middleware.

3. Aurasium Framework

Aurasium project has been developed in 2012 as a project at the University of Cambridge, UK. Aurasium use repackaging mechanism, wrapping around the DVM under which the Android applications run, with monitoring code. It do not require rooted Android device. To attach sandboxing code, Aurarium exploits Android's unique application architecture of mixed Java and native code execution and introduces `libc` interposition code. Because of this, Aurasium is capable to mediate almost all types of interactions between the application and the Android OS.

This project consists of three parts – automated repackaging system written in Python programming language named `pyAPKRewriter`, sandboxing code included in `ApkMonitor` application and `Aurasium's Security Manager` (ASM) application enabling central handling of policy decision of all repackaged application on the device. [2]

Starting with sandboxing code, the top layer of the framework is written in Java. The aim is to create an well-documented easy-to-use abstraction layer upon cumbersome native layer of the framework. The upper layer creates interface for other possible programs and delegates all requests to the low-level part of the framework implemented in native C++ code. This layer consists of few shared objects that do all the real work, such as communication with the Dalvik VM or establishing the machanism for IPC communication.

The second part of Aurasium, the repackaging Python script utilizes the previously mentioned sandboxing code and deploy it to Android APK installation package. APK file is similar to Java JAR archive and contains AndroidManifest file, application logic in the form of dex bytecode, compiled XML resources and native libraries. Each application package is also signed with authorship information. Besides the sandboxing code, Aurasium has to include also several additional parts to APK in order to ensure the functionality.

The last part of the Aurasium is called *Aurasium Security Manager* (ASM). ASM handles the policy decisions centrally, so that all repackaged applications can be maintained at one place. Security policy is based on decision of application or user. Application decision works transparently without user interaction, while the user decision is consented by dialog window and can be remembered and used by default during next occurrence.

4. Android Binder

In order to perform the required mediation, the part of Android middleware called the Binder needs to be rewritten. The Binder was originally developed under the name *OpenBinder* by *Be Inc.* and later under *Palm Inc.* and provides high-level abstraction on top of traditional modern operating system services including the facility to provide bindings to functions and data from one execution environment to another [5]. In Android, OpenBinder is customized to provide Inter-component Communication as described before. All interposition code needs to be placed in the suitable position in the original Binder implementation. Therefore, there is important to understand the concepts and to analyse the architecture of this part of system.

The communication between two processes is ensured using Binder Objects (BO), which are instances of classes that implement ioctl-based Binder interface. The most important operation which is declared in this interface is `transact(int code, Parcel data, Parcel reply, int flags)`. The corresponding callback method in the Binder object is called `onTransact()`. The interface can be further extended by additional business operations using Android Interface Definition Language (AIDL). Each BO uses local and global identifier. The local ID is unique in the process and the global ID is created when the BO is passed to another process using Binder Driver (BD). The BD then works like network switch and persists the mapping from local ID to global ID in the table structure and translate it transparently, similarly than the mapping using ARP protocol. The Binder framework communication uses the client-server model. However, the process can implement the server as well as client, so the communication can be still bidirectional. The Binder Client (BC) invokes an operation on a remote Binder object called Binder Transaction (BT), which may involve sending or receiving data over the Binder Protocol. The communication is performed indirectly using Binder Driver. In the Android, the Binder Driver is exposed via `/dev/binder` file and simple API based on `open()`, `release()`, `poll()`, `mmap()`, `flush()` and `ioctl()` operations. Most communication happens via `ioctl(int fd, unsigned long request, ...` method. The first parameter is the file descriptor number which identifies currently opened file and is used in `/proc/<pid>/fd/<fd>` file. The second parameter specifies the `ioctl()`.

In fact, most communication happens via `ioctl(binderFD, BINDER_WRITE_READ, &bwd)` operation, where the binderFD is used to access the `/dev/binder` file and the `bwd` structure is defined as:

The `write_buffer` contains a series of commands for the driver to perform, while the `read_buffer` contains commands for the BO in user-space. The commands for driver are called Binder Call (BC) commands and the commands for the BO are called Binder Return (BR) commands. Each command is couple (operation code, data).

The Binder Transaction is a passing data from the client to the service, while the Binder Reply is a passing data from the service back to the client. This is shown in figure 1. The whole Binder framework mechanism is transparent for the Android developer, since the Binder Transaction is performed as a local function call using so-called thread migration. This is ensured by the proxies and stubs, which are auto-generated helper classes from the AIDL files [6]. The proxy is the helper class which transforms Java code to low-level commands for the Binder Driver. The stub works in reverse to proxy and automatically parses and performs read commands on the service side. Since the Binder Driver is implemented on the low layer using C language, there has to be mechanism for encapsulation of high-level Java objects. This is ensured by `Parcel` container and corresponding `Parcelable` interface. A procedure for converting this higher-lever applications data structures into parcels is called marshalling. The marshalling as well as unmarshalling are also in the responsibility of the proxies and stubs.

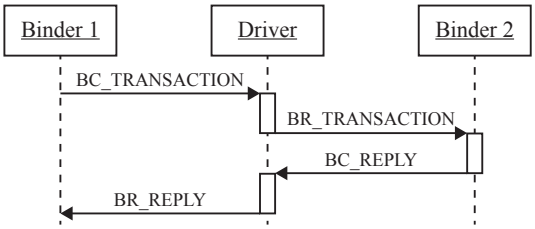

Figure 1. Binder Driver Interaction[1]

5. System Design and Implementation

Design of the system is based on previous analysis and various experiments, which were focused on the Android system behaviour. Design consists of three parts – design of architecture and principle of application, design of data structures and design of configuration. Design of architecture can be further devided into two parts – desing of tainting mechanism and design of restriction.

[1] Inspired by Schreiber's article [5]

Starting with the overall architecture and tainting principle, the tainting is based on the principle used in TaintDroid architecture. In order to perform real memory-level tainting, there has to be tracked each atomic memory transfer, which from a programator's point of view means each assignment to a variable. This can be done only through monitoring of instructions on the level of machine. In TaintDroid, there are monitored instructions on the level of virtual machines, because all possibly harmful applications are run under Dalvik virtual machine. TaintDroid uses *Virtual Taint Map* (VTM), which mirrors the address space, but does not contain the content of memory. It represents the division of memory into protected and public part. Before tainting process, the tainted files are marked in VTM. Then, every copying of memory invokes copying of blocks in VTM. Since the applications, which run on separate DVM can also exchange data, TaintDroid introduces message-level tainting as well.

In this project, there has been designed and integrated two granularities of taint propagation – file-level tainting and data-level tainting. The message-level tainting (between components) principle from TaintDroid is used only for final policy enforcement (restriction), because Aurasium intercepts only single applications and does not have possibility to monitor the unhardened ones. File-level tainting and data-level tainting uses the previously mentioned concept of VTM, but it is stored in higer-level abstract data structure and file instead of VTM.

File-level tainting between memory and the OS's file system can be performed in full scope, because Aurasium can fully intercept this communication using system calls `fopen()`, `open()`, `write()` and `read()`. Fuction `fopen()` is used for obtaining the opening mode. This is used for *tainting customization* (TA). If the untainted memory is written to tainted file in append mode, the files remains tainted, but if it is written in read mode, the file becomes untainted. The open() and read() calls are used for tainting the memory blocks as well as new files. The data in memory read from tainted file are marked similarily and the files, which are read from tainted memory blocks becomes tainted too. However, data in memory are also directly propagated.

Since the Aurasium can intercept only specific places (system calls) and not instruction itself, it is impossible to implement full-scope memory-level tainting as is introduced by TaintDroid. This is replaced by the newly designed data-level tainting concept. This concept together with the file-level tainting is depicted in figure 2. When the data are read from the file, content of data is read and tagged using hash function which assigns unique number. This tag, together with the size of block is used during the writing unknown memory block into file. Each unknown memory block is tested with respect to any existing hash and marked as tainted if the hash matches. Subsequently, the file is marked as tainted as well.

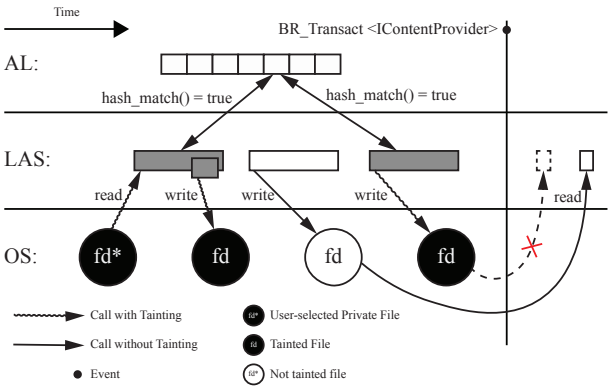

Figure 2. Design of Architecture

The final policy enforcement is performed using the interception of ioctl() call. Specifically, when the `BR_TRANSACTION` command which contains destination component ContentProvider is read, all the read() calls for the tainted files are in the mode of restriction.

The project is designed to secure the user-selected files or folders as a entity, which are intended to be invariable like images, pictures or videos. Documents that are often changed can be restricted for opening, or there can be assigned unique rights for opening to hardened application and the files are encrypted for other applications (reverse mode). In this reverse mode, data are protected with unhardened applications and uncovered and possibly exploited by the hardened application. The reverse mode is designed as optional and may not be implemented. Further extension is finer granularity of data-tainting. Unknown memory block which is being written to file is compared against the tainted memory blocks which are smaller than the unknown memory block, and the memory block which is being read from file has been divided into smaller units with separate hash.

Data structure which works as TaintDroid's VTM is designed as simple array of memory blocks, which are the interconnection part between the file system level and application logic performing described data-tainting. From designed perspective, each memory block is considered as tainted or untainted. Application can store only tainted data and other will be implicit. Initially, only the files are marked as tainted and during tainting process, the other files and new

memory blocks are added. Each memory block can have only one source file, one hash tag, but a lot of destination files to which this data are written. Due to Tainting Customization (TA), also the file modes need to be stored, because read() and write() functions does not dispose with this information in the passed arguments.

6. Testing and Conclusion

Application has been tested on platform 4.3.1 and several publicly-available applications, which share data using Bluetooth, Wi-fi, SMS, Email and other channels. In the first phase, there were performed tests and experiments related with Aurasium framework. The second step was developing of mocking environment according to previous stage. Here, the processing of gathered system call sequences has been tested and debugged. The last stage was to integrate these hook functions in real environment – firstly in developed test application and then in selected real application. Except for testing of configuration utility, which has been tested interacting with GUI, all the tests have been performed via logging files and manual evaluation.

The aim of this work was to provide the system for securing the user-selected private data of chosen applications with the sandboxing mechanism. The solution is focused on tracking the information flows from the privacy-sensitive sources to the system sink where they aim to leave the system. There has been examined Android platform from the security perspective and investigated the code and the possibilities of Aurasium framework. From this analysis, there has been designed and implemented solution, which uses file-level tainting and hash-tagging of memory in process's logical address space. The private data from the user-selected private files is restricted.

7. Acknowledgements

I would like to thank my supervisor Lukáš Aron for his help and motivation.

References

[1] JOEL ROSENBLATT. Google's android generates 31 billion dollars revenue, oracle says. [online], 2016. [cit. 2016-01-21].

[2] RUBIN XU, HASSEN SAÏDI, and ROSS ANDERSON. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, pages 539–552, 2012.

[3] ABRAHAM SILBERSCHATZ, PETER B. GALVIN, GREG GAGNE, and A. SILBERSCHATZ. *Operating system concepts*, volume 4. Addison-Wesley Reading, 9th edition, 1998.

[4] ANDROID. Android Developers. [online], 2016. [cit. 2016-01-26].

[5] THORSTEN SCHREIBER. Android binder. Master's thesis, Ruhr-Universität Bochum, 2011.

[6] ALEKSANDAR GARGENTA. Deep dive into android ipc/binder framework. In *AnDevCon: The Android Developer Conference*, 2012.