Secure Copy Protection for Mobile Apps

Nils T. Kannengiesser
Technical University of Munich
Chair for Operating Systems (F13)
Munich, Germany
Nils.Kannengiesser@tum.de

Uwe Baumgarten
Technical University of Munich
Chair for Operating Systems (F13)
Munich, Germany
baumgaru@tum.de

Sejun Song
University of Missouri-Kansas City
Computing and Engineering
Kansas City, USA
sjsong@umkc.edu

Abstract — **In the last few years there has been an increase in mobile app development especially on Android. Unfortunately, the rising app sales have been followed by an increasing number of piracies. Therefore Google has been forced to release the license verification library (LVL) to provide developers with basic copy protection for their apps. Not surprisingly, this copy protection was immediately cracked. Although Google tried to fix these problems by providing subsequent updates, we believe that there is still space for major improvements and that new and more secure solutions are possible for copy protection. In this paper we discuss the usage of secure elements to bind apps to their owners. It is part of our ongoing research and in an early stage.**

Keywords* — *Apps, Android, Copy Protection, Secure Element

I. Introduction

When Google's first smartphone, the Nexus One, hit the market in early 2010, nobody could have known whether it could effectively compete with existing smartphones. Today the Android's market share is almost 80% [1], and more and more developers are creating apps for Android. In comparison to the development of apps for other platforms, it's relatively easy to create and upload an app to Google's App Market – Google Play. Also Google released the LVL[1] in 2010 [2] to satisfy the needs for basic software protection. Nevertheless this release was immediately cracked [3]. A major issue with the protection was the reengineering possibility to change one line of code only, and get an unlicensed app working [3]. Another problem even today is that many developers are not aware of existing obfuscation tools. This allows their code to be processed by reengineering tools (e.g. APKtool[2]), which produce smali[3] or java output. The basic configuration of the IDE (which most Android developers are using) during the app creation does not typically include the use of obfuscation tools (e.g. ProGuard[4]), which is probably the reason that most apps (of our sample quantity) are still not protected. These apps can, however be readily modified. During current test-runs we found dozens of apps that were not protected by obfuscation tools. The LVL has been updated since its initial failure, but we still believe a major issue persists primarily, because critical data is stored in an insecure space – the phone itself. Root users can access any part of a phone, decompile apps as they like and remove or disable security features. Some manufacturers (e.g. Google) allow rooting by default. On other phones, there are multiple ways that allow users to gain root access, too. The timeline for root exploits (Figure 1) shows that every Android version is affected after only a few months on the market. Therefore we propose the usage of secure elements to add the required data security for sensitive information (e.g. license information).

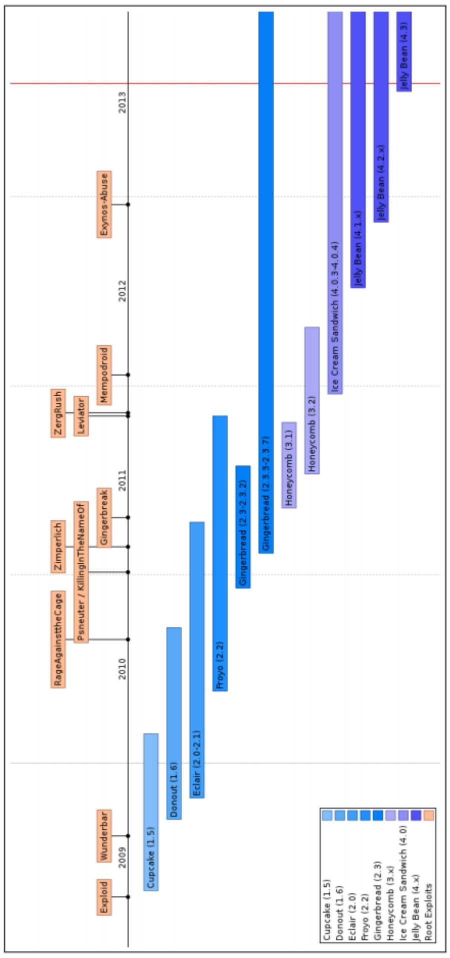

Fig. 1. Timeline of Root Exploits [4]

[1] License Verification Library
[2] Reengineering Tool - https://code.google.com/p/android-apktool/
[3] Smali = Assembly language (output of APKtool)
[4] Obfuscation Tool - http://proguard.sourceforge.net/

II. Foundations

Google began to worry about security early in the development of Android [5]. Android, itself, enforces security by separating apps to run as isolated processes with their unique user- and group-ids. The access to sensitive hardware is controlled by predefined permissions that need to be accepted by a user during the initial installation phase [5].

To support the security of commercial apps Google released the License Verification Library (LVL) in 2010 [2]. This library provides developers with the ability to integrate license checks in their apps. Figure 2 illustrates the basic functionality of this library:

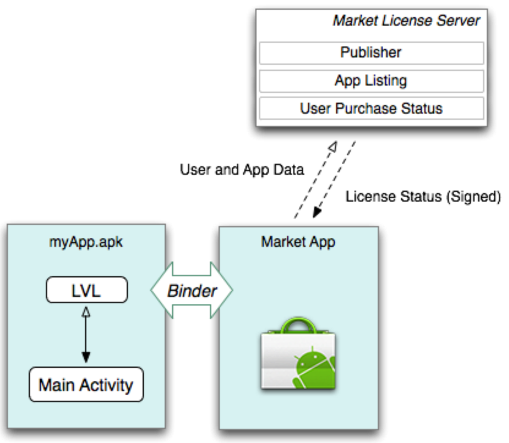

Fig. 2. Licence Verification Library [6]

Most of the later piracy issues arising from using the License Verification Library came from how developers used the library. Because of the available cracking tools[5], it seems like many developers copied the LVL package to their apps without making any major changes and especially without using any obfuscation tools. Google describes this problem in their own documentation by declaring "the security […] ultimately relies on the design of your implementation" and "actual enforcement and handling of the license [...] are up to you" [6]. Even today, the LVL uses signed responses [6], and can be assumed to be insecure, since the actual application can be modified using the aforementioned reengineering possibilities.

A possible way to solve this security issue might be the usage of a so called *Secure Element*. Such an element provides a secure space that is separated from the smartphone [8] and its vulnerabilities (cf. Android exploits, rooting, malware like trojans etc.). For instance, Google is using it as part of their Google wallet application [7]. Secure elements are available in form of UICCs[6], commonly known as SIM cards, as an external flash memory card or even already embedded in the hardware of the phone itself [7]. One of the manufactures for external secure elements in form of memory cards is Giesecke and Devrient. Their product, the MSC[7], is used in our research.

[5] Names not included due to legal reasons
[6] Universal Integrated Circuit Card
[7] Mobile Security Card by Giesecke & Devrient

III. Proposal

Because it has been relatively easy for users to gain root access to many different smartphones as well as the history of such root access exploitation for all kinds of Android versions and devices, it can be assumed that Android will very likely be insecure in the future. Therefore data, which shouldn't be accessed by a user (c.f. license information), isn't stored securely.

For this reason we focus on design ideas that combine apps and secure elements.

Besides issues involving the storage of critical data, we also face the problem of identifying a certified device or user for purposes of copy protection.

We propose the following:

Device/User identification

- Since the usual API calls for getting the device ID might be emulated (e.g. using AppGuard[8]), there is the need to define new methods for identifying devices and users. Here we distinguish between short and long term identification. For instance at the beginning the Google account being used might be taken into consideration, along with available information on hardware such as MAC addresses or available contacts and their (rarely changed) details. In addition there is long term information available that might be collected within a few days or even weeks of the first access that will be used to improve the identification. For instance, this can be a record of the usual locations of the phone over a period of a week or the available WiFi networks. A score might be used to allow developers to specify a limit before the app declares itself pirated. Nevertheless, there might be the risk of false alarms (e.g., the relocation of a user from Germany to the USA).

Server / Exchange of information in a secure manner

- A fundamental step needed for improved security is the ability of the MSC to connect to the internet for upgrading its secured content based on the used apps. The current idea is to add a proxy part to each protected app to allow the establishment of a secure P2P[9] connection between the MSC and a license server. Figure 3 illustrates this idea.

Content protection

Google proposed at Google IO 2011 that developers should implement the LVL in a modified way, by moving it to a safe place and

[8] A tool to redefine privacy-relevant API calls, e.g. to hide the real device ID, http://www.backes-srt.com
[9] Peer to Peer connection

encoding all strings. They also suggest that nonsense should be added to the source codes, that checksums be used for verification (cf. license checks) and that native code be used to complicate matters even more. Google also proposed a server-side validation of users / downloads [9].

Based on some of their ideas, we propose the following:

- The encoding of strings does not help against deobfuscation tools (e.g., “Binary Refactor”[10]). Therefore strings should be exported to the MSC and retrieved, when necessary only. This will make it much harder for attackers to identify important program parts.
- Since the APK file[11] is stored in an insecure place, its resources need to be protected. Therefore one initial option might be the encryption with the MSC providing the required keys during runtime. This will make the reengineering more difficult.
- In general, all URLs will be removed from the original source code and replaced by MSC calls. These removed URLs might be recovered in a modified way by the MSC/license server later on and are usable for a very limited time only. We call them One Time URLs (OTUs). The real URL is never transferred to the app and kept secret within the secure element. This step binds an app and secure element together.
- Another approach might be to fetch most of the additional resources of an app during its initial installation by downloading the encrypted content from a server and storing the required keys in the MSC for later use. We also make use of OTUs here.
- Integrity checks might be integrated into certain parts to discover modifications and to protect against attacks like “service injection” [10].
- In addition, some known countermeasures against reengineering might be added. For instance, methods to “break disassemblers, debuggers” [12].

<u>Protect code / Obfuscate execution</u>

- In addition, the complexity of an application might be highly increased by adding lots of nonsense functions to the source code that are based on available functions of the real source code. Most function calls need to be replaced by MSC calls, while this one replies with the required function name to proceed. This approach will finally bind the MSC and an app together. Attackers trying to reengineer such an app, will find lots of needless code. This means that they will need to employ advanced monitoring tools to accomplish their attack.

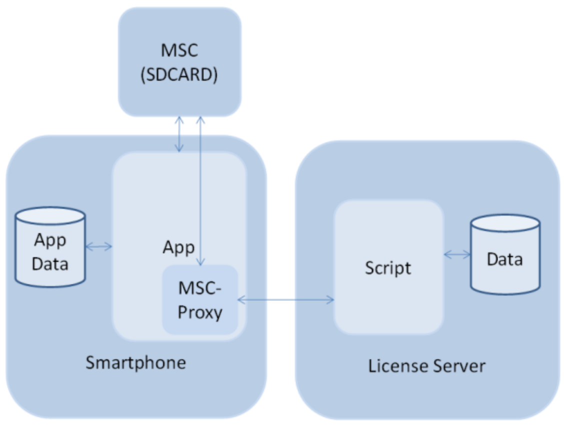

Fig. 3. Draft of the proposed solution

IV. RELATED WORK

Many similar research- and commercial projects exist. So far, the focus of most of these projects is on privacy protection or data protection in general.

For instance, a common problem in industry is the usage of smartphones by company employees commercially as well as privately. Commercial data might be at risk in this case.

Most related work uses software-based optimizations (e.g. the kernel modifications in “SEAndroid”[12]), while other vendors try to establish security by proposing new hardware (e.g. TEE[13] by Trustonic).

In terms of copy protection, research papers propose software related solutions. For instance, a paper by researchers of the Dankook and KonKuk University suggest using “Class Separation and Dynamic Loading for Android Applications” [11].

V. PERFORMANCE

Applets on Secure Elements run much slower than typical apps on a smartphone. At this stage of our research, it is unknown how much the proposed solutions will affect the performance of an app, while communicating to an secure element’s applet. If the performance denigrates significantly, the secure element solution may not be viable.

[10] Binary Refactor - https://github.com/argan/binary-refactor
[11] Application package file. It consists of resources, binaries and certificates of an app.

[12] Modified Android version by the NSA
[13] Trusted Execution Environment

VI. Conclusion

We are assuming that the proposed method of using secure elements (MSC) is going to improve the overall protection against piracy. For instance, important data can be exported to a secure place and the app can be bound to the hardware, itself. Nevertheless there might exist performance issues, because of the weak power of such secure elements. After the implementation is built, both performance and security tests are needed. For the security test, we suggest some sort of hacking contest to demonstrate the security of the approach we are suggesting. For the performance test, we will compare multiple apps to their protected versions in terms of execution time. Overall, our target solutions will add another security layer, but other security tools (e.g. obfuscation tools) should still be applied.

References

[1] Matthias Brandt, “Fast 80 Prozent Marktanteil fuer Android”, http://de.statista.com/themen/581/smartphones/infografik/1326/smartphone-absatz-weltweit/ , last access: 26th August 2013

[2] Red, “Kopierschutz von Android Market geknackt”, http://derstandard.at/1282273487603/App-Piraterie-Kopierschutz-von-Android-Market-geknackt, last access: 26th August 2013

[3] Justin Case, “Google’s Android Market License Verification Easily Corcumvented, Will Not Stop Pirates”, http://www.androidpolice.com/2010/08/23/exclusive-report-googles-android-market-license-verification-easily-circumvented-will-not-stop-pirates/ , last access: 26th August 2013

[4] Janosch Maier, “Enhanced Android Security to prevent Privilege Escalation”, p.16

[5] Google, “Android Security Overview”, http://source.android.com/devices/tech/security/index.html, last access: 27th August 2013

[6] Google, “Licensing Overview”, http://developer.android.com/google/play/licensing/overview.html , last access: 27th August 2013

[7] Thomas Zefferer, A-SIT, “Secure Elements am Beispiel von Google Wallet”, http://www.a-sit.at/pdfs/Technologiebeobachtung/20120428%20Studie_Google_Wallet.pdf, p.1 , last access: 30th August 2013

[8] Josef Langer, Andreas Dyrer, “Secure Element Development”, p.6, http://www.nfc-forum.org/events/oulu_spotlight/2009_09_01_Secure_Element_Programming.pdf, last access: 30th August 2013

[9] Google, “I/O 2011: Evading Pirates and Stopping Vampires”,

http://www.youtube.com/watch?feature=player_embedded&v=TnSNCXR9fbY, last acccess: 30th June 2013

[10] Damien Cauquil, Pierre Jaury, “Small footprint inspection techniques for Android”, http://events.ccc.de/congress/2012/Fahrplan/attachments/2103_SmallFootprintInspectionAndroid.pdf , last access: 31st August 2013

[11] Youn-Sik Jeong, Jae-Chan Moon, Dongjin Kim, Yeong-Ung Park, Seong-Je Cho, Minkyu Park, “An Anti-Piracy Mechanism based on Class Separation and Dynamic Loading for Android Applications”

[12] StackOverflow, “Protecting executable from reverse engineering?”, http://stackoverflow.com/questions/6481668/protecting-executable-from-reverse-engineering , last access: 6th September 2013