Research Article
The Dangers of Rooting: Data Leakage Detection in Android Applications

Luca Casati and Andrea Visconti
Department of Computer Science, Università degli Studi di Milano, Via Comelico 39/41, 20135 Milano, Italy
Correspondence should be addressed to Andrea Visconti; andrea.visconti@unimi.it
Received 31 July 2017; Revised 11 October 2017; Accepted 28 November 2017; Published 1 February 2018

Mobile devices are widely spread all over the world, and Android is the most popular operative system in use. According to Kaspersky Lab’s threat statistic (June 2017), many users are tempted to root their mobile devices to get an unrestricted access to the file system, to install different versions of the operating system, to improve performance, and so on. The result is that unintended data leakage flaws may exist. In this paper, we (i) analyze the security issues of several applications considered relevant in terms of handling user sensitive information, for example, financial, social, and communication applications, showing that 51.6% of the tested applications suffer at least of an issue and (ii) show how an attacker might retrieve a user access token stored inside the device thus exposing users to a possible identity violation. Notice that such a token, and a number of other sensitive information, can be stolen by malicious users through a man-in-the-middle (MITM) attack.

1. Introduction
In everyday routine, smartphones, laptops, tablets or, more in general, mobile devices have become an essential need for everyone. They are widely used to read e-mails, carry out financial transactions, browse maps, chat with other people, and so on. Mobile devices have to face a number of issues due to the resource constraints (performance issue [1, 2], e.g.) and also security issues (data leakage [3, 4], privacy concern [5, 6], etc.). In particular, the latter may be affected by the applications installed. Usually users choose such applications focusing on the number of total downloads [7], the reviews provided by users [8, 9], and so on. A typical environment where ratings can be easily found is Google Play Store, the largest app store which counts over 3 million applications available [10] split into two major categories: Apps and Games—with 2.5 million and 500 thousand apps, respectively [11]. However, it often happens that people who provide ratings evaluate the appearance, functionality, usability, and performances of an application without focusing on security aspects. In addition, as reported in Kaspersky Lab’s threat statistic (June 2017) [12] summarized in Table 1, security issues are further amplified by users when they root their phones. Notice that users obtain superuser access privileges to change the current Android version, to get access to the file system without restrictions, to install modified apps and gain more privileges, to improve performance, and so on. However, these access privileges may affect the security of installed applications [12, 13, 15], providing an access door to many sensitive information [16–18]. In this scenario, unintended data leakage flaws may exist.

In order to identify such flaws, in this paper, we extend and improve our previous work [19]. In particular, we improve our testing activities by analyzing not only the security issues of Android password managers but also those applications that are considered particularly relevant in terms of handling user sensitive information, such as financial, social, and communication applications. Notice that we do not describe innovative techniques but rather we measure the impact of a well-known technique (e.g., Xposed framework) on a rooted device, executing an extensive testing activities and observing that several applications do not implement the minimum security requirements.
In addition, we show the possibility to retrieve an access token, exposing users to a possible identity violation. Finally, we show that the same token (and many other sensitive information) can be retrieved through a man-in-the-middle (MITM) attack because several applications do not implement adequate cryptographic techniques for data protection or do not implement them at all.

The remainder of the paper is organized as follows. In Section 2, we describe a number of approaches that can be used to analyze applications. In Section 3, we show the solution adopted to retrieve sensitive information from Android applications. Particular attention is paid to describe hooking techniques. In Section 4, we present our testing activities, showing how malicious users might retrieve sensitive information. Finally, conclusions are drawn in Section 5.

2. Different Approaches to Analyze Applications

When an application lands on the market, it becomes suddenly available to be used by everyone. This means that it can be tested and analyzed under all possible conditions. Every internal element of an app should share the necessary information to perform a specific task without any data leakage. Unfortunately, this does not always happen.

In order to recognize possible data leakages, two well-known approaches can be used: static and dynamic analysis.

(i) Static analysis is based on the examination of an application without the execution of it [20]. Its radius of action is quite limited because many applications adopt obfuscation [21, 22] and dynamic code loading [23] to restrict access to internal information. However, it may be interesting to understand if the application’s associated files, such as database, backup, or log files, are encrypted. In this case, entropic techniques are very useful [24].

(ii) Dynamic analysis, instead, relies on the execution of the applications [25, 26]. The main idea is to collect (at runtime) the values that gradually come out from the called instructions. The advantage of this approach is to be less susceptible to code obfuscation.

In general, Android applications can assume many behaviors; thus it is necessary to monitor their activities, for example, through interface or automatic event injectors [27–29].

But there is also a third approach, situated halfway between the previous: the hybrid analysis [30, 31]. To work well, a system which adopts this technique must be designed in such a way that if the first was lacking, the second would take place, covering the gap [30].

In mobile device analysis, there is no a standard approach (static or dynamic) to collect data optimally. More precisely, we collect data via static analysis, and then we employ them in a dynamic scanning. This was accomplished through hooking techniques (hooking means to intercept methods with a known signature called by an application, acquiring its complete control), setting up the scenario shown in Figure 1. Taking into account a Java class named Signature, we notice that (a) the method initSign is invoked, (b) initSign receives a PrivateKey object, (c) initSign pass the object itself to another method, that is, engineInitSign of Figure 1, and (d) Hooker could take control of the method call, spying, or replacing its contents.

To better understand how this mechanism works, we explain in detail the hooking techniques—Xposed framework [32]—in Section 3.

3. How to Retrieve Sensitive Information

A generic Android application is a single compressed archive which includes essential information about the app [33]. Among all this information, we focus on the DEX file (Figure 2) because it provides interesting features related to the target application [34, 35].

We developed a tool, called Apk2Method, which

(i) opens the APK of the target application;

(ii) identifies the classes.dex file that looks for a specific marker, that is, 6465 780a 3033 3500 in Hex;

(iii) reads all methods invoked related to cryptographic field;

(iv) finally, outputs a text file where all gathered data are stored in a convenient format for a subsequent parsing. For the sake of simplicity, we call such a file file.txt.

Then, we developed an Android application which

(v) inputs data previously stored in file.txt and parses such a file using Java reflections and regular expressions;

(vi) runs inside a module of the Xposed framework, called Prober, which is able to select the target application.

More precisely, Prober represents the real execution engine of the hooking technique, implemented by Xposed. The Xposed framework, in turn, takes control of each method called by the target application, spying, or replacing each passed argument. Doing so, the control flow of an application can be changed, providing us the ability to execute our own code enriched with specific security tests.

<table>
<thead>
<tr>
<th>Country</th>
<th>Rooted devices</th>
<th>Place in top 100 countries attacked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bangladesh</td>
<td>13%</td>
<td>2</td>
</tr>
<tr>
<td>Indonesia</td>
<td>12%</td>
<td>3</td>
</tr>
<tr>
<td>Nepal</td>
<td>12%</td>
<td>5</td>
</tr>
<tr>
<td>Algeria</td>
<td>19%</td>
<td>7</td>
</tr>
<tr>
<td>Nigeria</td>
<td>13%</td>
<td>9</td>
</tr>
<tr>
<td>Ghana</td>
<td>12%</td>
<td>10</td>
</tr>
<tr>
<td>Venezuela</td>
<td>26%</td>
<td>13</td>
</tr>
<tr>
<td>Moldova</td>
<td>15%</td>
<td>22</td>
</tr>
<tr>
<td>Ecuador</td>
<td>11%</td>
<td>25</td>
</tr>
<tr>
<td>Italy</td>
<td>12%</td>
<td>66</td>
</tr>
</tbody>
</table>

Table 1: The top 10 (out of 100) countries where Android devices are rooted most frequently and where mobile devices are attacked most often by a malware [12].

In mobile device analysis, there is no a standard approach (static or dynamic) to collect data optimally. More precisely, we collect data via static analysis, and then we employ them in a dynamic scanning. This was accomplished through hooking techniques (hooking means to intercept methods with a known signature called by an application, acquiring its complete control), setting up the scenario shown in Figure 1. Taking into account a Java class named Signature, we notice that (a) the method initSign is invoked, (b) initSign receives a PrivateKey object, (c) initSign pass the object itself to another method, that is, engineInitSign of Figure 1, and (d) Hooker could take control of the method call, spying, or replacing its contents.

To better understand how this mechanism works, we explain in detail the hooking techniques—Xposed framework [32]—in Section 3.

3. How to Retrieve Sensitive Information

A generic Android application is a single compressed archive which includes essential information about the app [33]. Among all this information, we focus on the DEX file (Figure 2) because it provides interesting features related to the target application [34, 35].

We developed a tool, called Apk2Method, which

(i) opens the APK of the target application;

(ii) identifies the classes.dex file that looks for a specific marker, that is, 6465 780a 3033 3500 in Hex;

(iii) reads all methods invoked related to cryptographic field;

(iv) finally, outputs a text file where all gathered data are stored in a convenient format for a subsequent parsing. For the sake of simplicity, we call such a file file.txt.

Then, we developed an Android application which

(v) inputs data previously stored in file.txt and parses such a file using Java reflections and regular expressions;

(vi) runs inside a module of the Xposed framework, called Prober, which is able to select the target application.

More precisely, Prober represents the real execution engine of the hooking technique, implemented by Xposed. The Xposed framework, in turn, takes control of each method called by the target application, spying, or replacing each passed argument. Doing so, the control flow of an application can be changed, providing us the ability to execute our own code enriched with specific security tests.
Notice that it may happen that a portion of the target application’s information is encrypted or obfuscated [36], using specific tools such as Proguard, DashO, and DexProtector. These tools rename classes, methods, and variables assigning them meaningless names [37]. Consequently, the parsing activity will be very difficult and sometimes impossible (even with the support of the reflections [30]). In all other cases, if applications release sensitive information, our approach is able to detect these leaks.

3.1. The Xposed Framework. The framework used [32] is identified by four individual components: the Xposed, the XposedBridge, the XposedInstaller, and the XposedMods system. Among these, the first two are responsible for preparing the device to accommodate the framework. Let us briefly explain what happens when two generic methods, A and B, are called (Figures 3 and 4).

When the device is switched on,

1. the boot sequence starts: (a) the Boot ROM code starts executing from a predefined location, loading the bootloader into RAM, (b) the bootloader sets up the necessary resources in two stages—network and memory—needed to run the kernel, (c) the Android kernel sets up a group of resources—cache, protected memory, scheduling, and drivers—and looks for init in the system files, (d) init is the very first process, which sets the environment for Zygote [38] and daemons, and (e) daemons are invoked;

2. once the daemons are invoked, an extended version of process /system/bin/app_process [39] is called, which is meant to load the necessary classes designed to perform hooking—XposedBridge.jar;

3. as soon as an application calls a generic method (A), it is intercepted and redirected firstly to hookMethodNative, which increases the privilege level of the method received as argument, and secondly to handleHookedMethod, which links the method implementation to its own native generic method. In this way, it is possible to read all the arguments;

4. finally, the flow resumes naturally.

4. Testing Activity

We download and analyze several applications from Google’s official Android Market, using two mobile devices—Wiko Wax (Android KitKat, rooted with KingRoot [40])
and Samsung Galaxy Nexus (Android Lollipop, rooted with Nexus Root Toolkit [41]) (at time of writing, Android KitKat and Lollipop represent nearly half (about 47%) of the market) [42].

Our analysis follows two main directions. A first approach targets events resulting from data leakage of the method calls. These leaks are usually characterized by an improper use of objects as arguments, for example, using string as passwords, making whole structures visible, and so on. Then, to improve the ability to recognize data leakage, a second approach has been developed with the aim to find leaks on data transmitted over the Internet by phone.

4.1. First Approach. We downloaded 135 Android applications from Google Play Store, where 36 applications belong to “TOOLS” category, 54 to “PRODUCTIVITY,” 7 to “SOCIAL,” 8 to “COMMUNICATION,” and 30 to “FINANCE,” taking care of the installation count value. Such indicators represent the number of users who installed the chosen application, and it can be found at the information panel of each application [43]. In addition, let us remark that the choice of a particular application was taken relying on the fact that is used for security purposes and deal with data that are particularly sensitive for user side. For each application, we collect and store classes, methods, arguments, and return values.

More precisely, our approach works as follows (Figure 5):
An application alpha.apk is downloaded from Google Play Store and installed on the device.

Then alpha.apk is transferred to the computer, using the Android Debug Bridge (ADB) [44].

The Apk2Method tool inputs alpha.apk.

The Apk2Method tool outputs classes and methods, storing them in file.txt previously mentioned in Section 3. The top of Figure 6 shows a toy example, pointing out that classes and methods of an application might be obfuscated.

Such a file is copied in a specific path of our application Prober, and a rebooting of the mobile device is required to apply changes to the system.

When the alpha application runs—for example, the user input ID, password, e-mail, personal data, and so on—Prober stores methods invoked, arguments, and return values in file.log, as shown in the lower part of Figure 6.

Finally, in file.log, we are able to identify the presence of data leakage.

All apps analyzed have been cataloged using four levels of granularity: (1) no leakage: the application is safe; (2) abnormal behavior: the application suddenly freezes or crashes; (3) privacy concerns: the application releases unprotected sensitive information, that is, IMEI, phone number, geolocation, OS, and so on; and (4) account info: the application reveals account information—login IDs and passwords.

As shown in Tables 2 and 3 and in Figure 7, the testing results suggest that some issues have been identified for the category tools, productivity, and finance. In particular, in such categories, 51.6% of the tested applications suffer from one (at least) of the following issues:

(i) The application does not perceive to be observed.

(ii) The application does not warn the user about the presence of a jailbroken/rooted device.

(iii) Private keys used during a communication (e.g., the OpenSSLRSAPrivateCrtKey or the RSAPrivateKey and the associated parameters) are in plaintext.

(iv) Personal data, such as IMEI and geolocation, are not protected.

(v) The master password (of the password manager) or the users account password (login IDs and password) are handled in plaintext.

On the contrary, the applications tested which belong to social and communication are not affected by the same issues.

4.2. Second Approach. A second issue is related to the leakage of encrypted data transmitted over the Internet and stored in the device itself. To avoid a user being forced to create a new account, a common practice is to exploit a third-party app that handles the authentication phase using a delegation protocol—for example, OAuth 2.0 [45]. In particular, the authentication phase is done through an access token that is stored in the application’s internal directory, preventing user from entering the login credentials...
Since the access token can be seen as a set of user attributes used to prove that a user is authenticated, the client application usually does not use a mechanism to validate the access token, and in rooted devices, this token can be easily found by browsing the application’s folder; an attacker may retrieve such a token and inject it during a new authentication phase, stealing the identity of the victim (see Eve in Figure 8). Moreover, for all users who ignore the alerts and unknowingly accept everything, the token may be stolen on the channel through a man-in-the-middle attack.

For this set of users, we also tried to identify different types of possible attacks. Therefore, we downloaded and analyzed 67 Android apps that send data over the Internet and should take care about user sensitive information. As described in Section 4.1, these applications belong to the following categories: 2 apps belong to “TOOLS,” 16 to “PRODUCTIVITY,” 4 to “SOCIAL,” 10 to “COMMUNICATION,” and 35 to “FINANCE.” The main issue found is that several applications do not perform the SSL/TLS client authentication, thus making them potentially vulnerable to a man-in-the-middle attack. Tables 4 and 5 summarize our testing activities. More precisely, we found leaks on 55.2% of the apps tested, where 50.0% comes from “TOOLS,” 75.0% from “PRODUCTIVITY,” 25.0% from “SOCIAL,” 60.0% from “COMMUNICATION,” and 48.6% from “FINANCE.”

5. Conclusions

Since mobile devices are widely spread and used for everything, the protection of information, transaction data, and privacy have to be taken into account seriously.

In this paper, we focused on the real case scenario of rooted devices, analyzing the most installed Android applications with the aim to check how safe they are. We

<table>
<thead>
<tr>
<th>Installation count</th>
<th>No leakage</th>
<th>Abnormal behavior</th>
<th>Privacy concerns</th>
<th>Secret data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000,000–5,000,000,000</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>500,000,000–1,000,000,000</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100,000,000–500,000,000</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50,000,000–100,000,000</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10,000,000–50,000,000</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1,000,000–5,000,000</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>500,000–1,000,000</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>100,000–500,000</td>
<td>19</td>
<td>8</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>50,000–100,000</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>10,000–50,000</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5,000–10,000</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>50–5,000</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>18</td>
</tr>
</tbody>
</table>

Figure 7: The histogram shows the results of all ranges of Table 2.
showed that 62 out of 135 apps suffer data leakage and 37 out of 67 apps, which send sensitive information over the Internet, are potentially vulnerable to man-in-the-middle attacks. The most significant flaws found concern (a) password managers (we assume that password managers store user passwords implementing the minimum requirements for cryptographic applications, for example, adopting a password-based key derivation function [46, 47] and avoiding the well-known issues described in literature [14, 48–50]) that may release ID password of several accounts or the master password of password manager themselves; (b) financial applications that sometimes release secret codes or

Table 4: Number of apps that are potentially vulnerable to a MITM attack.

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of apps</th>
<th>MITM vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Productivity</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Social</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Communication</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>Finance</td>
<td>35</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 5: Correlation between the installation count and MITM vulnerability.

<table>
<thead>
<tr>
<th>Installation count</th>
<th>MITM vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000,000–5,000,000,000</td>
<td>3</td>
</tr>
<tr>
<td>500,000,000–1,000,000,000</td>
<td>2</td>
</tr>
<tr>
<td>100,000,000–500,000,000</td>
<td>5</td>
</tr>
<tr>
<td>50,000,000–100,000,000</td>
<td>1</td>
</tr>
<tr>
<td>10,000,000–50,000,000</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000–5,000,000</td>
<td>5</td>
</tr>
<tr>
<td>500,000–1,000,000</td>
<td>1</td>
</tr>
<tr>
<td>100,000–500,000</td>
<td>11</td>
</tr>
<tr>
<td>50,000–100,000</td>
<td>3</td>
</tr>
<tr>
<td>10,000–50,000</td>
<td>1</td>
</tr>
<tr>
<td>5,000–10,000</td>
<td>0</td>
</tr>
<tr>
<td>50–5,000</td>
<td>1</td>
</tr>
</tbody>
</table>
account credentials; and (c) applications which do not implement a SSL/TLS client authentication, making them potentially vulnerable to a MITM attack. Notice that the issues described in this paper can be easily faced by app developers—for example, exploiting obfuscation/encryption mechanisms, passing sensitive data using objects, or implementing two-step verification techniques—and users—for example, installing a stock ROM instead of a custom one.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to thank Marco Mauri and Giovanni Intorre who executed part of the testing activity described in Section 4.2.

References


[24] R. Lyda and J. Hamrock, "Using entropy analysis to find dangerous are your smartphones? App usage recommenda-


