Hindawi
Mobile Information Systems
Volume 2018, Article ID 6020461, 9 pages
https://doi.org/10.1155/2018/6020461

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

Academic Editor: Jinglan Zhang

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.

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].

Country	Rooted devices	Place in top 100 countries attacked
Bangladesh	13%	2
Indonesia	12%	3
Nepal	12%	5
Algeria	19%	7
Nigeria	13%	9
Ghana	12%	10
Venezuela	26%	13
Moldova	15%	22
Ecuador	11%	25
Italy	12%	66

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.

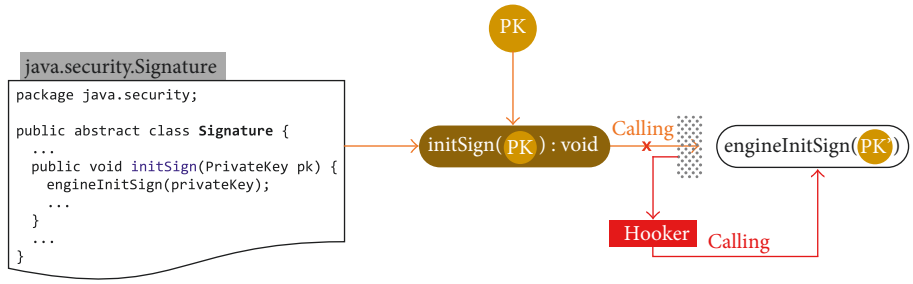

Figure 1: An example of the hooking technique in action, specialized in spying.

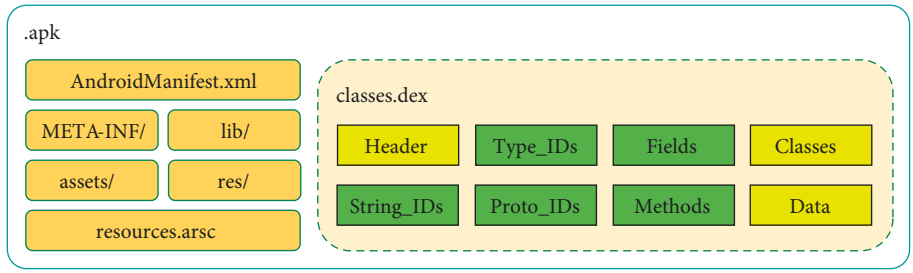

Figure 2: A compact view of an APK file, pointing out the DEX file components.

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 Dex-Protector. 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*;

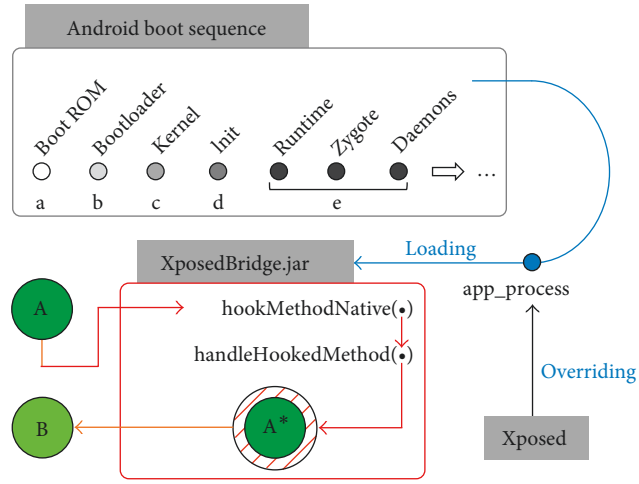

Figure 3: A diagram that shows how Xposed works in detail while intercepting a method.

(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])

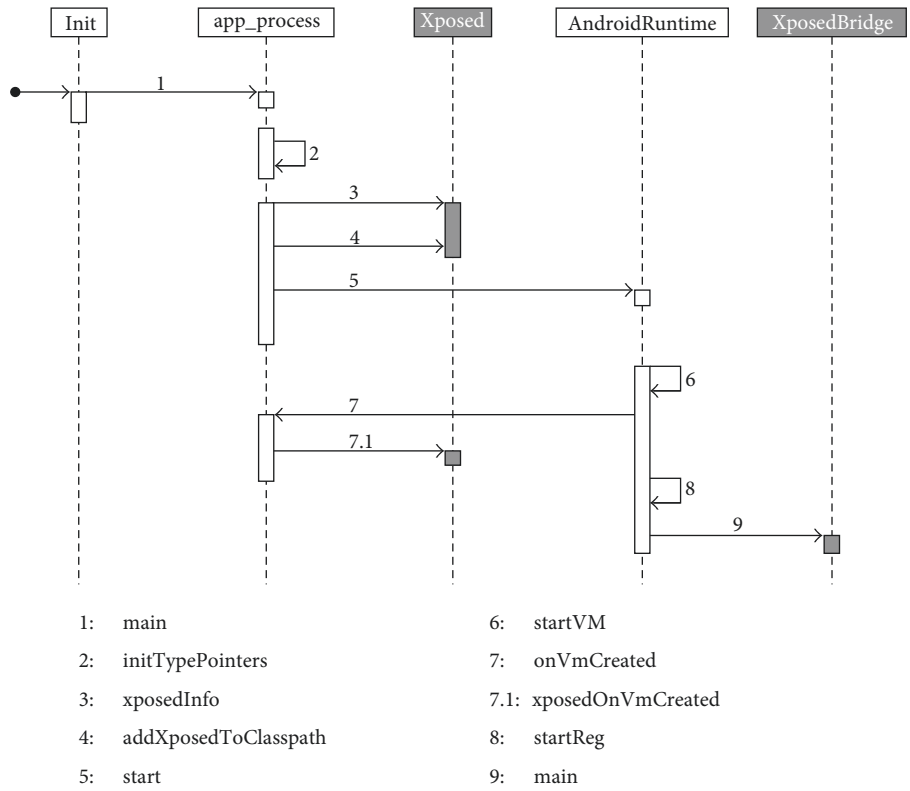

Figure 4: The sequence diagram illustrates how the system changes while the framework is active.

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):

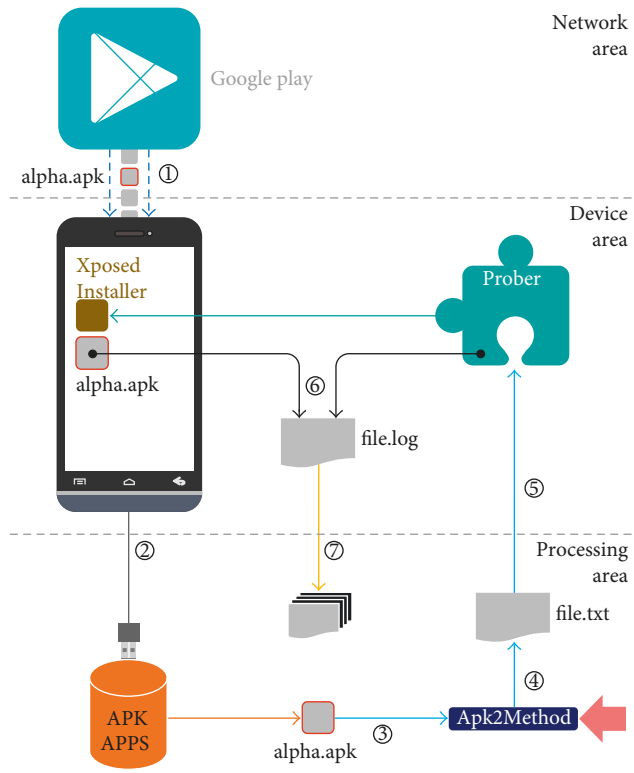

Figure 5: The entire project control flow which represents how an Android application is analyzed.

Figure 6: A toy example of the outputs obtained by analyzing an application *alpha.apk*.

(1) An application *alpha.apk* is downloaded from *Google Play Store* and installed on the device.

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

(3) The *Apk2Method* tool inputs *alpha.apk*.

(4) 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.

(5) 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.

(6) 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.

(7) 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.

Table 2: The results of the analysis obtained with the Android *5.x* device.

	No leakage	Abnormal behavior	Privacy concerns	Secret data
Tools	18	0	2	18
Productivity	23	3	1	31
Social	7	5	0	0
Communication	8	3	0	0
Finance	17	16	7	13

(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

Table 3: Correlation between the installation count and the 4 levels of granularity.

Installation count	No leakage	Abnormal behavior	Privacy concerns	Secret data
1,000,000,000–5,000,000,000	4	1	0	0
500,000,000–1,000,000,000	3	1	0	0
100,000,000–500,000,000	6	4	0	0
50,000,000–100,000,000	2	2	0	0
10,000,000–50,000,000	2	0	0	1
1,000,000–5,000,000	3	5	2	9
500,000–1,000,000	4	2	0	7
100,000–500,000	19	8	5	11
50,000–100,000	7	3	1	3
10,000–50,000	10	1	2	9
5,000–10,000	3	0	0	4
50–5,000	10	1	0	18

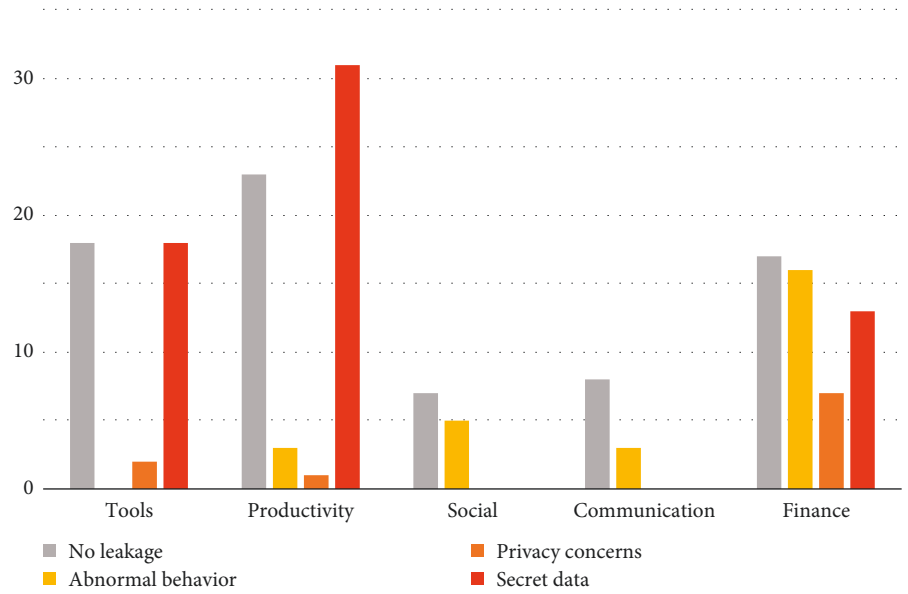

Figure 7: The histogram shows the results of all ranges of Table 2.

(see Alice in Figure 8). 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

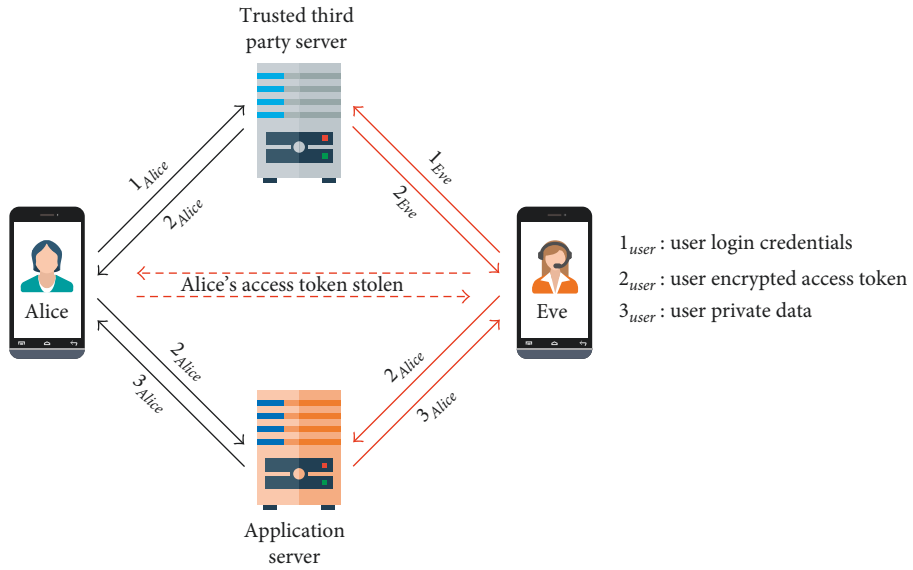

Figure 8: A graphical representation of the problem concerning the delegation scheme implemented by some applications.

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

	Number of apps	MITM vulnerability
Tools	2	1
Productivity	16	12
Social	4	1
Communication	10	6
Finance	35	17

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

Installation count	MITM vulnerability
1,000,000,000–5,000,000,000	3
500,000,000–1,000,000,000	2
100,000,000–500,000,000	5
50,000,000–100,000,000	1
10,000,000–50,000,000	3
1,000,000–5,000,000	5
500,000–1,000,000	1
100,000–500,000	11
50,000–100,000	3
10,000–50,000	1
5,000–10,000	0
50–5,000	1

showed that 62 out of 135 apps suffer of 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

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

[1] M. Louk, H. Lim, and H. Lee, "An analysis of security system for intrusion in smartphone environment," *The Scientific World Journal*, vol. 2014, Article ID 983901, 12 pages, 2014.

[2] M. Lettner, M. Tschernuth, and R. Mayrhofer, "Mobile platform architecture review: Android, iPhone, QT," in *Proceedings of the International Conference on Computer Aided Systems Theory*, pp. 544–551, Springer, Las Palmas de Gran Canaria, Spain, February 2011.

[3] P. Feng, J. Ma, and C. Sun, "Selecting critical data flows in Android applications for abnormal behavior detection," *Mobile Information Systems*, vol. 2017, Article ID 7397812, 16 pages, 2017.

[4] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on Android security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 623–634, ACM, Berlin, Germany, November 2013.

[5] K. Zhu, X. He, B. Xiang, L. Zhang, and A. Pattavina, "How dangerous are your smartphones? App usage recommendation with privacy preserving," *Mobile Information Systems*, vol. 2016, Article ID 6804379, 10 pages, 2016.

[6] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, p. 3, ACM, New York, NY, USA, July 2012.

[7] J. Alegre-Sanahuja, J. Camacho, J. C. Cortés López, F. J. Santonja, and R. J. Villanueva Micó, "Agent-based model to study and quantify the evolution dynamics of Android malware infection," *Abstract and Applied Analysis*, vol. 2014, Article ID 623436, 10 pages, 2014.

[8] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of Google Play," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 221–233, 2014.

[9] E. Guzman and W. Maalej, "How do users like this feature? A fine grained sentiment analysis of app reviews," in *Proceedings of the IEEE 22nd International Requirements Engineering Conference (RE)*, pp. 153–162, IEEE, Karlskrona, Sweden, August 2014.

[10] AppBrain, "Number of Android applications," 2017, http://www.appbrain.com/stats/number-of-android-apps.

[11] AppBrain, "Most popular Google Play categories," 2017, http://www.appbrain.com/stats/android-market-app-categories.

[12] Kaspersky Lab, "Rooting your Android: advantages, disadvantages, and snags," 2017, https://www.kaspersky.com/blog/android-root-faq/17135/.

[13] W. Jeon, J. Kim, Y. Lee, and D. Won, "A practical analysis of smartphone security," in *Human Interface and the Management of Information. Interacting with Information*, pp. 311–320, 2011.

[14] S. Bossi and A. Visconti, "What users should know about full disk encryption based on LUKS," in *Proceedings of the 14th International Conference on Cryptology and Network Security*, Marrakesh, Morocco, December 2015.

[15] C. Vorakulpipat, S. Sirapaisan, E. Rattanalerdnusorn, and V. Savangsuk, "A policy-based framework for preserving confidentiality in BYOD environments: a review of information security perspectives," *Security and Communication Networks*, vol. 2017, Article ID 2057260, 11 pages, 2017.

[16] S. T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 3–14, ACM, Denver, CO, USA, October 2015.

[17] Y. Kim, T. Oh, and J. Kim, "Analyzing user awareness of privacy data leak in mobile applications," *Mobile Information Systems*, vol. 2015, Article ID 369489, 12 pages, 2015.

[18] M. Nauman, S. Khan, X. Zhang, and J. P. Seifert, "Beyond kernel-level integrity measurement: enabling remote attestation for the android platform," in *Proceedings of the Trust and Trustworthy Computing*, pp. 1–15, Berlin, Germany, June 2010.

[19] L. Casati and A. Visconti, "Exploiting a bad user practice to retrieve data leakage on android password managers," in *Proceedings of the International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 952–958, Springer, Torino, Italy, July 2017.

[20] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 421–430, IEEE, Miami Beach, FL, USA, December 2007.

[22] I. You and K. Yim, "Malware obfuscation techniques: a brief survey," in *Proceedings of the International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA 2010)*, pp. 297–300, IEEE, Fukuoka, Japan, 2010.

[23] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 14, pp. 23–26, San Diego, CA, USA, February 2014.

[24] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40–45, 2007.

[25] O. Somarriba, U. Zurutuza, R. Uribeetxeberria, L. Delosières, and S. Nadjm-Tehrani, "Detection and visualization of android malware behavior," *Journal of Electrical and Computer Engineering*, vol. 2016, Article ID 8034967, 17 pages, 2016.

[26] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: mining API-level features for robust malware detection in android," in *Proceedings of the International Conference on Security and*

Privacy in Communication Systems, pp. 86–103, Springer, Guangzhou, China, October 2013.

[27] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, vol. 48, no. 10, pp. 641–660, 2013.

[28] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 224–234, ACM, Saint Petersburg, Russia, August 2013.

[29] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: segmented evolutionary testing of Android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 599–609, ACM, Hong Kong, China, November 2014.

[30] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Computing Surveys*, vol. 49, no. 4, pp. 1–41, 2017.

[31] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: automatic reconstruction of Android malware behaviors," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2015.

[32] *Xposed Module Repository*, 2017, http://repo.xposed.info/.

[33] L. Li, D. Li, T. F. Bissyandé et al., "Understanding Android app piggybacking: a systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[34] H. S. Oh, B. J. Kim, H. K. Choi, and S. M. Moon, "Evaluation of Android Dalvik virtual machine," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, ACM, New York, NY, USA, October 2012.

[35] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to java bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 6, ACM, Cary, NC, USA, November 2012.

[36] J. Park, H. Kim, Y. Jeong et al., "Effects of code obfuscation on Android app similarity analysis," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 6, no. 4, pp. 86–98, 2015.

[37] F. Sierra and A. Ramirez, "Defending your Android app," in *Proceedings of the 4th Annual ACM Conference on Research in Information Technology*, ACM, Chicago, IL, USA, October 2015.

[38] J. Andrus and J. Nieh, "Teaching operating systems using Android," in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*, ACM, New York, NY, USA, 2012.

[39] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing Android-powered mobile devices using SELinux," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 36–44, 2010.

[40] KingRoot, 2017, https://kingroot.net/.

[41] Nexus Root Toolkit v.2.1.9, 2016, http://www.wugfresh.com/nrt/.

[42] Dashboards–Platform Versions, 2017, https://developer.android.com/about/dashboards/index.html.

[43] Google Play Console Help Center, 2017, https://support.google.com/googleplay/android-developer/.

[44] Android Debug Bridge, 2017, https://developer.android.com/studio/command-line/adb.html.

[45] D. Hardt, *The OAuth 2.0 Authorization Framework, RFC 6749*, 2012, https://www.rfc-editor.org/info/rfc6749.

[46] Password Hashing Competition, https://password-hashing.net.

[47] K. Moriarty, B. Kaliski, and A. Rusch, *PKCS#5: Password-Based Cryptography Specification Version 2.1. RFC 8018*, 2017, https://www.rfc-editor.org/info/rfc8018.

[48] J. Steube, "Optimising computation of hash-algorithms as an attacker," 2013, https://hashcat.net/events/p13/js-ocohaaaa.pdf.

[49] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the weaknesses of PBKDF2," in *Proceedings of the 14th International Conference on Cryptology and Network Security*, Marrakesh, Morocco, December 2015.

[50] A. Ruddick and J. Yan, "Acceleration attacks on PBKDF2: or, what is inside the black-box of oclHashcat?," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies*, Austin, TX, USA, August 2016.