The Dangers of Rooting: Data Leakage Detection in Android Applications[*]

Luca Casati[†] and Andrea Visconti[†]

Abstract Mobile devices are widely spread all over the world and Android is the most popular operative system in use. According to a 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 (a) analyze the security issues of several applications considered relevant in term of handling user sensitive information, e.g. financial, social, and communication applications, showing that 51.6% of the tested applications suffer at least of an issue; (b) 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.

Key words: Data leakage · Mobile app · Rooted device · Hooking · Code injection

Luca Casati
Department of Computer Science, Università degli Studi di Milano, via Comelico 39/41, 20135, Milano, Italy, e-mail: `luca.casati1@studenti.unimi.it`

Andrea Visconti
Department of Computer Science, Università degli Studi di Milano, via Comelico 39/41, 20135, Milano, Italy, e-mail: `andrea.visconti@unimi.it`

[*] This paper extends and improves our previous work "Exploiting a Bad User Practice to Retrieve Data Leakage on Android Password Managers" presented at the 11th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2017).

[†] The authors declare that there is no conflict of interest regarding the publication of this paper.

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 [26, 24], for example) and also security issues (data leakage [18, 48], privacy concern [50, 17], etc.). In particular, the latter may be affected by the applications installed. Usually users choose such applications focusing on the number of total downloads [9], the reviews provided by users [45, 19], 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 [12] 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, performances of an application without focusing on security aspects. In addition, as reported in the Kaspersky Lab's threat

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 [22].

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

statistic (June, 2017) [22] 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 [22, 21, 47], providing an access door to many sensitive information [42, 23, 32]. 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 [15]. 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 term 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 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 adequately 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*.

- Static analysis is based on the examination of an application without the execution of it [16]. Its radius of action is quite limited, because many applications adopt obfuscations [31, 49] and dynamic code loading [36] 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 [27].

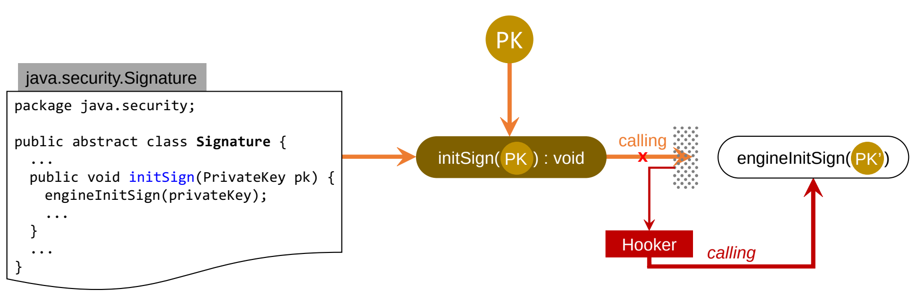

Fig. 1 An example of the hooking technique in action, specialized in spying.

- Dynamic analysis, instead, relies on the execution of the applications [40, 8]. 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 [13, 28, 29].

But there is also a third approach, situated halfway between the previous: the *hybrid analysis* [43, 44]. 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 [43].

In mobile device analysis, there is not 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[3] techniques, setting up the scenario shown in Figure 1. Taking into account a Java class named *Signature*, notice that (a) the method *initSign* is invoked, (b) *initSign* receives a *PrivateKey* object, (c) *initSign* pass the object itself to another method — i.e., *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 [7] — in the next section.

3 How to Retrieve Sensitive Information

A generic Android application is a single compressed archive which includes essential information about the app [25]. Among all this information, we focus on the DEX file (see Figure 2) because it provides interesting features related to the target application [34, 33].

We developed a tool, called *Apk2Method*, which:

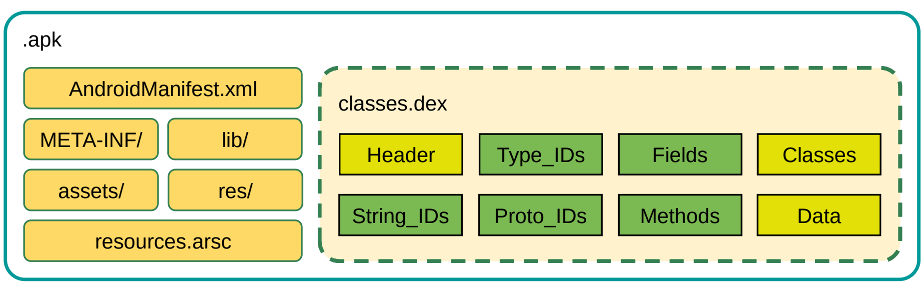

Fig. 2 A compact view of an APK file, pointing out the DEX file components.

[3] Hooking means to intercept methods with a known signature called by an application, acquiring its complete control.

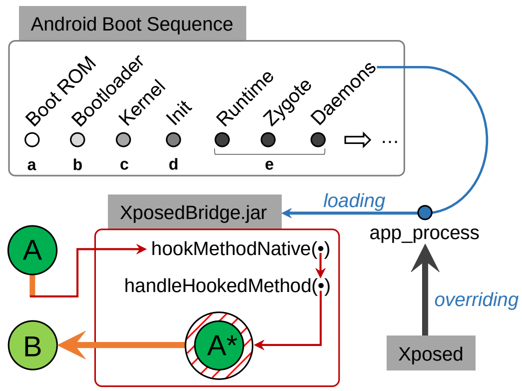

Fig. 3 A diagram that shows how Xposed works in detail while intercepting a method.

- opens the APK of the target application;
- identifies the *classes.dex* file looks for a specific marker — i.e., `6465 780a 3033 3500` in *Hex*;
- reads all methods invoked related to cryptographic field, and finally
- outputs a text file where all gathered data are stored in a convenient format for a subsequent parsing. For sake of simplicity, we call such a file *file.txt*.

Then, we developed an Android application which:

- inputs data previously stored in *file.txt* and parses such a file using Java reflections and regular expressions;
- 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 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 are encrypted or obfuscated [35], using specific tools such as Proguard, DashO, and DexProtector. These tools rename classes, methods and variables assigning them meaningless names [39]. Consequently, the parsing activity will be very difficult and sometimes impossible (even with the support of the reflections [43]). In all other cases, if applications release sensitive information, our approach is able to detect these leaks.

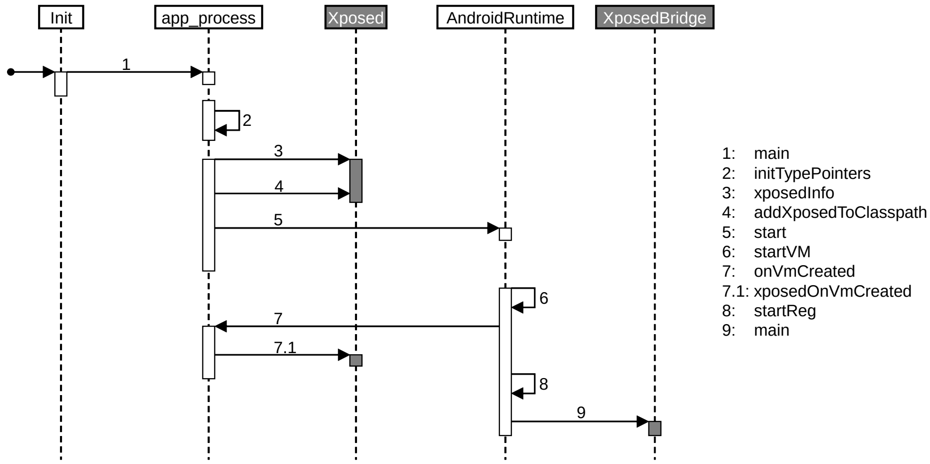

Fig. 4 The sequence diagram illustrates how the system changes while the framework is active.

3.1 The Xposed framework

The framework used [7] 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 (see Figure 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 setups in two stages the necessary resources — i.e., network, memory — needed to run the kernel, (c) the Android kernel setups a group of resources — i.e. 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* [10] and daemons, and (e) daemons are invoked;
2. once the daemons are invoked, an extended version of process */system/bin/app_process* [38] is called, which is meant to load the necessary classes designed to perform hooking — i.e., *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 — i.e., *Wiko Wax* (Android KitKat, rooted with *KingRoot* [3]) and *Samsung Galaxy Nexus* (Android Lollipop, rooted with *Nexus Root Toolkit* [5])[4].

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 the phone.

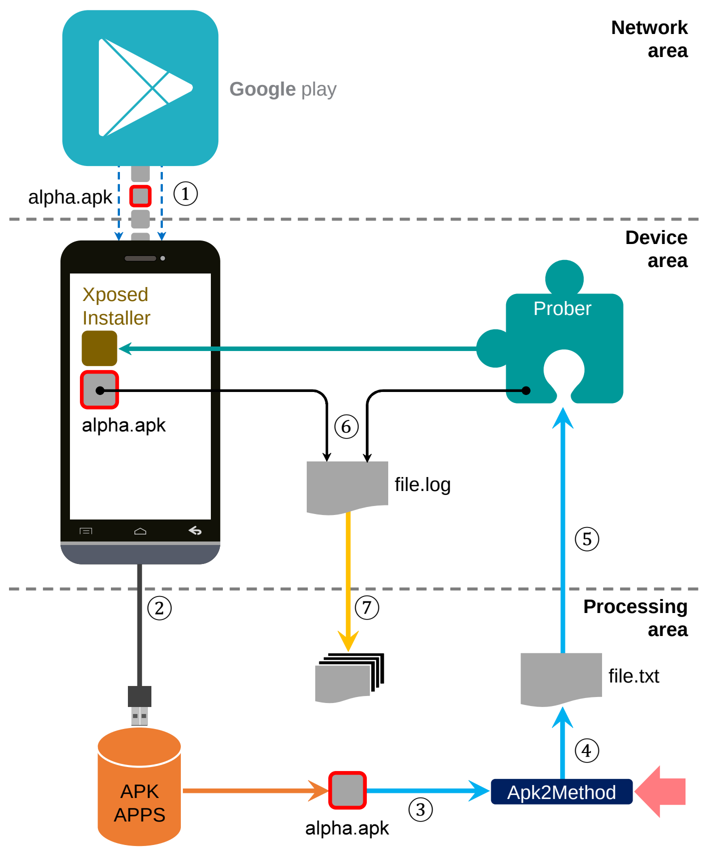

Fig. 5 The entire project control flow which represents how an Android application is analyzed.

[4] At time of writing, Android KitKat and Lollipop represent nearly half (about 47%) of the market [6]

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 indicator represent the number of users who installed the chosen application and it can be found at the information panel of each application [2]. 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 (see Figure 5):

1. an application *alpha.apk* is downloaded from *Google Play Store* and installed on the device;
2. then *alpha.apk* is transferred on the computer, using the Android Debug Bridge (ADB) [1];
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 system;

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

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

⑥ when the *alpha* application runs — e.g., the user inputs 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 — i.e., IMEI, phone number, geolocation, OS, and so on; (4) *account info*: the application reveals account information — i.e., login IDs and passwords.

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

- the application does not perceive to be observed;
- the application does not warn the user about the presence of a jailbroken/rooted device;
- private keys used during a communication (e.g. the *OpenSSLRSAPrivateCrtKey* or the *RSAPrivateKey* and the associated parameters) are in plaintext;
- personal data, such as IMEI and geolocation are not protected;

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

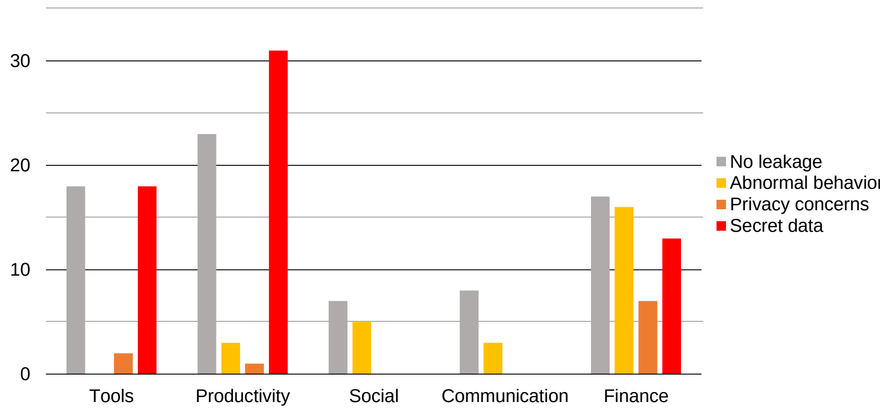

Fig. 7 The histogram shows the results of all ranges of Table 2.

- 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 handle the authentication phase using a delegation protocol — e.g. OAuth 2.0 [20]. 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 (see Alice in Fig. 8). Since (1) the access token can be seen as a set of user attributes used to prove that a user is authenticated, (2) the client application usually does not use a mechanism to validate the access token, and (3) in rooted devices this token

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

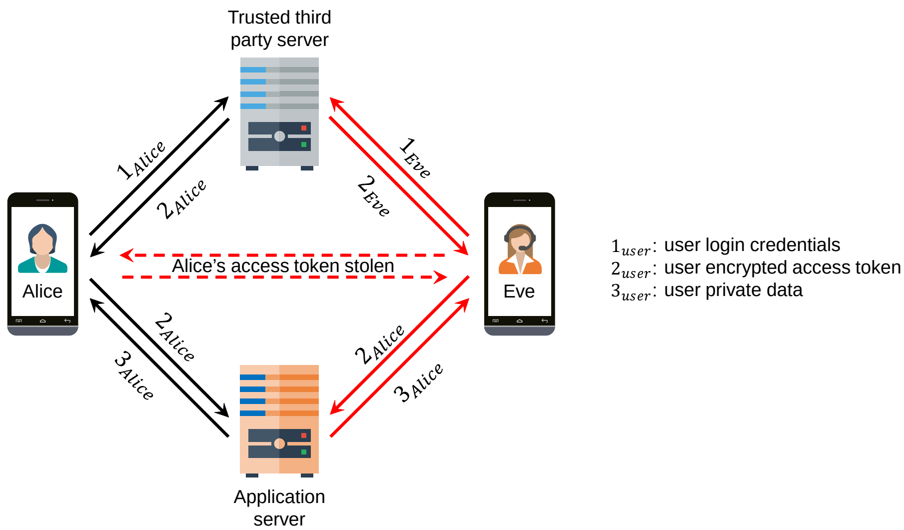

Fig. 8 A graphical representation of the problem concerning the delegation scheme implemented by some applications.

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 Fig. 8). Moreover, for all users who ignore the alerts and unknowingly accept everything, the token may be steal 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

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

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 4–5 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 has 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 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[5] that may release ID–password of several accounts or the master password of password manager themself; (b) financial applications that sometimes release secret codes or account credentials, and (c) applications who 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 — e.g., installing a stock ROM instead of a custom one.

6 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. Android Debug Bridge. URL https://developer.android.com/studio/command-line/adb.html. [Online, accessed 19 February 2017]

[5] We assume that password managers store user passwords implementing the minimum requirements for cryptographic applications, for example adopting a password-based key derivation function [4, 30] and avoiding the well-known issues described in literature [41, 46, 37, 14].

2. Google Play Console Help Center. https://support.google.com/googleplay/android-developer/. [Online, accessed 1 October 2017]
3. KingRoot. URL https://kingroot.net/. [Online, accessed 27 February 2017]
4. Password hashing competition. URL https://password-hashing.net
5. Nexus Root Toolkit v.2.1.9 (2016). URL http://www.wugfresh.com/nrt/. [Online, accessed 27 February 2017]
6. Dashboards - Platform Versions (2017). URL https://developer.android.com/about/dashboards/index.html. [Online, accessed 18 July 2017]
7. Xposed Module Repository (2017). URL http://repo.xposed.info/. [Online, accessed 27 February 2017]
8. Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: International Conference on Security and Privacy in Communication Systems, pp. 86–103. Springer (2013)
9. Alegre-Sanahuja, J., Camacho, J., Cortés López, J.C., Santonja, F.J., Villanueva Micó, R.J.: Agent-based model to study and quantify the evolution dynamics of android malware infection. In: Abstract and Applied Analysis, vol. 2014. Hindawi Publishing Corporation (2014)
10. Andrus, J., Nieh, J.: Teaching operating systems using android. In: Procs of the 43rd ACM technical symposium on Computer Science Education. ACM (2012)
11. AppBrain: Most popular Google Play categories. URL http://www.appbrain.com/stats/android-market-app-categories. [Online, accessed 12 July 2017]
12. AppBrain: Number of Android applications. URL http://www.appbrain.com/stats/number-of-android-apps. [Online, accessed 12 July 2017]
13. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of android apps. In: ACM SIGPLAN Notices, vol. 48, pp. 641–660. ACM (2013)
14. Bossi, S., Visconti, A.: What users should know about full disk encryption based on LUKS. In: Proc.s of the 14th International Conference on Cryptology and Network Security (2015)
15. Casati, L., Visconti, A.: Exploiting a bad user practice to retrieve data leakage on android password managers. In: International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, pp. 952–958. Springer (2017)
16. Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M.S., Conti, M., Rajarajan, M.: Android security: a survey of issues, malware penetration, and defenses. IEEE communications surveys & tutorials **17**(2), 998–1022 (2015)
17. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., Wagner, D.: Android permissions: User attention, comprehension, and behavior. In: Proceedings of the eighth symposium on usable privacy and security, p. 3. ACM (2012)
18. Feng, P., Ma, J., Sun, C.: Selecting critical data flows in android applications for abnormal behavior detection. Mobile Information Systems **2017** (2017)
19. Guzman, E., Maalej, W.: How do users like this feature? a fine grained sentiment analysis of app reviews. In: Requirements Engineering Conference (RE), 2014 IEEE 22nd International, pp. 153–162. IEEE (2014)
20. Hardt, D.: The oauth 2.0 authorization framework (2012)
21. Jeon, W., Kim, J., Lee, Y., Won, D.: A practical analysis of smartphone security. Human Interface and the Management of Information. Interacting with Information pp. 311–320 (2011)
22. Kaspersky Lab: Rooting your Android: Advantages, disadvantages, and snags. URL https://www.kaspersky.com/blog/android-root-faq/17135/
23. Kim, Y., Oh, T., Kim, J.: Analyzing user awareness of privacy data leak in mobile applications. Mobile Information Systems **2015** (2015)
24. Lettner, M., Tschernuth, M., Mayrhofer, R.: Mobile platform architecture review: android, iphone, qt. In: International Conference on Computer Aided Systems Theory, pp. 544–551. Springer (2011)
25. Li, L., Li, D., Bissyandé, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: A systematic study of malicious code grafting. IEEE Transactions on Information Forensics and Security **12**(6), 1269–1284 (2017)
26. Louk, M., Lim, H., Lee, H.: An analysis of security system for intrusion in smartphone environment. The Scientific World Journal **2014** (2014)

27. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security and Privacy **5**(2), 40–45 (2007). DOI 10.1109/MSP.2007.48. URL http://dx.doi.org/10.1109/MSP.2007.48
28. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: Procs of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 224–234. ACM (2013)
29. Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: Segmented evolutionary testing of android apps. In: Procs of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 599–609. ACM (2014)
30. Moriarty, K., and Kaliski, B., and A. Rusch: PKCS#5: Password-Based Cryptography Specification Version 2.1. RFC 8018 (2017)
31. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual, pp. 421–430. IEEE (2007)
32. Nauman, M., Khan, S., Zhang, X., Seifert, J.P.: Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. Trust and Trustworthy Computing pp. 1–15 (2010)
33. Octeau, D., Jha, S., McDaniel, P.: Retargeting android applications to java bytecode. In: Procs of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, p. 6. ACM (2012)
34. Oh, H.S., Kim, B.J., Choi, H.K., Moon, S.M.: Evaluation of android dalvik virtual machine. In: Procs of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems. ACM (2012)
35. Park, J., Kim, H., Jeong, Y., Cho, S., Han, S., Park, M.: Effects of code obfuscation on android app similarity analysis. J. Wireless Mobile Netw. Ubiquitous Comput. Dependable Appl **6**(4), 86–98 (2015)
36. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS, vol. 14, pp. 23–26 (2014)
37. Ruddick, A., Yan, J.: Acceleration attacks on pbkdf2: or, what is inside the black-box of oclhashcat? In: 10th USENIX Workshop on Offensive Technologies (2016)
38. Shabtai, A., Fledel, Y., Elovici, Y.: Securing android-powered mobile devices using selinux. IEEE Security & Privacy **8**(3), 36–44 (2010)
39. Sierra, F., Ramirez, A.: Defending your android app. In: Procs of the 4th Annual ACM Conference on Research in Information Technology. ACM (2015)
40. Somarriba, O., Zurutuza, U., Uribeetxeberria, R., Delosières, L., Nadjm-Tehrani, S.: Detection and visualization of android malware behavior. Journal of Electrical and Computer Engineering **2016** (2016)
41. Steube, J.: Optimising Computation of Hash-Algorithms as an Attacker. URL https://hashcat.net/events/p13/js-ocohaaaa.pdf
42. Sun, S.T., Cuadros, A., Beznosov, K.: 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 (2015)
43. Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L.: The evolution of android malware and android analysis techniques. ACM Comput. Surv. **49**(4) (2017). DOI 10.1145/3017427
44. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: Automatic reconstruction of android malware behaviors. In: NDSS (2015)
45. Viennot, N., Garcia, E., Nieh, J.: A measurement study of google play. In: ACM SIGMETRICS Performance Evaluation Review, vol. 42, pp. 221–233. ACM (2014)
46. Visconti, A., Bossi, S., Ragab, H., Calò, A.: On the weaknesses of PBKDF2. In: Proc.s of the 14th International Conference on Cryptology and Network Security (2015)
47. Vorakulpipat, C., Sirapaisan, S., Rattanalerdnusorn, E., Savangsuk, V.: A policy-based framework for preserving confidentiality in byod environments: A review of information security perspectives. Security and Communication Networks **2017** (2017)

48. Wu, L., Grace, M., Zhou, Y., Wu, C., Jiang, X.: The impact of vendor customizations on android security. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 623–634. ACM (2013)
49. You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on, pp. 297–300. IEEE (2010)
50. Zhu, K., He, X., Xiang, B., Zhang, L., Pattavina, A.: How dangerous are your smartphones? app usage recommendation with privacy preserving. Mobile Information Systems **2016** (2016)