Assessment of the Susceptibility to Data Manipulation of Android Games with In-app Purchases

Francisco Vigário, Miguel Neto, Diogo Fonseca, Mário Freire, Pedro Inácio

Assessment of the Susceptibility to Data Manipulation of Android Games with In-app Purchases

Francisco Vigário, Miguel Neto, Diogo Fonseca,
Mário M. Freire and Pedro R. M. Inácio

Instituto de Telecomunicações, Department of Computer Science,
University of Beira Interior
Rua Marquês d'Ávila e Bolama, 6201-001 Covilhã, Portugal
{fvigario,miguel.neto,diogompaf}@penhas.di.ubi.pt,
{mario,inacio}@di.ubi.pt

Abstract. This paper describes a study for assessing how many free Android games with in-app purchases were susceptible to data manipulation via the backup utility. To perform this study, a data set with more than 800 games available in the *Google Play* store was defined. The backup utility, provided by the Android Operating System (OS), was used to backup the app files into a Personal Computer (PC) in order to find and manipulate sensitive data. In the cases where sensitive data was found, the applications were restored and the games tested to assess if the manipulation was successful and if it could be used to the benefit of the user. The results included show that a significant percentage of the analyzed games save the user and app information in plaintext and do not include mechanisms to detect or prevent data from being modified.

Keywords: Android, Data Manipulation, Integrity, Mobile Operating System, Security, Storage

1 Introduction

In the last few years we have witnessed an significant growth in the use of mobile devices [15]. The massive adoption of these devices led several companies, as Google and Apple, to direct their efforts into the development of mobile Operating Systems (OSs), driven by the needs of users. Along with these OSs, they also provide app stores, (Google Play [12] and Apple Store [3]), which offer point and click access to commercial or free applications to their users. Internetworking is also increasing with the adoption of mobile devices, all contributing to a rich and heterogeneous environment where sensitive data is sometimes flowing in the network, or stored in mobile devices in an insecure manner. This data is a tempting target for malicious users or developers, which try to exploit vulnerabilities to steal it or manipulate it for their own profit.

Similarly, to traditional computer software, games comprise a substantial part of the revenue of this industry, reflected either in number of existing games or in the effort to develop them. There are several business strategies for such applications and, in the case of mobile applications, some developers prefer to provide their games for free, to then offer the users the possibility to expand the game or add functionalities in return for a payment or several micro-payments. These are nowadays known as *in-app purchases.* Many developers choose to ask for a fee during the installation process. Sometimes, the applications come with the additional functionalities already implemented, but blocked. Some programming logic prevents the user from accessing those parts of the application before the purchase. This programming logic may be based on the values of variables, which are stored as app information in the internal storage of the device. As such, it is often assumed that the internal storage is protected (e.g., by the OS) against manipulation by other applications, and difficult to directly access by the user.

In mobile applications, *in-app purchases* are used to remove advertisements included in the free versions of apps, and add additional advanced or premium functionalities. In the case of Android games, which are addressed in this work, these purchases can also be used to unlock additional levels, get extra points, progress faster in the game and obtain hidden items, apart from the aforementioned ones. This mechanism is, thus, an important block in the business model of developers.

This paper describes a study concerning the possibility of accessing and manipulating internal data storage of Android games with *in-app purchase*, using the backup utility provided by the Android OS. The data set used for this analysis consists of more than 800 free games offering *in-app purchases*, which were downloaded from the *Google Play* store. All games of the data set were subject to human analysis after their installation in a non-rooted smartphone and transfer to a Personal Computer (PC). The procedure includes backing up the applications to a PC, searching for interesting data, changing it, and restoring the application back to the smartphone, to then assess if the behavior of the game changes as a result of the manipulation. The results show that a notable part of the analyzed games are susceptible to data manipulation, which can be easily exploited by users. Sometimes, the procedure applied in the scope of this paper may be applied, by typical users, to enjoy blocked functionalities without paying for them. Results clearly show that developers should pay more attention to data integrity and encryption mechanisms in mobile OSs.

This paper is structured as follows. Related works and the motivation underlying this study are included in section 2. Section 3 discusses the data set used in the scope of this work and elaborates on the type of applications used. The method used to perform the analysis is described in detail on section 4. Section 5 discusses the results of the analysis, as well as the number and type of applications that are susceptible to data manipulation using the described method. The main conclusions and some lines of future work are described in section 6.

2 Related Work

Due to the popularity of the Android OS and also to the personal nature of mobile devices, security involving this OS is nowadays a hot research topic. The sub-topic discussed herein is also receiving a lot of attention lately, as shown by the recent works on this area, discussed below. In 2011, the non-for-profit organization Open Web Application Security Project (OWASP) began a project with focus on threats to the mobile environment. At the end of 2014, the threat occupying the first position of the OWASP Top 10 Mobile Risks was *Weak Server Side Controls*, with *Insecure Data Storage and Insufficient Transport Layer Protection* coming up next in the second position [14], which also motivated this work.

C. Håland, in his Masters thesis entitled *An Application Security Assessment of Popular Free Android Applications* [5], includes a study of 20 popular free applications, testing them for the OWASP Top 10 mobile risks. He found several vulnerabilities of this list in the applications, namely *Insecure Data Storage*, *Weak Server Side Controls*, *Insufficient Transport Layer Protection*, etc. The author states that most of the attacks were only possible in rooted devices. For example, he mentions that, with root privileges, the owner of the device can access any file or folder, which comprises a *Insecure Data Storage* problem. He was able to change the value of the coins used in the 4Pics1Word game, without paying for that feature, by simply searching the files storing the status of the application. Wordfeud Free was another application with a similar problem but, in this case, he was able to retrieve the username and password of the Facebook account that was used to login in the application, because the credentials were stored in plaintext in an Extensible Markup Language (XML) file.

C. Xiao, a researcher in Palo Alto Networks, delivered a talk entitled *Insecure Internal Storage in Android* in the Taiwan Conference (HITCON) [6] regarding the subject at hands. He processed a total of 12,351 applications downloaded from Google Play, having concluded that, from these applications, only 556 were not allowing backup by means of the backup utility, and that only other 156 applications were implementing a BackupAgent to protect the data. In other words, approximately 94,2% of the most popular applications allow transferring the package and all the internal storage files to a computer in a packed format. The authorization to backup applications can be set up by adjusting the `android:allowBackup` property to `true` or `false` in the manifest file. His study was focused on the applications with at least 500,000 downloads. The idea of manipulating data from Android apps via the backup utility was already circulating in specialized forums and its genesis is hard to obtain.

In the Masters thesis entitled *Android Application Security with OWASP Mobile Top 10 2014* [13], James King analyzed the FourGoats Android application, in which he identified several types of vulnerabilities, including *Insecure Data Storage*. The version of the application under analysis was using a local database file to store the credentials of the users in plaintext, and an attacker with physical access to the device could thus obtain them, even without root privileges (e.g., using the method described in section 4). In the Masters thesis, the author also describes problems related with *Insufficient Transport Layer Protection*, stating that the majority of the applications selected for analysis were not using encryption to transmit data over the network, which leaves them vulnerable to Man-In-The-Middle (MITM) attacks. In some cases, the credentials of users could also be obtained from traffic sniffing (because they were sent unencrypted). Other vulnerabilities discussed in this study include *Poor Authorization and Authentication*, *Broken Cryptography*, *Improper Session Handling* and *Lack of Binary Protections* related problems.

In [8], Fahl et al. tried to assess how, and to which extent, Secure Sockets Layer (SSL)/Transport Layer Security (TLS) was being used to protect the contents of network communications from Android applications. The inadequate usage or integration of the protocol could be as serious as not using it at all, and both situations lead to MITM related vulnerabilities. In the scope of their work, the authors developed `MalloDroid`, which is a small tool that can be used to find broken SSL certificate validation procedures in Android apps. A total number of 13,500 popular free apps downloaded from Google Play were then analyzed with this tool. They concluded that approximately 8% of the analyzed applications (1,074 apps) were potentially vulnerable to MITM attacks.

This paper is focused on the manipulation of locally stored data (no MITM attacks were performed). Nonetheless, the procedure described herein may be combined with MITM attacks to perhaps obtain an even higher success rate, since it was noticed that some apps were using the network to store values and to detect data modification.

3 Data Set

The analysis described in this paper was performed on a fairly large data set of Android games, which are free to download but have the *in-app purchase* characteristic. Even though simple tasks of this work used automated scripts, most of the analysis was performed manually, meaning that all games were subject to human analysis. The data set was collected between September and November of 2014 and it is consists of 849 games from the 15 different categories defined in *Google Play* for this type of software [12]. Table 1 summarizes the number of games-per-category considered in the analysis. Although some applications without *in-app purchases* were also analyzed in the meantime, they were not considered in the scope of this particular work, because the main objective was to evaluate whether a user could use the method described below to use premium or paid functionalities without purchasing them, i.e., by only modifying internal data.

Prior studies on Android OS security (e.g., [4, 7, 9, 10]) have mainly focused their analysis on the most popular apps available in Google Play (or Android Market, as it was designated previously). This work takes the number of downloads into account in the analysis of the results, but the popularity of the applications was not considered when choosing the games for installation. As such, our approach differs from some of the previously described ones in two ways: (i)

Table 1: Number of analyzed games by category.

	Category														
	Action	Arcade	Puzzle	Casual	Strategy	Sports	Racing	Simulation	Adventure	Role Playing	Card	Word	Family	Trivia	Music
Number of Games	141	138	99	91	91	58	50	39	37	35	34	16	13	6	1

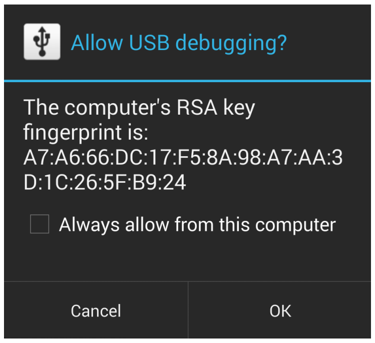
all applications were subject to manual analysis, so that minor details concerning the way that data was stored by different applications was not overlooked; (ii) the study is not limited to popular or to a small set of applications.

In order to give an idea of the popularity of the games comprising the data set, Table 2 shows the number of games in the data set for each of the 11 different download intervals defined by Google Play. A large slice of the data set was in the 1 to 5 billion download interval (approximately 35%), and approximately 71% had more than 1 billion downloads.

Notice that, not limiting the data set to the most popular games also provided a way to later on assess if there was an obvious relation between popularity and the problems related with *Insecure Data Storage*.

Table 2: Number of games in the data set per number of downloads.

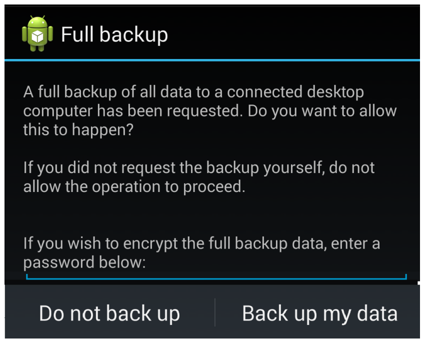
Number of Downloads	**Number of Games-per-Interval**
100 000 000 - 500 000 000	13
50 000 000 - 100 000 000	16
10 000 000 - 50 000 000	111
5 000 000 - 10 000 000	110
1 000 000 - 5 000 000	296
500 000 - 1 000 000	106
100 000 - 500 000	157
50 000 - 100 000	23
10 000 - 50 000	11
5 000 - 10 000	4
1 000 - 5 000	2


Allow USB debugging?

The computer's RSA key fingerprint is: A7:A6:66:DC:17:F5:8A:98:A7:AA:3D:1C:26:5F:B9:24

Always allow from this computer

(a) RSA Key fingerprint.


Full backup

A full backup of all data to a connected desktop computer has been requested. Do you want to allow this to happen?

If you did not request the backup yourself, do not allow the operation to proceed.

If you wish to encrypt the full backup data, enter a password below:

(b) Backup utility dialog.

Fig. 1: Screenshots of important steps of the method used in the scope of this work. (a) Screenshot of the RSA key fingerprint dialog for enabling USB debugging from a given computer; (b) Screenshot of the Backup utility provided by the OS.

4 Method

The method to perform the analysis described in this paper required the usage of a smartphone and a PC with specific software installed. The smartphone was running a non-rooted Android OS (version 4.4.2). Nonetheless, any version of the OS higher than 4.0.0 would suffice for all purposes of this work, since the backup utility (which is critical for the method) was provided natively from that version on [2]. As further explained below, a snapshot of each analyzed game needs to be copied to the PC. This was done via an Universal Serial Bus (USB) cable and the communications were managed by the Android Debug Bridge (ADB) tool [1], which was installed in the computer. The PC was running a Linux based OS also, with the following tools installed: `pax`, `tar`, `OpenSSL`, `dd` and `grep`. Some of these tools (e.g., `OpenSSL` and `grep`) come natively with most of the Linux distributions available, while others are very simple to obtain, e.g., via package managers. These tools were used to handle the package transferred from the smartphone. Typically, `OpenSSL` is used to perform cryptographic tasks but, in this case, it was used to compress and decompress packages resorting to the *zlib* library. `pax` was used to read and write files and copy directory hierarchies. `dd` and `tar` were responsible for converting and extracting files with the `.tar` format, respectively. Finally, `grep` was used for searching patterns in the data files of an application.

After the establishment of the initial setup and installation of the tools described above, a set of steps to explore the data storage problems was applied. The method can be divided into 10 different steps. Each step resorts to a set of commands, which are entered in a traditional shell. It should be emphasized that, apart from the already available and aforementioned tools, no particular secondary application needs to be developed to apply this method, and parts of it can be found online in specialized forums (e.g., [11]), which contributes to the severity of this problem. The method explained below presumes that the game to be analyzed was previously installed in the system. The steps are described with more detail as follows:

1. The first step consists of connecting the mobile device to the computer via USB. The debug mode should be active when the connection is performed (or needs to be activated on the device). The OS normally asks the user to allow the communication in debug mode, exhibiting a dialog with an RSA public key fingerprint, as shown on Figure 1a. The connection needs to be explicitly allowed for the method to work.
2. The app backup to the computer is performed in the second step, by issuing a command similar to the following one in the terminal:

   ```
   adb backup -f data.ab -apk PATH
   ```

   The `PATH` parameter represents the fully qualified path of the game in the smartphone internal storage. This command will trigger the backup utility in the Android OS, illustrated on Figure 1b, that asks the explicit permission to perform the backup operation for that app. An optional password for encrypting the package may be provided.
   Once the command is successfully executed, a compressed and non-encrypted archive (in this case referred to as `data.ab`), which contains a small header with 24 bytes, and the files composing the app, is transferred to the PC.
3. The third step consists of removing the header and converting the `data.ab` into a `tar` file, so as to enable the extraction of the compressed files in the subsequent step. This can be achieved by piping the following commands:

   ```
   dd if=data.ab bs=1 skip=24 | openssl zlib -d > data.tar
   ```

4. The fourth step consists of obtaining an exact snapshot of the names of the files and directory structure inside of the `data.tar` archive with:

   ```
   tar -tf data.tar > data.list
   ```

   This will enable packaging the application back perfectly after file manipulation and before restoring.
5. The fifth step is the one were the `data.tar` archive is decompressed using a command similar to the next one:

   ```
   dd if=data.ab bs=1 skip=24 | openssl zlib -d | tar -xvf -
   ```

6. The sixth step is where one tries to manipulate the data of the application. This work was just focused on trying to change data that could enable accessing paid functionalities or changing the behavior of the game without paying for them. For example, we were interested in changing the number of coins in a game. To achieve this objective, a simple procedure based on human analysis of the files and resorting to the `grep` tool was applied with commands similar to:

```
grep -R "xxx" app/PATH/
```

Several strategies were adopted to try to find out the interesting values. Some were based on thorough (human) analysis of the files. Another one, which proved to be very successful, consisted in playing the game in the smartphone for a limited period of time, leaving it with a given number of coins or in a given game level (which are numbers). Afterwards, those numbers were searched in the data files of the application using `grep` and all occurrences were further analyzed and manipulated. For example, we would legitimately play a game until 1234 coins were generated, to then look for and modify that value in this step. Sometimes, the `sqlite3` tool had to be used to perform the modification, as some apps use sqlite3 to store the data. This stage included the modification of XML, sqlite, JavaScript Object Notation (JSON) and text files.

7. The seventh step begins the *app* restore process. First of all, it is necessary to compress all *app* files in the exact same order they were decompressed using the information saved in the *data.list* file:

```
cat data.list | pax -wd > newdata.tar
```

8. The header that was previously stripped out needs to be properly inserted again so that the file has the right format, using:

```
echo -e "ANDROID BACKUP\n1\n1\nnone" > backup.ab
```

9. The application can then be compressed and concatenated to the end of the `backup.ab` file, which already contains the header:

```
openssl zlib -in newdata.tar >> backup.ab
```

The `backup.ab` is an archive compatible the Android OS.

10. The last step consists of issuing the command to restore the archive to the device, which will trigger a dialog similar to the one in Figure 1b.:

```
adb restore backup.ab
```

Notice that, all previous steps were performed on each one of the games included in the data set. After concluding the last step of the method successfully, the game was once again executed in the Android OS and it was assessed if the data modifications were producing the expected results. Sometimes, this procedure was repeated several times, to minimize the possibility of having overlooked some minor artifact. If the behavior of the game was changed without detection (some games detect modifications by storing values in the network) as a consequence of the manipulation, it was considered vulnerable. The results are discussed below.

5 Discussion and Results

The data set used in the scope of this work is constituted by 849 free (to download) games with the *in-app purchase* characteristic. From those 849, a total of 148 were susceptible to data manipulation performed using the method described in section 4, which corresponds to 17,43% of the tested games. This percentage is significant, taking into account that it refers to cases where developers are dependent of that specific income (the game is free to download, only the add-ons are paid). Also, worth of note is the fact that the method does not depend, in any way, of having the OS rooted or not and that the tools to perform data modification are readily available. Actually, it would be easy to automate the described method and pack the required tools to construct a program for cracking a given game, based on these findings.

The chart in Figure 2 compares the number of games that were found to be susceptible to data manipulation with the total number of games in the data set for each one of the *Google Play* categories. The results suggest that *Trivia* is the most affected category (with 50% of vulnerable games), but it is also one of the ones containing fewer games (only 6). Some of these games work both online and offline, and the paid add-ons are often aids to which the user may resort to answer questions right. In the vulnerable games, it was noticed that the access to such functionalities was controlled by values stored in plaintext files without any integrity mechanism.

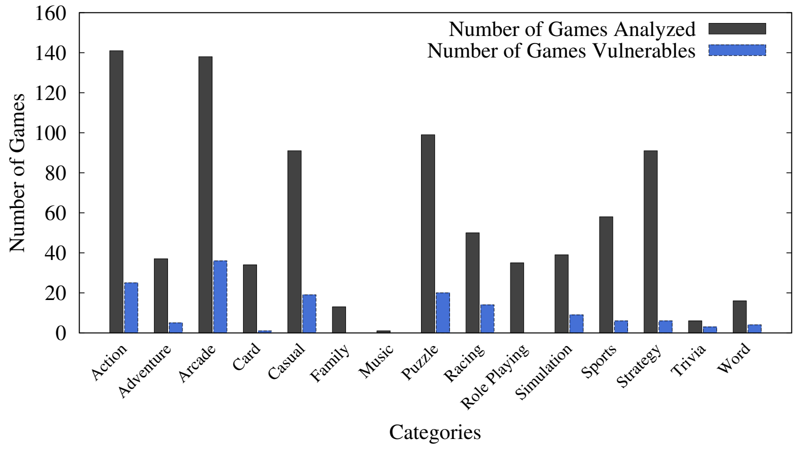


Fig. 2: Total number of games versus the number of games susceptible to data manipulation, per category.

After *Trivia*, the two categories with the most expressive results were *Racing* and *Arcade*, where 28% and 26% of the games were susceptible to data manipulation, respectively. In these types of games, the paid add-on or functionality is also normally comprised by means to have more virtual money or faster ways to progress in levels, namely by buying certain virtual items. The results derive from the fact that the money balance of a game is frequently stored in plaintext XML or text files, or in SQLite databases, without integrity or authentication codes, which are easy to find and modify.

The categories of *Role Playing*, *Family* and *Music* had no vulnerable games. These results are mostly due to the fact that, in this type of games, *in-app purchases* are used to remove advertisements or to, for example, buy the whole game or expansions (new levels, weapons or characters). In such cases, the purchase typically requires downloading new files (or a new version of the game) to the system and, therefore, these files were not previously available for manipulation. Purchases requiring the user to interact with a remote server are inherently less susceptible to data manipulation because, in such cases, it is not about changing the flow or status information of the application. *Role Playing* games usually have a higher longevity, and they are updated more times than, e.g., *Arcade* games. Some updates address previously known issues, namely the ones related with *Insecure Data Storage* when *crack* tools leak into the Internet. Probably, the effort in implementing security features in these categories is larger.

The *Cards* and *Strategy* categories had 2.9% and 6.5% of the games susceptible to data manipulation, respectively. These low values are related with the fact that both types of games are typically played online, which means that application information is either stored remotely or checked frequently against the last known (or normal) snapshot of such values. Some games allowed changing some app related values in the PC, but they were then restored when the connection was again established.

In order to assess if there was a relation between the popularity of a game and its susceptibility to data manipulation, an analysis similar to the previous one was conducted for the same data set segregated by number of downloads. The chart in Figure 3 summarizes this part of the work.

The most vulnerable games are the ones with the number of downloads in the ranges 5000–10000 and 50000–100000, though the first interval only contained 4 games (of which 2 were vulnerable to the method applied herein). In the second interval, 7 games were vulnerable, corresponding to approximately 30% of the total number of games in that range. The results suggest that the number of vulnerable games is not dependent of their popularity, even though no vulnerable games were found in the range 50000000–100000000 (there were only 16 games in this interval). In the range with most of the games (1000000–5000000), 15.4% were vulnerable to data manipulation, which corresponds to 46 games in a total of 296.

Focusing only on the vulnerable games, Figure 4 emphasizes the type of files used to sale application data in the internal storage of the Android OS. As shown in the pie chart, 76% of the vulnerable games resort to XML files, while 9% use SQLite databases and 14% use `.txt` or JSON files. This distribution was expected since XML comprises a simple and standard format for storing data and, in Android, one of the suggested storage options for saving application

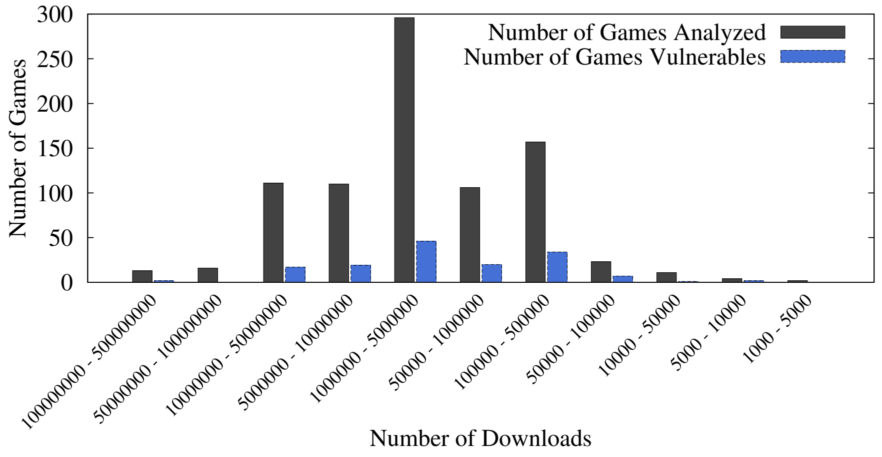


Fig. 3: Total number of games versus the number of games susceptible to data manipulation, segregated by popularity.

related data is known as *Shared Preferences*, which is used to save values from primitive JAVA types in XML format. Nonetheless, these results also show that manipulation is possible in a variety of formats. Data stored within SQLite databases is also stored in ASCII, except if encoded using application logic, which also enables one to easily find patterns in the files.

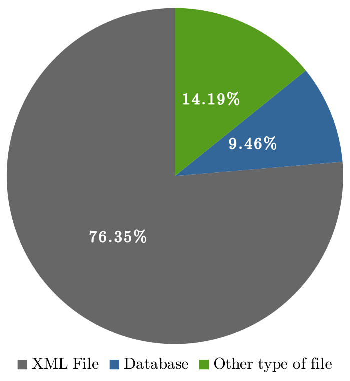


Fig. 4: Type of file used to save application data in Android internal storage in vulnerable games.

Finally, it should be mentioned that only 21 games, corresponding to approximately 2.5% of the data set, had the property `android:allowBackup` set to `false` in the `AndroidManifest.xml`. By default, this property is set to `true`, which means that, if the developer does not explicitly adjust it to `false`, the game will be inherently more prone to the method described herein. The possibility to backup an application is a commodity utility for the user and it makes sense to exist. The problem is that it can be easily exploited with different intentions. Setting the aforementioned property to `false` will solve the problem in non-rooted OSs, but legitimate users may miss the functionality. Apart from that, such setting would not solve the issue in rooted OSs.

Due to the reasons mention in the previous paragraph, integrating data integrity mechanisms into the application may comprise the best means to address this issue or, at least, make the exploitation more difficult. For example, developers may use hash functions, Message Authentication Code (MAC) or digital signature algorithms to calculate and store digests, MACs or signatures for the files where the sensitive data is stored (databases, text or implementation files, etc.). Encrypting the files will have a similar effect. Nonetheless, this imposes some additional computational burden, since the application needs to calculate the aforementioned codes when the contents of the files change and verify them at each execution. If encryption is used, the application needs to decrypt and encrypt the data before accessing or storing it in the files, respectively. Since the cryptographic secrets, used in these mechanisms, need to be available for the application, a specially motivated attacker may still try to manipulate the data by finding the keys and the data integrity mechanisms. If the cryptographic secrets are transferred with the backup, the manipulation is harder to achieve, but still within reach. As such, this topic needs more attention.

Curiously, 3.2% of the games in the data set (corresponding to 27 games) were publicized with the *in-app purchase* characteristic in *Google Play*, but had really nothing to buy in the application. In another 3.9% of the data set (i.e., 27 games), the *in-app purchase* was to remove advertisements or to buy their respective full version, which involves downloading additional files and, as such, they are not vulnerable to data manipulation.

6 Conclusions and Future Work

This paper is focused on the *Insecure Data Storage* threat defined by OWASP for mobile devices. It elaborates on the specific problem of manipulating application data from Android games to get free access to paid add-ons or functionalities. Herein, it was discussed that the backup and restore functionalities, provided natively with recent versions of the Android OS, could be used to transfer games to and from a PC, in which they could be tampered to obtain access to the aforementioned add-ons or functionalities. This procedure does not require the system to be rooted, only the users consent (for allowing the USB connection and for transferring the specific application). This procedure is already used by some tools for cheating some games since it can be easily automated for a specific application.

In order to quantify the problem, a total of 849 games offering *in-app purchases* were downloaded, from Google Play, and then manually analyzed using a method based on searching patterns on the data transferred with the application during the backup. An expressive number of 148 games were found vulnerable to this method, meaning that it was possible to change the application flow or status by searching for ASCII patterns on XML, JSON, `txt` or SQLite files. Results suggest that the vulnerability is not related with the popularity of the games and we argue that disabling the backup feature for a game should not be seen as a preventive measure against this problem, though it would certainly contribute for its attenuation. The possibility to backup the applications constitutes a commodity utility and it should not be disabled. Data integrity and encryption mechanisms comprise the best mechanisms to prevent *Insecure Data Storage* problems, like the ones discussed herein.

A possible line of future work consist on performing a similar assessment for other mobile OSs. For example, *iOS* provides also a means to transfer applications to a PC, e.g., for the cases when the user wants to switch phones.

Another line of work consists of expanding the data set to other types of applications and broaden the scope of the analysis. During the study described herein, a few other (non-game) applications were analyzed and it was found that many of them were saving user related data in plaintext files. Worst than that, some applications were enabling the access to others users online profiles via data manipulation, because they were storing other users information locally. A more detailed analysis is nonetheless required.

Acknowledgements Authors acknowledge the financial support from EyeSee, Lda. This work was performed in the scope of the R&D Unit 50008, financed by the applicable financial framework (FCT/MEC through national funds and when applicable co-funded by FEDER – PT2020 partnership agreement).

References

1. Android Developers: Android Debug Bridge (2014), `http://developer.android.com/tools/help/adb.html`, accessed Dec. 2014
2. Android Developers: Dashboards — Android Developers (2014), `https://developer.android.com/about/dashboards/index.html`, accessed Dec. 2014
3. Apple: Official Apple Store (20), `http://store.apple.com/us`, accessed Jan. 2015
4. Barrera, D., Kayacik, H.G., van Oorschot, P.C., Somayaji, A.: A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 73–84. CCS '10, ACM, New York, NY, USA (2010), `http://doi.acm.org/10.1145/1866307.1866317`
5. Christian Håland: An Application Security Assessment of Popular Free Android Applications. Master's thesis, Norwegian University of Science and Technology (2013)
6. Claud Xiao and Ryan Olson: Insecure Internal Storage in Android - Palo Alto Networks BlogPalo Alto Networks Blog (2014), `http://researchcenter.paloaltonetworks.com/2014/08/insecure-internal-storage-android/`, accessed Dec. 2014

7. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 235–245. CCS '09, ACM, New York, NY, USA (2009), `http://doi.acm.org/10.1145/1653662.1653691`
8. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: An analysis of android ssl (in)security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 50–61. CCS '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2382196.2382205`
9. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 627–638. CCS '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/2046707.2046779`
10. Felt, A.P., Greenwood, K., Wagner, D.: The effectiveness of application permissions. In: Proceedings of the 2Nd USENIX Conference on Web Application Development. pp. 7–7. WebApps'11, USENIX Association, Berkeley, CA, USA (2011), `http://dl.acm.org/citation.cfm?id=2002168.2002175`
11. Forums, X.: [GUIDE] How to extract, create or edit android adb backups — Android Development and Hacking — XDA Forums (20), `http://forum.xda-developers.com/showthread.php?t=2011811`, accessed Jan. 2015
12. Google: Google Play (20), `https://play.google.com/store`, accessed Dec. 2014
13. James King: Android Application Security with OWASP Mobile Top 10 2014. Master's thesis, Luleå University of Technology (2014)
14. OWASP: Projects/OWASP Mobile Security Project - Top Ten Mobile Risks - OWASP (2014), `https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks`, accessed Nov. 2014
15. Pieterse, H., Olivier, M.: Android botnets on the rise: Trends and characteristics. In: Information Security for South Africa (ISSA), 2012. pp. 1–5 (Aug 2012)