Android Password Managers and Vault Applications: An Investigation on Data Remanence in Main Memory

Peter Sabev and Milen Petrov

Faculty of Mathematics and Informatics,
Sofia University "St. Kliment Ohridski",
5 James Bourchier Blvd., 1164, Sofia, Bulgaria,

{petar.sabev, milenp}@fmi.uni-sofia.bg,

Abstract. An application defining itself as password management / secure vault software should meet a number of security requirements in order to provide an adequate data protection. The data entrusted to such application is its most valuable asset that the application is responsible to protect. To check to what degree the popular Android password managers / vault applications are protecting their data loaded into main memory, we analyze the runtime behavior of two of them from security perspective. More specifically, we suggest a main memory inspection procedure helpful for evaluation of the extent of how secure a given software application is in regards to prevention of secrets exposure. Then we apply this procedure to conduct a digital investigation in a forensically sound manner by focusing on data remanence in main memory. In conclusion, we summarize the investigation results by showing what part of the data entrusted to applications examined remains exposed in clear text in main memory.

Keywords: Android Password Managers, Vault Applications, Main Memory Investigation, Data Remanence, Security Analysis, Garbage Collection.

CEUR-WS.org/Vol-2933/paper31.pdf

1 Introduction

An application defining itself as a password management / secure vault software should be built upon a strong security architecture, efficient security mechanisms and strong defense techniques in order to provide a secure environment. It must not only allowing for secure storage, processing and management of sensitive data, but also allowing for a much higher level of security compared to a general-purpose software application. That means, a password manager (PM) / vault application (VA) should meet a number of security requirements in order to provide an adequate data protection.

Large number of Android applications defining themselves as password management / secure vault software have been developed and published in the official Google Play Store [1]. Some of these applications are used by millions of

Copyright © 2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

users, which are relying on them to protect their data [1]. Given this and based on the experts security advice recommending the use of PMs [2]which may be unrealistic, time consuming, or not really worth the effort. To improve the security advice, our community must find out what practices people use and what recommendations, if messaged well, are likely to bring the highest benefit while being realistic to ask of people. In this paper, we present the results of a study which aims to identify which practices people do that they consider most important at protecting their security online. We compare self-reported security practices of non-experts to those of security experts (i.e., participants who reported having five or more years of experience working in computer security, we can assume that the PMs / VAs will become more and more widely used. Based on this and also based on the fact that PMs / VAs are meant to protect user's most valuable data, it can be concluded that it is especially important for the user to have as full as possible knowledge about the extent of how secure these applications are. Related to this, the developers of this type of software are trying to provide an informative technical documentation, including security white papers and other types of documents revealing important details about the security of these applications. However, it is common for these documentations to only provide a high-level overview of the security architecture and the algorithms in use, but with missed important technical information explaining how exactly the security architecture is implemented in order to be provided an adequate data protection. More specifically, often in these documentations are missed the security requirements upon which the security architecture is built as well as the important technical details explaining the security measures taken by the actual implementation for providing an adequate data protection. Because of this, we find important to be evaluated the real security of the popular Android PMs / VAs when they are placed in equal conditions.

In our previous work [3], we have defined the fundamental security requirement and the sub requirements based on it that are essential for building strong security architecture. They are summarized as follows: an application defining itself as an application build upon a strong security architecture is expected to provide a secure data storage, processing, and management environment ensuring at least the integrity and authenticity of both public and sensitive data and the confidentiality of sensitive data entrusted to the application. In this work, we use this fundamental security requirement, as well as the sub requirements and the definitions from our previous work [3] as a basis to conduct a digital investigation in a forensically sound manner by focusing on data remanence in main memory. Our goal here is to evaluate to what degree the popular Android PMs / VAs are protecting their data in main memory by using reverse engineering tools and techniques, including runtime analysis and debugging techniques.

2 Background

This section provides background on Android Software Development, memory management concepts and guidelines for building secure software.

Android is an open source operating system. It is based on a modified version of the Linux kernel and other open source software. Android provides a rich set of tools, techniques and development kits (DK) for developing, debugging and testing software applications. The two officially supported DK for Android applications development are Android Software Development Kit (Android SDK) and Android Native Development Kit (Android NDK). [4][5]

Android SDK includes a set of development tools allowing application developers to develop Android applications in one or more high-level programming languages. Among them Java and Kotlin are the two officially supported programming languages. Android application written in these languages are typically compiled to bytecode and then executed into a process virtual machine (VM) providing an application runtime environment. The current application runtime environment used by the Android operating system is Android Runtime (ART), which is a replacement of Dalvik (the process VM originally used by Android). [6] More specifically, ART is a managed runtime environment having a few different garbage collection (GC) plans that consist of running different garbage collectors. The two most notable among them are Concurrent Mark and Sweep (CMS) and Concurrent Copying (CC) that is the default GC plan starting with Android 8 (Oreo).

Android NDK includes a set of development tools and platform libraries allowing application developers to develop Android applications in C and C++ programming languages. It is typically used when it is needed to be squeezed extra performance out of a device in order to be achieved low latency or to be executed computationally intensive tasks [5]. Without GC, such as programming C and C++ languages, application developers have to manage memory usage manually, meaning that it is a responsibility of the developers to consider object lifetimes, explicitly allocate and deallocate memory. More specifically, the developers are provided a fine control over object lifetimes allowing them to implement strategies for secure memory management minimizing the chance for sensitive data leakage, but the developers are also provided the entire responsibility to ensure memory safety and memory leaks prevention.

An Android application may be built in such a way that one part of the application is written in C / C++ code and the other part of the application is written in Java / Kotlin code. The C / C++ code may be compiled into a native library and packaged together with the bytecode produced by the Java / Kotlin compilation. That means, thanks to the Java Native Interface (JNI), one part of the application will be able to run in a managed runtime environment where GC is available and the other part of the application will run without GC.

An application such as a PM / VA is meant to not only securely store sensitive data like passwords, credit card numbers, identity card numbers, etc., but it is also meant to securely process and manage the data entrusted to it. In order to do this effectively, the PM / VA should keep sensitive data in memory for as short a time as possible and should take care to ensure that data never gets written as a clear text to device's flash memory or another type of long-term memory, such as an external memory card. Clearing sensitive data promptly after use together with memory locking techniques on a per-page basis and core dumps disabling techniques are widely accepted security recommendations for strengthening the overall security [7].

3 Applications selection

A lot of Android PMs / VAs are claiming to be secure. We have reviewed and considered for selection a number of them. Part of these applications were already analyzed by independent security researchers from security perspective. Based on their results, we have decided to exclude the following three applications, because we consider them as totally insecure [9]:

- **Keepsafe (package name: com.kii.safe)** – This application has more than 50 million installs according to the Google Play Store [8]. But work [9] shows that while Keepsafe 7.3.1 is storing encrypted picture and videos files, it also stores the Master Password in clear text in the value of tag master-password of XML file **com.kii.safe_preferences.xml** in the application's **shared_prefs** folder and a 32 byte Key that can be found also in **com.kii.safe.secmanager.xml** file in **shared_prefs**.
- **AppLock (package name: com.domobile.applock)** – This application claims to be "#1 app lock in the world. Launched in 2012 and trusted by 300 million users in over 150 countries" [10]. However, it is:
  - storing unencrypted pictures and videos in separate directories under **/sdcard/.MySecurityData/dont_remove**; [9]
  - storing Base64 encoded Secure Hash Algorithm 1 (SHA1) hash value of the gesture for the pattern lock in the XML file **com.domobile.applock_preferences.xml** in the application's **shared_prefs** directory; [9]
  - known to be vulnerable to swap attack (to reset the gesture) and rainbow table attack (to crack the gesture); [9]
- **Calculator Vault (package name: com.calculator.vault)** – This application has more than 5 million installs according to the Google Play Store [11]. But work [9] shows that Calculator Vault 8.5 disguised itself as a calculator on the system and vault function is activated only when the correct password is provided, but it stores unencrypted photos and vid-

eos under the directory /data/data/com.calculator.vault/files/locker1762, which is only protected by Android's built-in application isolation. In addition, the password is stored in clear text as a value of the tag mpass, part of the XML file com.calculator.vault_preferences.xml, located in shared_prefs directory.

Among the other applications that we have reviewed, we have selected the following two applications to serve as representatives of the close sourced applications and the open source applications respectively, which are defining themselves as PMs / VAs:

- **Keeper (package name: com.callpod.android_apps.keeper)** – At the time of the writing, this application has rating ~ 4.6 with more than 10 million installs according to the Google Play Store and based on the description there: "Keeper is the Most Secure Password Manager in the Industry" [12].
- **Bitwarden (package name: com.x8bit.bitwarden)** – At the time of the writing, this application has rating ~ 4.6 with more than 500 000 installs according to the Google Play Store and based on description there "Bitwarden is focused on open source software. The source code for Bitwarden is hosted on GitHub and everyone is free to review, audit, and contribute to the Bitwarden codebase" [13].

For the above applications, we have analyzed the findings and results [9], [14] for them from several publicly available security researches. In these findings and results [9], [14], we were unable to find strong evidences proving that the above two applications are relying on security through obscurity as a main method of providing security or are using broken or weak hashing / encryption / protection approaches or techniques as a main method to protect the data entrusted to them. This motivates our choice to select exactly the above applications, especially given the fact that the goal of our current work is focused on applications striving to provide as much security as possible.

4 Main memory inspection procedure

In this section, we suggest a Main Memory Inspection Procedure helpful for evaluation of the extent of how secure a given PM / VA is in regards to prevention of secrets exposure in main memory. It consists of six main phases visually presented in Fig. 1.

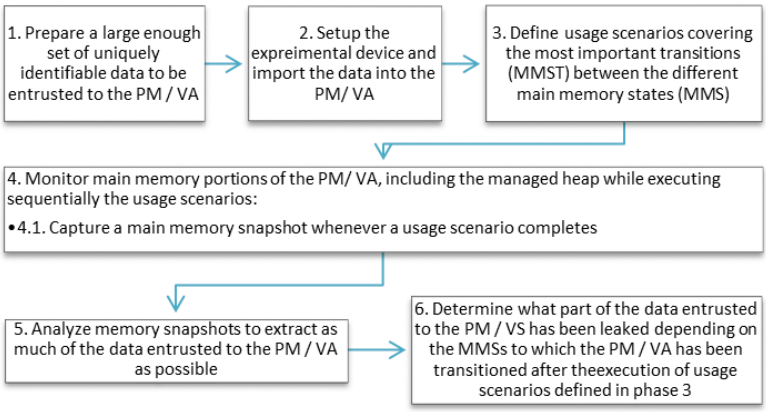

Fig. 1. Inspection procedure of the main memory.

To support the suggested main memory inspection procedure, we use the execution states defined in our previous work [3] as a foundation to define the following three Main Memory States of a PM / VA. These are essential for both the correctness and the relevance of the usage scenarios definitions (phase 3 that is shown on Fig. 1), as well as for the accuracy of the results of the data leakage analysis process (phase 6 that is shown on Fig. 1):

MMS1. DESTROYED – PM / VA, is in this state when there is no running instance of the application in the operating memory, the application is fully stopped, all processes and services owned by the application are fully destroyed and there are no their previous instances (if any were existed before transition to this state) left running in the operating memory. An example of a PM / VA that is considered to be in MMS1.DESTROYED is an application that is explicitly killed by the force stop feature and all processes and services owned by the application are destroyed. Another example is a PM / VA that is implicitly killed by the operation system itself due to full system restart or if the device is completely turned off and then turned on.

MMS2. RUNNING_UNTRUSTED – PM / VA is in this state when there is at least one running instance of the application (a running process and / or a service owned by the PM / VA) in the operating memory, there is no active trusted user session established and the access to the data entrusted to the PM / VA is strictly forbidden. An example of a PM / VA that is considered to be in MMS2. RUNNING_UNTRUSTED is a running application that is started for the first time right after it is installed on the device. Another example is a running PM / VA in which an explicit (lock button is clicked, log out button is clicked, etc.) or implicit (auto lock security feature is triggered due to user inactivity, switch accounts button is clicked, etc.) transition from MMS3.RUNNING_TRUSTED to

MMS2.RUNNING_UNTRUSTED is completed successfully. As result of that, the previously established trusted user session is destroyed and the access to the data entrusted to the PM / VA is strictly forbidden.

MMS3. RUNNING_TRUSTED – PM / VA is in this state when there is at least one running instance of the PM / VA (a running process and / or a service owned by the PM / VA) in the operating memory, there is a trusted user session established and the user is provided a full access to the data entrusted to the PM / VA. A PM / VA is allowed to complete a transition from MMS2.RUNNING_UNTRUSTED to MMS3.RUNNING_TRUSTED, only after user's identity is successfully verified by the authentication mechanism of the application – for example when the user is prompted to provide a correct combination of Email and Master Password. The transition from MMS2.RUNNING_UNTRUSTED to MMS3.RUNNING_TRUSTED is completed successfully only if user's identity is successfully verified and a trusted user session is established.

We denote the above three Main Memory States as **MMS = {MMS1.DESTROYED, MMS2.RUNNING_UNTRUSTED, MS3.RUNNING_TRUSTED}**.

In addition to them, we also define the following two terms:

Main Memory State Transition (MMST) – For a PM / VA, it is a transition from one MMS to another MMS caused by an interaction between the user and the PM / VA, the operating system and the PM / VA or the PM / VA itself based on its internal logic. A MMST from a **MainMemoryStateX** to a **MainMemoryStateY** is denoted as follows: **MainMemoryStateX => MainMemoryStateY, where MainMemoryStateX ∈ MMS and MainMemoryStateY ∈ MMS**

Sequence of Main Memory State Transitions (SMMST) – For a PM / VA, it is a numbered sequence of one or more MMST representing the order in which the transitions had occurred. SMMST representing n consecutive MMST is denoted as follows:

1. MainMemoryStateA => MainMemoryStateB
2. MainMemoryStateB => MainMemoryStateD
…
n. MainMemoryStateD => MainMemoryStateN,

where **n** is a positive natural number, **MainMemoryStateA ∈ MMS**, **MainMemoryStateB ∈ MMS**, **MainMemoryStateD ∈ MMS**, … , **MainMemoryStateN ∈ MMS**.

The focus of the suggested main memory inspection is primarily to evaluate the protections and security measures taken by the inspected PM / VA itself. Because of this, it suggests to be monitored only the portions of main memory owned by the processes of the inspected PM / VA, including the managed heap as part of phase 4 (shown on Fig. 1). It excludes from the inspection the memory portions of the other user-mode processes, the kernel, drivers, caches, CPU / GPU

specifics, etc. However, we find important to note that all of these may be used by a potential attacker to break the confidentiality of the data entrusted to the PM / VA.

5 Device setup and data preparation

To complete phases 1 and 2 from inspection procedure of the main memory (shown on Fig. 1) and to prepare for the digital investigation, we performed the following actions:

1. A new Android virtual device emulating Pixel 3 device was created. More details about this device and the exact versions of software installed are provided in Table 1.

Table 1. Emulated test device details.

Device Emulated	Android Version	Build Number	CPU/ABI	Google Play Version	Is Rooted
Pixel 3	Android 9.0 (Google APIs), API 28	sdk_gphone_x86_ arm-userdebug 9 PSR1.180720.117 5875966 dev-keys	x86	19.3.36-all [0] [PR] 302041649	Yes

2. The latest versions of the applications from representative sample were downloaded from Google Play Store and then installed on the emulated test device. More details about their exact versions are provided in Table 2.

Table 2. Representative sample applications.

Application	Package Name	Version
Keeper	com.callpod.android_apps.keeper	14.5.31.1
Bitwarden	com.x8bit.bitwarden	2.3.1

3. Two free accounts were created. More details are provided in Table 3.

Table 3. Accounts details.

Application	Email	Master Password	Free Premium
Keeper	secure.vault.app.test@gmail.com	analy1@te$t@db500	Yes
Bitwarden	secure.vault.app.test@gmail.com	analy1@te$t@db500	No

4. A large set of data to be entrusted to the applications from representative sample was prepared and imported. It consists of the following:
   a. **Credentials_f0x1f2x5.csv** – A CSV file containing 10 000 text-only records representing synthetic login data for external sites (the

data was generated by a special tool written in Kotlin). More details are provided on Fig. 2.

b. Two records containing file attachments. More details are provided in Table 4. (**Note: These records were not imported in Bitwarden, because its file attachments feature requires paid account**).

Table 4. Records containing file attachments.

Name	URL	File Attachment
TextSingleLineFile_x2f5.txt	https://WebSingleLineFile_x2f5.txt	SingleLineFile_x2f5.txt (text file containing a line of 1500000 alpha-numeric characters)
ImageBankCard_b0x2b.jpg	https://WebImageBankCard_b0x2b.jpg	BankCard_b0x2b.jpg (binary jpg file containing a photo of test bank card)

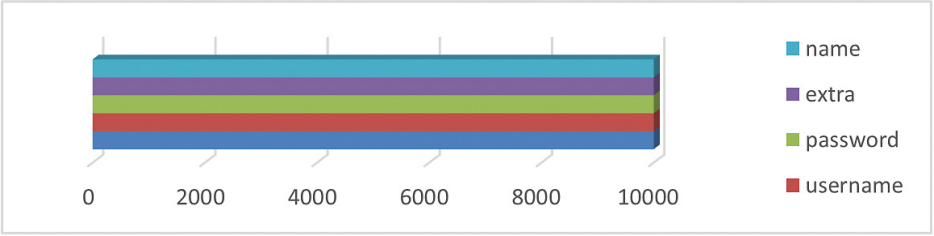

Fig. 2. Credentials_f0x1f2x5.csv – Number of values grouped by field names.

6 Usage scenarios definition

To complete phase 3 from inspection procedure of the main memory (shown on Fig. 1) and to prepare for the digital investigation, we defined a number of usage scenarios (US) covering multiple MMST, but in this study, we will focus on the following ones:

US 2. Log in, and then open the default view of application's Vault screen

Sequence of Main Memory State Transitions:

1. MMS1.DESTROYED => MMS2.RUNNING_UNTRUSTED
2. MMS2.RUNNING_UNTRUSTED => MMS3.RUNNING_TRUSTED

Prerequisite:

PR1; PR2; PR4; PR5; PR6; PR8; PR11;

Steps:

1. Enter the credentials, provided in Table 3.
2. Click the Log In button.
3. Wait the login process to complete.
4. Open the default view of application's Vault screen (if not opened by default).

5. Wait until the screen is loaded.
6. Observe the result.

US 4. Open an existing record

*Sequence of Main Memory State Transitions***:**

1. MMS3.RUNNING_TRUSTED => MMS3.RUNNING_TRUSTED

Prerequisite:

PR1; PR2; PR4; PR5; PR7; PR9; PR11;

Steps:

1. Select an existing record at random.
1. Open the selected record by tapping its name.
2. Wait until the record is loaded and displayed on the screen.
3. Observe the result.

US 6. Open the text file attached to an existing record

*Sequence of Main Memory State Transitions***:**

1. MMS3.RUNNING_TRUSTED => MMS3.RUNNING_TRUSTED

Prerequisite:

PR1; PR2; PR4; PR5; PR7; PR9; PR11;

Steps:

1. Open the TextSingleLineFile_x2f5.txt record by tapping its name.
2. Wait until the record is displayed on the screen.
3. Observe the result.
4. Open the attached file by tapping its name.
5. When prompted by the application to select an external application to open the file, choose the default HTML Viewer.
6. Wait until the file is loaded and displayed on the screen.
7. Observe the result.

Note: **Not applicable to Bitwarden, because file attachments feature requires paid account.**

US 9. Log out

Sequence of Main Memory State Transitions:

1. MMS3.RUNNING_TRUSTED => MMS2.RUNNING_UNTRUSTED

Prerequisite:

PR1; PR2; PR4; PR5; PR7; PR9; PR11;

Steps:

1. Log out.
2. Observe the result.

To support the usage scenarios above, we also defined the following prerequisites:

PR1. The device described in Table 1 is used.

PR2. The PMs / VAs listed in Table 2 are already installed on the device and their shortcuts are already present on the Home screen.

PR4. The accounts with the details provided in Table 3 are already created.
PR5. The dataset described in Section 5 is already imported.
PR6. The user is logged out.
PR7. The user is already logged in and some time is passed since then.
PR8. The current screen is the application's Login screen.
PR9. The current screen is application's Vault screen (all records view).
PR11. Autofill feature in application settings is enabled; application's Autofill service in Accessibility services is selected and turned on.

7 Findings and results

In this section, we summarize the digital investigation that we have conducted, our findings and the results from the investigation. As part of the preparation for the digital investigation, we performed the actions needed to ensure that the phases 1 to 3 from inspection procedure of the main memory (shown on Fig. 1) are fully completed. After that, we moved to the actions needed for phase 4 (shown on Fig. 1). We sequentially executed the US defined in Section 6 for each of the representative applications. During the execution of each US's steps we were monitoring the main memory portions of each application, including its managed heap to analyze the runtime behavior of each application. As part of this process we were capturing PM'S / VA's main memory contents to make it available for offline analysis. More specifically, we were creating a separate memory dump after the execution of each scenario's last step and after ensuring that each US is fully completed (the full memory dumps can be requested on our email).

After we ensured that phase 4 (shown on Fig. 1) is fully completed, we moved to phases 5 and 6 of the Main memory inspection procedure (shown on Fig. 1). More specifically, we tried to extract as much of the data entrusted to each of the reprehensive PMs / VAs as possible by analyzing the memory dumps created in a special tool written in Kotlin (the raw result produced by the tool can be requested on our email). Then, we summarized our findings.

Fig. 3 is showing the total count of Master Password copies found in clear text in the memory dumps. Fig. 4 and Fig. 5 are showing the total count of values (values of user's data entrusted to the PM /VA) found in clear text in the memory dumps. We grouped the results by the respective memory dumps, applications and field values as follows:

- **Keeper's memory dumps**:

  Keeper_US2_MD2 – A memory dump created right after the completion of US 2. Based on what is shown on Fig. 3, we can conclude that Keeper has failed to timely (promptly after use) scrub / clear all copies of the Master Password that are present in Keeper's process main memory.

The results shown on Fig. 4 are also confirming this. As it can be seen on the figure, Keeper is loading all entrusted to it data in main memory right after the transition in MMS3.RUNNING_TRUSTED (all 50 000 values are present).

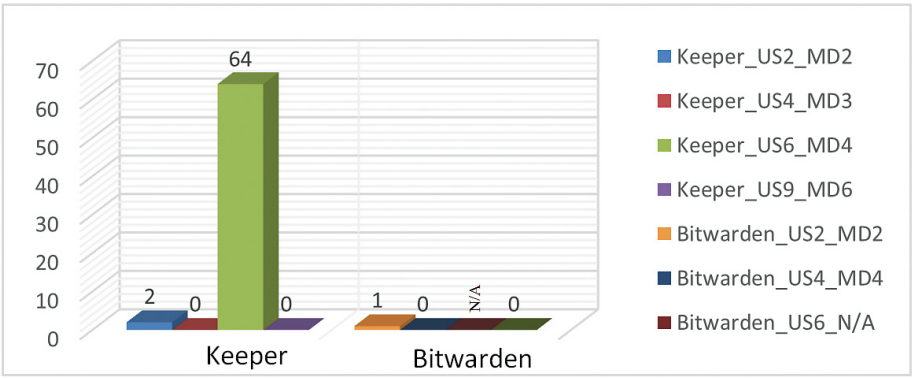

Fig. 3. Count of Master Password copies found in clear text in the memory dumps.

Keeper_US4_MD3 – A memory dump created right after the completion of US 4. Based on what is shown on Fig. 3, we can conclude that Keeper has performed some actions to scrub / clear all copies of the Master Password that are present in main memory, but it is also possible for this to be caused by an implicit data overwriting due to memory exhaustion, garbage collection, etc. About the data entrusted to Keeper, Fig. 4 shows that the results here are the same as the previous results.

Keeper_US6_MD4 – A memory dump created right after the completion of US 6. As shown on Fig. 3, 64 copies of Keeper's Master Password remained exposed in cleartext in main memory even after the full completion of **US 6**. This is the highest count of Master Password copies so far. About the data entrusted to Keeper, Fig. 4 shows that the results here are the same as the previous results.

Keeper_US9_MD6 – A memory dump created right after the completion of US 9. As shown on Fig. 3, Keeper's Master Password is not present in main memory. About the data entrusted to Keeper, Fig. 4 shows that the results here are the same as the previous results.

- **Bitwarden's memory dumps**:

Bitwarden_US2_MD2 – A memory dump created right after the completion of US 2. As shown on Fig. 3, Bitwarden's Master Password is exposed in main memory similarly to Keeper's result, but this time there is only one copy of it. About the data entrusted to Bitwarden, Fig. 5 shows that Bitwarden is holding only 198 values loaded in main memory.

Bitwarden_US4_MD4 – A memory dump created right after the completion of US 4. As shown on Fig. 3, Bitwarden's Master Password is not present in main memory. About the data entrusted to Bitwarden, Fig. 5 shows that the results here are similar to the previous ones.

Bitwarden_US6_N/A – A memory dump created right after the completion of US 6 is not applicable (N/A) to Bitwarden, because file attachments feature requires paid account ("File Attachments are available for Premium users, including members of Paid Organizations (Families, Teams, or Enterprise)." [15]).

Bitwarden_US9_MD6 – A memory dump created right after the completion of US 9. As shown on Fig. 3, Bitwarden's Master Password is not present in main memory. About the data entrusted to Bitwarden, Fig. 5 shows that the results here are similar to the previous ones.

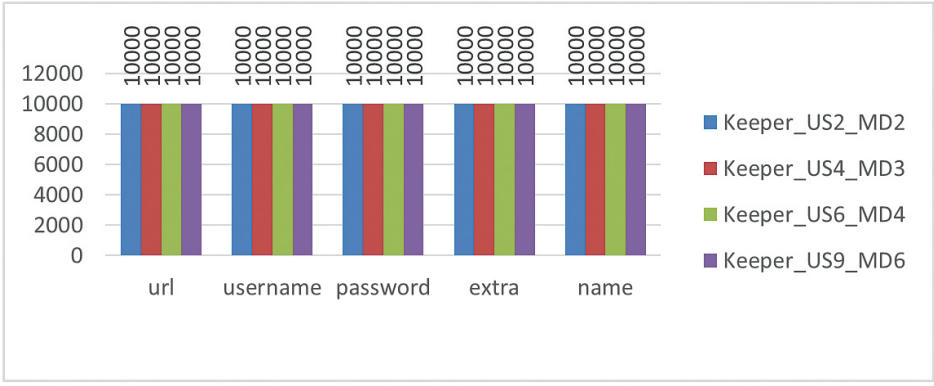

Fig. 4. Count of values found in cleartext in the memory dumps of Keeper.

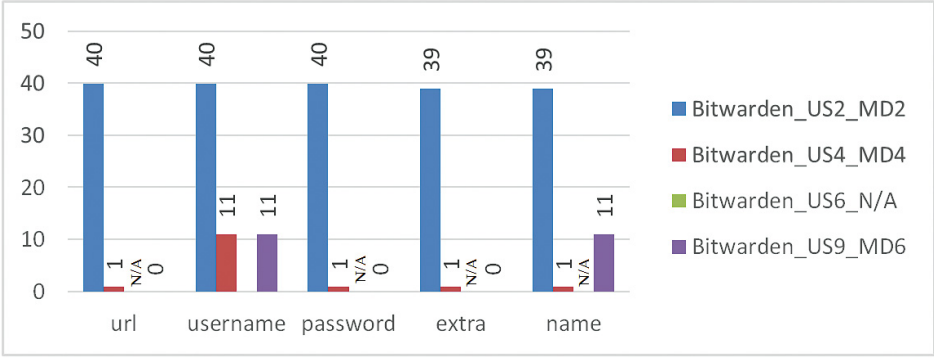

Fig. 5. Count of values found in cleartext in the memory dumps of Bitwarden.

8 Conclusion and future work

Based on the results above, we can conclude that Keeper is providing a big enough time window for a potential attacker to steal not only user's Master Password, but also the data entrusted to Keeper and this is achievable by a simple inspection of the main memory portions of Keeper's process. Moreover, Keeper is holding all entrusted to it data in cleartext in main memory for the whole lifetime of its process and the services on which it depends, including the case when the user is explicitly logged out and the application is in MMS2. RUNNING_UNTRUSTED (transition is fully completed).

In contrast to Keeper, Bitwarden's Master Password was only found in Bitwarden_MD2. In addition to this, the results have showed that Bitwarden is loading a very small amount of the data entrusted to it in clear text in main memory. Based on this, we can conclude that Bitwarden is taking a much better security measures to limit data exposure in main memory compared to Keeper. However, Bitwarden similarly to Keeper allowed some of the values to remain in main memory even after the transition to MMS2. RUNNING_UNTRUSTED was completed.

Future research should investigate on the development of better approaches and security mechanisms for limitation of data exposure in main memory by taking into consideration the specifics of Android's managed runtime environment.

9 Acknowledgments

Research presented in this paper was supported by the FNI-SU-80-10-152/05.04.2021, FNI project of Sofia University "St. Kliment Ohridski" (Bulgaria) "Challenges of developing advanced software systems and tools for big data in cloud environment (DB2BD-4)".

References

1. Password Manager - Android Apps on Google Play, https://play.google.com/store/search?q=password+manager&c=apps, last accessed 2016/04/19.
2. Ion I., R. Reeder, S. Consolvo: '...No one can hack my mind': Comparing expert and non-expert security practices. In: SOUPS 2015 - Proceedings of the 11th Symposium on Usable Privacy and Security (2019).
3. Kaloyanova K., Ed.: Requirements for Securing User Data in Android Applicationsat at Software Level in Information Security in Education and Practice, pp. 114–130. Cambridge Scholars Publishing (2021).
4. Download Android Studio and SDK tools | Android Developers, https://developer.android.com/studio, last accessed 2016/04/19.
5. Get started with the NDK | Android NDK | Android Developers, https://developer.android.com/ndk/guides, last accessed 2016/04/19.
6. Android Runtime (ART) and Dalvik | Android Open Source Project, https://source.android.

com/devices/tech/dalvik/, last accessed 2016/04/19.
7. OWASP Top 10 -2017 The Ten Most Critical Web Application Security Risks, https://owasp.org/www-project-top-ten/2017/, last accessed 2016/04/19.
8. KeepSafe - Google Play, https://play.google.com/store/apps/details?id=com.kii.safe, last accessed 2016/04/19.
9. X. Zhang, I. Baggili, F. Breitinger: Breaking into the vault: Privacy, security and forensic analysis of Android vault applications. Comput. Secur., vol. 70, pp. 516–531, doi: 10.1016/j.cose.2017.07.011 (2017).
10. AppLock - Protect Your Privacy, http://www.domobile.com/best/applock.html, last accessed 2016/04/19.
11. Calculator - Google Play, https://play.google.com/store/apps/details?id=com.calculator.vault, last accessed 2016/04/19.
12. Password Manager - Keeper - Google Play, https://play.google.com/store/apps/details?id=com.callpod.android_apps.keeper&hl=en_US, last accessed 2016/04/19.
13. Bitwarden Password Manager - Google Play, https://play.google.com/store/apps/details?id=com.x8bit.bitwarden, last accessed 2016/04/19.
14. Bitwarden Security Assessment Report, https://cdn.bitwarden.net/misc/Bitwarden Security Assessment Report.pdf, last accessed 2016/04/19
15. File Attachments | Bitwarden Help & Support, https://bitwarden.com/help/article/attachments/, last accessed 2016/04/19