# Analysis of Secure Storage in Android Applications

Rasha Altamimi, Jerrold Ansman, Niharika Arora, Kyle Hogan, Amal Kadi, and Blake Tribou

## I. Introduction

Android applications are becoming increasingly responsible for managing sensitive information. Phones are used for banking, email, social media, and can be a part of many aspects of the users life. Each of these applications must be able to store and access this data - passwords, bank account numbers, address and phone number - securely, without leaking any information to a would be attacker. Many applications choose to store this data within the phone as this allows users to access it without an internet connection.

Data stored locally must be protected against the case that an attacker manages to gain physical access to the device. Possession of the device gives an attacker the ability to gain access to any of the applications data stored in shared preferences or a SQLite database through requesting a backup or rooting the phone and accessing the filesystem itself[12]. In this case the data is only as secure its decryption key. Anything stored in plaintext will be clearly visible to the attacker and some developers choose to use encoding rather than encryption for their data - resulting in a decode function within the application that can be used to reverse any encoded passwords or keys. Even encrypted data is vulnerable to the case that a decryption key was stored within the application or the possibility that, while the data or key is stored securely, upon access the running application will transfer it in plaintext and an attacker performing a dynamic analysis could recover it.

Here we will focus primarily on the storage of passwords and other sensitive data within android applications. In particular we have chosen to analyze applications whose primary function is security related: password managers, data lockers, etc. Our expectation is that developers of these apps will have been more conscious of the importance of securing their data and will thus have attempted to follow best practice. Due to the lack of restrictions placed on the content of the playstore, we are not considering a random selection of applications as we expect that this would result in a significant percentage where developers did not make any attempt to provide security. We assess whether, out of a subset of security conscious applications, there is a significant difference between the security of the data in those created by independent developers versus those created by a corporate dev team. From this, we provide recommendations on which were the most effective measures and whether these are a cost and effort effective option for an indie developer.

## II. Tools

### A. JADx, JD, & dex2jar

dex2jar converts Androids .dex format to Javas .class format. This allows Java decompilers such as FernFlower or JD to decompile the .class files to display the Java source code[6], [16], [24]. JD was used in the form of a plugin for Intellij Idea that allows the fileviewer to display Java source code from .class files, as as a standalone decompiler in the form of JD-GUI[24]. JADx is a full decompiler that takes .dex, .class, .apk, or .jar files as input and produces .java files as output. It has options to choose whether or not to decode resources or decompile source code[14]. This is useful for cases where the application may have been run through an obfuscator and a full decompilation would not have been possible. Additionally, it provides some deobfuscation features such as renaming variables into readable format and producing a control flow graph[14].

### B. Drozer

drozer is a comprehensive security audit and attack framework for Android. In our analysis we checked the activities exported by an app, which are the activities it makes accessible to other applications running on the device. [10], [11]. We have analyzed 33 password manager apps from Google Play (see Appendix C), from which, only 1 has a vulnerability in which they display the user's saved passwords without prompting for the master key (bypasses security check).

### C. apktool

APKTool is a decoder for Android .apk files, it will decode both code and resources such as the application's manifest. It utilizes smali for assembling/building of dex format code and baksmali as part of the decoding/disassembly process[2]. Previously, it supported debugginf for smali files, but that has been depreciated since the creation of smaliidea[5], [25].

### D. adb

The Android Debug Bridge is a service that allows a developer to communicate with a connected Android Device or emulator. It can be used to create backups, install applications, and access the filesystem of the device - including databases[23]. It runs as a client, a server, and a daemon process. The client is started from the command line of the development machine and issues commands to the daemon running on the Android device. The communication between the client and the daemon is managed by the adb server which is also running on the same machine as the client [23].

### E. Intellij Idea

Intellij Idea is a Java IDE that provides plugins to support the reverse engineering of Android applications. Distributed with the software is an extension for the .class viewer that integrates the FernFlower decompiler to display Java source code instead of bytecode[16]. There is also a plugin available for download that does the same, but is based on JD-GUI instead[15]. (Note: experience working with both of these plugins is that obfuscated bytecode is neither decompiled nor displayed, though they work correctly with non-obfuscated code. JADx was superior for decompilation of obfuscated applications with the added benefit of going directly from Dalvik bytecode[14]) Another plugin available for download, smaliidea, provides synatax highlighting for smali files and debug support for bytecode including breakpoints and instruction level single stepping. The support for smali files is particularly helpful in the case that the application being analyzed had been run through an anti-reverse engineering framework such as ProGuard or DexGuard. These files are disassembled, but not decompiled so, while decompilers such as dex2jar or JADx fail on certain methods in obfuscated classes - removing the possibility of recompiling the application, baksmali[2] leaves the application in Dalvik bytecode so the resultant application can be modified and then reassembled into an apk. These two features allow for a combination of static and dynamic analysis of applications as the decompiled code is more human readable, making the identification of relevant parts of the code more convenient, but it cannot be recompiled so smaliidea is better for assessing the flow of an application.

### *F. APK Studio*

APK Studio is a QT5 based IDE designed specifically for the reverse engineering of Android applications, It essentially provides a subset of the features of Intellij Idea with plugins as described above, but its simplicity makes the assembly and running of previously disassembled applications more straightforward[7].

## III. Applications

### *A. my password einglish*

my password einglish is a, for good reason, unpopular password management application available on the play store with less than 500 downloads. To analyze it we first launched the drozer agent on the phone and ran the following command to get the list of activities exported by the app: 'run app.activity.info -a appinventor.ai_ashersport.my_password_einglish'[10], [11]. Next we ran this command to run one of the activities: 'run app.activity.start –component appinventor.ai_ashersport.my_password_einglish appinventor.ai_ashersport.my_password_einglish.Screen2'
The command displays a screen on the phone with the password list in plaintext, just like what you would get on your phone when you use the application, except it bypassed the security check of asking for the master password. This displayed the name and value of every password stored within the application.

### *B. Password*

Password (it.bonavita.password) is a slightly more popular password manager, but no more secure than the previously discussed one. Though the passwords the user stores in the application are encoded before being entered into the database and the application does attempt to hash and encrypt some of the information (using DES, which is easily breakable at this point and MD5 which is not collision resistant), the master password for the application is stored in plaintext in a different table. The database is not encrypted so it is easy to access and read and/or modify this password. Since this password unlocks the application, obtaining it makes obtaining the rest of the passwords trivial as, once the app is running, an attacker could simply enter the password and the application would give them the rest of the passwords for free.

The use of encode and decode functions written into the application is also problematic as an attacker can simply grab the decode function and run the passwords from the database back through it, since their encoded values are easily accessible.

### *C. CryptFS*

The application is developed by Nikolay Elenkov. Cryptfs Password is an android application that is used to change the android disk encryption password. This application requires root access to the users device and that it be encrypted. It will not run if these conditions are not met[13].

To analyze, the code was decompiled using JADx. It was slightly obfuscated, but not to the degree of SAASPASS or LastPass. It was found that the decompiled code contained a java file named SuShell.java which contained shell commands prefixed with su".

Providing any application root access can be a security threat to the users device. In order to provide root access to an application, a user should permanently root his/her device which can lead to loss of user data and application data on the device. Also, if an application gains the super user access, there can be a possibility that the application is performing some other tasks in the background putting the user data on a vulnerable end. Additionally, if a users device is permanently rooted, the device loses its sandbox and any kind of protection management present. This will result in a security threat to the users device and users private data. This is especially relevant in this case as we have shown above that this application could then access the passwords stored by either of the other applications analyzed.

### *D. KeePass*

Developed by Dominik Reichl around 2004. KeePass is a desktop application designed to manage the increasing number of complex passwords required by current computer operations. KeePass is free and open source to allow all users to review the source code and verify that the encryption algorithms are correctly implemented[19]. KeePass utilizes AES and Twofish encryption algorithms to to perform full database encryption. It hashes the user's password plus 128-bit random salt using SHA-256 to get the final key. KeePass allows users to use either a master password or a key file to encrypt the database[17]. Users are also allowed to use both methods in unison for a more secure encryption method. In this case the key would be formed via: (SHA-256(SHA-256(password), SHA-256(key file contents)) and both the password and the key file would be required to decrypt[21].

All of a user's usernames and passwords are stored in the encrypted database and transferred to forms when needed using an AutoType feature. This utilizes clipboard obfuscation and virtual keystrokes in an attempt to fool any keyloggers that may be present on the device and keep the password protected while it is unencrypted[19].

*1) KeyPass2 Android Offline:* Keepass2Android is a port of KeePass 2.x to the Android platform that reads and writes KDBX files which is KeePass's extension for its encrypted database files. These databases are composed of a password-protected file contianing an encrypted body and an unencrypted header[28]. The user interface is based on KeePassDroid (by Brian Pellin).

Users are able to copy their password database from the desktop version of KeePass to the Android version, or vice versa, which ensures that they have all their passwords on each device [20].

It also provides a random password generator which forms a password out of a user-specified set of characters and a AutoFill similar to that in the version discussed above [29].

KeePass2Android does not automatically store any passwords on the cloud, but provides an option to sync with a user's Dropbox account [18].

### *E. SAASPASS*

SAASPASS is an application designed to authenticate its users with their various accounts and devices from their phone. It provides password management, two factor authentication, secure storage, and proximity based authentication among other features. It is designed for both personal and commercial use and is the product of a well funded industry team. Thus, developers for this application should have had the resources, capability, and motivation to follow through on their promises of security.

Our choice of this application for analysis is based on this expectation - our analysis of an indie application with a small userbase showed basic flaws in the development, we are interested in assessing the difference a larger, better funded development team makes on an application that was designed with similar goals in mind.

Our intention was to focus on Locker functionality of SAASPASS which is designed to be a place within the application for users to store their passwords or other sensitive data such as bank account information. This section of the application is secured with the same four digit PIN used to access the application itself and the data within is stored in an encrypted SQLite database. Our intention was to a) acquire this PIN as it was being checked for validity b) locate the decryption key for the database or c) prevent the application from enforcing the number of attempts on the PIN (as four digits could be easily brute forced).

However, this application utilized the enterprise version of DexGuard (see Appendix A) which not only obfuscated the code, encoded strings, and hid calls using reflection, but also provided checks that prevent the application from running in a debugger, on an emulator, or on a rooted phone[3]. Additionally, it checks to ensure that the application was signed with the original certificate, preventing a recompiled application from running unless this check is located and removed before the application is recompiled and loaded onto the phone.

The application also does not allow data backup, which normally could be easily changed through modification of the AndroidManifest, but cannot be modified in this case without first removing the certificate check implemented by DexGuard[12].

Due to the difficulties attaching a debugger to the unmodified application running on a phone, primarily static analysis was performed. Initially, the application was decompiled with JADx and the .java files were examined for references to validating the PIN or to the locker functionality. We quickly ran into one of the obfuscated classes which contained encoded strings and commented out decoder method that JADx had not been able to decompile out of bytecode. We translated this into Java code by hand (see Appendix B for a discussion of DexGuard obfuscation) to see if we could obtain a plaintext version of the string. Our translation and result appeared consistent with the bytecode for the method, but the string obtained was not recognizable as a filename and did not appear elsewhere within the application. This is not particularly surprising as DexGuard renames variables and resources so the decoder would only have produced a recognizable filename if the file being accessed was stored external to the application, preventing its name from being modified[9]. Without attempting to translate each method where the decompiler failed by hand to determine whether this file was created elsewhere in the application, it did not seem like this direction was likely to be fruitful. We thus switched to working with the .smali files directly. As these can be recompiled and run we were interested in attempting to determine the portion of the code responsible for the emulator, debugger, and certificate checks. Due to the encoded variable and class names and the call reflection (2323 files worth) we were unable to find and remove the certificate check. This prevented us from performing a dynamic analysis on SAASPASS and was ultimately the largest hurdle in its analysis.

### *F. LastPass*

LastPass is a multi-platform password and secure notes storage program designed for both individual and corporate use. For our purposes, we focused only on the free Android version of the program (there were additional versions for premium and corporate use that require purchase)[26]. It was very similar in functionality to that of SAASPASS.

As with SAASPASS, the motivation behind choosing this program was to add another corporate app to the list and compare it with the others we had analyzed. It came as little surprise that LastPass was utilizing some of the same techniques in protecting their program.

The developers of LastPass claim that all passwords are encrypted and decrypted on the device before being transferred to the cloud, and that they use PBKDF2 to generate encryption and decryption keys for use with AES-256[26]. Instead of providing a simple 4 digit PIN, as with SAASPASS, users would have a username/password combination for the secure storage of their data with the password being used as part of the key derivation. In addition to storing passwords, LastPass was also able to store other sensitive information supplied by the user which it would encrypt using the same techniques.

The LastPass developers also appeared to be employing the Enterprise version of DexGuard or similar type of professional obfuscation tool[3], [9]. The .apk decompiled into over 3000 .java files with many functions unable to be decompiled due to generating inconsistent code. Dynamic Analysis was prevented due to emulator detection and code modification was nullified due to code signature checking.

Since LastPass does support auto form filling[27], it is possible that it is using the Android Clipboard, which depending on the version of Android being used, may present a vulnerable attack vector, but that goes beyond the scope of this report.

## IV. Conclusion

From our analysis of these applications we determined that the data contained within the corporate developed applications was generally significantly better secured than that within the indie applications. Interestingly, this was due primarily to the level of code obfuscation provided by DexGuard. SAASPASS and LastPass did do a better job of encrypting their database, but beyond that DexGuard made it incredibly difficult to determine whether they had used strong crypto, left any keys within the application, or were exporting security centric activities. All the strings were encoded, variable names were changed, application flow was hidden through call reflection, and root, emulator, debug, and certificate checks prevented the sort of dynamic analysis that had revealed information for some of the indie apps. The main flaws in the indie applications were exactly the things DexGuard made it difficult to find: they used outdated crypto like DES, used in-app encoding instead of encryption, did turn off backup or debugging, or were not checking access privileges. Additionally they often failed to encrypt the entire database and would instead try to encode or encrypt only its contents - leaving many opportunities to make a mistake and allowing an attacker to modify the contents (in a denial of service sort of way, if crypto with non-malleability was used they could not change anything to a value that would be meaningful to them) even if they could not decrypt them.

Unfortunately, DexGuard is expensive and the mild encoding provided by the free version, ProGuard, is not nearly as effective[3], [9]. It is unlikely that indie developers will be able to afford this or that they will have the time and expertise necessary to duplicate some of its most effective features against static analysis (call reflection and string encoding/decompiler failures) themselves. However, root checks, emulator detection, debug checks, and certificate verification are less time consuming to implement and seriously hinder dynamic analysis of the applications. Anyone attempting to reverse engineer the application would have to first find and remove the certificate verification before they could begin attempting to get rid of emulator checks or anti-debug. This is obviously more difficult for the heavily obfuscated applications, but will still hinder anyone attempting to perform a dynamic analysis of even unobfuscated code. Combining this with full database encryption and careful checks of the source code to make sure that no keys are hardcoded into the application, to reduce the efficacy of static analysis, and taking the time to verify that all logging, debug features, and backup were turned off and that no security centric activities were accessible by other applications would have prevented all of the information leaks that we found, without requiring developers to pay for DexGuard or to expend significantly more time or effort.

Since no code validation is performed by the Play Store, and even if it was, it is the responsibility of developers to protect the data that their users store within their application. As we have shown here, this is often easier said than done, even for developers who should have been security minded when writing their applications. We have identified features of more secure applications and assessed which of these are time and cost effective for developers who possess fewer resources to implement. Our findings indicate that, while expensive, DexGuard was very effective in slowing down or even stopping reverse engineering of the application and that, though difficult to replicate in its entirety, certain features such as emulator and debug checking, can increase the security of the application without requiring significant effort on the part of the developer. When this is combined with appropriate use of encryption and enforced privacy of sensitive activities, it increases the effort required to exploit information leaks, if not eliminating them entirely.

## Appendix A
## Detecting DexGuard

Use of DexGuard can be identified by its use of non-ASCII characters in file and variable names[9]. This must be checked before files are decompiled as most decompilers tested will rename the vars and files for readability. Additionally, the method of using integer operations on elements of a byte array within a decode method at runtime is distinctive to DexGuard[8]. Proguard will not perform encoding in this way and APK Protect inserts the string APKProtected into the dex [9]. The Enterprise version of DexGuard can be detected by the addition of anti-debug, emulator, root detection, and certificate checking features[3].

## Appendix B
## Failure to Decompile DexGuarded Apps

We were not able to determine exactly what DexGuard does that causes decompilation to fail. Based on our use of different tools we are fairly confident that this failure occurs as the code is being translated from Android .smali files to Java .class files. (disassembling from .dex to .smali (baksmali) works, and no new errors are introduced going from .class to .java (numerous tools, including IDE plugins[5]), but crossing the Android/Java platform line using either dex2jar or any of the full Android decompilers causes failures[6].)

Our theories for why this might be the case, based on error messages returned from decompilers and conversations with Ethan Heliman, are that a) in certain cases the byte arrays used for decoding are too large to be placed into a class file (error from dex2jar) and b) misleading information can be added to the code that would be ignored by a running program, but that a decompiler might try to use for optimization purposes. Based on observation of the decompiled code and the error from dex2jar (that the method was too large to be placed in a class file), both options seem valid. Smaller byte arrays were decompiled correctly, and the failed methods were fairly easily decompilable by a human - lending credence to the idea that the decompiler had been misled by false information - or heavy optimization - about the structure of the method.

## APPENDIX C
## LISTING OF APPS ANALYZED USING DROZER

allmypasswordswithtrial.byengashdigital (All My Passwords)
appinventor.ai_ashersport.my_password_einglish (my_password_einglish)
awesomelifemaker.password.lite (My Password)
com.MyPasswords (My Passwords)
com.baki.mypasswordbank (My Password Bank)
com.er.mo.apps.mypasswords (My Passwords)
com.fenapps.android.mypassword (My Passwords)
com.jin_engineering.mypasswords (My Passwords)
com.kaspersky.passwordmanager (Kaspersky Password Manager)
com.mmitco.mypasswordsaver (Password Saver)
com.mypasswords.mypass (My Pass)
com.passwordbox.passwordbox (PasswordBox)
com.ssmobile.keys (My Passwords)
com.superappslab.mypasswords (My Passwords)
eti.victor.minhassenhas (My Passwords)
it.drd.UltimatePasswords (My Password)
net.sugar.mypasswords (MyPasswords)
org.igears.passwordlocker (Password Locker)
aaro.Remembrall.Android (Remembr'all)
com.agilebits.onepassword (1Password)
com.callpod.android_apps.keeper (Keeper)
com.dashlane (Dashlane)
com.reneph.passwordsafe (Password Safe)
com.safeincloud.free (SafeInCloud)
com.siber.roboform (RoboForm)
com.symantec.mobile.idsafe (Norton Identity Safe)
keepass2android.keepass2android (Keepass2Android)
org.awallet.free (aWallet Password Manager)
com.lastpass.lpandroid (LastPass)
com.mirsoft.passwordmemory (Password Manager)
com.green.banana.app.lockscreenpassword (Lock screen)
com.gsonly.passbook (Password Saver)
org.sam.applications.passwordmanager (Password Manager)

## ACKNOWLEDGMENT

Thanks to Ethan Heilman for discussion of DexGuard, to our computers for running 5 different IDEs and an emulator at the same time, to the students (and professor) of EC521 for engaging questions, and to Dr. Egele for his assistance with the project.

## REFERENCES

[1] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson, ”Password Managers: Attacks and Defenses” *Proceedings of the 23rd USENIX Security Symposium.* 2014.

[2] A tool for reverse engineering Android apk files *APKTool.* http://ibotpeaches.github.io/Apktool/.

[3] ”Feature Comparison” *guardsquare.com.* https://www.guardsquare.com/comparison-proguard-and-dexguard.

[4] Nico, A Look Inside DexGuard *PNF Softwares Blog.* 2 April 2013. https://www.pnfsoftware.com/blog/a-look-inside-dexguard/

[5] Jesus Freke, smailidea https://github.com/JesusFreke/smali/wiki/smalidea

[6] dex2jar https://github.com/pxb1988/dex2jar

[7] APKStudio http://www.vaibhavpandey.com/apkstudio/

[8] Opensecurity.in Reversing Dexguards String Encryption 11 June 2015 http://opensecurity.in/reversing-dexguards-string-encryption/

[9] Axelle Apvrille, Ruchna Nigam. Obfuscation in Android Malware, and How to Fight Back 2 July 2014. https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation

[10] MWR Infosecurity Drozer User Guide 23 March 2015 https://www.mwrinfosecurity.com/system/assets/937/original/mwri_drozer-user-guide_2015-03-23.pdf

[11] Adita Agrawal. Android Application Security Series https://manifestsecurity.com/android-application-security/

[12] Godfrey Nolan, Where is the Best Place to Store a Password in Your Android App 31 March 2015 http://www.androidauthority.com/where-is-the-best-place-to-store-a-password-in-your-android-app-597197/

[13] Simon Roses, Build Root Detection in Your App 4 June 2013 http://www.simonroses.com/2013/06/appsec-build-rooted-detection-in-your-app/

[14] JADx https://github.com/skylot/jadx

[15] JD-GUI for Intellij https://bitbucket.org/bric3/jd-intellij

[16] FernFlower https://github.com/fesh0r/fernflower

[17] Reichl, D. ”KeePass: Password Safe.” 2015 http://keepass.info/features.html

[18] Zukerman, E. ”Manage Your Passwords On The Go With KeePassDroid [1.5+]”. *MakeUseOf.* http://www.makeuseof.com/tag/manage-passwords-keepassdroid-15/ 21 February 2012.

[19] Zukerman, E. ”Review: KeePass makes strong passwords and keeps them safe.” *PCWorld.* 28 January 2013 http://www.pcworld.com/article/2026547/review-keepass-makes-strong-passwords-and-keeps-them-safe.html

[20] Security-in-a-box. ”KeePassDroid For Android. Security-in-a-box.” 21 July 2014 https://securityinabox.org/en/guide/keepassdroid/android

[21] ”Security” *KeePass help center* http://keepass.info/help/base/security.html

[22] Eric Ravenscraft. ”The Easiest Way to Install Android's ADB and Fastboot Tools on Any OS” *Lifehacker* 6 June 2014 http://lifehacker.com/the-easiest-way-to-install-androids-adb-and-fastboot-to-1586992378

[23] Android Debug Bridge http://developer.android.com/tools/help/adb.html

[24] JD-GUI https://github.com/java-decompiler/jd-gui

[25] Connor Tumbleson. ”Remove SmaliDebugging” 14 October 2015. https://github.com/iBotPeaches/Apktool/issues/1061

[26] LastPass How It Works. https://lastpass.com/how-it-works/

[27] Last Pass Form-Fill https://helpdesk.lastpass.com/fill-form-basics/

[28] Paolo Gasti and Kasper B. Rasmussen. ”On The Security of Password Manager Database Formats” https://www.cs.ox.ac.uk/files/6487/pwvault.pdf

[29] ”Password Generator”. KeePass Help Center. http://keepass.info/help/base/pwgenerator.html