8-2-2017

# Breaking Into the Vault: Privacy, Security and Forensic Analysis of Android Vault Applications

Xiaolu Zhang
*University of New Haven*

Ibrahim Baggili
*University of New Haven*, ibaggili@newhaven.edu

Frank Breitinger
*University of New Haven*, fbreitinger@newhaven.edu



## Publisher Citation

Xiaolu Zhang, Ibrahim Baggili, Frank Breitinger, Breaking into the vault: privacy, security and forensic analysis of android vault applications. Computers & Security, Volume 70, September 2017, pages 516-531.

## Comments

This is the authors' accepted version of the article published in *Computers & Security*. The version of record may be found at http://dx.doi.org/10.1016/j.cose.2017.07.011 .

Dr. Baggili was appointed to the Elder Family Endowed Chair in 2015.

# Breaking into the vault: Privacy, security and forensic analysis of Android vault applications

**Xiaolu Zhang[a], Ibrahim Baggili[a,*], Frank Breitinger[a]**

*[a]Cyber Forensics Research & Education Group, Tagliatela College of Engineering, ECECS, University of New Haven, 300 Boston Post Rd., West Haven, CT, 06516*

**Abstract**

In this work we share the first account for the forensic analysis, security and privacy of Android vault applications. Vaults are designed to be privacy enhancing as they allow users to hide personal data but may also be misused to hide incriminating files. Our work has already helped law enforcement in the state of Connecticut to reconstruct 66 incriminating images and 18 videos in a single criminal case. We present case studies and results from analyzing 18 Android vault applications (accounting for nearly 220 million downloads from the Google Play store) by reverse engineering them and examining the forensic artifacts they produce. Our results showed that 12⁄18 obfuscated their code and 5⁄18 applications used native libraries hindering the reverse engineering process of these applications. However, we still recovered data from the applications without root access to the Android device as we were able to ascertain hidden data on the device without rooting for 10⁄18 of the applications. 6⁄18 of the vault applications were found to not encrypt photos they stored, and 8⁄18 were found to not encrypt videos. 7⁄18 of the applications were found to store passwords in cleartext. We were able to also implement a swap attack on 5⁄18 applications where we achieved unauthorized access to the data by swapping the files that contained the password with a self-created one. In some cases, our findings illustrate unfavorable security implementations of privacy enhancing applications, but also showcase practical mechanisms for investigators to gain access to data of evidentiary value. In essence, we broke into the vaults.

*Keywords:* Forensics, Mobile applications, Privacy, Security, Vault Applications, Android.

## 1. Introduction

Mobile vault applications facilitate secure storage of personal data, preventing data leakage when a user's device is borrowed or lost. Some applications disguise themselves by mimicking widely used programs such as a calculator. Only when a user types in the correct passcode (like a password), hidden data will be displayed. By searching 'vault app' in the Google Play Store, many vault applications can be found. At the time of writing, the most downloaded vault application – AppLock had been downloaded over a hundred million times.

From a privacy perspective, vault applications can be used to hide benign personal data (typically media files such as pictures and videos). They have also been misused with criminal intent for hiding photos / videos of victims. This means that they are also considered as anti-digital forensic tools and fall under the data hiding category since they hinder the forensic process (Conlan et al., 2016). Furthermore, vault applications have been used by minors to hide photos from their peers and parents. The increased use of vault applications has brought challenges for school systems as well as security and digital forensic practitioners.

For example, in a 2015 case, high school students in Colorado exchanged hundreds of nude photos of themselves (*Colorado sexting scandal: High school faces felony investigation*, 2016). Nearly half of the students in the incident hid their photos from their parents in a calculator-like vault application. In another criminal case in December 2015, the Connecticut Glastonbury Police Department (PD) investigated a report that a man had been observed secretly taking pictures with his phone up the skirt of a woman in an office building parking lot while he assisted her with her vehicle. During the investigation, the suspect's phone was received at the Connecticut Center for Digital Investigations (CDI) for examination. The phone was a Samsung Galaxy 6 Edge running Android. A logical and physical extraction were completed using what most law enforcement regard as the golden standard tool for mobile forensics: Cellebrite UFED 4PC. The data was analyzed using Cellebrite's Physical Analyzer. During the analysis, it was noted that the suspect had been using several programs on the phone that were designed to hide files. One of the programs in particular was titled *GalleryVault*.

*Corresponding author.
*Email addresses:* XZhang@newhaven.edu (Xiaolu Zhang), IBaggili@newhaven.edu (Ibrahim Baggili), FBreitinger@newhaven.edu (Frank Breitinger)
*URL:* http://www.unhcfreg.com (Xiaolu Zhang), http://www.Baggili.com (Ibrahim Baggili), http://www.FBreitinger.de (Frank Breitinger)

With traditional forensic tools, the police were only able to recover a few photos that were located on the device, none of which were guarded by the vault applications.

It is of note that work presented in this paper was used on the data hidden by the *GalleryVault* application. The researchers collaborated with CDI and reconstructed 66 new files that were previously irrecoverable. These files included 18 additional videos that would be prosecutable in the Connecticut jurisdiction, as well as 38 videos that may be prosecutable in other jurisdictions. All together, 42 new victims were revealed in these recovered videos allowing the police to demonstrate the full scope of the suspect's actions, having a direct impact on the outcome of the case. This showcased the importance of our research findings to the law enforcement community.

Our work had two complementary goals. The first was to test security implementations in vault applications to examine their impact on privacy. The second was to explore the digital forensic implications of our findings. We therefore conducted an investigation based on the logical acquisition of artifacts from vault applications complimented by (in most instances) reverse engineering the 18 most popular free vault applications which at the time of writing amounted in total for over 219,500,000 downloads from Google Play.
This work presents the following contributions:

- Methods for breaking into 18 vault applications is presented in detail allowing the reconstruction of original files hidden by most of the applications.
- The location and type of relevant forensic artifacts for the 18 vault applications such as passwords (cleartext and hashed), database files and encrypted and unencrypted media files.
- In $^{16}/_{18}$ cases, we present ways to reconstruct or retrieve hidden files without the need to log into the applications.
- Software tools for reconstructing / decrypting data from some of the vault applications.

The rest of the paper presents related work in Sec. 2 followed by the apparatus used in Sec. 3. We share our four-phase overarching methodology in Sec. 4. The core of our work is presented in Sec. 5 with our case study and findings for the 18 vault applications are portrayed. We then follow up with Sec. 6 which summarizes our results and findings. The discussion and limitation sections are then presented in Sec. 8 and Sec. 9 respectively. We conclude and set directions for future work in Sec. 10.

## 2. Background information & related work

We review related work in four major areas: methods for Android data acquisition (Sec. 2.1), artifact analysis of popular messaging and social networking applications (Sec. 2.2), Android application privacy (Sec. 2.3) and approaches for reverse engineering Android applications (Sec. 2.4).

### *2.1. Android acquisition methods*

Readers familiar with mobile phone acquisition may want to skip this section. Data acquisition can be achieved both logically and physically. Logical acquisition extracts user data recognized by the filesystem whereby deleted files are excluded. Physical acquisition achieves a bit stream copy of flash memory and all the data physically stored. One approach for achieving logical acquisition is utilizing *adb* – an official Android versatile command line tool (*Android Debug Bridge — Android Studio*, n.d.). With this, user data can be transferred from the Android device to a forensic workstation. However, establishing an *adb* connection requires the target device to have Universal Serial Buss (USB) debugging mode enabled. Additionally, whether the data can be extracted relies on the established user privilege.

Early research used *adb* to support physical acquisition (Lessard and Kessler, 2010; Hoog, 2011). In these articles, achieving physical acquisition required root privilege on the device to leverage an *adb* connection. Even though Lessard and Kessler (2010) presented an approach for gaining root access, it is no longer wildly applicable for the new generation of Android devices. In situations where root privilege, an unlocked screen or enabled *adb* is unavailable, researchers achieved physical acquisition by flashing a custom image to a recovery partition of an Android device (Vidas et al., 2011; Son et al., 2013). This was executed by rebooting the device to recovery mode to acquire user data with minimal loss of integrity.

In recent work, Yang et al. (2015) discovered in the firmware update protocol for different brands of Android devices a flash memory read command. By reverse engineering the protocol in the *bootloader*, the researchers were able to physically acquire the entire flash memory of the device without gaining root privilege and unlocking the screen.

While most acquisition typically happens using software, acquisition may also be achieved at the hardware level and is usually substantially more intrusive to a device's hardware. Joint Test Action Group (JTAG) and chip-off are the most popular approaches (Breeuwsma et al., 2007). JTAG leverages a testing port on a mobile device and can be utilized by examiners to physically connect to a Printed Circuit Board (PCB), which is available for variety Android devices (Kim et al., 2008). In Chip-off, investigators detach the flash memory chip from the PCB and acquire the data using specialized equipment (Hoog, 2011).

### *2.2. Application artifact analysis*

Other Android research focused on recovering and exploring application artifacts. Much of the work has focused on Instant Messenger (IM) applications that provide evidence of interpersonal communications. IMs such as Wickr

and ChatON on Android have been subjected to artifact analysis (Mehrotra and Mehtre, 2013; Iqbal et al., 2013). Other work also focused on extracting GPS geodata from all applications on an experimental Android device (Maus et al., 2011) or target specific mobile navigation applications like Google maps or Waze (Jason Moore, 2016). Earlier work on mobile phones by Husain and Sridhar (2009) also forensically examined the iPhone messaging applications: AIM, Yahoo! Messenger and Google Talk. They were able to recover potential evidence such as usernames and passwords. Similar work was conducted on multiple Android devices where WhatsApp and Viber were examined (Mahajan et al., 2013).

Past research also examined the security of nine VoIP and messaging applications on both iOS and Android (Schrittwieser et al., 2012). Results indicated that in most tested applications, vulnerabilities could be exploited to hijack accounts, enumerate subscribers and more. Anglano (2014) fully analyzed artifacts acquired from WhatsApp Messenger, which included several database files (e.g. contacts database, chat database etc.) and sent or received media files. Unclear data was decoded and interpreted. Sahu (2013) applied data acquisition and analysis on both non-volatile and volatile memory for WhatsApp. They were able to retrieve artifacts such as contacts, location, media files and messages.

In other recent work, Karpisek et al. (2015) were able to decrypt WhatsApp's network traffic and examined the new voice call signaling protocol. They were able to identify and visualize the messages exchanged between the client and server as well as the audio codec used.

Social networking applications have also been studied by forensic / security researchers. For instance, the widely used social networking applications: Facebook, Twitter and MySpace were forensically analyzed (Al Mutawa et al., 2012). Walnycky et al. (2015) conducted forensic analysis on 20 popular Android social messaging applications by capturing the applications' network traffic as well as examining the data they stored. Lastly, Saltaformaggio et al. (2016) were also able to recover multiple previous screens of a terminated application on an Android device from a memory image.

In summary, mobile application analysis has attracted attention. However, past scientific work was limited to application artifacts or network traffic of messaging applications. At the time of writing, there were minimal exploratory attempts aimed at examining the security of some vault applications presented in blog entries (David Auerbach, 2015; E2e, 2016; Gary Hawkins, 2012). Some of these early results are outdated. More importantly, the blog entries only scratched the surface in exploring the strength of data protection in vault applications and the practitioners did not reverse engineer the applications at the source code level - thus our findings are more comprehensive and rigorous. For example, the authors were unable to retrieve encrypted data from Private Photo Vault (Table 1), whereas our work was unable to recover th data protected by that application. It is also of note that at the time of writing, no scientific papers have paid attention to vault applications. Given their wide adoption, our work fills that literature gap. Since our study focused on a specific types of Privacy Enhancing Technologies (PET) – vault applications – literature pertaining to Android application privacy is of relevance to our inquiry.

### *2.3. Android application privacy*

Other than forensic related literature, the privacy leakage / security issues of Android applications have inspired an active body of research. Past work in this area focused on potential attacks for leaking private user data. Work by Jana and Shmatikov (2012) illustrated a side-channel attack by tracking changes in an application's memory footprint. They showed that using a concurrent process belonging to a different user can grab an application's secrets. They used web browsers as an application target and illustrated how an unprivileged, local attack process e.g., a malicious Android application can infer the page the user is browsing as well as finer-grained information like if the user is a paid customer, interests, etc. This motivated studies on the automatic detection of privacy leakage from Android applications. For example, AppAudit merged static and dynamic analysis to provide effective real-time Android application auditing by simulating the execution of part of the program and performing customized checks at each program state (Xia et al., 2015).

Tools such as *AndroidLeaks* and *AppIntent* focused on detecting a user's information privacy leakage. Even though Android applications are sandboxed, vulnerabilities were identified that allowed applications to gain the capabilities of other applications, and *DroidChecker* was the tool created to detect those vulnerable applications (Gibler et al., 2012; Yang et al., 2013; Chan et al., 2012). Taming Information-Stealing Smartphone Applications (on Android) or *TISSA* extended a new privacy mode on Android to prevent information stealing from applications (Zhou et al., 2011). By tracking the information transferred in the control flow, Static Analyzer for Detecting Privacy Leaks in Android Applications *SCANDAL* was also designed for privacy leak detection (Kim et al., 2012). Secure Password Tracking for Android *SpanDex*, however, was created to ensure that passwords do not leak from Android applications by creating a set of extensions to Android's Dalvik virtual machine (Cox et al., 2014).

There exists abundant literature on the detection of privacy leakage from Android applications, and although this body of knowledge is relevant to our work, it is not fully applicable. Since our work focused on reversing vault applications to understand their security implementations, and since every security implementation is different, we had to revert to manual analysis and reverse engineering techniques. This is why tools and techniques for Android application analysis and reverse engineering is of strong relevance our work.

### 2.4. Android application analysis

The purpose for statically analyzing the source code of or reverse engineering an Android application is to provide a comprehensive understanding of its inner workings. As Android applications are programmed in JAVA, different methods can decompile the compiled JAVA code (Dalvik bytecode) into a human-readable representation. The human readable code is typically in the form of JAVA source code or *smali* code.

Enck et al. (2011) presented a step-by-step tutorial for converting the Dalvik bytecode to JAVA source code, where the tool *ded* (*ded Homepage*, n.d.) converts the Dalvik bytecode into JAVA bytecode (`.jar`, `.class`) and the tool *Soot* (*A framework for analyzing and transforming Java and Android Applications*, n.d.) decompiled JAVA bytecode to JAVA source code. Gibler et al. (2012) utilized other tools with similar functions – *dex2jar* (*dex2jar*, 2016) similar to *ded* and – *JD-GUI* (*JD-GUI A Java Decompiler*, n.d.) – similar to *Soot* but with a Graphical User Interface (GUI). Another possibility is to convert Dalvik bytecode to *smali* code which is an assembly-like representation based on Jasmin syntax (*Jasmin User Guide*, n.d.). The tools to achieve this are *Apktool* (*Apktool - A tool for reverse engineering Android apk files*, n.d.) and *smali/baksamli* (*smali/baksamli*, 2016). If the analysis encompasses a large sample of Android applications, an open source parser called Rapid Android Parser for Investigating DEX files *(RAPID)* can effectively reduce the processing time (Zhang et al., 2016).

As Android applications can call native libraries[1], the well-known commercial tool Interactive Disassembler *(IDA) Pro* (*IDA pro*, n.d.) may be used for their analysis. The benefits of using *IDA Pro* are the disassemble and debug functions for both APK files and native libraries followed by the function of converting assembly code to C-like pseudocode. Note that *IDA Pro* is not limited to low level Android application analysis and may be used for disassembling Windows and other types of executables. Acar et al. (2016) also pointed out, due to the fact that most Android developers are likely to use existing open source code / libraries, insecure implementations and vulnerabilities may be inherited. Other work by Bläsing et al. (2010) proposed a sandbox for conducting dynamic analysis of Android applications. However, dynamic analysis was not employed in our work. To go over all the tools and techniques that foster Android application analysis is beyond this paper's scope. One may explore work by Faruki et al. (2015) to survey the various tools.

## 3. Apparatus

This section outlines the apparatus used in our testing. The Android device used when installing the vault applications was a *GalaxyS4* mobile phone, model number `SAMSUNG-SGH-I537`, system version `5.0.1` and kernel version `3.4.0-4554112`. An acquisition workstation connected to the phone via USB running Microsoft Windows 8.1 was employed. Even though root access may not always be a prerequisite for acquisition of potential artifacts (see Sec. 6), user privilege on the device was maximized in order to locate necessary files and artifacts; the *GalaxyS4* was rooted and USB debugging mode was enabled. Other tools were configured on the acquisition workstation. *adb* was used to connect the forensic workstation to the Android device for acquiring relevant vault application artifacts. *ApkTool* and *IDA Pro* were used in reverse engineering the Android applications and a *SQLite Browser* (*DB Browser for SQLite*, n.d.) was used for exploring SQLite database files (*SQLite Home Page*, n.d.).

## 4. Methodology

To verify if an Android vault application was able to protect a user's personal files, the testing was divided into four phases: scenario creation, data acquisition, case analysis and tool development.

### 4.1. Scenario creation

The purpose of this phase was to simulate real user data on the Android device. Therefore, the 18 most popular free vault applications were downloaded from the Android official application market – Google Play store. Appendix 1 shows the official name, name on the device and package name of the downloaded applications.

As soon as the vault applications were installed, we took photos (`.jpg`) and videos (`.mp4`) with clearly numbered identifiers (in each image / video) for each application and stored them in each 'vault'. We had to also initially set up passwords / passcodes on the vault applications. For the application *Keeper* (No. 4 in Appendix 1), we set the password '951234' since it required a minimum 6 character password. All other vault applications were configured with the same four digit password – 9512. This was chosen because it was a forced requirement to have a maximum of four digits for some of the applications – so we opted to standardize the password (as much as possible) across all of our testing. We also configured the same 9 pointer pattern lock – 0124678 (if that option was available) which represented the gesture of the letter 'Z'. Finally, when applicable, we set up the recovery e-mail '`vaultapptest@gmail.com`', and the security answer '3' with the security question 'What is your favorite number?' to applications that implemented a password recovery feature.

### 4.2. Data acquisition

To explore potential artifacts that may be utilized for breaking into the vault applications, two key items (listed in the following paragraphs) for each application were

[1] A native library for an Android application is also known as a `.so` (Shared Object) file which is usually compiled by C / C++ code and only executed on the structure of Central Processing Unit (CPU) of mobile devices.

pulled from the Android device to the acquisition workstation (more on how this was accomplished over the *adb* is discussed later in this section).

**App-generated folders** which are folders created by the application on the Android filesystem. We hypothesized that some files may contain hidden photos and videos, user passwords or other forensically relevant application artifacts that may help us break into the vaults.

**APK files** which are Android Application Packages. These files contain installer files for each of the vault applications and are an important resource for reverse engineering the executable code.

Precisely, for each vault application, three folders on the Android system were generally targeted during our data acquisition:

**/data/app/package_name** which is a root user accessible folder that stores the APK file of each application. 'package_name' refers to the package name of each application (see Appendix 1 for the package names).

**/data/data/package_name** is another folder accessible for root users that stores the private data / artifacts for each application.

**/sdcard** which is the folder for the mounted Secure Digital (SD) card physically located in the Android device. In our tests, we found different paths that pointed to the same folder such as: `/storage/emulated/0`, `/storage/emulated/legacy` and `/mnt/sdcard`.

It is important to note that in our work, acquisition was accomplished logically. We did not focus on deleted data as we were looking for low hanging fruit artifacts allowing us to break into each of the vault applications and reconstruct hidden files[2]. For each vault application, we employed the `pull` command in the *adb* tool to extract the following folders: `/data/app/package_name`, `/data/data/package_name` and `/sdcard`. To replicate our work, one may use the commands presented in the following steps:

**Step 1:** The command `chmod [OPTION] <MODE> <FILE>` was used to modify the assigned folders to a normal user accessible under a root user's access.

**Step 2:** The command `adb pull [-p] [-a] <remote> [<local>]` was used to acquire the folders mentioned above.

The resultant data from the acquisition process for each vault application is referenced in the case study in Sec. 5. Note that although data acquisition was achieved via *adb*, other methods to pull data from the aforementioned folders may be employed as well.

[2]It is of note that more research needs to be conducted to analyze how the media (images and videos) are stored by the vault applications in case data is copied and then deleted from the device, thus, there may be potential evidence residing on the device's storage when physical acquisition is employed.

### *4.3. Analysis*

Once the artifacts were extracted from the Android device, we sequentially analyzed each vault application through artifact analysis and executable code reverse engineering. Executable code analysis was primarily employed when the security implementation was needed in order to gain access to encrypted material stored in the vault application.

**Artifact analysis** was carried out manually by exploring the acquired data. During exploration, meaningful artifacts such as labels in Extensible Markup Language (XML) files (e.g. `<string name="password">`), column names in SQLite files (e.g. `aeskey`) or obvious folder / file names (e.g. `/sdcard/.EncryptedFolder`) were noted. Some findings became 1) Direct artifacts e.g. unencrypted photos / videos or passwords found in cleartext and 2) Indirect artifacts such as hashed passwords or files suspected to be the encrypted photos and videos.

**Executable code analysis** was employed mainly if all artifacts retrieved through our analysis were deemed indirect. We focused on reverse engineering the APK files. Precisely, we disassembled APK files into *smali* code. By employing manual code analysis, we were able to retrieve 1) Artifacts missed by our primary artifact analysis, where access to information or hidden media files were stored and 2) The security implementations for authentication / authorization or data hiding. This allowed us to understand how to reconstruct hidden data and explore methods of decrypting encrypted data.

### *4.4. Tool development*

Based on our findings, we constructed software tools if breaking into the vault application required such implementations to aid in the reconstruction and decryption of vault-stored data. We posit that developing these tools is necessary to assist investigators in the recovery of potential evidence they might have missed. ***We note that these tools are not shared publicly and are provided on need basis, after the identity of the requesting party is vetted.***

## 5. Case study and findings

In this section, we present a comprehensive case study and findings that entail each of the 18 vault applications shown in Table 1. The subsections follow the ordering of the table starting with the most prominent application. Each case details the results from our analysis augmented by an approach for recovering hidden photos / videos. It is of note that we attempted two major methods to gain access to data stored by the vault applications. The first focused on finding ways of recovering media files from data acquired directly from the phone – which is usually more relevant to forensic investigations. The second focused on gaining unauthorized access to the vault applications

Table 1: Most downloaded Android vault applications.

| | Application | Name on device | package name | Version | Downloads |
|---|---|---|---|---|---|
| 1 | AppLock | AppLock | com.domobile.applock | 2.16.3 | 100,000,000+ |
| 2 | LEO Privacy-Applock,Hide,Safe | LEO Privacy | com.leo.appmaster | 3.7.7 | 50,000,000+ |
| 3 | Vault-Hide SMS, Pics & Videos | Vault | com.netqin.ps | 6.4.22.22 | 10,000,000+ |
| 4 | Keeper®: Free Password Manager | Keeper | com.callpod.android_apps.keeper | 10.1.2 | 10,000,000+ |
| 5 | Hide Pictures Keep Safe Vault | Keepsafe | com.kii.safe | 7.3.1 | 10,000,000+ |
| 6 | Hide Pictures &Videos - Vaulty | Vaulty | com.theronrogers.vaultyfree | 4.3.3 | 5,000,000+ |
| 7 | Gallery Vault - Hide Pictures | GalleryVault | com.thinkyeah.galleryvault | 2.9.1 | 5,000,000+ |
| 8 | Hide Photos, Video-Hide it Pro | Audio Manager | com.hideitpro | 5.4 | 5,000,000+ |
| 9 | Hide Photos in Photo Locker | Photo Locker | com.handyapps.photoLocker | 1.2.1 | 5,000,000+ |
| 10 | Video Locker - Hide Videos | Video Locker | com.handyapps.videolocker | 1.2.1 | 5,000,000+ |
| 11 | LOCX Applock Lock Apps & Photo | LOCX | com.cyou.privacysecurity | 2.3.1.013 | 5,000,000+ |
| 12 | Secret AppLock for Android | Secret AppLock | com.amazing.secreateapplock | 6.9 | 5,000,000+ |
| 13 | Pic Lock- Hide Photos & Videos | Pic Lock | com.xcs.piclock | 1.9 | 1,000,000+ |
| 14 | HidePhoto | HidePhoto | com.domobile.hidephoto | 1.9 | 1,000,000+ |
| 15 | Hide Pictures PhotoSafe Vault | PhotoSafe | com.slickdroid.vaultypro | 2.0.1 | 1,000,000+ |
| 16 | Private Photo Vault | Private Photo Vault | com.enchantedcloud.photovault | 1.8.3 | 500,000+ |
| 17 | Calculator Vault- Gallery Lock | Calculator | com.calculator.vault | 8.5 | 500,000+ |
| 18 | Private Text Messaging & Calls | Coverme | ws.coverme.im | 2.6.4 | 500,000+ |

through various attack vectors. To ensure the reader's comprehension of the 18 vault application examinations, we wanted to reinforce the following terms used throughout the case study:

**Folder "shared_prefs"** represents the folder that is located in /data/data/package_name for the vault applications.

**Swap attack** is a method in which an examiner can swap an assigned value in a certain file with a self-created one in order to reset the password, gesture for login or the status of an application (e.g. switch on / off the pattern lock).

**Password attacks** which represents a variety of attacks against application passwords and gestures. In some instances, a rainbow table attack was used. In other instances, a brute force attack was used. Generally, these password attacks helped us in identifying the passwords used to access the application. In many of the applications, the password space was low (4-6 characters), and sometimes limited to digits, making a brute force attack a practical option.

### *5.1. AppLock 2.16.3*

Our artifact analysis showed that AppLock stored unencrypted pictures and videos in separate folders under /sdcard/.MySecurityData/dont_remove. The folders were named with different random strings / hash value. In each folder, the photos / videos were stored in the subfolder .video / .image respectively. Also, the application stored the 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 folder. Even though this approach was available only if the pattern lock of the application was activated, the pattern lock can be activated either by a previous user or an examiner can swap the value of the tag is_image_lock_pattern to true by modifying file com.domobile.applock_preferences.xml. As long as the pattern lock was activated, examiners can swap the original value in tag image_lock_pattern in the same XML file with a new one that was created by encoding the SHA1 hash value of a new gesture to Base64 format. For example, the value agYrmzRS42ZAcYGhv5Lqc+ntTEg= is the hash value for a 7 byte hex value 0x00, 0x01, 0x02, 0x04, 0x06, 0x07, 0x08 which represents gesture 'Z' (every byte maps to 0 - 8 points of the pattern lock). This will allow an examiner to log into the application using the newly created gesture. If the gesture was set previously, another option was to decode the Based64 value in the tag image_lock_pattern in the file com.domobile.applock_preferences.xml. For instance, in our case, the value was agYrmzRS42ZAcYGhv5Lqc+ntTEg=. When decoded with Base64, it became a 20 byte SHA1 hash value – 6a062b9b3452e366407181a1bf92ea73e9ed4c48. This hash was easily broken using a rainbow table attack and revealed the original byte sequence / pattern representing the gesture 'Z'.

### *5.2. LEO Private 3.7.7*

The first weakness we found in LEO Private is the storage of the password which was found in cleartext in shared_settings.xml in the shared_prefs folder tagged with password. Thus, if the phone is active, an examiner can log into the application using the mobile phone directly. If the phone is unavailable, we observed that the photo and video were stored and treated differently. The video was still stored in the original folder /sdcard/DCIM/Camera without any encryption. In order to hide video files, LEO Private only changed the file extension to '.leotmi'. Photos were actually encrypted and stored in /sdcard/.DefaultGallery/DCIM/Camera with the extension '.leotmi' as well. Having a closer look showed that the encryption / decryption is packed in the native library /lib/armeabi-v7a/libLeoImage.so. The library

runs the eXtended Tiny Encryption Algorithm (XTEA) to encrypt the first 1024 bytes of the photo where the Key is hard coded into the library. To decode, one can either call the function `leo_decrypt_v1(const char *a1)` or copy the encrypted files to `/sdcard/.DefaultGallery/DCIM/Camera` on another device that has LEO Private installed (regardless of the actual user password).

### *5.3. Vault 6.4.22.22*

Vault stored the encrypted content in the directory `/sdcard/SystemAndroid/Data/MTc1MDU4OQ==` which contained two subfolders named `.video` and `.image`. Inside these subfolders, we found our target files with `.bin` extensions. Note, the `Data` folder contained several Base64 encoded folders. The only one of relevance in our testing was the `MTc1MDU4OQ==` folder. While analyzing the application source code, we found that Vault only encrypts the first 80 bytes of each media file by using a repeating single byte which will be referred to as $B$. In order to retrieve $B$, we utilized the file header information, e.g. in case of JPEG it is `0xFFD8`, which revealed that $B$ = `0x3D`. In other words, to decrypt the videos and images, one simply has to XOR `0x3D` against the first 80 bytes. An interesting side note is that Vault takes and stores a picture every time someone enters an incorrect password. These pictures are stored in `/sdcard/SystemAndroid/Data/LTIxMDY4ODk5OTA=` and uses the identical encryption mechanism (XOR) but with $B$ = `0xFA`.

### *5.4. Keeper 10.1.2*

Out of all 18 applications, we deemed Keeper as the most secure. We were not able to reconstruct the original content except by employing a brute force attack on the password hash. Although we could not gain unauthorized login into the application, we will explain our findings as well as how to retrieve the hash value of the password. In contrast to many other applications, Keeper organizes its metadata in a SQLite database instead of XML which can be found in `/data/data/com.callpod.android_apps.keeper/databases/vaultapptest@gmail.com.sql` where the e-mail address became the username during installation. The most important table in the database was `setting` (shown in Fig. 1). On the other hand, the encrypted content can be found in `/data/data/com.callpod.android_apps.keeper/cache/record_files`.

The actual encryption scheme is best explained in Fig. 1 where we will focus on the right side first. Keeper uses Key-1 in order to encrypt all images / videos which is created by `SecureRandom()` when the application is initialized. This Key-1 is encrypted using Key-2 which is generated from the user password, a Salt-1 and IterationCount. Salt-1, IterationCount and encrypted Key-1 can be found in the '`setting_str`' column when the column '`name`' = '`encryption_params`' and is stored in a BLOB data type. The BLOB is separated into 1 byte for Mode, 3 bytes for IterationCount, 16 bytes for Salt-1 and 80 bytes for Key-1.

The left side of Fig. 1 shows the login process which is handled in a similar manner. Again, the data is stored in the `setting` table where column '`name`' = '`enc_pass`'. The content of the cell (marked in red) consists of a 22 bytes randomly generated Salt-2, and the 31 bytes Base64 encoded password hash. Hash and Salt-2 are both encoded with *bcrypt* - an open source library (Provos and Mazieres, 1999). Note, while Fig. 1 shows the actual names of the encrypting / decrypting functions, Keeper used obfuscation techniques and hence the function names were different. Names in the figure are based on the open source *bcrypt.java* (*BCrypt source code*, n.d.). Also note that *bcrypt* uses its own encoding and decoding implementation for Base64.

In order to attack this algorithm, a brute force attack can be implemented as follows:

**Step 1:** Decode the Salt that was encoded as the second part of the highlighted string in Fig. 1 with the function `decode_base64(..)` of *bcrypt.java*.

**Step 2:** Hash the attempted password with the function `crypt_raw(..)` and compare the result with the third part of the highlighted string. The correct password can be acquired if the values match.

Given that there is no limitation on the password length or complexity, the success rate depends on the chosen password.

### *5.5. Keepsafe 7.3.1*

This application stored the cleartext password in the value of tag `master-password` of XML file `com.kii.safe_preferences.xml` in the application's `shared_prefs` folder. Thus, investigators can simply read the file and then log in. Having a closer look revealed that the photos and videos were renamed with a hash value (we did not investigate this further as it is not relevant for decrypting the files) and stored encrypted data in subfolders of `/sdcard/.keepsafe/manifests/primary`. Each subfolder was named using the first 2 letters of the file, e.g., subfolder `2e` stored the encrypted media file `2e3c1f4aa10da568c5a26251457 79ad01163acc8`. The en / decryption functions were found in a native library in `/lib/armeabi-v7a/libcrypt_user.so` of the APK file of the application. An overview of the encryption mechanism is provided in Fig. 2 employing the following three major steps:

**Step 1:** The media file is divided into blocks of 16,384 bytes of data (the last block was equal to or smaller than 16384).

**Step 2:** Every block was then encrypted using the same encryption method, Initialization Vector (IV) and password. The size remained identical.

**Step 3:** Once all the blocks were encrypted, an Initialization Block (block zero) is added to the beginning of the media file which has a constant size of 3728 bytes. This block starts with Keepsafe's logo (3711 bytes), 16 byte IV and a one byte terminator. If a user clicks on any of

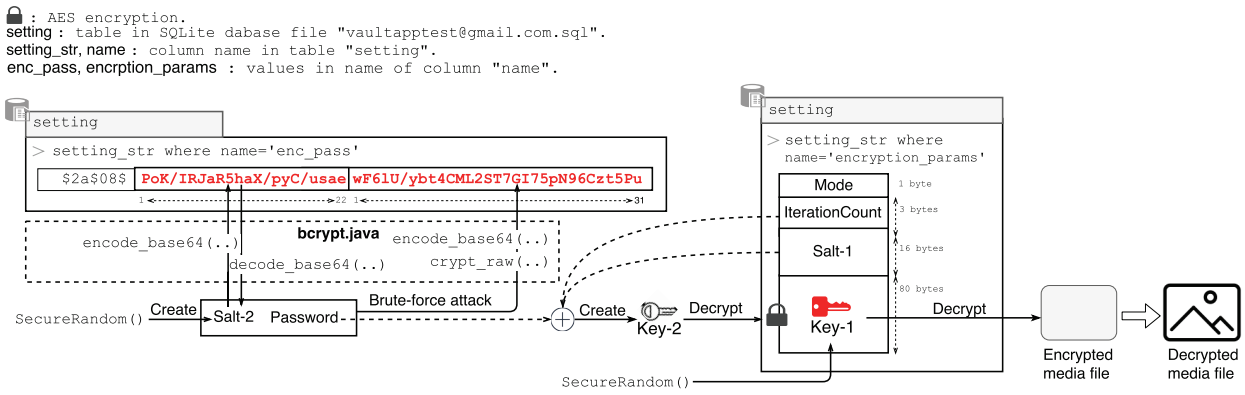


Figure 1: The decryption procedure for media files in Keeper.

the encrypted files, Keepsafe's logo would appear faking an average user to think that it is a PNG logo file.

The decryption is performed with the `processBlock(..)` function in the native library which needs the Key and the IV. While the Key is stored in the Initialization Block (block zero), the 32 byte Key can be found in tag `aec5a0aa33e25e4c9108939f6d6292c4507b045eAe` of the `com.kii.safe.secmanager.xml` file in `shared_prefs`. Once all encrypted blocks were decrypted using the `processBlock(..)` function, the Initialization Block was finally removed from the encrypted media file to recover the file to its original state.

### *5.6. Vaulty 4.3.3*

Vaulty's media files were found in `/sdcard/Documents/Vaulty/data` but with a new file extension '`.vdata`'. The videos were unencrypted. Vaulty additionally prepends the word '`obscured`' (`0x6F, 0x62, 0x73, 0x63, 0x75, 0x72, 0x65, 0x64`) to the photos' header. Thus, removing the '`obscured`' string and changing the file extension allows swift recovery of the pictures. The password of the application was stored as a Message Digest 5 (MD5) hash value in tag `password_hash` of the XML file `com.theronrogers.vaultyfree_preferences.xml` in `share_prefs`. Similar to case 1 in Sec. 5.1, examiners can either adopt a swap attack or a rainbow attack on the hash value.

### *5.7. GalleryVault 2.9.1*

In order to hide photos and videos, GalleryVault deconstructed them under `/sdcard/.galleryvault_DoNotDelete_1466617307/file/` where the first 10 bytes were overwritten with `0x00` and placed the original 10 bytes in a database. To reconstruct the files, one may simply query the database, identify the correct file and replace the header with the following steps:

**Step 1:** Query the database file `/sdcard/galleryvault_DoNotDelete_146610307/backup/galleryvault.db` using '`Select name, path, org_file_header_blob FROM file`' to retrieve the original header string. This returns the original name of the media file, the path of the deconstructed media files and the 10 bytes of wiped data for each file.

**Step 2:** After identifying the correct file by using the name and path, the first 10 bytes of the file was replaced with its original header.

If the phone is active, one may also perform a swap attack on the MD5 or SHA1 values which can be found in `Kidd.xml` in `shared_prefs` tagged `LockPin`. It remained unclear why the developers decided to store a 72 hex string value in this field which consists of the MD5 hash (32 hex characters) followed by the SHA1 hash (40 hex characters).

### *5.8. Audio Manager 5.4*

This vault application disguised itself as an audio manager on the device. Once a user holds a tap on the logo at the top of the screen, the real application is activated. Audio manager stored unencrypted photos and videos separately in the subfolder `New Album` of folder `Pictures` and `Videos` under `/sdcard/ProgramData/Android/Language/.fr/`. Also, the cleartext password was found in tag `password` of XML file `com.hideitpro_preferences.xml` in `shared_prefs`. Note that `New Album` is the default name of the album storing the hidden files, which may be customized by users.

### *5.9. Photo Locker 1.2.1 & Video Locker 1.2.1*

Photo Locker and Video Locker were released by the same developer. These two applications adopted the Advanced Encryption Standard (AES) algorithm to store the password. For the media files, they utilized AES to encrypt the first 16384 bytes. As shown in Fig. 3, a hard-coded string `"HANDY_APPShandyapps@gmail.com"` was found which was utilized to generate the secret Key object – Key-1 (object `javax.crypto.spec.SecretKeySpec` in JAVA SDK) for an AES algorithm. Key-1 can then be used for decrypting the AES-encrypted system e-mail address which was stored in label `SECRET_KEY` of XML file `com.handyapps.photoLocker_preferences.xml` for Photo Locker or `com.handyapps.videolocker_preferences.xml` for Video Locker in the `shared_prefs`

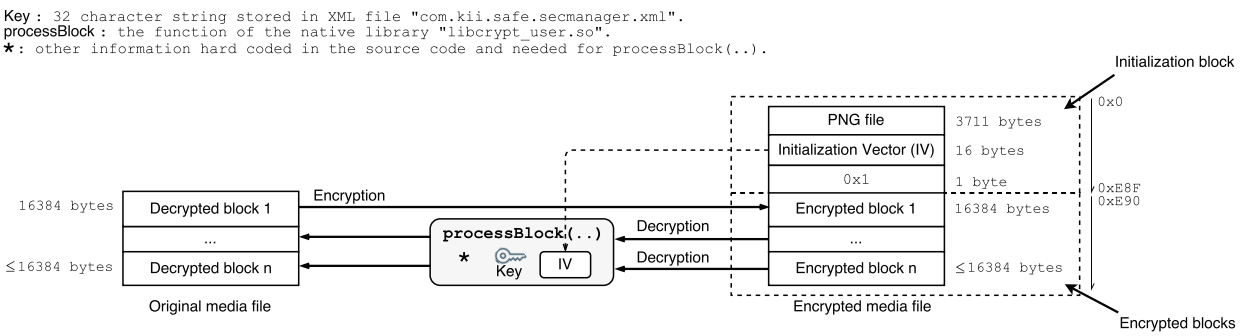


Figure 2: The decryption procedure for media files in Keepsafe.

folder. The e-mail is the input for generating the second Key object Key-2 which is utilized for protecting the password and media files. The encrypted password was found as the first line in file `/sdcard/.PL/.config` for Photo Locker and file `/sdcard/.VL/.config` for Video Locker. The media files were stored for these two applications in folder `/sdcard/.PL/Private Photo` and `/sdcard/.VL/Private Video` respectively. Reusing Key 2 while adopting AES allowed us to decrypt and recover the media files.

### *5.10. LOCX 2.3.1.013*

LOCX stored the media files in the hidden folder `/sdcard/.EncryptedFolder`. The image file `.enc_14668978236175365.jpg` and the video file `.2683375985d8b519e43a1fcf5e86c95c.mp4` were stored in that folder. Our code analysis revealed that `2683375985d8b519e43a1fcf5e86c95c` is the MD5 hash value for the path of the original video. For example, in this case, the path was `/storage/emulated/0/DCIM/Camera/20160627_102536.mp4`. To protect the images, LOCX prepends the original path as hex to the image header followed by 4 `0x0`s. The actual image is then XORed with the constant `0x7B` for every single byte. To recover a photo, one simply removes the prepended information and XORs again.

On the other hand, the video file was handled differently. Specifically, the first 524,288 bytes (`0x80000`) of the video were moved to the end of another created file by the vault application. Once the 524,288 bytes were moved, they were simply replaced with `0x0`s. We note that each of the 524,288 bytes inside the created file was XORed with the constant value `0x6E`. The content of the created video file included the file path of the deconstructed video file with a thumbnail of the video. From our testing, we located `.enc_14670617949374302.mp4`. Since inside `.enc_14670617949374302.mp4` we can observe the path of the deconstructed video file, we were able to hash the path of the deconstructed file, and locate the file name with same hash value (similar to the example shared above). This process allowed us to match the deconstructed video file with the vault application created file necessary for the reconstruction of the video. Therefore, the video can be reconstructed by adding the last 524,288 bytes of `.enc_14670617949374302.mp4` to the beginning of the deconstructed video and then XORing each of the 524,288 bytes with `0x6E`. Lastly, we found the MD5 hash value of the password in the tag `pinp` of file `com.cyou.privacysecurity_preferences.xml` under the application's `shared_prefs` folder.

### *5.11. Secret AppLock 6.9*

Secret AppLock stored unencrypted photos and videos in folder `/sdcard/.PixnArt12/.Photos` and `/sdcard/.PixnArt12/.Videos`. It also stored the cleartext of the password in tag `pin` of XML file `ApplockPreferences.xml` in the `shared_prefs` folder. Note that once the user typed in an incorrect password, photos would be taken from the front and back cameras and stored in folder `/sdcard/.hackImages`. This is forensically important because investigators may note who was trying to break into the vault application, but may also be useful for reasons to validate the integrity of the collected digital evidence.

### *5.12. Pic Lock 1.9*

Identical to the previous application, all information is simply hidden but not encrypted. The photos and videos were found in the subfolder of `Photos` and `Videos` in the folder `/sdcard/.AndroidLibs/33e75ff09dd601bbe69f351039152189/.SafeBox1`. Note, the random string / hash value might change for different devices / new installations. Additionally, we found the password in tag `pass` of XML file `Password.xml` in the `shared_prefs` folder.

### *5.13. HidePhoto 1.9*

HidePhoto stored the unencrypted photos and videos in folder `/sdcard/.MySecurityData2/dont_remove/68d30a9594728bc39aa24be94b319d21/.image` and `/sdcard/.MySecurityData2/dont_remove/d82c8d1619ad8176d665453cfb2e55f0/.video`, respectively. The subfolders in folder `dont_remove` were generated for different media files, which were labeled with random strings / hash values. Moreover, this application stored the MD5 hash value

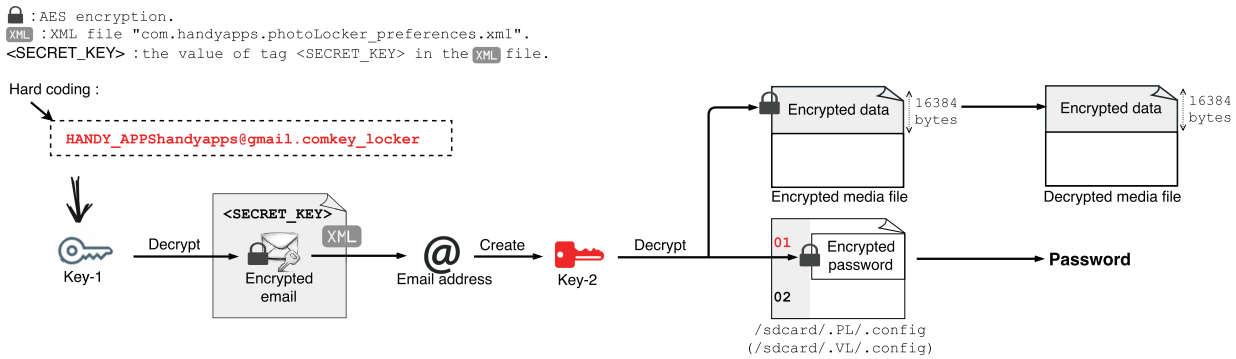


Figure 3: Media file and password decryption in Photo Locker and Video Locker.

of the password in tag `password` of XML file `com.domobile.hidephoto_preferences.xml` in the `shared_prefs` folder. Thus, examiners can achieve unauthorized access by swapping the hash value in tag `password` or utilizing a password attack (see Sec. 5.1).

### *5.14. PhotoSafe 2.0.1*

PhotoSafe stored the cleartext of the password in tag `password` of file `vaultypro.xml` in the `shared_prefs` folder. On the other hand, the application protected photos and videos using a similar approach discussed in Sec. 5.3. In subfolders `images` and `videos` under `/sdcard/.photosafe_DoNotDelete/camera`, the first 10 bytes of the photos and videos were XORed with the constant value `0xE7`.

### *5.15. Private Photo Vault 1.8.3*

Private Photo Vault stored the SHA1 hash of the password in `com.enchantedcloud.photovault_preferences.xml` in the application's `shared_prefs` folder with the tag `pin`. Since the password is limited to four digits, it can be easily brute forced. On the other hand, a swap attack is not possible as the application employs the password for encryption.

As shown in Fig. 4, the application utilized the Facebook open source library *Conceal* to perform the encryption and decryption of the files. Employing the decryption function `Crypto.getCipherInputStream(..)` in the library required a secret Key object – Key-1 (object `javax.crypto.SecretKey` in JAVA SDK) that is highlighted in red in Fig. 4. Based on our code analysis, Key-1 can be acquired by decrypting the Base64 decoded value of tag `enc_keys_pin` in file `com.enchantedcloud.photovault_preferences.xml`. The password used to log into the application is used to create the Key object `javax.crypto.spec.DESKeySpec`. This Key can be used for decrypting the value of tag `enc_keys_pin`.

Therefore, in order to bypass the login and recover encrypted photos and videos stored in folder `/data/data/com.enchantedcloud.photovault/files/media/orig`, some data was needed which included (a) The correct password (acquired using a password attack) and (b) The value of tag `enc_keys_pin`. After obtaining these two pieces, examiners can reuse the function `Crypto.getCipherInputStream(..)` to acquire the data stream of the decrypted media file.

### *5.16. Calculator 8.5*

This application disguised itself as a calculator on the system. Only when as user typed in the correct password the vault function is activated. The application stored the unencrypted photos and videos in `Pictures` and `Videos` in folder `/data/data/com.calculator.vault/files/locker1762`. Therefore, the only protection for the photos and videos was storing them in the folder that only root user can open. Additionally, the cleartext password was stored in tag `mpass` of XML file `com.calculator.vault_preferences.xml` in the `shared_prefs` folder.

### *5.17. Coverme 2.6.4*

For Coverme artifact analysis did not yield any results and therefore all our findings are based on analysis of the source code. Fig. 5 illustrates the encryption method adopted by the application from a high level perspective. Coverme applied triple-encryption for achieving the security of the media files and the password, where three different Keys were created.

**Key-1** was created using the password when the user first signed up. The application stored the hash value of Key-1 in column `password` of table `kexinuser` in the SQLite database `kexin.db` which is stored in `/data/data/ws.coverme.im/databases/`.

**Key-2** was created based on both the password and random bytes which were influenced by the current system time. Key 2 was encrypted using Key-1 and resultant ciphertext was stored in column `aeskey` of table `kexinuser` in the same database.

**Key-3** was the Key utilized for the encryption of the media files. It was created using the password and random bytes. The application stored the ciphertext of Key 3 which was encrypted using Key 2. The ciphertext was stored in column `keyByte` of table `localAesKey` in database file `kexin.db`.

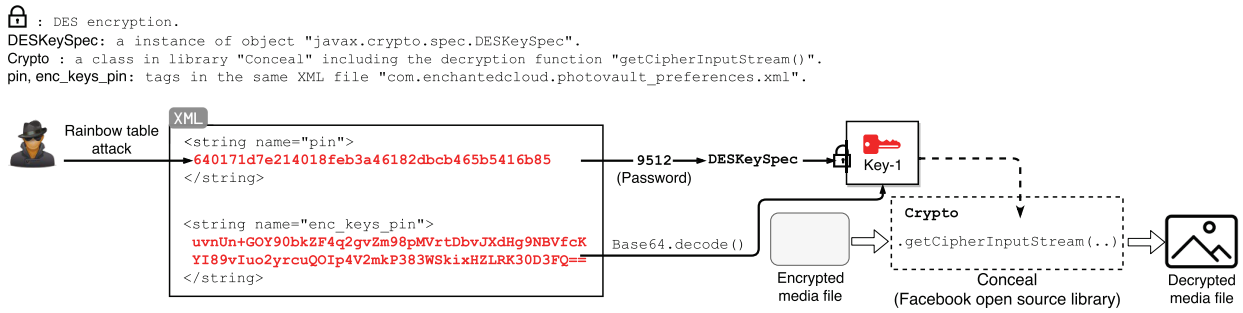


Figure 4: Media file decryption in Private Photo Vault.

Coverme's encryption implementation(Fig. 5), is relatively secure when compared to most of other applications. Without foreknowing the password, we were not able to create or recover the three Keys. However, we can still apply a brute force attack for retrieving the password. The password space for the application was 1-16 digits, but we anticipate that it is highly unlikely that users would choose a 16 digit password.

Since column `password` stored the hash value of the password (hashed twice using MD5 and SHA1 respectively), we reused the same function in the application for creating a Key-1 to the exhaustive passwords and then matched the hash value of Key-1 with the record found in column `password`. As soon as the hash values matched, Key-1 was used for decrypting Key-2; thus Key-2 can decrypt Key-3; and finally Key-3 can decrypt the media files that were stored with extension '`.dat`' in folder `/sdcard/coverme/images/hidden` and `/sdcard/coverme/video/hidden` respectively.

Even though a brute force attack was a possible solution, to crack a large password may be time consuming. We also note that because this application implemented encryption and decryption in a native library – `libNative-aes.so`, it can only be used on an ARM-based Central Processing Unit (CPU). In order to reuse the library, we implemented our decryption code in a separate Android project that we created.

## 6. Summary of findings

From the case studies, we see that most application developers protected their code using obfuscation and by implementing native libraries as shown in Table 2 in columns O and N under Code protection. This hindered the process of reverse engineering the applications. Notwithstanding, we observed that applications generally adopted similar security implementations with that could be exploited by examiners. For example, corresponding to column U, D and E, Table 2 shows that the photos and videos may be unencrypted, deconstructed or encrypted. As Table 2 shows, 6⁄18 and 8⁄18 applications only relocated / renamed the photos and videos respectively. Since the content of the media files was not changed, an examiner can easily recover such files.

On the other hand, 5⁄18 and 4⁄18 applications stored deconstructed photos and videos respectively by wiping data from the header, conducting a simple computation or storing partial data in other files. Comparatively, 7⁄18 applications actually encrypted the Photo, which we considered a more secure implementation. However, in the four cases tagged with $\times^H$ in Table 2, the encryption was implemented insecurely because the information for creating the Key that was used for encrypting the photos was hard coded in source code. The same issue was found in 3⁄6 applications that stored encrypted videos. We expected to recover the photos and videos for most of the 18 applications from the logical data we acquired. However, in some cases, if the recovery was more time consuming, or difficult to accomplish, we took the second best alternative of retrieving or swapping the password to recover the media files through unauthorized logins. As Table 2 shows, 7⁄18 vault applications (marked in column C) stored the password in cleartext on the Android device. The other 8⁄18 applications stored the hash value of the password where we were able to adopt a swap attack (marked as $\times^S$), rainbow table attack (marked as $\times^R$) or brute force attack (marked as $\times^B$).

There were two applications we were only able to apply a brute force attack for retrieving the password. Keeper utilized a salted password hashing function. Coverme did not store the hash value of the password but the hash value of the Key that was created by the password. We considered these two applications relatively more secure because they allowed a more complex password scheme or employed a more secure hashing function. Comparing these two applications, Coverme offered a better chance for the brute force attack because its password was limited to a maximum of 16 digits. Only 2⁄18 vault applications, encrypted the password. However, in both cases the applications generated their secret Keys through hard coded strings found in the source code. Lastly, in the column 'without root' we also marked if root privilege was necessary for the recovery as well as applications that needed a brute force attack were marked in column 'complex recovery'. In other words, except for these two applications, we can retrieve the hidden photos and videos from the vault applications all the time without difficulty, under the identified settings.

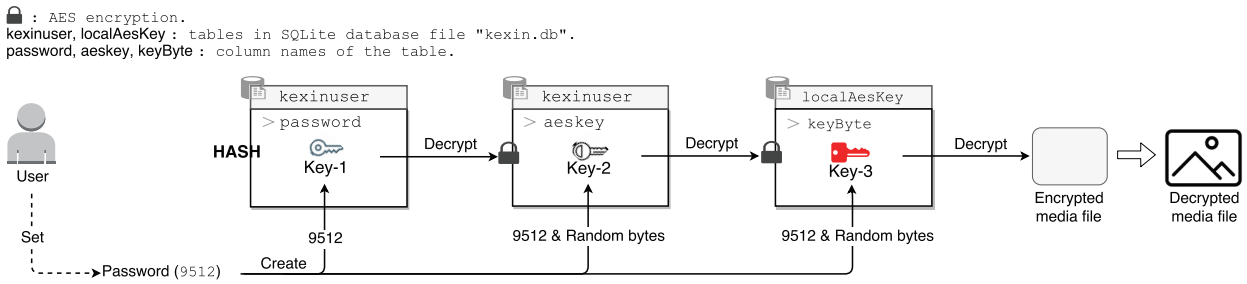


Figure 5: Secret files and password encryption in Coverme.

Table 2: Case study summary of results.

| | Application | Code protection | | Photo | | | Video | | | Password | | | Without root | Complex recovery |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | O | N | U | D | E | U | D | E | C | # | E | | |
| 1 | AppLock | × | | × | | | × | | | | $\times^{SR}$ | | ✓ | |
| 2 | LEO Privacy | × | × | | | $\times^{H}$ | × | | | × | | | ✓ | |
| 3 | Vault | × | × | | × | | | × | | | | | ✓ | |
| 4 | Keeper | × | | | | × | | | × | | $\times^{B}$ | | | ✓ |
| 5 | Keepsafe | × | × | | | $\times^{H}$ | | | $\times^{H}$ | × | | | | |
| 6 | Vaulty | | | | × | | × | | | | $\times^{SR}$ | | ✓ | |
| 7 | GalleryVault | × | | | × | | | × | | | $\times^{SR}$ | | | |
| 8 | Audio Manager | | | × | | | × | | | × | | | ✓ | |
| 9 | Photo Locker | × | | | | $\times^{H}$ | | | $\times^{H}$ | | | $\times^{H}$ | | |
| 10 | Video Locker | × | | | | $\times^{H}$ | | | $\times^{H}$ | | | $\times^{H}$ | | |
| 11 | LOCX | × | | | × | | | × | | | $\times^{SR}$ | | ✓ | |
| 12 | Secret AppLock | × | | × | | | × | | | × | | | ✓ | |
| 13 | Pic Lock | × | | × | | | × | | | × | | | ✓ | |
| 14 | HidePhoto | × | | × | | | × | | | | $\times^{SR}$ | | ✓ | |
| 15 | PhotoSafe | | | | × | | | × | | × | | | ✓ | |
| 16 | Photo Vault | | × | | | × | | | × | | $\times^{R}$ | | | |
| 17 | Calculator | | | × | | | × | | | × | | | | |
| 18 | Coverme | | × | | | × | | | × | | $\times^{B}$ | | | ✓* |
| | **Count** | 12/18 | 5/18 | 6/18 | 5/18 | 7/18 | 8/18 | 4/18 | 6/18 | 7/18 | 8/18 | 2/18 | 10/18 | 2/18 |

O: JAVA source code of the given application was obfuscated.
U: Photos or videos were found *unencrypted*.
E: Password, photos or videos were *encrypted* using a crypto algorithm.
#: Password *hash* was found in a file.
$^{R}$: *Rainbow table attack* to crack the password / gesture is possible.
$^{B}$: *Brute force* was the only approach to crack the password.
N: Application implements the critical functions in a native library.
D: The photos or videos were found being *deconstructed*.
C: Password was found in *cleartext* in a file.
$^{S}$: *Swap attack* to reset password / gesture is possible.
$^{H}$: Info for creating the encryption key found *hard-coded* in source code.
*:Password consisted of a maximum of 16 digits.

## 7. Other Vulnerabilities

Beyond retrieving / decrypting the passwords we also found two other vulnerabilities that can be used for achieving unauthorized access in some applications.

**Security e-mail:** If the password is forgotten, a user can submit a request for sending the password or a temporary password to a preset security e-mail. In the applications we tested, Applock (Sec. 5.1) and Pic Lock (Sec. 5.12) were found storing the cleartext of the security e-mail in in application related files. Examiners can modify the e-mail in these files using a self-owned e-mail address. Therefore, the user's passwords may be sent to that e-mail address instead.

**Security answer & question:** By correctly answering the preset security question a user can re-own access to a vault application. In our test applications, both the security answer and question were found in cleartext in Vaulty (Sec. 5.6) and Calculator (Sec. 5.16). Submitting the answer at the login page allows for unauthorized access to the applications.

Full artifact analysis findings, the results for each vault application, including the mentioned vulnerabilities is presented in a single Table in Appendix A.

## 8. Discussion

In this work we had to break the security of the vault applications to recover digital evidence. In fact, this work has already been used in a real world investigation as mentioned in the introduction. Whilst examining the forensic feasibility of retrieving data from vault applications, we also found that although applications were designed as privacy enhancing technologies, researchers and practitioners with thorough knowledge in reverse engineering and security may still be able to recover evidentiary data from them.

We learned that even though developers protected applications using techniques like obfuscation, encryption and the use of native libraries, there may be still security holes that can be exploited by digital forensic examiners. For

example, code obfuscation can only hinder examiners without reverse engineering knowledge; native libraries can be hijacked and reused in decryption; encryption may be bypassed using a swap attack, rainbow table attack or brute force attack etc. Generally speaking, our findings show that most security issues in the tested vault applications were due to (1) storing data locally (on the Android device) without encryption or with simple encryption (2) developers hard-coding constants as encryption Keys, Salts or other data related to the encryption implementation and (3) the minimal password space for some applications.

Regarding applications with relatively secure implementations, our testing offers ideas for resolving the aforementioned issues. For example, Keeper stored most of the important information on a server as opposed to local storage and the password could be made up of all possible characters (see Sec. 5.4). Additionally, to provide better source code protection, Keeper implemented code obfuscation[3] on the source code before compilation. Coverme for example packed the encryption method in a native library (see Sec. 5.17). These approaches used in combination can effectively increase the time and cost for reverse engineering. More importantly, a common feature for these relatively secure applications is that they use user credentials as part of the encryption process. Lastly it is of note that both secure applications generated Initialization Vectors / Salts through randomization.

## 9. Limitations

Our work has limitations. Primarily, artifact analysis can be conducted only if the targeted mobile device is qualified for data acquisition that usually requires root privilege, switched-on USB debugging mode or one of the requirements in Sec. 2.1. Although this is a limitation for a potential adversary, we argue that in real investigations, forensic examiners usually have physical access to the devices and the data on them. Second, our work focused on the logical acquisition of data from the mobile device. Should experimenters focus on physical acquisition, which may also include deleted data, we hypothesize that some applications may leave traces of unencrypted media files on the phone's memory. Third, compared to how many applications with 'vault' functions exist, the 18 applications we tested may be considered a rather small set of samples. We did focus our research however on the most widely downloaded applications. Fourth, we only focused on examining if the user's secret media files could be reconstructed. Some applications also have features for hiding applications or sending secure massages. These features were not part of our analysis. Fifth, our research was limited to stored data and did not focus on Random Access Memory, nor did it target network traffic. Lastly, our work was limited to only Android vault applications and did not take into account other mobile operating systems.

[3]Obfuscation changes the function name and variable name to irregular letters.

## 10. Conclusion & future work

As shown from our results, vault application developers indeed code protect their applications, yet through extensive reverse engineering efforts, we are still able to acquire hidden evidence. While this may not be privacy preserving, the discovered security implementations aid digital forensic examiners in reconstructing media files that may be relevant to a case. Future work should examine vault applications on iOS and also explore network traffic analysis. Work should also test the viability of reconstructing media files from media that is physically acquired. Future research should replicate our methods and focus on other privacy enhancing technologies such as mobile password vault applications. Lastly, work should explore the feasibility of designing a tool to automate the discovery of the security implementations of privacy preserving mobile applications.

## References


Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L. and Stransky, C. (2016), You Get Where Youre Looking For, *in* 'IEEE Symposium on Security and Privacy', pp. 289–305.

*A framework for analyzing and transforming Java and Android Applications* (n.d.). http://engineering.purdue.edu/~mark/puthesis.

Al Mutawa, N., Baggili, I. and Marrington, A. (2012), Forensic analysis of social networking applications on mobile devices, *in* 'Digital Investigation', Vol. 9.

*Android Debug Bridge — Android Studio* (n.d.). https://developer.android.com/studio/command-line/adb.html.

Anglano, C. (2014), 'Forensic analysis of WhatsApp Messenger on Android smartphones', *Digital Investigation* **11**(3), 1–13.

*Apktool - A tool for reverse engineering Android apk files* (n.d.). https://ibotpeaches.github.io/Apktool/.

*BCrypt source code* (n.d.). https://gist.github.com/cavneb/651613.

Bläsing, T., Batyuk, L., Schmidt, A. D., Camtepe, S. A. and Albayrak, S. (2010), An android application sandbox system for suspicious software detection, *in* 'Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software, Malware 2010', pp. 55–62.

Breeuwsma, M., De Jongh, M., Klaver, C., Van Der Knijff, R. and Roeloffs, M. (2007), 'Forensic data recovery from flash memory', *Small Scale Digital Device Forensics Journal* **1**(1), 1–17.

Chan, P. P. F., Hui, L. C. K. and Yiu, S. M. (2012), DroidChecker: Analyzing Android Applications for Capability Leak, *in* 'Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks', pp. 125–136.
**URL:** *http://doi.acm.org/10.1145/2185448.2185466*

*Colorado sexting scandal: High school faces felony investigation* (2016). http://www.cnn.com/2015/11/07/us/colorado-sexting-scandal-canon-city/.

Conlan, K., Baggili, I. and Breitinger, F. (2016), 'Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy', *Digital Investigation* **18**, S66–S75.

Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B. and Cheemalapati, S. (2014), 'SpanDex: Secure Password Tracking for Android', *USENIX Security 2014 (23rd USENIX Security Symposium)* (Vm), 481–494.

David Auerbach (2015), 'Fghcvq Is as Fghcvq Does'. http://www.slate.com/articles/technology/bitwise/2015/04/nq_mobile_vault_the_popular_encryption_app_has_laughably_crackable_encryption.html.
*DB Browser for SQLite* (n.d.). http://sqlitebrowser.org/.
*ded Homepage* (n.d.). http://siis.cse.psu.edu/ded/.
*dex2jar* (2016). https://sourceforge.net/projects/dex2jar/.
E2e (2016), 'How safe are your photos - Android Photo Vault app analysis'. https://www.e2e-assure.com/blog/AndroidPhotoVault_app_analysis_1/.
Enck, W., Octeau, D., McDaniel, P. and Chaudhuri, S. (2011), 'A Study of Android Application Security.', *USENIX Security* **39**(August), 21–21.
Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M. and Rajarajan, M. (2015), 'Android Security: A Survey of Issues, Malware Penetration, and Defenses', *IEEE Communications Surveys & Tutorials* **17**(2), 998–1022.
Gary Hawkins (2012), 'Just how well do Android privacy apps hide your sexy photos and secret texts?'. https://nakedsecurity.sophos.com/2012/11/12/android-privacy-apps/.
Gibler, C., Crussell, J., Erickson, J. and Chen, H. (2012), AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale, *in* 'Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)', Vol. 7344 LNCS, pp. 291–307.
Hoog, A. (2011), *Android forensics: investigation, analysis and mobile security for Google Android*, Elsevier.
Husain, M. I. and Sridhar, R. (2009), iForensics: forensic analysis of instant messaging on smart phones, *in* 'International Conference on Digital Forensics and Cyber Crime', Springer, pp. 9–18.
*IDA pro* (n.d.). https://www.hex-rays.com/products/ida/.
Iqbal, A., Marrington, A. and Baggili, I. (2013), Forensic artifacts of the ChatON Instant Messaging application, *in* 'Systematic Approaches to Digital Forensic Engineering (SADFE), 2013 Eighth International Workshop on', IEEE, pp. 1–6.
Jana, S. and Shmatikov, V. (2012), Memento: Learning secrets from process footprints, *in* 'Proceedings - IEEE Symposium on Security and Privacy', pp. 143–157.
*Jasmin User Guide* (n.d.). http://jasmin.sourceforge.net/guide.html.
Jason Moore, Ibrahim Baggili, F. B. (2016), 'Find Me If You Can: Mobile GPS Mapping Applications Forensics Analysis & SNAVP The Open Source, Modular, Extensible Parser', *The Journal of Digital Forensics, Security and Law: JDFSL* .
*JD-GUI A Java Decompiler* (n.d.). http://jd.benow.ca/.
Karpisek, F., Baggili, I. and Breitinger, F. (2015), 'WhatsApp network forensics: Decrypting and understanding the WhatsApp call signaling messages'.
Kim, J., Yoon, Y., Yi, K. and Shin, J. (2012), Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications, *in* 'IEEE Workshop on Mobile Security Technologies (MoST)', pp. 1–10.
Kim, K., Hong, D. and Ryou, J.-C. (2008), Forensic Data Acquisition from Cell Phones using JTAG Interface., *in* 'Security and Management', pp. 410–414.
Lessard, J. and Kessler, G. C. (2010), 'Android Forensics : Simplifying Cell Phone Examinations', *Small Scale Digital Device Forensics Journal* **4**(1), 1–12.
Mahajan, A., Dahiya, M. and Sanghvi, H. (2013), 'Forensic Analysis of Instant Messenger Applications on Android Devices', *International Journal of Computer Applications* **68**(8), 38–44.
Maus, S., Höfken, H. and Schuba, M. (2011), Forensic Analysis of Geodata in Android Smartphones, *in* 'International Conference on Cybercrime, Security and Digital Forensics, http://www. schuba. fh-aachen. de/papers/11-cyberforensics. pdf'.
Mehrotra, T. and Mehtre, B. M. (2013), Forensic analysis of Wickr application on android devices, *in* '2013 IEEE International Conference on Computational Intelligence and Computing Research, IEEE ICCIC 2013'.
Provos, N. and Mazieres, D. (1999), 'A future-adaptable password scheme', *USENIX Annual Technical Conference, ...* pp. 1–12.
Sahu, S. (2013), 'Forensic Analysis of WhatsApp on Android Smartphones', *International Journal of Engineering Research* **3**(5), 349–350.
Saltaformaggio, B., Bhatia, R., Zhang, X., Xu, D. and Richard Iii, G. G. (2016), Screen After Previous Screens: Spatial-Temporal Recreation of Android App Displays from Memory Images, *in* 'USEC'.
Schrittwieser, S., Frühwirt, P., Kieseberg, P., Leithner, M., Mulazzani, M., Huber, M. and Weippl, E. R. (2012), Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications., *in* 'NDSS', Citeseer.
*smali/baksamli* (2016). https://github.com/JesusFreke/smali.
Son, N., Lee, Y., Kim, D., James, J. I., Lee, S. and Lee, K. (2013), 'A study of user data integrity during acquisition of Android devices', *Digital Investigation* **10**, S3–S11.
*SQLite Home Page* (n.d.). https://www.sqlite.org/.
Vidas, T., Zhang, C. and Christin, N. (2011), 'Toward a general collection methodology for Android devices', *digital investigation* **8**, S14–S24.
Walnycky, D., Baggili, I., Marrington, A., Moore, J. and Breitinger, F. (2015), 'Network and device forensic analysis of Android social-messaging applications', *Digital Investigation* **14**, S77–S84.
Xia, M., Gong, L., Lyu, Y., Qi, Z. and Liu, X. (2015), Effective real-time android application auditing, *in* 'Proceedings - IEEE Symposium on Security and Privacy', Vol. 2015-July, pp. 899–914.
Yang, S. J., Choi, J. H., Kim, K. B. and Chang, T. (2015), 'New acquisition method based on firmware update protocols for Android smartphones', *Digital Investigation* **14**, S68–S76.
Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P. and Wang, X. S. (2013), 'AppIntent: analyzing sensitive data transmission in android for privacy leakage detection', *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* pp. 1043–1054.
Zhang, X., Breitinger, F. and Baggili, I. (2016), 'Rapid Android Parser for Investigating DEX files (RAPID)', *Digital Investigation* **17**, 28–39.
Zhou, Y., Zhang, X., Jiang, X. and Freeh, V. W. (2011), Taming information-stealing smartphone applications (on android), *in* 'Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)', Vol. 6740 LNCS, pp. 93–107.

## Appendix A. Acquisition summary

To provide a fast look up for readers, we summarized our findings from our artifact analysis which includes important files with critical data such as a user's password, the directory of the hidden photos / videos etc. that may be potentially required by forensic examiners. Ordered similar to Table 1, Table A.3 contains data about the first 9 applications and Table A.4 contains data about the rest of the 9 applications.

Table A.3: Summary of findings from artifact analysis for vault applications number 1 – 9.

| | Application | File | Found |
|---|---|---|---|
| 1 | AppLock | /sdcard/.MySecurityData/dont_remove/<HASH>/.image | Unencrypted photos |
| | | /sdcard/.MySecurityData/dont_remove/<HASH>/.video | Unencrypted videos |
| | | /data/data/com.domobile.applock/shared_prefs/com.domobile.applock_preferences.xml | SHA1-hashed pattern lock:<string name=image_lock_pattern>agYrmzRS42ZAcYGhv5Lqc+ntTEg=</string><br>Security e-mail:<string name="mail">vaultapptest@gmail.com</string> |
| 2 | LEO Privacy | /sdcard/.defaultGallery/DCIM/Camera | Encrypted photos |
| | | /sdcard/DCIM/Camera | Unencrypted videos |
| | | /data/data/com.leo.appmaster/shared_prefs/shared_settings.xml | Password:<string name="password">9512</string> |
| 3 | Vault | /sdcard/SystemAndroid/Data/MTc1MDU4OQ==/.image | Converted photos |
| | | /sdcard/SystemAndroid/Data/MTc1MDU4OQ==/.video | Converted videos |
| 4 | Keeper | /data/data/com.callpod.android_apps.keeper/databases/<E-MAIL>.sql | Encryption parameter, bcrypt-hashed password |
| | | /data/data/com.callpod.android_apps.keeper/cache/record_files | Encrypted media files |
| 5 | Keepsafe | /sdcard/.keepsafe/manifests/primary | Encrypted media files |
| | | /data/data/com.kii.safe/shared_prefs/com.kii.safe_preferences.xml | Password:<string name="master-password">9512</string> |
| | | /data/data/com.kii.safe/shared_prefs/com.kii.safe.secmanager.xml | 32 byte Key:<string name="aec5a0aa33e25e4c9108939f6d6292c4507b045eAe">4a1dd381b63e324a15ddc77e38553ed90136f6102aca62304b32db356f5cf206</string> |
| 6 | Vaulty | /sdcard/Documents/Vaulty/data/ | Converted photos and unencrypted videos |
| | | /data/data/com.theronrogers.vaultyfree/shared_prefs/com.theronrogers.vaultyfree_preferences.xml | MD5-hashed password:<string name="password_hash">C5ttbRVOmM40s/Lk73bq6Q==</string><br>Security question & answer:<string name="security_question">Favorite number</string><br><string name="security_answer"> 3 </string> |
| 7 | GalleryVault | /sdcard/.galleryvault_DoNotDelete_1466617307/file/ | Deconstructed photos and videos |
| | | /data/data/com.thinkyeah.galleryvault/shared_prefs/Kidd.xml | (SHA1+MD5) Hashed password:<string name="LockPin">640171D7E214018FEB3A46182DBCB465B5416B850B9B6D6D154E98CE34B3F2E4EF76EAE9</string> |
| | | /data/data/com.thinkyeah.galleryvault/databases/galleryvault.db | Original name of the media files; path of the deconstructed media files; the wiped 10 bytes for each media file |
| 8 | Audio Manager | /sdcard/ProgramData/Android/Language/.fr/Pictures/NewAlbum | Unencrypted photos |
| | | /sdcard/ProgramData/Android/Language/.fr/Videos/NewAlbum | Unencrypted videos |
| | | /data/data/com.hideitpro/shared_prefs/com.hideitpro_preferences.xml | Password:<string name="password">9512</string> |
| 9 | Photo Locker | /sdcard/.PL/PrivatePhoto | Encrypted photos |
| | | /sdcard/.PL/.config | Encrypted password |
| | | /data/data/com.handyapps.photoLocker/shared_prefs/com.handyapps.photoLocker_preferences.xml | Encrypted e-mail address |

<HASH>: the folder was named with a random string / hash value that may change in different cases

<E-MAIL>: The e-mail address previously set up by users.

Table A.4: Summary of findings from artifact analysis for vault applications number 10 – 18.

| | Application | File | Found |
|---|---|---|---|
| 10 | Video Locker | /sdcard/.VL/PrivateVideo | Encrypted videos |
| | | /sdcard/.VL/.config | Encrypted password |
| | | /data/data/com.handyapps.videolocker/shared_prefs/com.handyapps.videolocker_preferences.xml | Encrypted e-mail address |
| 11 | LOCX | /sdcard/.EncryptedFolder | Deconstructed photos and videos |
| | | /data/data/com.cyou.privacysecurity/shared_prefs/com.cyou.privacysecurity_preferences.xml | MD5-hashed password:<string name="pinp">0b9b6d6d154e98ce34b3f2e4ef76eae9</string> |
| 12 | Secret AppLock | /sdcard/.PixnArt12/.Photos | Unencrypted photos |
| | | /sdcard/.PixnArt12/.Videos | Unencrypted videos |
| | | /data/data/com.amazing.secreateapplock/shared_prefs/ApplockPreferences.xml | Password:<string name="pin">9512</string> |
| 13 | Pic Lock | /sdcard/.AndroidLibs/33e75ff09dd601bbe69f351039152189/.SafeBox1/DefaultFolder/Photos | Unencrypted photos |
| | | /sdcard/.AndroidLibs/33e75ff09dd601bbe69f351039152189/.SafeBox1/Privatevideos/Videos | Unencrypted videos |
| | | /data/data/com.xcs.piclock/shared_prefs/Password.xml | Password:<string name="pass">9512</string> |
| | | /data/data/com.xcs.piclock/shared_prefs/Email.xml | Security e-mail:<string name="mail">vaultapptest@gmail.com</string> |
| 14 | HidePhoto | /sdcard/.MySecurityData2/dont_remove/<HASH>/.image | Unencrypted photos |
| | | /sdcard/.MySecurityData2/dont_remove/<HASH>/.video | Unencrypted videos |
| | | /data/data/com.domobile.hidephoto/shared_prefs/com.domobile.hidephoto_preferences.xml | MD5-hashed password:<string name="password">0b9b6d6d154e98ce34b3f2e4ef76eae9</string> |
| 15 | PhotoSafe | /sdcard/.photosafe_DoNotDelete/camera/images | Converted photos |
| | | /sdcard/.photosafe_DoNotDelete/camera/videos | Converted videos |
| | | /data/data/com.slickdroid.vaultypro/shared_prefs/vaultypro.xml | Password:<string name="password">9512</string> |
| 16 | Private Photo Vault | /data/data/com.enchantedcloud.photovault/files/media/orig | Encrypted media files |
| | | /data/data/com.enchantedcloud.photovault/shared_prefs/com.enchantedcloud.photovault_preferences.xml | SHA1-Hashed password:<string name="pin">640171d7e214018feb3a46182dbcb465b5416b85</string> |
| | | | Encrypted key:<string name="enc_keys_pin">uvnUn+GOY90bkZF4q2gvZm98pMVrtDbvJXdHg9NBVfcKYI89vIuo2yrcuQOIp4V2mkP383WSkixHZLRK30D3FQ==</string> |
| 17 | Calculator | /data/data/com.calculator.vault/files/locker1762/Pictures | Unencrypted photos |
| | | /data/data/com.calculator.vault/files/locker1762/Videos | Unencrypted videos |
| | | /data/data/com.calculator.vault/shared_prefs/com.calculator.vault_preferences.xml | Password:<string name="mpass">9512</string> |
| | | | Security question & answer:<string name="seq_question">Favorite number</string> <string name="seq_answer">3</string> |
| 18 | Coverme | /sdcard/coverme/images/hidden | Encrypted photo |
| | | /sdcard/coverme/video/hidden | Encrypted video |
| | | /data/data/ws.coverme.im/databases/kexin.db | Value of the password column in table kexinuser; value of the keyByte column in table localAesKey |

<HASH>: the folder was named with a random string / hash value that may change in different cases.